Asymptote 范例教程

刘海洋

2010 年 6 月 25 日

本文是基于 Asymptote 1.91 的一组教程。
http://asy4cn.googlecode.com/

目录

第一章　赵爽的弦图

赵爽博士是位老知识分子，研究兴趣是天文历法与算学，一生精研《周髀算经》。不过赵老爷子年轻的时候书籍都是手写，最近才紧跟时代潮流用上了电脑。现在他要修订他研究《周髀算经》的札记，决定使用 LaTeX 来排版。

现在他遇到一个难题，就是他要画出笔记中讲解勾股定理的一幅弦图。听人介绍，几经比较之后，他决定使用现在炒得火热的 Asymptote。

赵博士的原图是手画的，线框多有不直不准的，图 1.1 就是旧年据手稿做的雕版图，赵博士并不满意。赵爽博士理想中的图，线条要平整美观，文字要清楚整齐，图形还要上色：朱实自然得用红色，黄实也该用黄色，以与注文一致——就是图 1.2 的样子。

计议已定，赵博士要开始正式的绘图了。

1.1　绘图环境

Asymptote 的安装并不复杂，在 Windows 下面就是下载运行那个安装包，在 Linux 下面一般也只需要下载对应的压缩包，解压就可以使用了。哦，赵博士用的就是 Windows。

点图标运行 Asymptote，就出现了交互式的命令行，提示符是一个 >。输入命令：

```
draw((0,0) -- (3cm,4cm));   // 一条直线
```

赵博士装的 GSView 立即弹了出来，里面已经画出一条倾斜的直线。再输入 `quit`，程序退出，并留下了一个叫做 `out.eps` 的图形文件，小菜一碟。

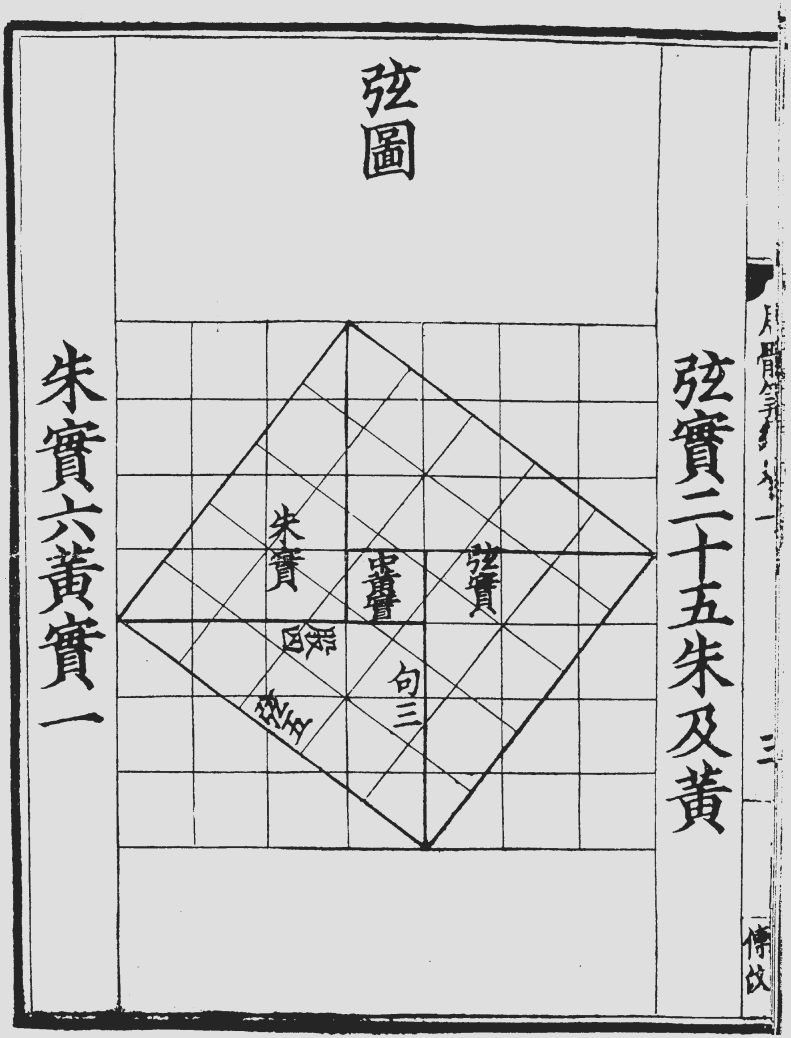


图 1.1: 旧年做的雕版

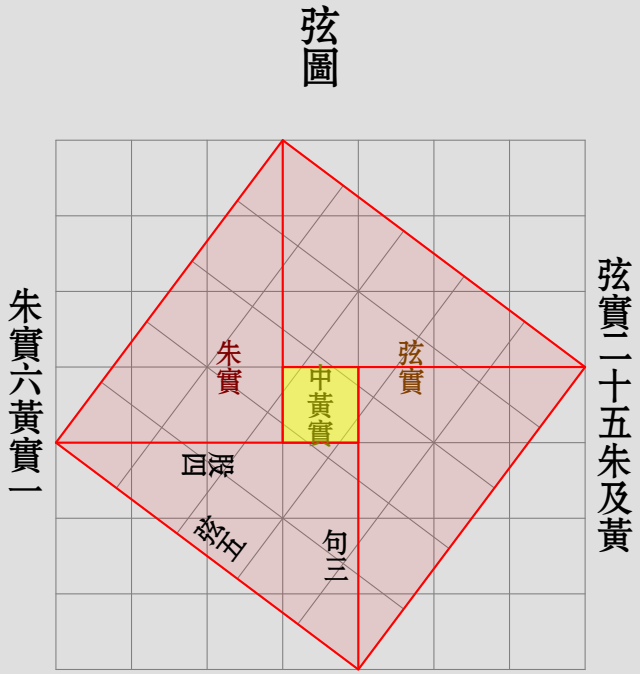

弦圖
朱實六黃實一
弦實二十五朱及黃
朱實
弦實
中黃實
句三
股四
句股各自乘，併之為弦實，開方除之即弦。案：弦圖又可以句股相乘為朱實二，倍之為朱實四，以句股之差自相乘為中黃實，加差實亦成弦實。

图 1.2: 理想中的新版本

这里稍稍解释一下上面的一句代码。draw 是画线的命令，更准确地说，是 Asymptote 中的函数：它带有一个参数 (0,0) -- (3cm,4cm)，参数外面是圆括号，整个命令以分号结束。里面的 (0,0) -- (3cm,4cm) 是由两个坐标连接而成的直线，坐标是在直角坐标系下的，可以带单位 mm, cm, pt, bp, inch, inches，其意义与在 TEX 中的一样，如果不带单位，则默认为 bp。行末以 // 开头的是注释，另有一种在 /* */ 之间的注释，与 C 语言相同。

不过赵博士写书讲求胸有成竹方才下笔，因此他更愿意使用更一种方式：打开一个文本编辑器，把上面的绘图命令都录入完毕，保存为一个名为 line.asy 的文件，最后把这个文件拖动到 Asymptote 的图标上面，就完成了整个作图。

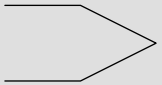
赵博士的小侄子不屑于这种快捷图标的编译方式，他直接进入命令行，输入 asy，就进入了 Asymptote 的交互环境；要是输入 asy line，就画出了刚才的保存的直线。

1.2 直线与绘图命令

弦图的图形其实很简单，都是直线、方块、三角形这些，而且为了计算的简便，所有长度也都是整数值。那么，首先就是在 Asymptote 中来画直线。

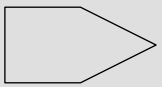
正如前面试验的时候做的那样，一条直线就是用 -- 把坐标连起来，再使用 draw 命令，直线就画出来了。事实上，可以用 -- 把坐标点连成折线：

```
draw( (0,0) -- (1cm,0) -- (2cm,0.5cm) -- (1cm,1cm) -- (0,1cm) );
```



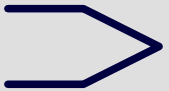
像这样把坐标用 -- 连结起来的，就成为一条路径。把直线而稍做修改，在后面连上一个特殊的坐标 **cycle**，就可以得到一条首尾相接的闭路径。如：

```
draw( (0,0) -- (1cm,0) -- (2cm,0.5cm) -- (1cm,1cm) -- (0,1cm) -- cycle );
```



画一条路径可以使用不同的颜色、粗细的笔，这只要给 draw 命令多加一个画笔 参数（多个参数用逗号分开）：

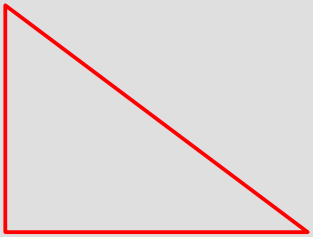
```
draw((0,0) -- (1cm,0) -- (2cm,0.5cm) -- (1cm,1cm) -- (0,1cm), darkblue+1mm);
```



这里 darkblue 是颜色，1mm 是线的粗细。darkblue+1mm 即指一毫米宽的深蓝色粗线。（可用的颜色名称可以参考 [3]）

赵博士画的是“勾三股四弦五”的红色三角形，这很容易：

```
draw( (0,0) -- (4cm,0) -- (0,3cm) -- cycle, red+0.5mm );
```

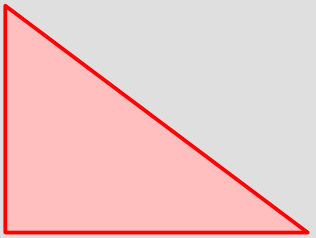


不过现在还需要的是实心的三角形，因此就需要一个新的绘图命令 fill，即填充。于是，红色三角形就成为

```
fill( (0,0) -- (4cm,0) -- (0,3cm) -- cycle, red );
```

但这样一来三角形的颜色就太重了，而且边界也不清楚。因此似乎应该先用浅红色填充一遍，然后再用深红色勾边。好在可以使用一个命令 filldraw 同时完成这两件事情，这样就不需要把一条路径写两遍了。即有：

```
filldraw((0,0) -- (4cm,0) -- (0,3cm) -- cycle,
    fillpen=palered, drawpen=red+0.5mm);
```



这里可以简单地直接写两个参数 palered, red+0.5mm，不过为了清晰起见还是使用“键 = 值”的写法，明确表示出填充的画笔和描线的画笔。

现在，赵博士的整个弦图的框架就呼之欲出了，就是画出四个三角形（图 1.3）：

```
filldraw( (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle,
    fillpen=palered, drawpen=red+0.5mm);
filldraw( (7cm,4cm) -- (4cm,4cm) -- (4cm,0) -- cycle,
    fillpen=palered, drawpen=red+0.5mm);
filldraw( (3cm,7cm) -- (3cm,4cm) -- (7cm,4cm) -- cycle,
    fillpen=palered, drawpen=red+0.5mm);
filldraw( (0,3cm) -- (3cm,3cm) -- (3cm,7cm) -- cycle,
    fillpen=palered, drawpen=red+0.5mm);
```

还应该画出弦图的中间的“黄实”，用黄色填充。这部分是一个正方形，可以使用现成的 box(角点，角点) 命令来产生矩形的路径，因而填充正中间的正方形就可以用：

```
fill( box((3cm,3cm), (4cm,4cm)), yellow );
```

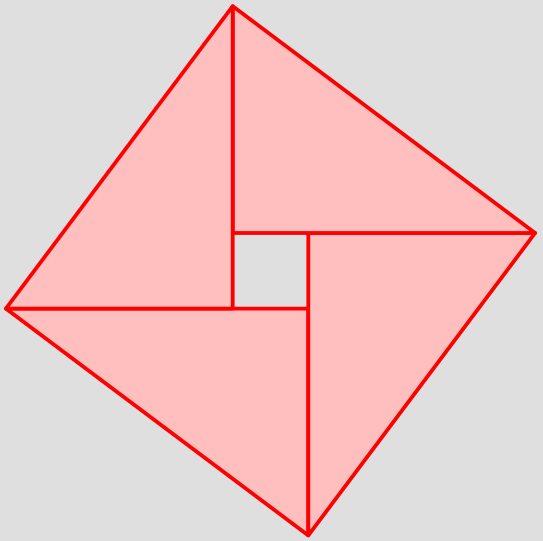


图 1.3: 弦图的初步框架

这个填充的命令应该放在画线之前（以免覆盖描的红线）。

回顾前面的代码，赵博士觉得连续地写四个 filldraw 命令太重复了。他读了 Asymptote 的文档 [3]，才知道多条路径可以用符号 ^^ 连起来，一起使用，于是立即着手改进原来的代码。

而且，由于还打算在图的后面画出参考网格，图形的颜色还应该设置为半透明的。好在这并不难实现，只要稍稍改动一下填充的画笔，使用 opacity(数值) 来设定有一定不透明度（取值为 0 ～ 1）的画笔，并把它加在原来的画笔上 [1]。于是赵博士最后写出了这样的代码：

```
fill( box((3cm,3cm), (4cm,4cm)), opacity(0.5)+yellow );
filldraw( (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle
    ^^ (7cm,4cm) -- (4cm,4cm) -- (4cm,0) -- cycle
    ^^ (3cm,7cm) -- (3cm,4cm) -- (7cm,4cm) -- cycle
    ^^ (0,3cm) -- (3cm,3cm) -- (3cm,7cm) -- cycle,
    fillpen=opacity(0.1)+red, drawpen=red+0.5mm );
```

至此，弦图的主要框架（图 1.4）就此完成。

[1] Asymptote 的 EPS 格式输出看不到透明效果，必须输出为 PDF 格式或从 PDF 格式转化为其他格式才能有透明效果。

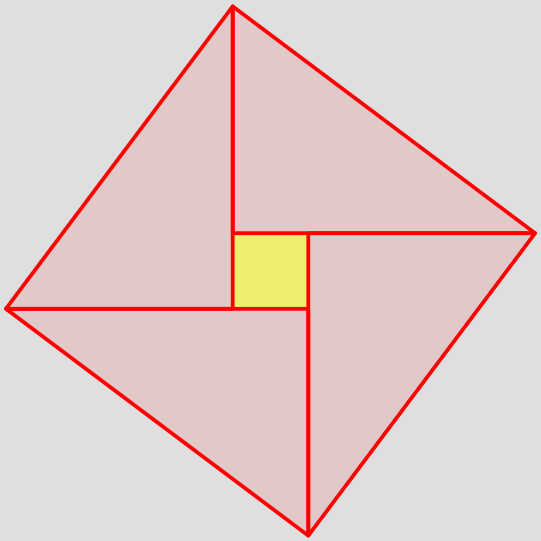


图 1.4: 弦图的进一步优化的框架

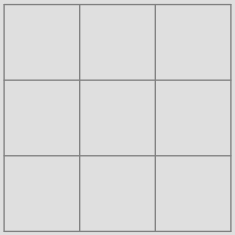
1.3 图形变换与功能模块的使用

然后是画作为长度参考的网格。其实网格一开始就该在图中画出来，这样后面画图准确与否才能看得清楚。不过现在赵博士是重作旧图，图样已定，网格就搁在弦图的主要图形后面才画了。

按说这个网格是十分简单的，无非就是画一些灰色的纵横细线。比如这个 3×3 的网格：

```
draw( (0,0) -- (3cm,0)
    ^^ (0,1cm) -- (3cm,1cm)
    ^^ (0,2cm) -- (3cm,2cm)
    ^^ (0,3cm) -- (3cm,3cm),
    gray );
draw( (0,0) -- (0,3cm)
```

```
    ^^ (1cm,0) -- (1cm,3cm)
    ^^ (2cm,0) -- (2cm,3cm)
    ^^ (3cm,0) -- (3cm,3cm),
    gray );
```



可无疑这个办法显得太麻烦了，赵博士要画的是 7×7 的网格，就要分别画出 16 条直线。这样的代码不仅不好写，而且容易出错，修改一下也很麻烦。

赵博士看到了手册 [3] 中讲循环语句的用法，似乎可以完成这件事。可是以赵博士的年龄，再去看什么编程什么变量的，命令不能一条一条执行下来，很不习惯，头脑就往往转不清楚。

于是赵博士就去论坛上咨询，一些人劝他去用几行循环语句，甚至有人已经把完整的函数做好了。但有一个结果特别引人注目，有人指出，在 math 模块中已经定义好了一个 grid 函数，只要拿来用就可以了。赵博士立即精神大振，来看这个 grid 函数：

```
picture grid(int Nx, int Ny, pen p=currentpen)
```

这个是在 math 模块中 grid 函数的原型。它说明 grid 函数有 Nx, Ny 两个整数类型的必需参数，一个可选的画笔，并且返回一个 **picture**（图）类型的对象。

要使用模块的功能，需要在绘图之前导入这个模块，这只要使用

```
import 模块名;
```

因此，要使用 math 模块中的 grid 函数，只要在代码中写

```
import math;
```

就可以了。

grid 函数的行为看起来很奇怪，调用它会在一个单独的图上画出一个 Nx × Ny 的网格，网格的左下角在原点，间距为 1。要使用 grid 函数画的图形，要使用 add(图) 命令，把这个图形加在当前的图上：

```
import math;
add(grid(10,10,gray));
```

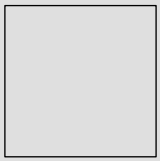
不过直接这样做的结果是只能得到一个小得已经看不清的网格。因此，必须对图形进行放缩。

Asymptote 提供了平移、旋转、放缩、倾斜、反射等各种的仿射变换，来对坐标、路径、图形等元素进行变换（严格的函数原型参考 [3]）：

```
shift(坐标)            // 按坐标平移
shift(x, y)            // 按（x，y）平移
scale(倍数)            // 按倍数放缩
xscale(倍数)           // x 轴方向按倍数放缩
yscale(倍数)           // y 轴方向按倍数放缩
scale(x, y)            // 在 x 轴、 y 轴方向分别按倍数 x，y 放缩
rotate(角度, z=(0,0)) // 按角度绕中心 z（默认为原点，逆时针）旋转
slant(因子)            // 按一定因子向右倾斜
reflect(a, b)          // 相对直线 a--b 反射
```

使用一个变换就是把这个变换乘在被变换对象的左边。例如一个放缩一个单位正方形：

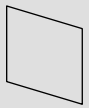
```
draw( scale(2cm) * box((0,0), (1,1)) );
```



这样的变换可以连续地做下去，例如

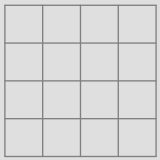
```
draw( rotate(90) * slant(0.3) * scale(1cm) * box((0,0), (1,1)) );
```

就是把一个单位正方形先放大，再倾斜，再旋转 90°：



有了这些变换，画出一个合适大小的网格就不再是什么难事了：

```
import math;
add( scale(5mm) * grid(4, 4, gray) );
```



赵博士的弦图有两个网格，不仅要大小合适，而且其中一个需要进行旋转和平移。平移的位置很明显，但旋转仍然需要一些计算。当然这难不倒精研天文算学多年的赵博士，这里弦实（5×5 的正方形）可以看作是顺时针旋转得到的，从朱实的三角形容易看出旋转的角度正好是 $\arctan(3/4)$。Asymptote 中也可以方便地调用返回角度的反三角函数 `aTan` 来计算这个角度[2]。更详细的数学函数列表，参看 [3]。（注意：C/C++ 中 `3/4` 表示求带余除法 $3 \div 4$ 的商，得到 0；但在 Asymptote 中 `3/4` 的结果是实数 0.75，整数的带余除法则用 `quotient(a,b)` 函数。）

于是，赵博士弦图中的网格，就可以这样方便地画出来了（图 1.5）：

```
// 网格
import math;
add( scale(1cm) * grid(7, 7, gray) );
```

[2]也可以使用 atan2 函数，但注意返回值是弧度。

```
add( shift(0,3cm) * rotate(-aTan(3/4)) * scale(1cm) * grid(5, 5, gray) );
// 弦图主体
fill( box((3cm,3cm), (4cm,4cm)), opacity(0.5)+yellow );
filldraw( (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle
    ^^ (7cm,4cm) -- (4cm,4cm) -- (4cm,0) -- cycle
    ^^ (3cm,7cm) -- (3cm,4cm) -- (7cm,4cm) -- cycle
    ^^ (0,3cm) -- (3cm,3cm) -- (3cm,7cm) -- cycle,
    fillpen=opacity(0.1)+red, drawpen=red+0.5mm );
```

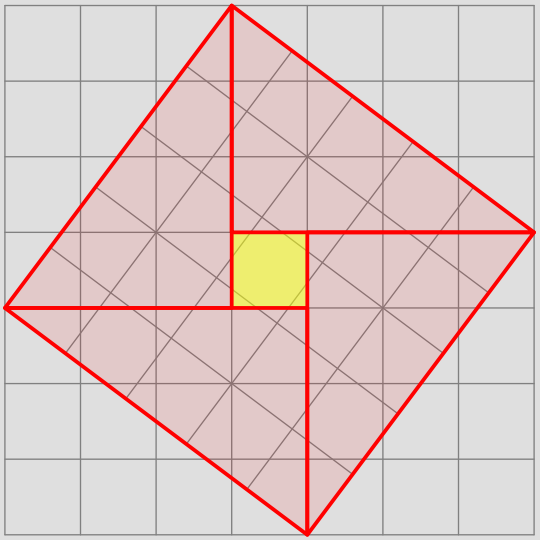


图 1.5: 带网格的弦图草图

1.4 标注文字

现在要进行的是文字的标注。按照勾股定理的约定，赵博士打算在一个红色三角形内标注“朱实”，在黄色矩形处标注“黄实”，并为拼得的整个大矩形标注“弦实”；在另一红色三角形的三边标注“勾三”、“股四”、“弦五”的尺寸；最后在图形两侧加上说明的文字。

在标注文字之前，对于中文标签，应该先定义好中文环境和字体。Asymptote 会调用 LaTeX 来进行标签的处理，因而需要设置的就是 LaTeX 的编译引擎与一般的中文 LaTeX 文件导言区。在这里，赵博士决定使用 XeLaTeX 引擎与 xeCJK 宏包来处理中文。为此，在 Asymptote 源文件中，他使用了下面的设置代码：

```
settings.tex = "xelatex";
usepackage("xeCJK");
texpreamble("\setCJKmainfont{SimSun}");
```

这里第一行是设置编译时所用的 TeX 引擎。后面 usepackage 命令就是 LaTeX 中的 \usepackage 命令的一个包装形式，里面的字符串参数就是宏包名；而 texpreamble 命令则把接收的参数直接放进 LaTeX 的导言区。

进行上述设置后，就可以正确使用中文标签了。标注的命令很简单，就是 label，参数正是标签文字和标签的位置。例如：

```
draw( (0,0) -- (1cm,1cm) -- (2cm,0) );
label( "中间", (1cm,0cm) );
```

就得到

可以在标签中使用任意的 LaTeX 代码，包括数学公式，例如：

```
label("$x = \sin\alpha$", (0,0));
```

就会正确地得到 $x = \sin\alpha$ 的标签。

现在，我们可以给弦图加上“朱实”、“黄实”和“弦实”的标签了。在前面的框架代码后面加上

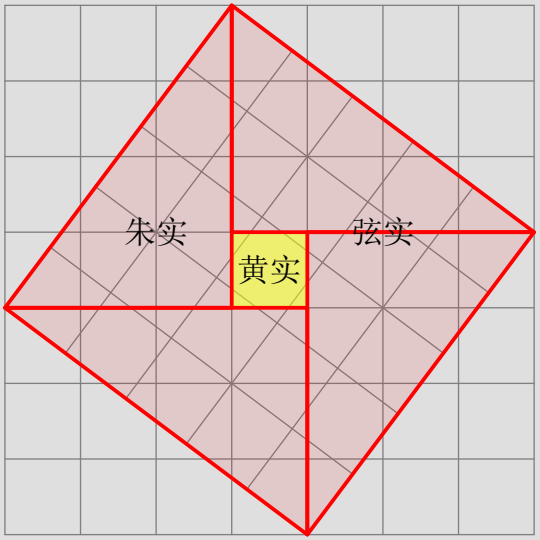


图 1.6: 带部分标注的弦图

```
label("朱实", (2cm,4cm));
label("黄实", (3.5cm,3.5cm));
label("弦实", (5cm,4cm));
```

以及设置字体的代码，就得到图 1.6 的结果。

下面则是要给三角形的三边进行标注。

与前面在一个点处标注不同，这里实际是给一条路径（直线）标注标签。因此，想要得到的是距离这条路径的中点一定方向距离加一个标签，而不是简单地取路径上的一点作为标签的位置。好在 Asymptote 确实也提供了这样的功能，仍然使用 label 命令，基本语法是：

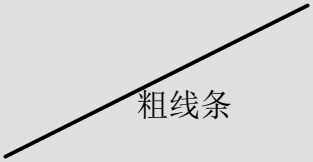
```
label(标签，路径)
```

这里默认会在路径中间的右侧（沿着路径行进方向）加标签。例如：

```
draw( (0,0) -- (4cm,2cm), linewidth(0.5mm) );
```

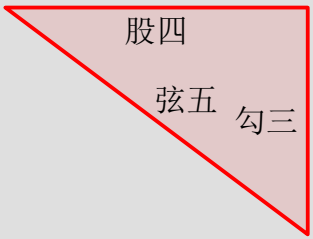
```
label("粗线条", (0,0) -- (4cm,2cm));
```

（其中的 linewidth 函数用来表示具有一定线宽的画笔）这段代码将得到



现在，赵博士就可以给三角形的三边加上“勾三”、“股四”、“弦五”的标签了，只要稍稍注意一下标签摆放的默认方向：

```
filldraw( (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle,
    fillpen=opacity(0.1)+red, drawpen=red+0.5mm );
label( "勾三", (4cm,3cm) -- (4cm,0) );
label( "股四", (0,3cm) -- (4cm,3cm) );
label( "弦五", (4cm,0) -- (0,3cm) );
```



不过，为了得到正确的标签位置，不得不把原来逆时针画的线用顺时针方向重写，这多少让赵博士有些恼火：为什么不能在路径的左边标注标签呢？确实可以，很简单，只要给 label 命令再加上 align=LeftSide 选项（或者简单地只用 LeftSide）就指定了在左边放置对齐。同理，还有向右对齐的 RightSide，在中间对齐的 Center 以及一般意义的相对方向 Relative(方向)。例如：

```
draw( (0,0) -- (4cm,2cm), blue, Arrow );
label( "LeftSide", (0,0) -- (4cm,2cm), align=LeftSide );
label( "RightSide", (0,0) -- (4cm,2cm), align=RightSide );
label( "Center", (0,0) -- (4cm,2cm), align=Center );

draw( (6cm,0)--(8cm,2cm), blue, Arrow );
label( "E", (6cm,0)--(8cm,2cm), Relative(E) );
label( "S", (6cm,0)--(8cm,2cm), Relative(S) );
label( "W", (6cm,0)--(8cm,2cm), Relative(W) );
label( "N", (6cm,0)--(8cm,2cm), Relative(N) );
```

E, S, W, N 分别是东南西北四个罗盘方向，用在 Relative 函数里面就表示相对于路径方向的四个方向。为明确，这里用 Arrow 选项在画线时加了箭头。

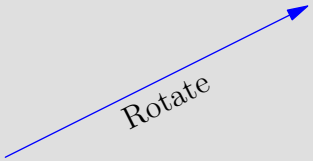
不过，这样的标签还是不能令赵博士满意。赵博士给三角形加标签，还希望标签随着三角形的边作旋转，使标签沿着边排列。为此，赵博士不得不又仔细查看了 [3]，在讲标注文字一节（label）他找到了一个构造标签的更高级的办法，即不仅仅是使用一个简单的字符串，而是使用

Label(标签)

函数进行构造。里面的“标签”参数仍然可以是原来的字符串，或是通过这个函数构造出来的高级的标签。这个函数可以带许多可选的其他参数，如 **position=位置** 的参数就可以指定标签放在路径中点之外的其他地方；而 **embed=嵌入变换方式** 的参数则可以解决标签自动旋转的问题。

这里暂且放下 **position** 参数。只来看 embed 参数的一个特例：Rotate(方向)。这个参数会让标签向着给定的方向旋转，如：

```
draw( (0,0)--(4cm,2cm), blue, Arrow );
label( Label("Rotate", Rotate((4,2))),
    (0,0)--(4cm,2cm) );
```



这里坐标 $(4,2)$ 正是这条直线的绘制方向。

终于，使用了上面的全部功能，赵博士完成了全部的图形标注工作（图 1.7）：

```
settings.tex = "xelatex";
usepackage("xeCJK");
texpreamble("\setCJKmainfont{SimSun}");

import math;
add( scale(1cm) * grid(7, 7, gray) );
add( shift(0,3cm) * rotate(-aTan(3/4)) * scale(1cm) * grid(5, 5, gray) );

fill( box((3cm,3cm), (4cm,4cm)), opacity(0.5)+yellow );
filldraw( (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle
    ^^ (7cm,4cm) -- (4cm,4cm) -- (4cm,0) -- cycle
    ^^ (3cm,7cm) -- (3cm,4cm) -- (7cm,4cm) -- cycle
    ^^ (0,3cm) -- (3cm,3cm) -- (3cm,7cm) -- cycle,
    fillpen=opacity(0.1)+red, drawpen=red+0.5mm );

label("朱实", (2cm,4cm));
label("黄实", (3.5cm,3.5cm));
```

```
label("弦实", (5cm,4cm));
label( Label("勾三",Rotate(S)), (4cm,0)--(4cm,3cm), LeftSide );
label( Label("股四",Rotate(E)), (4cm,3cm)--(0,3cm), LeftSide );
label( Label("弦五",Rotate((4,-3))), (0,3cm)--(4cm,0), LeftSide );
```

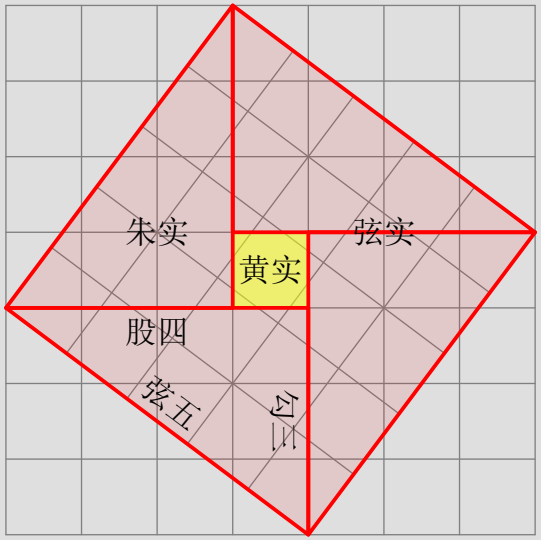


图 1.7: 带标注的完整弦图

1.5 习题和评注

1. 查阅参考手册 [3]，看看都有哪些颜色可用。看看你的系统中安装了哪些中文字体。然后修改图 1.7，使用另一种你喜欢的字体进行标注；并且将“朱实”、“黄实”和“弦实”分别用红、黄、橙色标注。

2. 查阅参考手册 [3]，看看 `draw`、`fill`、`label` 等命令都可以带哪些可选的参数，并自己举例子试验一下效果。

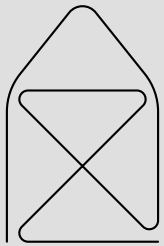
3. 代码

```
guide zhushi = (4cm,0) -- (4cm,3cm) -- (0,3cm) -- cycle;
```

可以把一个三角形的路径保存在变量 zhushi 中，以后就可以使用 draw(zhushi) 这样的命令对此路径进行操作了[3]。

考虑如何利用平移和旋转变换，只定义一个三角形的路径，就把弦图中的四个红色三角形都画出来。

4. 查阅手册 [3] 和 Asymptote 自带的例子，研究模块 roundedpath 的用法。并利用它尝试画出下面的图形：



看看 Asymptote 中还有什么用法简单而又有趣的模块。

5. （较难）研究在 LaTeX 里中文直排的方法，尽量精确地复现出赵博士理想中的弦图效果（图 1.2）。

本章的素材源自有关我国古代对勾股定理证明的一篇介绍性论文 [13]。本章的主要内容是 Asymptote 中直线几何图形的基本绘制方法。有关平面几何的数学图形是 Asymptote 以及 METAPOST 的传统长项和最重要的应用范围之一。不过 Asymptote 默认自动加载的 plain 模块功能比较原始，用来绘制平面几何图形往往比较麻烦，使用由 Philippe Ivaldi 开发的专门模块 geometry 可以更容易地画出各种 Euclid 平面几何的数学图形来，可参考此模块的英文文档 [6]。

[3]这里定义了一个 **guide** 类型的变量，我们译之为“路向”，以区别于“路径”（**path**）。两个概念在 Asymptote 中同源而有别，但本章中二者可以互换并无区别，因此为方便我们也不区别而通称为路径。有关 Asymptote 语言中路径与路向的区别，以及关于变量定义和使用的详细内容，请参考后续的章节及手册 [3]。

熟悉 C/C++ 语言的读者须注意，为了方便 TEX 指令在 Asymptote 中的书写，Asymptote 中的字符串内可以直接使用换行符，而不必手工使用转义符；另外，双引号内的字符串只有 \\ 和 \" 两种转义符，单引号也用来表示字符串，使用的转才与 C/C++ 语言相同。

习题 4 的素材源自 pgf/Ti*k*Z 宏包的文档 [11]。pgf/Ti*k*Z 宏包是基于 TEX 的绘图宏包，功能强大，一些方面更胜于老牌的 PSTricks 宏包。

第二章　André Deledicq 的铺砌插画

André 是一名兴趣广泛的法国数学教师，在他的新著《Le monde des pavages》（《铺砌世界》）中，打算画一幅有关羊的铺砌插画：

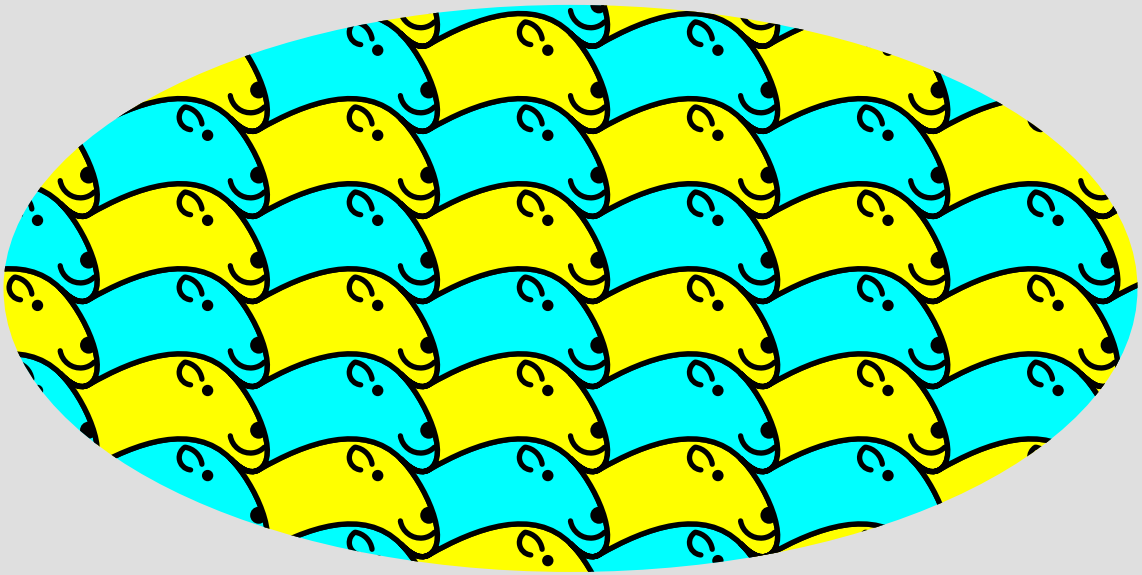
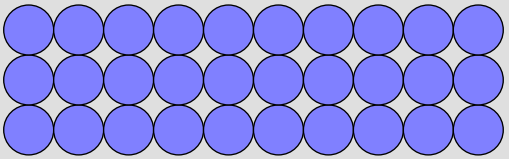


图 2.1: André 理想中的铺砌图

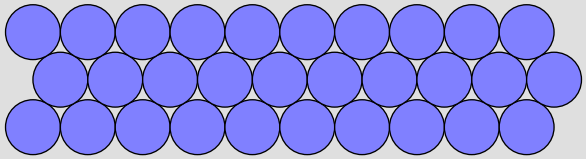
André 很清楚他要画的图形的数学理论，但 André 的朋友 Timothy 告诉他要画这样的图形多少是需要一些编程的知识的，对于他这样一位往日对计算机并不通晓的人来说可能会有困难。不过 André 并不以为意：这世上还有什么比数学更难的呢？于是他兴致勃勃的开始了。

2.1 从矩形到铺砌

铺砌图，顾名思义，就是像铺地板砖一样，把许多相同样式的图形平铺开来。不过，并不是什么图形都可以平铺填满整个平面的——比如圆形就不行。把许多圆形一个挨一个排列起来，也只能得到



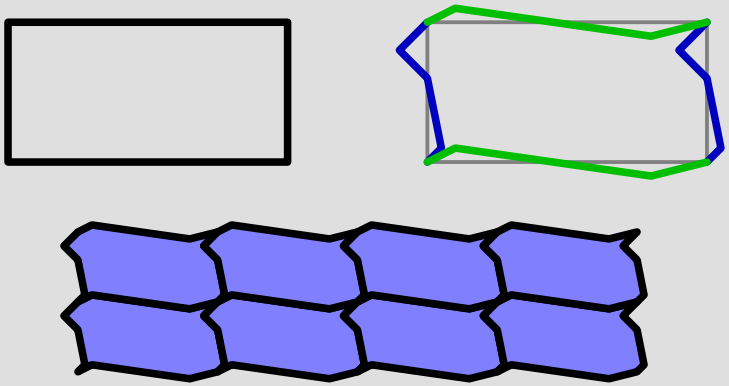
或者是



都会留下许多空隙。而矩形、平行四边形、六边形等等都可以不留空隙地把平面铺满。

但问题是，如何设计出 André 理想中的那种看起来形状不规则的铺砌图案呢？

身为数学教师的 André 当然有办法。其实不规则铺砌图案还是规则图案的变形。André 要画的羊形铺砌图，其实就是从矩形铺砌变化而来的。只要把一个矩形图案的上下两边、左右两边分别变形，使得变形后的上边与下边、左边与右边还对应重合，就依然可以完美地拼合起来。这正是铺砌图案最基本的构成方式：



有了这个方法，对复杂的铺砌图，也只要从一个基本形状（比如矩形、正六边形）开始变形，就等到铺砌所需要的一块“砖”。

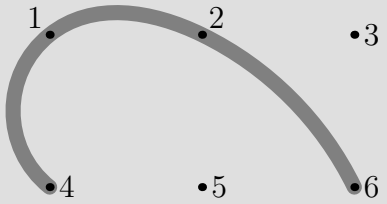
因此，要画出羊头形状铺砌图，只要把一个矩形按照上面的要求变形为一个羊头形状，在不同的位置重复画出就可以了。

2.2 变量与曲线

下面的问题就是，怎么画一个羊头呢？更具体地说，怎么画出羊头的曲线呢？

那么，首先要了解如何在 Asymptote 中描述曲线。1.2 节中提到 -- 连结一组坐标就成为直（折）线段；类似地，用 .. 连结坐标就得到经过这些坐标点的曲线：

```
size(5cm,0);
pair z1 = (0,1), z2 = (1,1), z3 = (2,1),
     z4 = (0,0), z5 = (1,0), z6 = (2,0);
guide p = z4 .. z1 .. z2 .. z6;
draw(p, gray+2mm);
```



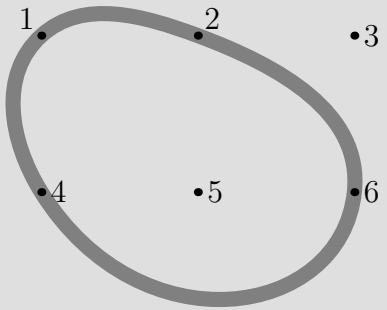
在这里，我们定义了一些变量以使代码清晰（这里略去了画点和标签的代码）。**pair** 类型的变量 z1, …, z6 保存六个坐标，**guide** 类型的变量 p 保存一条曲线的路向。因而上面 size 之后的绘图代码就相当于

```
draw( (0,0) .. (0,1) .. (1,1) .. (2,0), gray+2mm );
```

其中前面的一句 size(5cm,0) 表示代码中的坐标只是相对位置，最后将整个图形按比例放缩为 5 cm 宽[1]。类似地，也可以使用 size(0,4cm) 把图形放缩到 4 cm 高。

最重要的当然还是曲线的表示。以 .. 连结的坐标会以一种尽量接近圆弧的方式连为经过这些点的光滑曲线。与画直线类似，**cycle** 可以作为一个特殊的坐标产生闭合曲线，即一条闭路向：

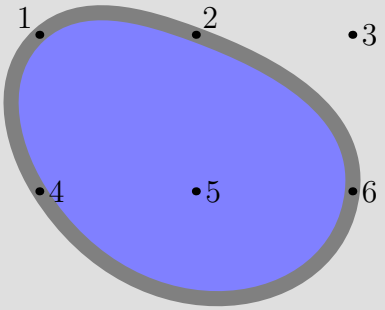
```
guide q = z4 .. z1 .. z2 .. z6 .. cycle;
draw(q, gray+2mm);
```



[1]注意坐标、图形会被放缩，但画笔的宽度不会放缩。

变量不仅仅是给了坐标、路向等对象一个简洁的名字，它也使得对同一个对象重复使用并进行不同的操作变得十分方便：

```
fill(q, lightblue);
draw(q, gray+2mm);
```

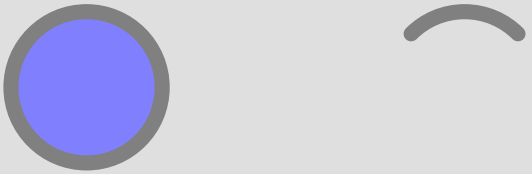


就像使用 box 可以直接得到矩形一样，最常用的曲线：圆、椭圆和圆弧，也可以使用现成的命令得到：

circle(c, r)	圆心 c，半径 r 的圆，这是逆时针方向的闭曲线；
ellipse(c, a, b)	中心为 c，长半轴 a，短半轴 b 的椭圆，这也是逆时针方向的闭曲线；
arc(c, r, angle1, angle2)	圆心 c，半径 r，角度从 angle1 到 angle2 的圆弧。

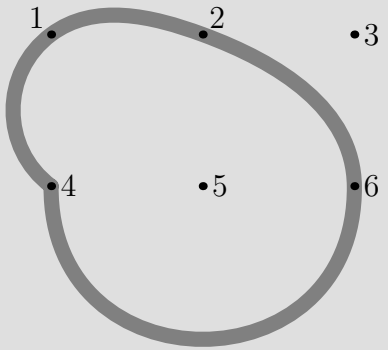
例如：

```
filldraw( circle((0,0), 1cm), lightblue, gray+2mm );
draw( arc((5cm,0), 1cm, 45, 135), gray+2mm );
```



一条用 **cycle** 产生的闭路向和简单地把首尾结点重合的路向是非常不同的。首先，只有闭路向可以填充颜色；其次，使用 **cycle** 连结的曲线在起点处是光滑连接的，而如果只是首尾结点重合则不会光滑连接。试将下面的曲线 q2 与上面的曲线 q 比较：

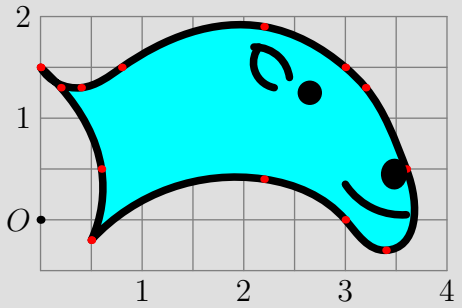
```
guide q2 = z4 .. z1 .. z2 .. z6 .. z4;
draw(q2, gray+2mm);
```



现在有了绘制曲线的方法，画出一个羊头就只是把草稿上的坐标连接起来而已。André 有一个纸上的草图，于是在描出几个点以后，他很快得到这样的结果（这里给图形增加了辅助网格）：

```
size(0,4cm);
pen outline = black+1mm;
// 头
guide head = (0.5,-0.2) .. (0.6,0.5) .. (0.2,1.3) .. (0,1.5) .. (0,1.5)
    .. (0.4,1.3) .. (0.8,1.5) .. (2.2,1.9) .. (3,1.5) .. (3.2,1.3)
    .. (3.6,0.5) .. (3.4,-0.3) .. (3,0) .. (2.2,0.4) .. (0.5,-0.2) .. cycle;
filldraw(head, cyan, outline);
dot(head, red+1mm); // 画出羊头曲线上的结点
// 五官
```

```
fill( circle((2.65,1.25), 0.12), outline );
fill( (3.5,0.3) .. (3.35,0.45) .. (3.5,0.6) .. (3.6,0.4) .. cycle, outline );
draw( (3,0.35) .. (3.3,0.1) .. (3.6,0.05), outline );
draw( (2.3,1.3) .. (2.1, 1.5) .. (2.15,1.7), outline );
draw( (2.1,1.7) .. (2.35,1.6) .. (2.45,1.4), outline );
```



在一开始，André 使用

```
pen outline = black+1mm;
```

定义一个 **pen** 类型的变量 outline 表示用来画羊头轮廓的画笔，以备使用。

然后，André 直接用 .. 连结一组坐标来定义羊的头部轮廓：

```
guide head = (0.5,-0.2) .. (0.6,0.5) .. (0.2,1.3) .. (0,1.5) .. (0,1.5)
    .. (0.4,1.3) .. (0.8,1.5) .. (2.2,1.9) .. (3,1.5) .. (3.2,1.3)
    .. (3.6,0.5) .. (3.4,-0.3) .. (3,0) .. (2.2,0.4) .. (0.5,-0.2) .. cycle;
```

需要尖角的时候，就使用重复的相同点（如这里的起点）；曲线变化大的地方，取的点也比较密集。

最后五官的绘制。眼睛是填充的小黑圆，鼻子是黑色的卵形，耳朵和嘴都是简单的曲线。

于是，只要把这样一个图形一个挨一个地重复画许多遍，就可以得到 André 想要的铺砌效果了。设计羊头形状的工作无疑是最关键也最复杂的，因此 André 的任务现在就已经完成了一半。

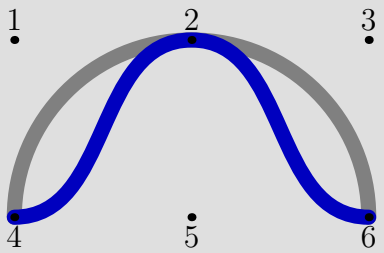
不过继承了法国完美主义风气的 Deledicq 老师，很快挑出了毛病：这只羊头部的轮廓，并不完全是按照 2.1 节对矩形变形得到的——他的手稿基本上是这样设计的，但在使用 Asymptote 上绘图时则只是在手稿上相当随意地取了一些结点连结得到曲线，这个轮廓想必也并不能严丝合缝地一个个拼起来。还有一件很令他恼火的事情则是：要画出羊头的轮廓，他要画的点太多了，一个尖角用两个结点表示，也太不符合他的简洁美学了。因此，这个看上去相当不错的羊头一号，就被 Deledicq 老师无情地否决掉了。他决定发扬数学教师严谨简洁的作风，再做出更完美的羊头二号来。

2.3 细致的曲线调整与曲线操作

在 Asymptote 中，除了简单地使用 `--` 和 `..` 连接坐标来定义直线与曲线以外，也提供了更丰富的手段来对曲线进行更为细致的调整与操作。于是，就有了方向、张力、卷曲值这些东西。

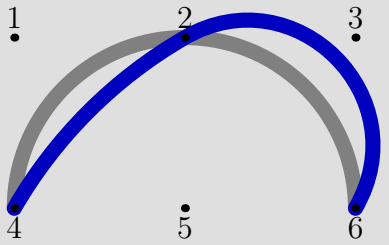
在曲线的结点处，可以使用以花括号括起的一个向量坐标来限定曲线在此处的切线方向 。例如：

```
draw(z4 .. z2 .. z6, gray+2mm);
draw(z4{right} .. z2{right} .. z6{right}, heavyblue+2mm);
```



在这里，常量 `right` 就相当于罗盘方向 `E`，也就是 `(0,1)`。事实上，在 Asymptote 中已经预定义好了上下左右四个方向：`up`, `down`, `left`, `right`，作为四个罗盘方向的同义词。此外，也可以使用预置的函数 `dir(角度)` 来表示指向此方向的单位向量，例如：

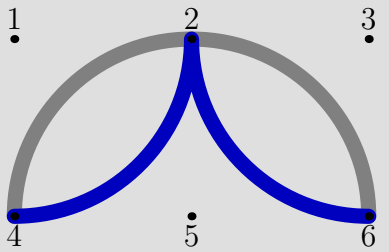
```
draw(z4 .. z2 .. z6, gray+2mm);
draw(z4{dir(60)} .. z2{dir(30)} .. z6{dir(-120)}, heavyblue+2mm);
```



当然，这里并不要求使用单位向量来限定曲线方向，使用任何长度的向量都得得到相同的效果。

就像上面的例子，限定曲线的切线方向无疑使得定义一条曲线更加直观和方便。不仅如此，我们甚至还可以在曲线同一结点的两侧设置截然不同的切线方向，此时曲线就会在这个结点的位置得到一个尖角。如：

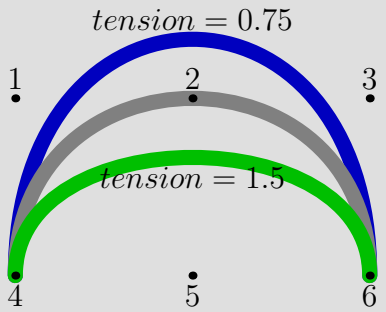
```
draw(z4 .. z2 .. z6, gray+2mm);
draw(z4 .. {up} z2 {down} .. z6, heavyblue+2mm);
```



了解了曲线的方向，再来看曲线的张力。大体上讲，这是一个相对模糊的概念，因为它并不像曲线切线方向那样具有明确的几何特征。如果把结点看做是钉子，而把 Asymptote 的曲线看做是用若干钉子固定起来的一根弹簧或是硬橡皮绳，或许有助于理解所谓张力的概念。在不同的钉子之间，弹簧的张力越大，弹簧就越趋于直线的紧绷状态；反之，张力小时，弹簧就会因为松驰而自动张成圆弧状。张力用 **tension 张力值** 来设定，张力值是不小于 0.75 的实数，默认值 1 使得曲线最接近圆弧形状。例如：

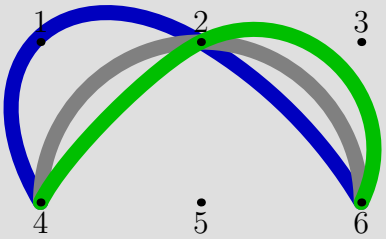
```
draw(z4 {up} .. tension 0.75 .. {down} z6, heavyblue+2mm);
draw(z4 {up} .. tension 1 .. {down} z6, gray+2mm);
```

```
draw(z4 {up} .. tension 1.5 .. {down} z6, heavygreen+2mm);
```



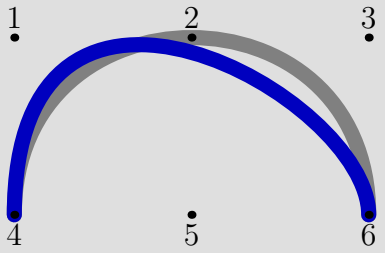
注意张力不仅影响两个结点之间的曲线形状，临近结点间的曲线形状也会受到影响。如：

```
draw(z4 .. tension 0.75 .. z2 .. z6, heavyblue+2mm);
draw(z4 .. tension 1 .. z2 .. z6, gray+2mm);
draw(z4 .. tension 1.5 .. z2 .. z6, heavygreen+2mm);
```



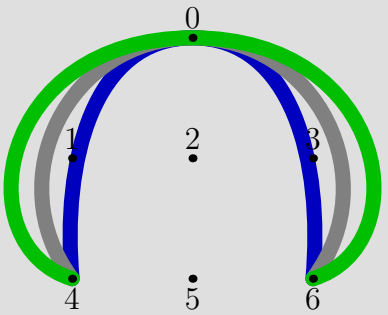
比弹簧或橡皮绳更强的是，Asymptote 曲线在结点两边的张力事实上也可以使用 **tension 起 and 末** 来分开设定：

```
draw(z4 {up} .. tension 1 .. {down} z6, gray+2mm);
draw(z4 {up} .. tension 0.75 and 2 .. {down} z6, heavyblue+2mm);
```



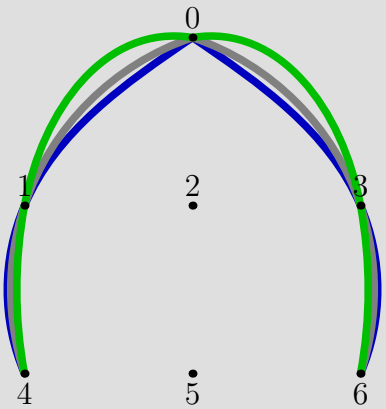
最后来看卷曲度。卷曲度影响曲线端点的弯曲程度，它的语法和设定曲线方向类似：**{curl 数值}**。卷曲度是一个非负值，其数值越大则曲线在端点的曲率越大，数值为 0 时则曲率趋于 0。在曲线的端点处，有默认的卷曲度 1，使得曲线接近圆弧。例如：

```
draw(z4 {curl 0} .. z0 .. {curl 0} z6, heavyblue+2mm);
draw(z4 {curl 1} .. z0 .. {curl 1} z6, gray+2mm);
draw(z4 {curl 5} .. z0 .. {curl 5} z6, heavygreen+2mm);
```



必须注意的是，卷曲度影响的是曲线的首尾端点的弯曲情况；而如果某个结点原本不是曲线的端点，那么曲线就会在此处折成尖角——特别地，如果相临端点都设置了卷曲度值，那么这两个结点之间就会是一段直线。例如：

```
draw(z4 .. z1 .. z0{curl 0} .. z3 .. z6, heavyblue+1mm);
draw(z4 .. z1 .. z0{curl 1} .. z3 .. z6, gray+1mm);
draw(z4 .. z1 .. z0{curl 5} .. z3 .. z6, heavygreen+1mm);
```

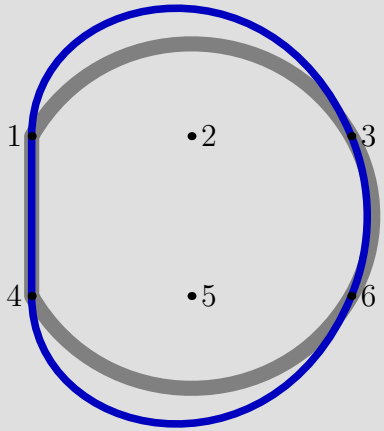


除了直线连接的符号 -- 和曲线连接的符号 ..，还有几个连接点坐标以得到曲线的连接符，它们分别是特殊的直线连接符 ---、特殊的曲线连接符 :: 以及将曲线首尾相接的 &。再加上连接几条不同曲线的 ^^，就是全部连接曲线的符号了。

连接符 --- 与 -- 类似，都得到直线，单独使用 --- 画出的直线和折线与使用 -- 得到的结果并不能看出区别。但事实上，用 --- 得到的并不是严格的直线折线，而只是非常接近直线的光滑曲线[2]。因而，如果把 --- 与 .. 配合使用，就能使得到的直线与曲线光滑地连接，而不像使用 -- 那样得到不需要的尖角。例如：

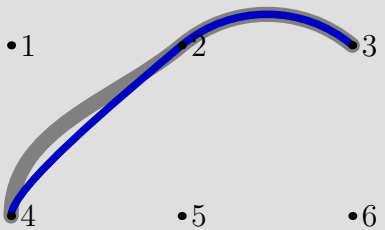
```
draw(z6 .. z4 -- z1 .. z3 .. cycle, gray+2mm);
draw(z6 .. z4 --- z1 .. z3 .. cycle, heavyblue+1mm);
```

[2]在 Asymptote 内部，连接符 --- 是 .. tension atleast infinity .. 的缩写。atleast 关键字一般并不直接使用，故本文不作讨论，可参考 [9]。



连接符 :: 则与 .. 类似，都得到曲线[3]。不过连接符 :: 有一个特性，就是它将尽可能使两个坐标之间的曲线没有拐点（即尽可能避免两坐标间出现 S 形曲线，效果是在 .. 的基础上自动增加张力的结果）。例如：

```
draw(z4{up} .. z2{dir(40)} .. z3, gray+2mm);
draw(z4{up} :: z2{dir(40)} .. z3, heavyblue+1mm);
```

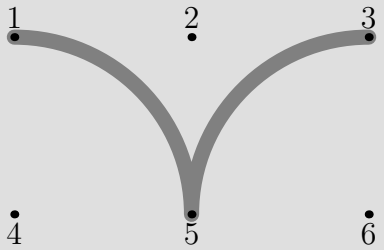


& 连接符用于把几段首尾点依次重合的曲线直接相连，而不改变原来连接前曲线的形状。这事实上与使用重复点直接用 .. 生成尖角的效果相同，但在 Asymptote 中，连接符 & 会在生成的曲线中去除第一条曲线的尾结点（它

[3]在 Asymptote 内部，连接符 :: 是 .. **tension atleast** 1 .. 的缩写。Asymptote 的曲线功能和语法大多源自 METAPOST，不过在 METAPOST 中与 :: 功能相同的连接符是 ...。

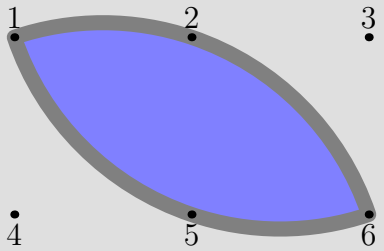
应该是与第二条曲线的首结点重合的），因而不会出现像之前直接使用 .. 生成的曲线那样会有重复的结点。例如：

```
draw(z1{right} .. {down} z5 & z5 {up} .. {right}z3, gray+2mm);
```



另一方面，相比直接单独绘制几条不相关的曲线，使用 & 连接符的一个重要作用是可以由此生成合适的闭曲线。例如：

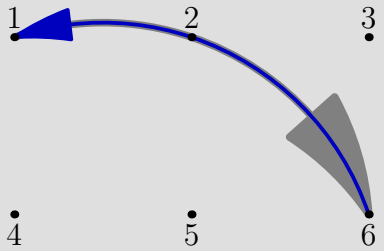
```
filldraw(z1 .. z2 .. z6 & z6 .. z5 .. z1 & cycle, lightblue, gray+2mm);
```



连接符 ^^ 在前面 § 1.2 中已经介绍过了。这里则指出它的一个特殊应用：在一个区域中“挖洞”填充。在数学中，平面上的一条简单闭曲线有的方向可以分为正负两种，逆时针方向为正，正时针方向为负。因此，在某个方向曲线围成的区域中用连接符 ^^ 加入一条反方向的闭曲线，就相当于在这个封闭区域中挖了一个洞。

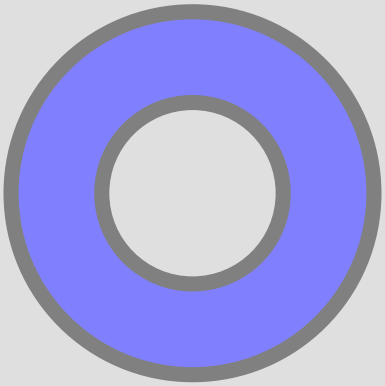
在给出相关的例子之前，我们首先给出 Asymptote 中把曲线方向逆转的函数 reverse(曲线)，它接受一条曲线（可以是路径或路向）作为参数，并返回此曲线的逆方向曲线。绘制时给曲线加上箭头可以明显看出其方向：

```
draw(z1 .. z2 .. z6, gray+1mm, Arrow);
draw(reverse(z1 .. z2 .. z6), heavyblue+0.5mm, Arrow);
```



于是，利用这种机制，在大圆内加上一个反方向的圆，把这两条曲线用 ^^ 连接，就可以用来填充得到环形：

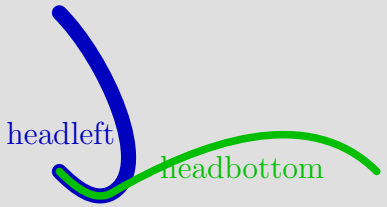
```
size(5cm, 0);
filldraw(circle((0,0), 2) ^^ reverse(circle((0,0), 1)), lightblue, gray+2mm);
```



需要说明的是，使用 & 连接两条曲线，得到的结果是一条连续的曲线；而使用 ^^ 连接两条曲线，得到的结果并不是一条曲线，而只是把两条曲线合并在一起的曲线数组（数组在稍后的章节介绍）。因而它们在使用时是不同的。

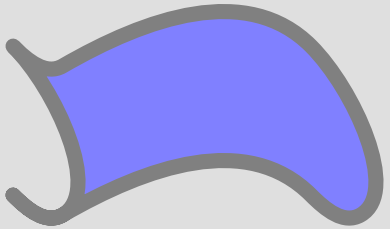
上面介绍的内容，差不多就是 Asymptote 中全部基本的曲线控制方式了。有了这些，Deledicq 老师就可以按照 § 2.1 中的方式，精确地设计羊头的轮廓了。他分别定义羊头左边和下面的两根线：

```
guide headleft = (0,1.5){SE} .. tension 1.4 .. (0.5,-0.2){dir(-150)} ..
      {NW}(0,0);
guide headbottom = (0,0){SE} .. {dir(30)}(0.5,-0.2) ..{SE}(3,0);
```



这里的两条曲线虽然看起来定义比较复杂，但它们都只是由三个结点连接而成的，定义更为直观，且在数学上更加简单。此外，明确限定了两条曲线在结点处的切线方向，可以保证它们可以精确拼接。于是，通过平移变换、使用 `reverse` 函数逆向并使用连接符 `&` 的拼接，就可以得到完整的羊头轮廓了：

```
guide head = headleft & headbottom &
    shift(3,0)*reverse(headleft) & shift(0,1.5)*reverse(headbottom) & cycle;
```

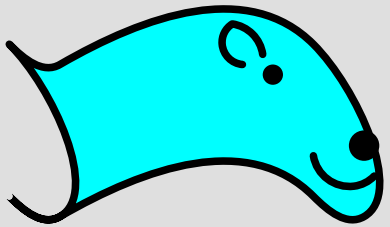


下面，羊的眼睛（`eye`）、耳朵（`ear`）、鼻子（`muzzle`）和嘴（`mouth`）就比较容易得到了：

```
guide eye = circle((2.6,1.2), 0.1);
guide ear = (2.3,1.3) .. (2.1,1.5) .. (2.2,1.7)
    & (2.2,1.7) .. (2.4,1.6) .. (2.5,1.4);
guide muzzle = circle((3.5,0.5), 0.15);
guide mouth = (3,0.4) .. (3.4,0.1) .. (3.6,0.2);
```

于是，André 就这样画出了一个完整的青色羊头：

```
size(5cm, 0);
filldraw(head, cyan, linewidth(1mm));
fill(eye ^^ muzzle);
draw(ear ^^ mouth, linewidth(1mm));
```



挑剔的 André 对这个羊头的曲线感到满意，因为作为一个铺砌图形，它绘制得完全精确。一个羊头已经可以很好地画出来了，剩下的问题就是重复地把这个羊头铺满平面，开始“铺砖”了。

2.4 循环与条件判断

要在平面上铺满相同的图形，就需要反复地在不同的位置进行绘制。Asymptote 中可以使用循环语句来完成这种重复的工作。

最常用的循环语句是 **for** 循环，其基本语法是：

```
for（初始化；执行条件；循环增量）
    循环语句体
```

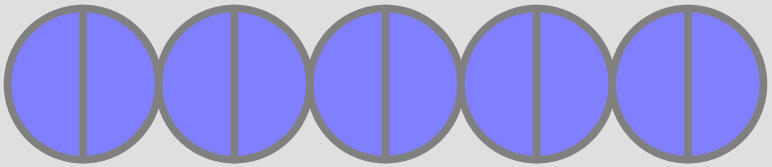
在一个循环语句中，Asymptote 首先运行初始化语句，随后开始不断运行循环语句体，每次运行语句体前检查执行条件以判断是否开始，而在运行语句体后运行循环增量的语句。例如下面的语句会画出起始角度为 0°, 30°, 60°, 90° 的四条曲线：

```
for (real angle = 0; angle <= 90; angle += 30)
    draw( (0,0){dir(angle)} .. {right}(6cm,0), gray+1mm );
```

在初始化部分定义了一个实数（**real**）类型的变量 angle，初始值为 0；进入循环的执行条件是角度变量 angle 不大于直角；而在每次循环体画完曲线后，变量 angle 的值就增加 30°。这里复合赋值运算符 += 表示在变量原有的基础上累加， var += x 就等价于 var = var + x，类似地减法、乘除法、取余数和乘方也有 -=、*=、 /=、%=、^= 这几种对应的赋值形式。对于整数 （**int**）类型的变量，还可以使用 ++var 和 --var 分别表示 var += 1 和 var -= 1。

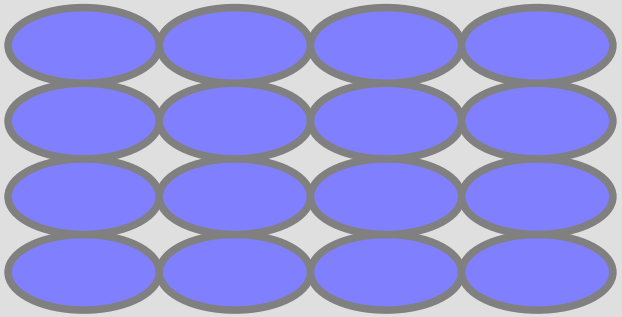
如果循环体的语句不止一个，可以用花括号把这些语句括起来。如：

```
for (int i = 0; i < 5; ++i) {
    filldraw(circle((2i*cm,0), 1cm), lightblue, gray+1mm);
    draw( (2i*cm,-1cm) -- (2i*cm,1cm), gray+1mm);
}
```



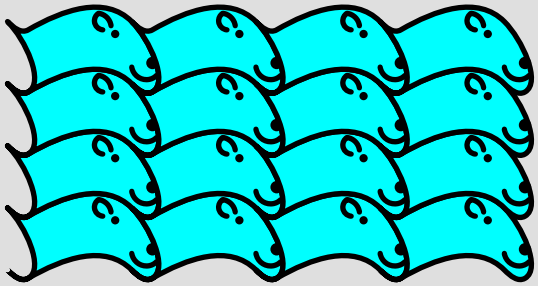
平面铺砌图形需要在横向和纵向两个方向重复延伸，因而可以使用两层嵌套的循环语句来完成这种图形：

```
for (real x = 0; x < 8cm; x += 2cm)
    for (real y = 0; y < 4cm; y += 1cm)
        filldraw(ellipse((x,y), 1cm, 0.5cm), lightblue, gray+1mm);
```



因此，有了前面的准备工作，André 就很容易通过一个双重循环来做出羊头的铺砌形状来：

```
// ……相关曲线的定义同前
size(7cm, 0);
for (int x = 0; x < 4; ++x) {
    for (int y = 0; y < 4; ++y) {
        transform pos = shift(3x, 1.5y);
        filldraw( pos * head, cyan, linewidth(2bp));
        fill( pos * (eye ^^ muzzle) );
        draw( pos * (ear ^^ mouth), linewidth(2bp));
    }
}
```



在这里，André 使用关于整数变量 x 和 y 的二重循环来控制整个铺砌图。在循环体中使用了变换类型 **transform** 的变量 pos 来保存从原点到每一块铺砌图形元的平移变换。在绘图时，给每条曲线都使用 pos 进行变换，就可以在正确的位置绘图，而不用直接修改原来的曲线。

现在，André 几乎已经得到了和图 2.1 相同的图形了。不过在 图 2.1 中，羊头的颜色是随位置交错变化的。一个笨办法是分别用两组循环来画出黄色和青色的羊头，但这会带来很多重复的代码。因此 André 决定在循环体内做一点手脚：出于数学教师的敏感，他发现羊头颜色的规律可以由整数 x + y 的奇偶性来判定，于是只要在循环体内增加一点判断……

Asymptote 的条件判断语句正好可以完成这项工作。条件判断语句的基本语法是：

```
// 只有一个分支
if (条件)
    条件为真执行的语句块
// 有 else 块，有两个分支
if (条件)
    条件为真执行的语句块
else
    条件为假执行的语句块
```

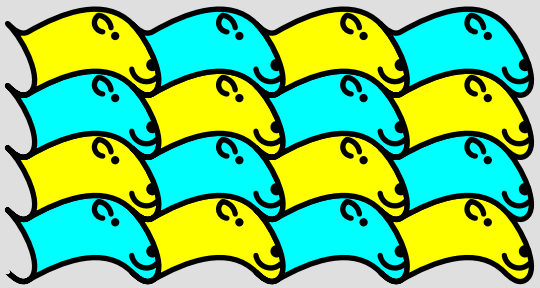
其中，可以用花括号括起多行的语句块。因而，可以使用如下的语句来通过判断 x + y 的奇偶性来决定画笔的颜色：

```
pen color;
if ( (x+y) % 2 == 0 )   // 判断 x+y 是否为偶数
```

```
    color = cyan;
else
    color = yellow;
```

把上面的控制颜色的判断语句用到前面的循环体中，就可以得到交错色彩的铺砌图了：

```
// ……相关曲线的定义同前
size(7cm, 0);
for (int x = 0; x < 4; ++x) {
    for (int y = 0; y < 4; ++y) {
        transform pos = shift(3x, 1.5y);
        pen color;
        if ( (x+y) % 2 == 0 )
            color = cyan;
        else
            color = yellow;
        filldraw( pos * head, color, linewidth(2bp));
        fill( pos * (eye ^^ muzzle) );
        draw( pos * (ear ^^ mouth), linewidth(2bp));
    }
}
```



至此，André 大松一口气，他的铺砌图形基本上就绘制完成了。不过在他结束这一切的时候，他的同事，Asymptote 的专家 Philippe 告诉他，其实有关条件判断的语句还可以使用一个条件表达式来简化。在这里，要判断的条件比较简单，而判断的结果也只是青、黄两种颜色的值，因此，可以使用条件运算符 ? : 来代替。条件运算符是一个特殊的三元运算符，它的语法是：

```
条件 ? 条件为真的值 : 条件为假的值
```

因此前面的颜色判断和赋值语句就可以简化为：

```
pen color = (i+j)%2==0 ? cyan : yellow;
```

现在，André 的铺砌图形就完成了。下面只要把他的图形剪裁一下，就大工告成了。

2.5 图形的剪裁与自定义图

André 最后的问题是把整个铺砌图形剪裁为椭圆形。为此，我们就需要引入在画线（draw）、填充（fill）、标注（label）之外的最后一种基本绘图命令，即剪裁（clip）。剪裁命令的基本语法是：

```
clip(图，曲线);      // 图的默认值是当前图 currentpicture
```

如果省略 **picture** 类型的图参数，则剪裁命令会把当前正在绘制的图用给定的曲线裁剪开来；显示指定图参数则只剪裁给定的图中的内容。

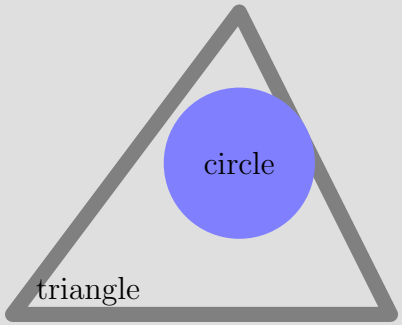
裁剪默认的当前图只是在绘图的最后进行的少许修饰，而事实上，clip 命令更多地是用在明确指定的图上，以完成一些用其他方法难以达成的效果。在第一 章中已经看到了产生一个带网格的图的 grid 函数与使用图的 add 函数，下面，我们首先说明如何自定义图并在其上绘图。

自定义一个图只要定义一个 **picture** 类型的变量。新定义的图相当于一块单独的画布，而画线、填充、标注这些命令都可以在最前面加上图的参数，表示在某一个图上进行绘图。例如：

```
picture mypic;       //自定义图
draw(mypic, (0,0) -- (3cm,4cm) -- (5cm,0) -- cycle, gray+2mm);  //画三角形
fill(mypic, circle((4cm,3cm), 1cm), lightblue);                 //画圆
label(mypic, "trangle", (0,0));      //标注
label(mypic, "circle", (3cm,2cm));
```

不过这些内容都是画在自定义的图 mypic 上的，而只有在默认图 currentpicture 上的内容才会真正输出被我们看到。因此，就又需要使用 § 1.3 中提到过的 add 命令把自定义图 mypic 中的内容加到 currentpicture 之上：

```
add(mypic);
```



add 函数还可以有更多的参数，更一般的形式是[4]：

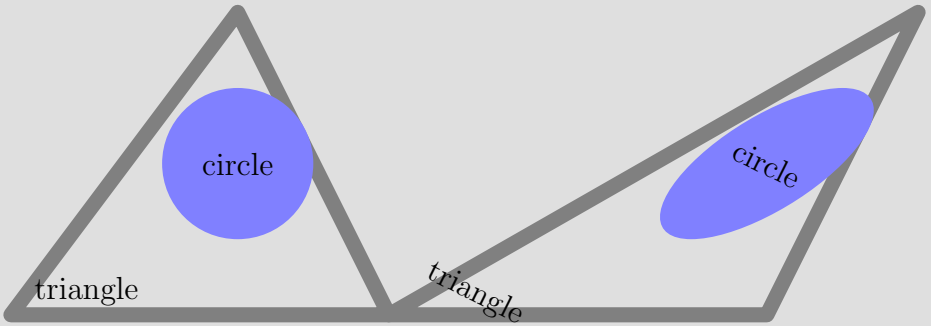
```
// 可以利用 size 函数自动缩放
add(目标图=currentpicture，源图，above=true);
// 不能自动缩放
add(目标图=currentpicture，源图，位置坐标，above=true);
```

add 函数的意义是：把源图的内容加到目标图的上方或下方，并使源图的原点重合于目标图的位置坐标，其中目标图的默认值是当前图 currentpicture，above 的真假值控制目标图是否在源图的上方，位置坐标不指定的形式两图原点重合。但需要注意，带位置坐标的形式不支持利用 size 函数自动缩放，此时只能通过 shift 变换改变源图的位置。通常，可以把目标图看做是已经有一些图案的背景画布，而源图是一些新的图案，add 命令会把这些新的图案贴在原来画布的某个位置上。

[4] add 函数有很多变形，还带有更多次要的参数，详细的语法形式参看手册 [3] 和 plain_picture.asy 中的源代码。

使用自定义图的一个好处是，画在自定义图上的所有内容作为一个整体，可以通过 add 函数对其重复使用，或者进行各种操作（如仿射变换）。例如，对于前面定义的图 pic，经过仿射变换[5]：

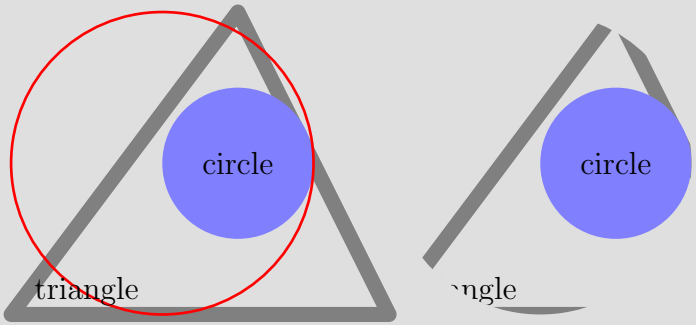
```
add(mypic);
add(slant(1)*mypic, (5cm,0));
```



把剪裁命令用在自定义的图上，则这个图的内容就会随之被剪裁，在使用 add 时，即可看到效果：

```
add(mypic);                              // 增加未剪裁的图
draw(circle((2cm,2cm), 2cm), red+1bp);   // 在未剪裁的图上画出剪裁圆形线
clip(mypic, circle((2cm,2cm), 2cm));     // 用圆形剪裁
add(mypic, (5cm,0));                     // 增加剪裁过的图
```

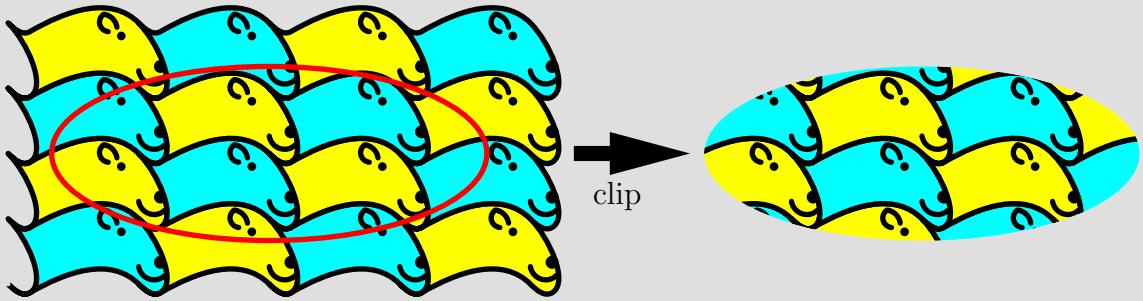
[5]为方便文字标签的阅读，在 Asymptote 中文字默认只会旋转，不会倾斜。如果要求在图变换时，标签也随之变换产生特殊效果，可以修改 Label 的 embed 参数，参见 [3]。



剪裁命令的效果正是 André 需要的，为了用椭圆形剪裁图形，André 只需要对当前图上的铺砌图进行剪裁。增加一句

```
clip(ellipse((6,3), 5, 2));
```

就得到下图右侧的剪裁效果：



最后，André 得到了完整画出图 2.1 的全部代码：

```
// André 的铺砌图案

// 全局图形缩放
size(15cm, 0);
```

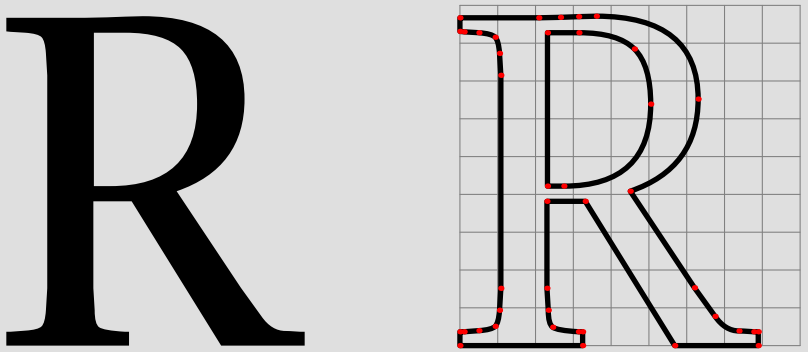
```
// 定义羊头曲线
guide headleft = (0,1.5){SE} .. tension 1.4 .. (0.5,-0.2){dir(-150)} ..
     {NW}(0,0);
guide headbottom = (0,0){SE} .. {dir(30)}(0.5,-0.2) ..{SE}(3,0);
guide head = headleft & headbottom &
    shift(3,0)*reverse(headleft) & shift(0,1.5)*reverse(headbottom) & cycle;
guide eye = circle((2.6,1.2), 0.1);
guide ear = (2.3,1.3) .. (2.1,1.5) .. (2.2,1.7)
    & (2.2,1.7) .. (2.4,1.6) .. (2.5,1.4);
guide muzzle = circle((3.5,0.5), 0.15);
guide mouth = (3,0.4) .. (3.4,0.1) .. (3.6,0.2);

// 绘制铺砌图形
for (int x = 0; x < 8; ++x) {
    for (int y = 0; y < 8; ++y) {
        transform pos = shift(3x, 1.5y);
        pen color = (i+j)%2==0 ? cyan : yellow;
        filldraw(pos * head, color, linewidth(2bp));
        fill(pos * (eye ^^ muzzle) );
        draw(pos * (ear ^^ mouth), linewidth(2bp));
    }
}

// 剪裁为椭圆形
clip(ellipse((12,6), 10, 5));
```

2.6 习题和评注

1. 试使用 Asymptote 设计下面字母的轮廓曲线：



图中的网格和结点取法仅供参考。虽然设计花体字母或许更有挑战性，但这类自由曲线并不适合数学描述，使用可视化的软件效果更好。

2. 诚然，Asymptote 并不适合那些没有明确数学描述的曲线，尤其是在并不要求完全精确的场合，但幸运的是，Asymptote 的一个附属工具 Xasy 可以弥补这个缺陷。 Xasy 是一个简单的图形界面绘图工具（图 2.2），它可以用鼠标控制画图，并保存为 Asymptote 的代码或 EPS 等图形文件格式，也能读取写好的代码并显示图形效果。Xasy 本身的绘图功能不强，但用来辅助生成一些自由曲线，是很方便的。

   请运行并熟悉 Xasy 的用法，查看用它生成的代码，并尝试与自己写的代码结合起来。

   （注：Xasy 会随 Asymptote 安装在系统中。但在 Windows 下运行 Xasy 需要自己下载安装 Python 2.6 系统及 Python Imaging Library，请参考 [3] 中的说明。）

3. 试画出下面的图形（提示：可以把叶子在一个自定义图中画好，在循环中变换后用 add 加入当前图）：

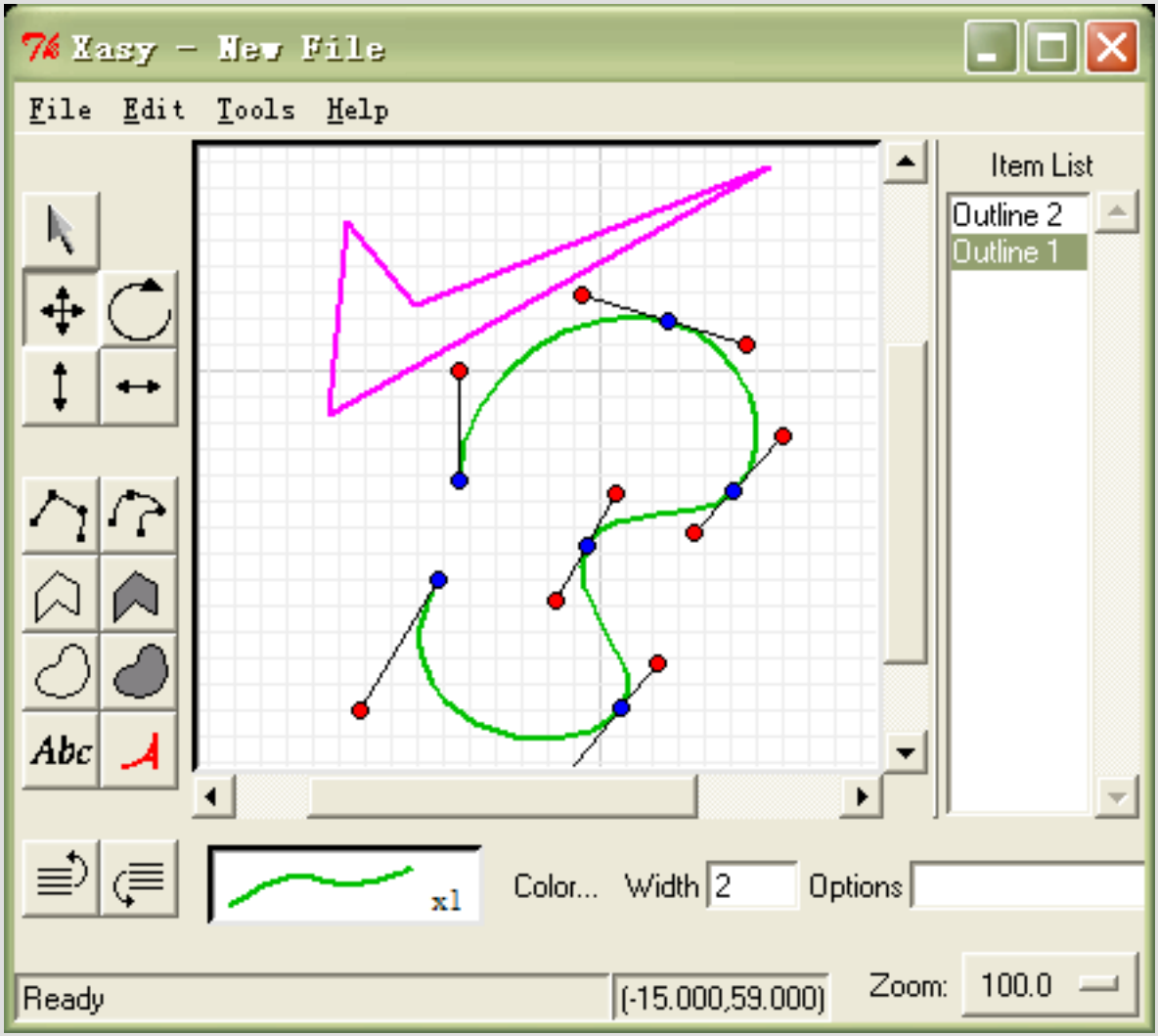
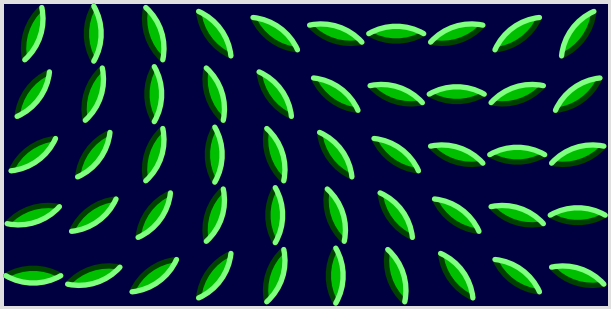
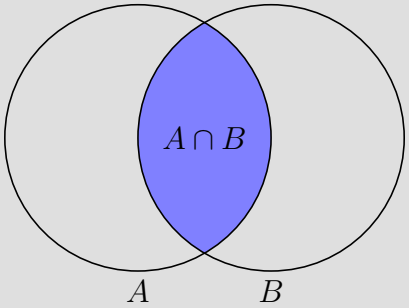


图 2.2: 微软 Windows 系统下运行的 Xasy 界面示例

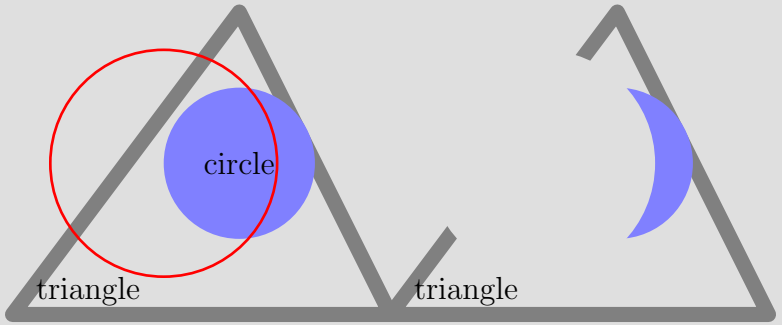


4. 利用图形剪裁，绘制两个集合交集 $A \cap B$ 的 Venn 图：

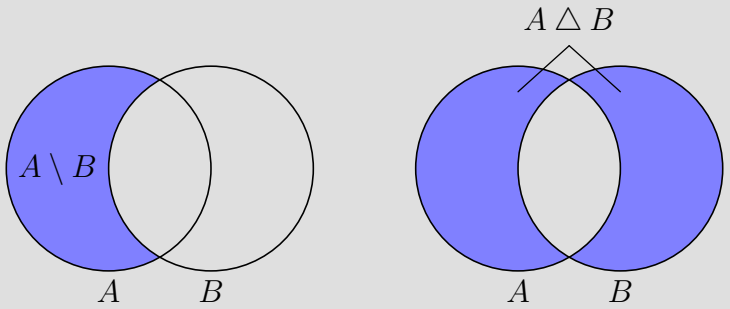


5. 与 `clip` 命令作用类似，还有一个 `unfill` 命令也可以用于图形的剪裁。它们的区别是，`clip` 命令留下剪裁曲线的内部，而 `unfill` 命令挖掉剪裁曲线的内部而留下外部。`unfill` 命令的语法与 `clip` 相同。借用 § 2.5 中的 `mypic` 举例：

```
add(mypic);
draw(circle((2cm,2cm), 1.5cm), red+1bp);
unfill(mypic, circle((2cm,2cm), 1.5cm));
add(mypic, (5cm,0));
```



下面，请利用 `unfill` 命令绘制两个集合差集 $A \setminus B$ 和对称差集 $A \triangle B$ 的 Venn 图[6]：



6. 请制做下面的艺术字效果：

[6]Venn 图的做法并不是唯一的。例如，对称差集 $A \triangle B$ 的 Venn 图也可以利用 § 2.3 绘制圆环的方式制做；在 [3] 中，还介绍了利用表示特殊填充规则的画笔 `evenodd` 绘制这种 Venn 图的方法；路径的 `buildcycle` 函数（参看 [3]）也可以用于绘制对称差集——这些方式对画一般的 Venn 图并不通用，但在一些情况下比使用剪裁功能要简洁，或能得到更丰富的效果。读者在实际使用中不必拘泥。

本章的素材源自 PSTricks 的文档 [12] 中的一个例子。 PSTricks 是 TEX 的一个老牌绘图宏包，功能强大，尤其是与 LATEX 文档有很好的结合性。在 [12] 中还可以找到其他一些有趣的铺砌图形。

Asymptote 的曲线控制方式脱胎于它的前身 METAPOST，而 METAPOST 的语法和曲线控制方式则与其前身 METAFONT 完全相同。METAFONT 是 Knuth 为配合 TEX 系统制做的字体设计软件，在 METAFONT 的经典文献 [9] 中详细介绍了这种曲线的控制。METAPOST 是 John D. Hobby 模仿 METAFONT 编写的绘图软件，它与 Asymptote 采用基本相同的方式使用 TEX 排版文字，METAPOST 的文档 [5] 也对曲线控制有一些说明。而介绍与 ConTEXt 配合使用 METAPOST 的著名文档 *METAFUN*[2] 更是以极其详实的例子展示了曲线绘制的各个方面。除个别连接符不同，Asymptote 的曲线控制语法与 METAFONT 及 METAPOST 基本相同，本章关于曲线控制的内容也主要参考了这些文档 [9, 5, 2]。

与在 METAFONT 及 METAPOST 不同的是，在 Asymptote 中，曲线（包括直线）被分为两种不同的数据类型：**path**（译为路径）和 **guide**（暂译为路向）。在 METAFONT 和 METAPOST 中的路径（**path** 类型）实际对应于 Asymptote 中的 **guide**。在计算机中曲线的数学表示是由一列结点和控制点决定的三次 Bézier 样条曲线，这种确定的曲线在 Asymptote 中对应于 **path** 类型；而由 `--`、`..` 等连接符说明的曲线，仍可能在延长时改变形状，在以一定算法转换为三次 Bézier 样条之前，就是 Asymptote 中的 **guide** 类型。本章的内容集中在使用更直观的曲线连接符生成 **guide** 类型的曲线，而完全用数学方式操控 **path** 类型，将在 § 3.4 中进一步介绍。

变量定义、循环和判断是使用 Asymptote 编程的基础。Asymptote 的程序设计语法基本来自 C++ 语言，而略有扩充和改动。本教程有意淡化编程语法，也不假定读者有很好的编程基础，但熟悉 C/C++/Java 语言的读者学习 Asymptote 会方便得多。如果对 Asymptote 语言有疑问，请参考 Asymptote 的官方手册 [3] 或中文的节译。

图类型（**picture**）也源自 METAPOST，它是分组绘图和图形复用的基础。不过 Asymptote 的图类型功能更复杂，使用 `size` 函数自动缩放就是 Asymptote 的特色功能。

剪裁功能可以产生许多特殊效果，本章正文只是最简单的应用，习题中展示了剪裁功能的一些更实际的应用，但仍然有一些问题没有涉及，留待读者自行思考了。

第三章　令狐庸的星空图

令狐庸是大学概率统计专业的学生，因为酷爱金庸的武侠小说，便给自己起了个“令狐庸”的网名。这日他读到《天龙八部》的一段：

> 阿朱道：“本来我不知道，看到阿紫肩头刺的字才知。她还有一个金锁片，跟我那个金锁片，也是一样的，上面也铸着十二个字。她的字是：‘湖边竹，盈盈绿，报平安，多喜乐。’我锁片上的字是：‘天上星，亮晶晶，永灿烂，长安宁。’我……我从前不知道是什么意思，只道是好口采，却原来嵌着我妈妈的名字。我妈妈便是那女子阮……阮星竹。这对锁片，是我爹爹送给我妈妈的，她生了我姊妹俩，给我们一个人一个，带在颈里。”

便打算把“天上星，亮晶晶，永灿烂，长安宁”这几句画成一幅插图。最近他在学习 Asymptote，尽管他清楚这种工作应该使用 Inkscape 之类的图形界面软件来做，但本来艺术修养不高的他还是决定直接用 Asymptote 编程画图。

令狐庸心目中的插图就像图 3.1 一样。

在他设想的图中，主体是深蓝色的夜空，空中有一弯月亮，有无数星星，还有一颗巨大的彗星。尽管艺术感并不强，但这幅图看起来相当复杂而且杂乱无章，对于初学 Asymptote 不久的令狐庸来说，也算是一种挑战了。

3.1　在循环中构造路向

星空图中最基本的元素就是星星。令狐庸的第一件事就是画出星星。简单的星星就是一个小圆点，但真正让令狐庸头疼的是四角星、五角星之类的形状。更进一步，令狐庸想搞明白的是，如何画出一般的星形。

图 3.1: 令狐庸理想中的星空图

令狐庸的数学不错，一个自然的想法是，把星形的每个角上的点坐标计算出来，依次连接得到整个星形的路径。而 n 角星形共有 $2n$ 个点，其中 n 个是外面的角点， n 个是内凹的角点，外面的 n 个点与里面的 n 个点分别都在以星形中心的两个圆上均匀分布，如图 3.2 所示。

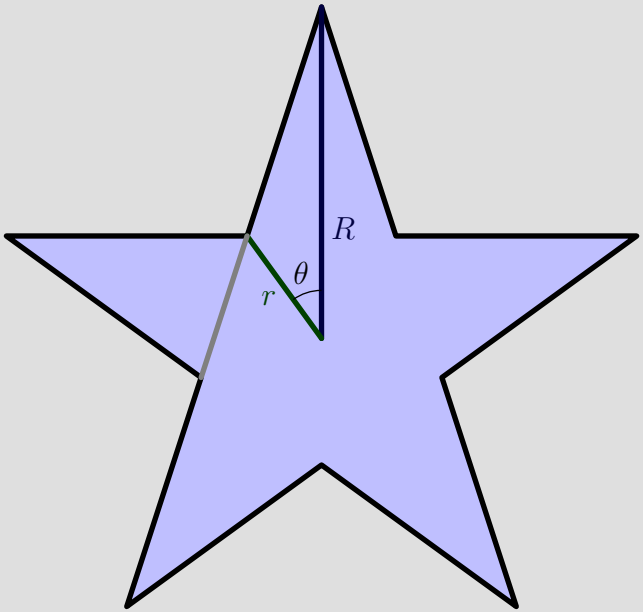


图 3.2: $n = 5$ 时的星形

其中，从中心点到 n 角星形的相邻角点（一凸一凹）的夹角容易得到：

$$\theta = \frac{360^\circ}{2n} = \frac{180^\circ}{n}.$$

设大圆半径是 R，则只要求出小圆半径 r，则整个 n 角星形的各点坐标就很容易得到了。这里，星形的形状要求是相隔一个角的两角的边共线（见图 3.2 中的灰色线）。

下面，就是考验令狐庸几何计算能力的时候了。他在图上添加了几条辅助线，问题立即变得清晰起来：

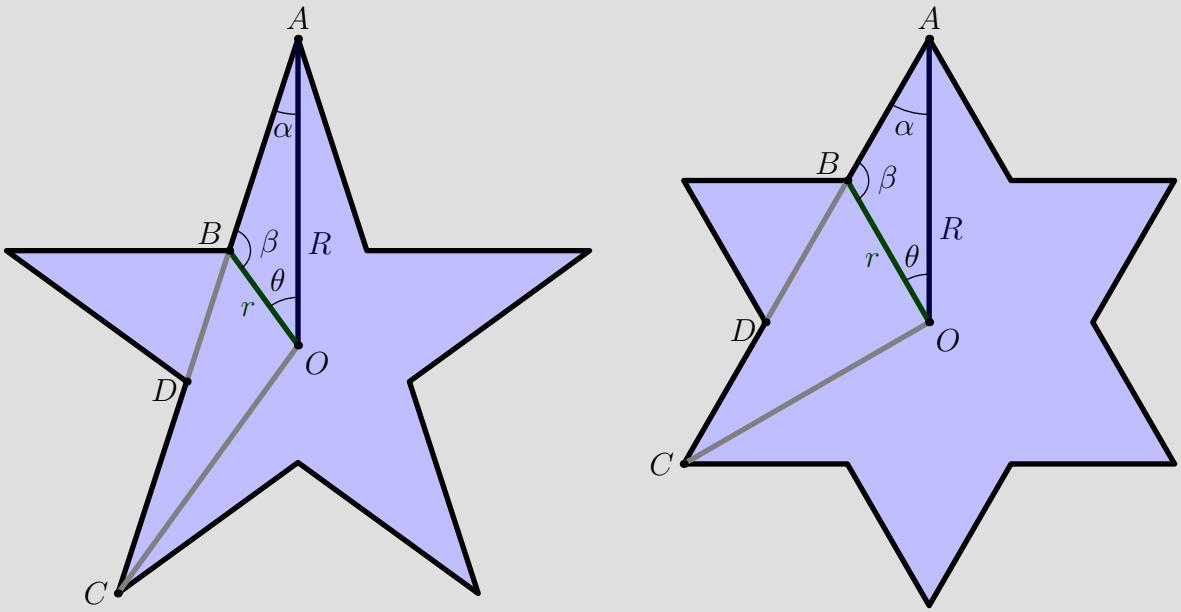


图 3.3: $n = 5, 6$ 时的星形分析

如图 3.3 所示，星形满足 A, B, D, C 四点共线的条件。在等腰三角形 AOC 中，容易看出 $\angle AOC = 4\theta$，从而

$$\alpha = (180^\circ - \angle AOC)/2 = 90^\circ - 2\theta.$$

于是在三角形 AOB 中即有

$$\beta = 180^\circ - \theta - \alpha = 90^\circ + \theta.$$

于是，利用正弦定理

$$\frac{r}{\sin\alpha} = \frac{R}{\sin\beta}$$

即得

$$\begin{aligned} r &= \frac{\sin\alpha}{\sin\beta}R \\ &= \frac{\sin(90^\circ - 2\theta)}{\sin(90^\circ + \theta)}R \\ &= \frac{\cos 2\theta}{\cos\theta}R. \end{aligned}$$

现在令狐庸已经成功地算出了 n, θ, R, r 的关系式，于是 n 角星上的任一个点都容易算出。不妨设 $R = 1$，利用 Asymptote 的 `dir` 函数，则容易在 Asymptote 中计算出 $n = 5$ 时 A, B 两点的坐标：

```
int n = 5;
real theta = 180 / n;
real r = Cos(2theta) / Cos(theta);
pair a = dir(90);
pair b = r * dir(90+theta);
```

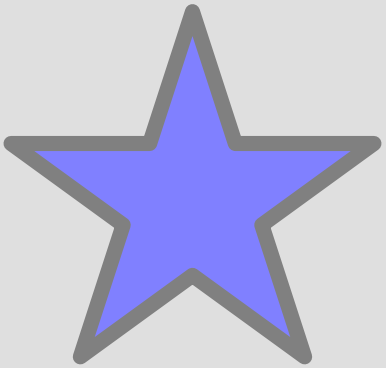
而五角星上的其他点就可以直接由 A, B 两点坐标绕原点旋转 2θ 的整数倍得到，再注意到 A, B 都是由 `dir` 函数的倍数定义的，所以这个旋转变换可以直接写进 `dir` 函数的参数，即 `dir(90+i*2theta), r*dir(90+theta+i*2theta)`（`i` 取 $0, 1, 2, \cdots, n-1$）。

现在，令狐庸已经得到所有的点坐标，剩下的问题就是把这些点连接为星形的路向。n 角星形的路向是由 n 组相同的点连接得到的，很自然地，就可以使用循环语句来逐步地构造路向：

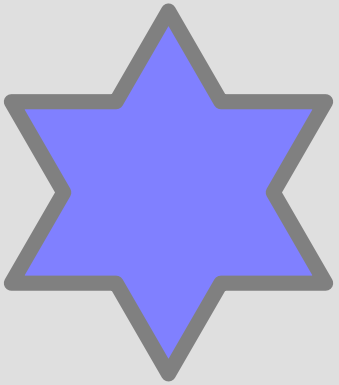
```
int n = 5;          // 五角星
real theta = 180 / n;
real r = Cos(2theta) / Cos(theta);
guide unitstar;     // 外圈半径为 1 的星形路向，初值为空
// 在循环中连接所有的点
for (int i = 0; i < n; ++i)
    unitstar = unitstar -- dir(90+i*2theta) -- r*dir(90+theta+i*2theta);
```

```
// 连接星形路向的起末点，构成闭路向
unitstar = unitstar -- cycle;

// 测试
size(5cm);
filldraw(unitstar, lightblue, gray+2mm);
```



哈，一个五角星就这样画成了。而且只要把上面代码中的 n 改为 6，就可以得到六角星了：



星形令狐庸已经可以画好，可整幅插画看起来却还是遥遥无期。如何把不同类型的星形统一起来，如何把星形均匀而又随机地放在图中，尤其是那个看起来十分复杂的彗星，现在成为令狐庸案头的新的难题。

3.2 自定义函数

在 § 3.1 中已经得到了五角星、六角星以至 n 角星的画法。不过，令狐庸发现，如果需要同时使用各种不同的星形，就不得不重复书写构造星形路向的代码——但这些代码除了 n 的取值外是完全一样的。这就需要使用 Asymptote 的函数定义，把构造星形路向的过程进行抽象。令狐庸这样定义他的函数：

```
// 返回半径为 1 的星形路向
guide star(int n)
{
    real theta = 180 / n;
    real r = Cos(2theta) / Cos(theta);
    guide unitstar;
    for (int i = 0; i < n; ++i)
        unitstar = unitstar -- dir(90+i*2theta) -- r*dir(90+theta+i*2theta);
    unitstar = unitstar -- cycle;
    return unitstar;
}
```

于是要画一个五角星，只需要再用

```
size(5cm);
draw(star(5));
```

调用这个函数即可。而同时画五角星和六角星也可以方便地做到：

```
size(5cm);
draw(star(5));
draw(shift(2) * star(6));
```

函数的定义分成两个部分：函数原型和函数体。在上面星形路向的函数中，函数原型部分是

```
guide star(int n)
```

它表示一个名字是 star 的函数，接受一个 int 类型名为 n 的整数参数，返回值的类型是 guide（路向）。而后面括在花括号中的部分则是函数体，表示整个函数进行的操作。在函数体中可以使用在函数原型中声明的所有参变量，就如同这些变量是在函数体中定义的一样。在函数体的最后一句通常是一个 return 语句，它将退出整个函数的运行，并把 return 语句后面的表达式作为这个函数的返回值。一般地，函数的定义形如：

```
返回类型 函数名(参数类型 1 参数 1，参数类型 2 参数 2，……)
{
    各种语句
    return 返回值;
}
```

在函数体的内部是一个独立的环境（叫做作用域），函数的参数和函数体内新定义的变量都只在函数体内部这个作用域有效，而不会影响作用域之外的内容。这就使得代码可以分成函数一块一块互不影响地分别设计，而不必担心在这里的变量 n 会不会影响别处的另一个变量 n。

函数定义的这些部分并不都是必需的。函数可以没有参数，此时函数只相当于给一段语句的集合起一个名字。没有参数的函数较少使用，通常这种函数都会使用在函数体外定义的变量来产生不同的行为，例如后面 § 3.3 将提到的伪随机数函数 rand()。更常见的情况是没有返回值的函数，此时函数体就不需要有 return 语句（或者使用一个没有值的 return 语句表示退出函数），但仍然需要声明函数的返回类型为 void——即空类型。和数学中的函数不同，Asymptote 中一个函数有用不仅因为它可以计算出一个返回值，还在于它能进行许多操作，比如画图。所以令狐庸写了这样一个画星形的函数：

```
void drawstar(int n, pen drawpen)
{
    draw(star(n), drawpen);
}
```

这样，drawstar(7, red) 就可以画出一个红色的七角星。

像在 C++ 中一样，在定义函数时，函数原型中函数参数还可以有一个默认值，这样调用函数的时候就可以省略这个参数。例如，令狐庸的画星形函数可以改为

```
void drawstar(int n = 5, pen drawpen = currentpen)  // currentpen 是当前画笔
{
    draw(star(n), drawpen);
}
```

于是，如果没有改变当前画笔，只要用 drawstar() 就可以画出一个黑色的五角星轮廓；而用 drawstar(red) 则画出红色五角星；用 drawstar(6) 画出黑色六角星[1]；用 drawstar(7, red) 画出红色七角星。

调用函数时，除了直接写出参数值，还有一种在 § 1.2 中就使用过的一种方式，即用 键 = 值 的方式书写参数，这样即使与函数原型中参数声明的顺序不同，也可以正确调用，如用 drawstar(drawpen = red, n = 7) 画出红色七角星——这也是 C++ 中不具备的一种语法。

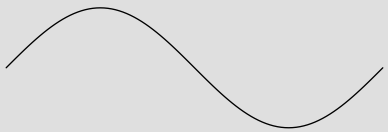
事实上，与数值、画笔等类型一样，函数也是一种数据类型。函数的类型名就是函数原型去掉函数名字的部分，如正弦函数 real sin(real x) 的类型就是 real (real x)（在不混淆时或可简写为 real (real)）。定义一个函数就是定义这个类型的一个变量。因此，一个函数也可以作为另一个函数的参数，例如在 Asymptote 绘制函数图形的基本模块 graph 中，就有函数（这里省略了一些有默认值的参数）：

```
guide graph(real f(real), real a, real b);
```

因而可以用

```
import graph;
size(5cm);
draw(graph(sin, 0, 2pi));
```

画出正弦函数的图像来：



[1] C++ 的参数可以有缺省值，但允许从最后一个参数开始从后向前省略，不能这样跳跃缺省参数值。这是因为 C++ 是按位置解析参数的，但 Asymptote 则还会考虑参数的类型。注意这个例子中 drawpen(red, 7) 的形式是不成立的。

因为函数也是一种数据类型，所以一个函数并不必须有名字。使用 **new** 表达式可以构造匿名函数。如计算 $f(x) = x^2$ 的函数：

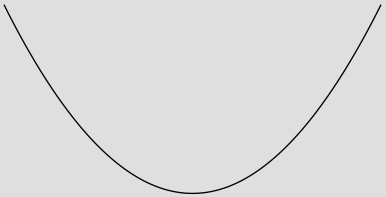
```
new real (real x)  { return x^2; }
```

匿名函数可以直接求值或赋给一个函数变量名，但这都没有什么实际意义：

```
(new real (real x)  { return x^2; })(5);    // 直接求值得 25
real f(real x) = new real (real x)  { return x^2; };    // 赋值
```

匿名函数实际的作用，是在调用另一函数时作为参数传递，或作为另一函数的返回值。例如，要画出 $f(x) = x^2$ 的函数图像，就可以用：

```
import graph;
size(5cm);
draw(graph(new real(real x) {return x^2;}, -1, 1));
```



唔，现在回到令狐庸的星星。数学的分析方法很有效，五角星甚至六角星的画法都解决了，可令狐庸现在还需要四角星：

但四角星根本没有任何角的边共线，内圆的半径也就不可能用前面的几何分析得到。此时，就必须单独给四角星设定半径。另外，令狐庸还想要四角星并不是“正”的（一个角向正上方向），也需要设定一个初始的角度。因此，经过反复修改，令狐庸最后写出了这样的函数：

```
guide star(int n = 5, real r = 0, real angle = 90)
{
    guide unitstar;
    if (n < 2) return nullpath;
    real theta = 180/n;
    if (r == 0) {
        if (n < 5)
            r = 1/4;
        else
            r = Cos(2theta) / Cos(theta);
    }
    for (int k = 0; k < n; ++k)
        unitstar = unitstar -- dir(angle+2k*theta) -- r * dir(angle+(2k+1)*theta);
    unitstar = unitstar -- cycle;
    return unitstar;
}
```

它对任何 $n \geq 3$ 都能得出星形路向，而且所有的参数都给出了一个合理的默认值。而当 $n < 2$ 时，函数返回空路向 `nullpath`（这也是 **guide** 类型的默认初始化值）。上面的四角星就是用

```
size(3cm);
filldraw(star(4), lightblue, linewidth(2));
```

画出来的。

3.3 随机数和数组

令狐庸设计的插画，最重要的部分就是分布在夜空中的星星。星星的位置是随机的，这就需要使用 Asymptote 中的随机数。

Asymptote 中主要由两个函数生成均匀分布的随机数：

```
int rand();         // 返回 [0, randMax] 之间的随机整数
int unitrand();     // 返回 [0, 1] 之间的随机实数
```

rand() 函数自源于 C 语言，它的返回值比较特别，是 0 到一个大整数 randMax 之间的一个随机整数，一般都会经过变换后使用[2]。通常需要的是特定区间的随机整数或实数，都由 unitrand() 函数变换得到。任意区间的随机实数，可以对 unitrand() 的返回值作线性变换得到：(b-a)*unitrand()+a 就是 a 到 b 之间的随机数。注意到对随机整数取余数的结果 rand() % n 正是 0 到 $n-1$ 之间的随机整数（$n>0$），任意区间的随机整数能类似地变换得到。为此概率统计专业出身的令狐庸写了下面的函数：

```
// 随机实数
real random(real a, real b=0)
{
    return (b-a) * unitrand() + a;
}
// 随机整数
int randomInteger(int a, int b=0)
{
    if (a > b) {
        int t = a;
        a = b;
        b = t;
    }
```

[2]常数 randMax 对应于 C 语言的 RAND_MAX 宏，其取值不确定，与具体实现环境有关。如官方在 Windows 平台发布的 1.88 版本中，randMax 值为 $2^{31}-1$，即 32 位带符号整数的最大值。unitrand() 实际就是由 rand()/randMax 得到的。

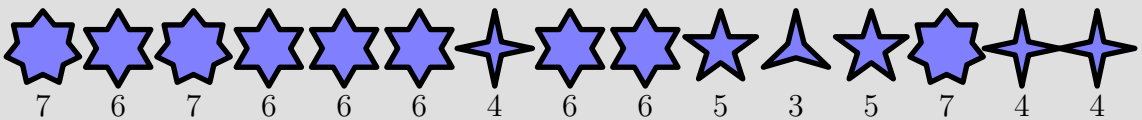
```
    return a + rand() % (b-a+1);
}
```

令狐庸给他的生成随机整数的函数起了一个很长的名字 randomInteger，一个更简洁的名字（如原来的 rand）用起来会方便得多。事实上，确实可以把前面的 randomInteger 函数还起名为 rand，因为两个函数的参数个数、类型不同，所以 Asymptote 可以在定义和使用时都清楚地区分二者，而对令狐庸来说，这只是相当于给 rand 函数增加了一个功能。这种用一个名字给不同参数的函数命名的定义方式叫做函数的重载。于是 randomInteger 可以重定义为

```
// 随机整数
int rand(int a, int b=0)
{
    // 原 randomInteger 的定义……
}
```

生成随机数的函数是比较特别的，每次调用时的参数是相同的，而返回值则是一个随机数列。下面这段代码显示随机数的使用：

```
size(15cm);
for (int i = 0; i < 15; ++i) {
    int r = rand(3, 7);
    filldraw(shift(2i,0)*star(r), lightblue, linewidth(2));
    label(string(r), (2i,-1), align=S);
}
```



其中 **string** 函数可以把各种数值类型转换为字符串类型。

使用随机数时，令狐庸发现每次使用上面的代码都得到同样的结果，而并不是如想像中的那样是随机的。进一步他还发现，每当在 Asymptote 中第一次调用 rand() 函数时，都返回 0，之后反复调用的序列在每次运行 Asymptote 时也都是相同的。这提示令狐庸一点：Asymptote 并不是真的产生随机数序列，而只是以 0 为初始值，不断地用某个函数迭代产生一个很像随机数的序列——即伪随机数序列。为了让这个伪随机数序列更随机一些，令狐庸就需要自己给这个函数迭代过程设定一个真正随机的数作为初始值——这是可以做到的。给伪随机数序列赋初值（这个初值通常被称为种子［seed］）的函数是：

```
void srand(int seed);
```

而由于令狐庸本人使用 Asymptote 编译代码的时间是随机选取的，因而可以把系统的当前时间设定为伪随机序列的种子。系统时间可以用 seconds() 函数读入，它返回一个整数，表示从 1970 年 1 月 1 日零点整到现在的秒数。于是，只要在使用随机数前加上

```
srand(seconds());
```

就可以保证生成的图形是真正随机的（相反，当需要调试使用伪随机数的程序时，使用 srand(0); 语句则可以使程序变成确定的）。

令狐庸需要随机地画一些星形，它们可能有随机的形状（角的多少）、随机的大小、随机的位置以及随机的颜色。这些都可以通过生成随机数实现。令狐庸的想法是：为了决定星形的形状，他首先生成一个 $[0,1]$ 间的随机数，根据随机数的取值区间，分段地决定星形形状；星形大小就是一个区间内的随机实数；而星形的位置可以用两个随机实数构成的坐标来表示；最后，通过生成一个 $[0,k-1]$ 之间的随机整数，令狐庸可以确定出 k 种颜色中的一种。他很快写出了下面的代码框架：

```
srand(seconds());
for (int i = 0; i < 100; ++i) {                          // 100 颗星
    real shape = unitrand();                             // 控制形状
    pair pos = (random(xmin, xmax), random(ymin, ymax));// 位置
    real size = random(sizemin, sizemax);                // 大小
    pen color = 第 rand(0,k-1) 种颜色;
    if (shape < 1/3)
        画四角星;
    else if (shape < 2/3)
```

```
        画五角星;
    else
        画点;
}
```

只要补全所有的代码，所有的星星就画出来了。

不过，令狐庸在这里又遇到了一个问题：如何选择 k 种颜色中的一种？因为已经有了随机整数的生成，问题就变成：如何表示 k 种颜色？如何表示其中第 i 种颜色？这正好是数组的应用范围。

数组是 Asymptote 中的一类数组类型，任何一个类型都对应有这个类型的数组类型，表示这种类型的序列。数组类型用类型名后面加上方括号表示，如定义一个画笔数组类型的变量：

```
pen[] pen_arr;
```

数组变量在定义时可以用花括号括起一列数据来对变量赋初值（即初始化）。例如：

```
// 红橙黄绿青蓝紫
pen[] pen_arr = {red, orange, yellow, green, cyan, blue, purple};
```

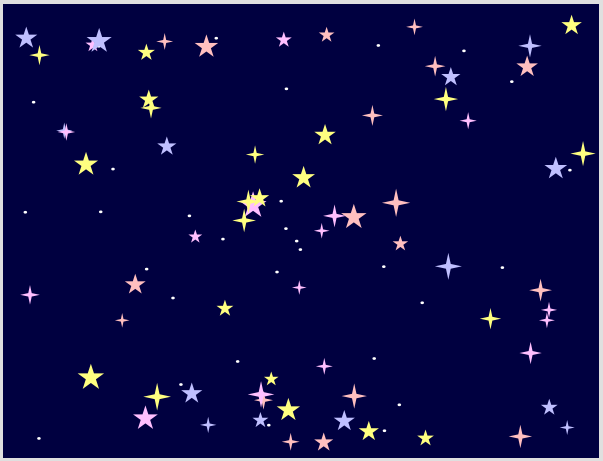
使用数组变量时，则可以在变量名后用方括号括起的下标来表示序列中的一个元素。下标从 0 开始，到元素个数减 1。如对上面定义的 pen_arr 数组，pen_arr[0] 就是红色 red，而赋值 pen_arr[6] = pink 则把数组末尾的紫色 purple 改为粉色。数组元素的个数可以在变量后加 .length 来得到，因而 pen_arr.length 就是 7。

除了 length 之外，数组还有许多有用的成员和相关的函数，可以对数组元素插入、删除、查找、筛选、排序、数值计算等等，这些内容可以参看 [3]。而对令狐庸的要求来说，简单地定义、使用数组就已经足够了。

随机颜色就是用随机整数作为下标来访问数组得到的。由于数组的下标是从 0 开始，到数组长度减 1，因此其实并不需要再定义任意区间的随机整数函数，只要简单地对 rand() 的结果取余数就行了。这样，令狐庸就画出了他纷繁的星空：

```
srand(seconds());   // 放在程序所有随机函数出现之前
size(0,6cm);        // 较小的画面
real xmin=0, ymin=0, xmax=40, ymax=30, sizemin=0.5, sizemax=1;
pen[] colors = {palered, paleblue, lightyellow, pink};
fill(box((xmin-1,ymin-1), (xmax+1,ymax+1)), darkblue);  // 背景比星星位置的边界大一些
```

```
for (int i = 0; i < 100; ++i) {
    real shape = unitrand();
    pair pos = (random(xmin, xmax), random(ymin, ymax));
    real size = random(sizemin, sizemax);
    pen color = colors[rand() % colors.length];
    if (shape < 1/3)
        fill(shift(pos)*scale(size)*star(4), color);    // 四角星
    else if (shape < 2/3)
        fill(shift(pos)*scale(size)*star(5), color);    // 五角星
    else
        draw(pos, white+1bp);   // 1bp 大小的白点
}
```



3.4 从路向到路径

令狐庸下面的难点是那个看起来特别复杂的彗星。彗星的头只是一个圆形，但使用了放射形状的渐变色填充：

```
size(5cm,0);
fill(box((-7,-2), (3,2)), darkblue);     // 背景
radialshade(circle((0,0), 1),            // 彗星头
            yellow, (0,0), 0.2,          // 彗核
            darkblue, (0,0), 1);         // 彗头边缘
```

放射渐变填充函数的原型是：

```
void radialshade(picture pic=currentpicture, path g, bool stroke=false,
                 pen pena, pair a, real ra,
                 pen penb, pair b, real rb);
```

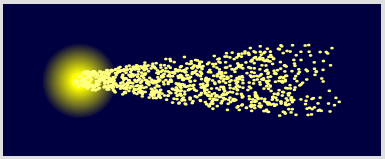
这个函数在图 pic 上对路径 g 内部的区域作放射渐变填充，从以 a 为圆心 ra 为半径的圆上的颜色 pena，到以 b 为圆心 rb 为半径的圆上的颜色 penb 进行放射梯度光滑渐变。而如果 stroke 为真，表示填充路径 g 本身（一定粗细的线）而不是路径内部的区域。类似还有一些其他类型的渐变填充，可参看 [3]。

真正让令狐庸难办的是由许多点分布而成的彗尾。彗尾的点是随机分布的，但又不是完全均匀分布，而是限定在一定区域内，在不同的区域具有不同的密度。然而 Asymptote 只给出了均匀分布的随机数，怎么能变成所需要的随机分布形式呢？

概率统计出身的令狐庸知道，要用数学公式给出从这样一种分布的变换并不容易。然而对于画出这个图形来说，也并不需要什么精确的数学描述，令狐庸立即想到了一种近似的办法：在边长为 $2r$ 的小正方形区域内随机取点，移动这个方形区域的同时不断改变 r 的大小，就可以得到不均匀的、大量随机点扫过的轨迹。让小正方形从彗核沿彗尾移动，半径不断增大，就得到这样的效果：

```
size(5cm,0);
fill(box((-2,-2), (8,2)), darkblue);
radialshade(circle((0,0), 1),
            yellow, (0,0), 0.2,
            darkblue, (0,0), 1);

for (int i = 0; i < 1000; ++i) {          // 1000 个点
    real x = 6*i/1000;                    // 方形中心从 0 到 6
    real r = 0.2 + 0.8*i/1000;            // 方形半径从 0.2 到 1
    draw((x,0)+(random(-r,r), random(-r,r)), lightyellow+1bp);  // 画点
}
```



这个方法很巧妙，然而只能对简单的直线画出彗尾。对于彗尾是曲线的情况就不成了。但这个想法给了令狐庸一个启示：如果把上面表示 (x,0) 换成任意曲线上点的坐标，就可以把这个方法推广到任意曲线上了。例如，圆形的参数曲线是

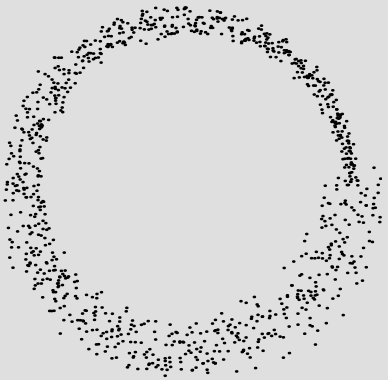
$$\begin{cases} x(t) = r\cos(2\pi t), \\ y(t) = r\sin(2\pi t), \end{cases} \qquad t \in [0,1).$$

那么，就可以用

```
size(5cm,0);
for (real t = 0; t < 1; t += 1/1000) {       // 1000 个点
    pair c = 5(cos(2pi*t), sin(2pi*t));      // 半径为 5
    real r = 0.2 + 0.8t;
```

```
    draw(c+(random(-r,r), random(-r,r)), black+1bp);
}
```

得到环形彗尾：



可是，在 Asymptote 中，并不是用参数方程构造出曲线，而是使用 `--`、`..` 等曲线连接符直观地得到曲线的路向。能不能把曲线看成是像上面圆形一样的参数方程呢？令狐庸为了搞清楚这个问题，又仔细研读了 Asymptote 的手册 [3]，发现确实是可以的。

是的，在 Asymptote 的内部，确实把曲线表示为参数方程。如果曲线是由 n 个结点连接而成的，那么参数曲线的时间参数 t 的取值范围就是 $[0, n-1]$（如果是封闭曲线，取值范围是 $[0, n]$）。既然 Asymptote 的曲线都是参数曲线，那么前面画彗尾的方法也是可以使用的。

令狐庸很早就懂得使用各种曲线连接符把坐标点连接为曲线。这种曲线的描述，在 Asymptote 中就对应于 **guide** 类型，我们生造了一个名词，称它为路向。也就是说，在 Asymptote 中的一个 **guide** 类型的变量，就保存了描述这个曲线时使用的各个结点，以及各点处的切线方向、张力大小、卷曲值等等这些名目繁多的东西。路向保存了构造它用到的全部信息，这些信息往往也都是具有鲜明的直观几何意义的。

然而当曲线要被画出来的时候，路向的描述就显得过于模糊不清了：确定了曲线经过的点和在这些点上的切线方向，怎么就能唯一决定一条曲线？张力为 2 到底算有多大？ **{curl 0.5}** 和 **{curl 1}** 有多大区别？……因此，Asymptote 内部还有一套算法，把这些相对模糊而直观的信息，转换成标准的参数方程的数学表示，这就是 **path** 类型 ，我们称之为路径。路向一经转换为路径，就成为可以精确确定每一个点的参数曲线了。如果把路向比做用钉

子连接固定的橡皮绳的话，那么转换得到的路径就是依照橡皮绳伸展的形状做成的铁丝，铁丝一经定形，就不再变化。在 Asymptote 中从路向到路径的转换是自动进行的，所有绘图的命令（如 draw）实际接受的是 **path** 类型的参数，但可以自动把收到的 **guide** 类型参数转换为 **path** 类型。

正因为路向在转化为路径时形状会固定下来，所以在循环中使用曲线连接符逐步连接曲线时，就必须使用路向即 **guide** 类型，而不能用 **path** 类型，否则可能因为 **path** 的刚性而产生意想不到的结果。试看下面的例子：

```
size(5cm);
pair[] arr = {(0,0), (1,1), (2,0), (3,1), (4,0)};
guide g;
path p;
for (int i = 0; i < arr.length; ++i) {
    g = g .. tension 2 .. arr[i];
    p = p .. tension 2 .. arr[i];
}
draw("guide", g, heavyblue+1mm);
draw("path", shift(5,0)*p, heavygreen+1mm);
```

一般说来，除非曲线已经完全确定，否则在 Asymptote 中总使用 **guide** 类型作图是比较合适的。

数学上，路径是用三阶 Bézier 样条曲线描述而成的，原来路向的结点就是路径样条曲线的采样点，而路向的其他描述则会经过一系列复杂算法的计算（这些算法尽力确保路向的描述符合直观），转换为样条曲线每一段的另外两个控制点。样条曲线的每段是三阶 Bézier 参数曲线，它是关于时间参数 t 的三次多项式，系数就由端点 z_0, z_1 和两个控制点 c_0, c_1 决定：

$$r(t) = z_0(1-t)^3 + 3c_0(1-t)^2t + 3c_1(1-t)t^2 + z_1t^3, \qquad t \in [0,1], \quad z_0, z_1, c_0, c_1 \in \mathbb{C}.$$

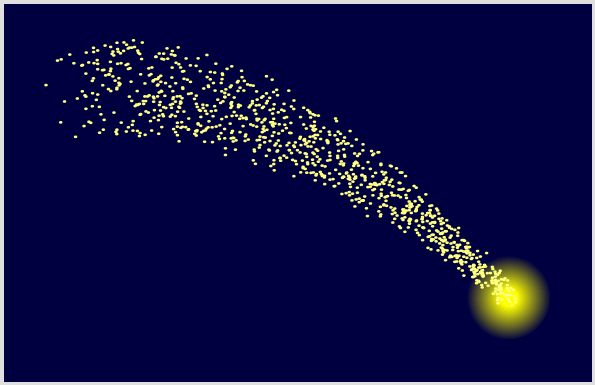
每段 Bézier 曲线的时间参数都从 0 开始，而在样条曲线中，第 k 段曲线的时间参数则从 k 开始。这样，整条曲线就完全精确地由一组坐标点确定了。**path** 类型的变量，在 Asymptote 内部就只需要保存这一系列坐标点。

不过，大多数情况下，并不需要从底层的数学公式来控制路径。Asymptote 已经提供了许多函数来获得有关路径的信息，常用的一些如（更多路径函数见手册 [3]）：

```
int length(path p);          // 路径 p 的 Bézier 曲线段数
int size(path p);            // 路径 p 的结点数（闭路径与 length(p) 相同）
pair point(path p, int t);   // 时刻（参数）t 处的点
pair point(path p, real t);
pair dir(path p, int t);     // 时刻 t 处的切向量
pair dir(path p, real t);
path reverse(path p);        // 反向路径
path subpath(path p, int a, int b);     // 时刻 a、b 之间的子路径
path subpath(path p, real a, real b);
pair[] intersectionpoints(path p, path q);  // p 与 q 的交点数组
pair intersectionpoint(path p, path q);     // p 与 q 的第一个交点
```

令狐庸的要求其实很简单：由时间参数 t 得到路径上的点。这只需要用 **point** 函数。因此他可以这样画出一颗彗星：

```
size(0,5cm);
fill(box((-12,-2), (2,7)), darkblue);
radialshade(circle((0,0), 1),
            yellow, (0,0), 0.2,
            darkblue, (0,0), 1);
path tail = (0,0){NW} .. {W}(-10,5);
for (real t = 0; t < 1; t += 1/1000) {
    real r = 0.2 + t;
    draw(point(tail,t) + (random(-r,r), random(-r,r)), lightyellow+1bp);
}
```

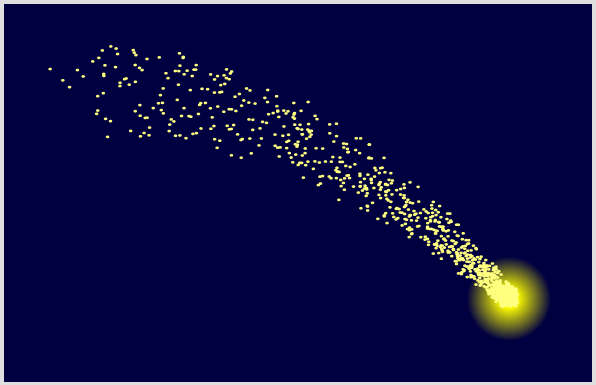


不过，令狐庸还是对这个彗星不满意。彗星的彗尾，应该是靠近彗核的地方稠密些，远离彗核的地方稀疏些，而不是现在比较均匀的样子。这就要求，放置随机点的小矩形在靠近彗核的地方运动得慢一些，而在远离彗核的地方运动得快一些。这看上去似乎很麻烦，但令狐庸很快又找到了一个技巧：由于参数 $t \in [0,1]$，则对任意的 n 也有 $t^n \in [0,1]$，所以参数曲线 $r(t) = (x(t), y(t))$ 其实和参数曲线 $\tilde{r}(t) = r(t^n) = (x(t^n), y(t^n))$ 是图像是相同的，但第二个参数曲线的时间运行速度是不均匀的，当 $n > 1$ 时前慢后快。因此，令狐庸对他的彗星又作了少许修改：

```
real random(real a, real b=0)
{
    return (b-a) * unitrand() + a;
}
size(0,5cm);
fill(box((-12,-2), (2,7)), darkblue);
radialshade(circle((0,0), 1),
            yellow, (0,0), 0.2,
            darkblue, (0,0), 1);

path tail = (0,0){NW} .. {W}(-10,5);
for (real t = 0; t < 1; t += 1/1000) {  // 循环体内 t 变为 t^3
    real r = 0.2 + t^3;
```

```
    draw(point(tail,t^3) + (random(-r,r), random(-r,r)), lightyellow+1bp);
}
```

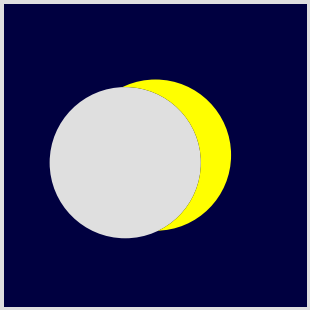


经过以上的努力，令狐庸的插图就几乎完成了，剩下的两个比较简单的部分：月亮和几行文字。
利用 § 2.6 习题 4 中画 Venn 图的方法，不难利用 `unfill` 挖洞画出月亮：

```
size(3cm);
fill(circle((0,0), 1), yellow);
unfill(circle((-0.4,-0.1), 1));
```

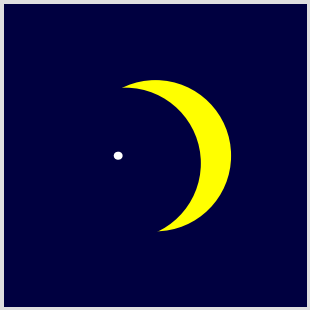

可一旦在画月亮之前再画背景，这样画月亮时用 `unfill` 会把背景的夜空也挖掉，这就很有问题了：

```
size(4cm);
fill(box((-2,-2), (2,2)), darkblue);
fill(circle((0,0), 1), yellow);
unfill(circle((-0.4,-0.1), 1));
```



另一种办法是不使用 unfill 这种剪切办法画月亮，而是直接定义月亮的路径；或者先把月亮画到一个 picture 类型的图上，然后再加在当前图 currentpicture 上：

```
size(4cm);
fill(box((-2,-2), (2,2)), darkblue);
dot((-0.5,0), white);   // 一颗在月亮后面的星
picture moon;
fill(moon, circle((0,0), 1), yellow);
unfill(moon, circle((-0.4,-0.1), 1));
add(moon);
```



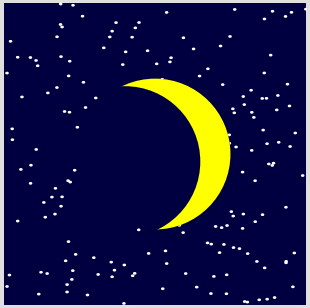
可这样月牙儿内部就可能会冒出星星——然而令狐庸清楚月亮是圆的，阴影部分同样应该遮蔽后面的星星。因此这样做也是有问题的。

一个相对合理的解决办法是：不是把月亮单独画在一个图中，而是把月亮和星星同时画在这个图中，最后再加进当前图。于是，进行 unfill 剪裁时就会同时挖掉月亮和星星，但不会影响背景的夜空。不过，这还要求 unfill 剪裁只挖去月亮的阴影部分，而不能挖去不属于月亮阴影的部分，所以用与月亮圆形错位的另一个圆形进行 unfill 操作是不合适的（尽管相差并不大）。但月亮阴影部分的曲线并不容易找出来，所以这种办法也很难做好。

最后令狐庸采用的办法是，在画出月亮的同时也同时画出月亮后面的阴影（与夜空颜色相同），把它单独画在一个图中，最后再加进当前图。结果的代码与最初的想法相比有些复杂：

```
size(4cm);
fill(box((-2,-2), (2,2)), darkblue);
for (int i = 0; i < 200; ++i)   // 200 颗星
    dot((random(-2,2),random(-2,2)), white+1bp);
// 单独画月亮
picture moon_nobg, moon;
fill(moon_nobg, unitcircle, yellow);
unfill(moon_nobg, shift(-0.4,-0.1)*unitcircle);
fill(moon, unitcircle, darkblue);
add(moon, moon_nobg);
// 把月亮加入当前图
```

```
add(moon);
```



最后再加上文字。由于文字和星星都是浅色，同样要考虑文字被星星影响的问题。这可以给 label 命令或标签构造函数 **Label** 加上 Fill(标签背景色) 参数解决。为了显示比较平滑，令狐庸使用半透明的背景色：

```
size(4cm);
fill(box((-2,-2), (2,2)), darkblue);
for (int i = 0; i < 100; ++i)   // 100 颗星
    dot((random(-2,2),random(-2,2)), white);
label("Stars", (0,0), lightyellow+fontsize(2cm), Fill(darkblue+opacity(0.7)));
```

最后，令狐庸把所有这些内容集合在一起，就得到了完整的星空图（图 3.1）代码：

```
// # -*- coding:utf-8 -*-
srand(seconds());
// 随机实数
real random(real a, real b=0)
{
    return (b-a) * unitrand() + a;
}
// 半径为 1 的 n 角星形路向
guide star(int n = 5, real r = 0, real angle = 90)
{
    guide unitstar;
    if (n < 2) return nullpath;
    real theta = 180/n;
    if (r == 0) {
        if (n < 5)
            r = 1/4;
        else
            r = Cos(2theta) / Cos(theta);
    }
    for (int k = 0; k < n; ++k)
        unitstar = unitstar -- dir(angle+2k*theta) -- r * dir(angle+(2k+1)*theta);
    unitstar = unitstar -- cycle;
    return unitstar;
}
// 全局定义
size(15cm);
real xmin=0, ymin=0, xmax=40, ymax=30, sizemin=0.3, sizemax=0.8;
```

```
pen[] colors = {palered, paleblue, lightyellow, pink};
// 夜空
fill(box((xmin-1,ymin-1), (xmax+1,ymax+1)), darkblue);
// 星星
for (int i = 0; i < 100; ++i) {
    real shape = unitrand();
    pair pos = (random(xmin, xmax), random(ymin, ymax));
    real size = random(sizemin, sizemax);
    pen color = colors[rand() % colors.length];
    if (shape < 1/3)
        fill(shift(pos)*scale(size)*star(4), color);
    else if (shape < 2/3)
        fill(shift(pos)*scale(size)*star(5), color);
    else
        draw(pos, white+1bp);
}
// 彗星
picture comet;
radialshade(comet, unitcircle,
            yellow, (0,0), 0.2,
            darkblue, (0,0), 1);
path tail = (0,0){NW} .. {W}(-20,10);
for (real t = 0; t < 1; t += 1/1000) {
    real r = 0.2 + 2*t^3;
    draw(comet, point(tail,t^3) + (random(-r,r), random(-r,r)),
         lightyellow+1bp);
}
add(shift(30,3) * comet);
```

```
// 月亮
picture moon_nobg, moon;
fill(moon_nobg, unitcircle, lightyellow);
unfill(moon_nobg, shift(-0.5,-0.2)*unitcircle);
fill(moon, unitcircle, darkblue);
add(moon, moon_nobg);
add(shift(5, 25) * scale(2) * moon);
// 文字
settings.tex = "xelatex";
usepackage("xeCJK");
texpreamble("\setCJKmainfont{FZSongHeiTi_GB18030}"); // 方正宋黑体
string text = "\begin{minipage}{3em}
天上星\par
亮晶晶\par
永燦爛\par
長安寧
\end{minipage}";
label(text, (xmax,ymax), align=SW, yellow+fontsize(0.7cm),
      Fill(darkblue+opacity(0.5)));
```

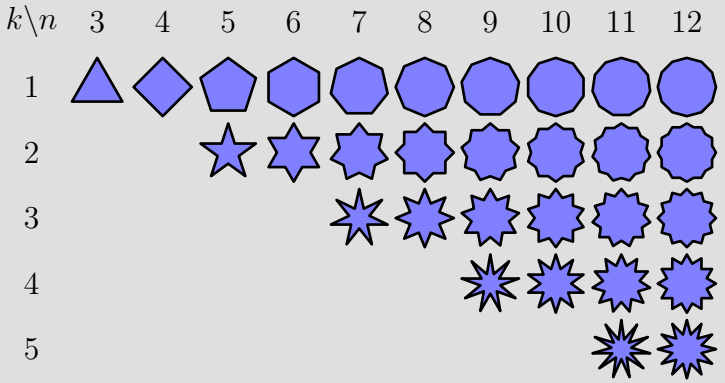
3.5 习题和评注

1. 继续研究星形的画法。

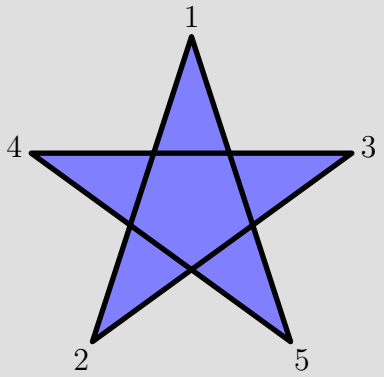
   (a) 在 § 3.1 定义的星形图，要求相距为 2 的两个角各自有一条边在一条直线上。请推广此要求，给出相距为 k 的两个角有边共线条件下，求 n 角星形路向的函数（$k \geq 1, n \geq 2k+1$）。设此函数原型为：

   ```
   guide star(int n, int k = 2);
   ```

并利用这个函数画出下面的表格（特别地，当 $k=1$ 时，函数给出正 n 边形的路向）：



(b) 星形的另一种画法不是基于星形外凸和内凹角点各自共圆均匀分布，而是直接基于前述星形角的共线关系，把角点相连得到。例如，五角星可以按下图所示的次序连接角点得到：

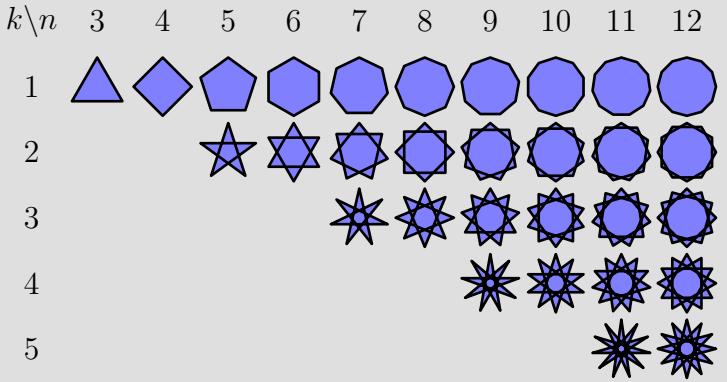


这种画法能更清晰地表示出星形的几何特征，但缺点是在星形内部也有线条，这可能是在描绘星形轮廓时所不需要的。试编写代码，画出上面的图形，并推广至任意 n 角星（n 为奇数）。

(c) （较难）推广上一小题的结果，编写函数，它返回相距为 k 的角有边共线时， n 角星的路向数组（可能由多条路向组成），其中 $k \geq 1, n \geq 2k+1$：

```
guide[] star_new(int n, int k = 2);
```

特别地，当 $k = 1$ 时，此函数也返回正 n 边形的路向。并利用这个函数画出下面的表格：



提示：对 n 与 k 互素的情况（即 n 与 k 的最大公因数 $\gcd(n,k) = 1$），画出此星形路径的方法与上一小题并无二致。但当 n 与 k 不互素时问题则较复杂，必须对这种方法进行变化。一些初等数论的知识可能会对分析此问题有所帮助。

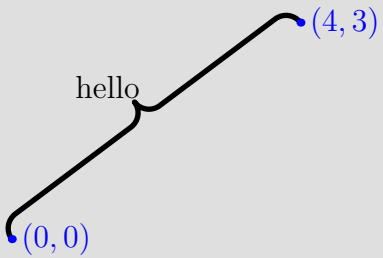
2. 实现函数 bracelabel，其函数原型为：

```
void bracelabel(picture pic=currentpicture, Label L, pair a, pair b,
                pen p=currentpen);
```

其使用效果为：

```
size(5cm);
bracelabel("hello", (0,0), (4,3), linewidth(2bp));
dot("$(0,0)$", (0,0), blue);
```

```
dot("$(4,3)$", (4,3), blue);
```



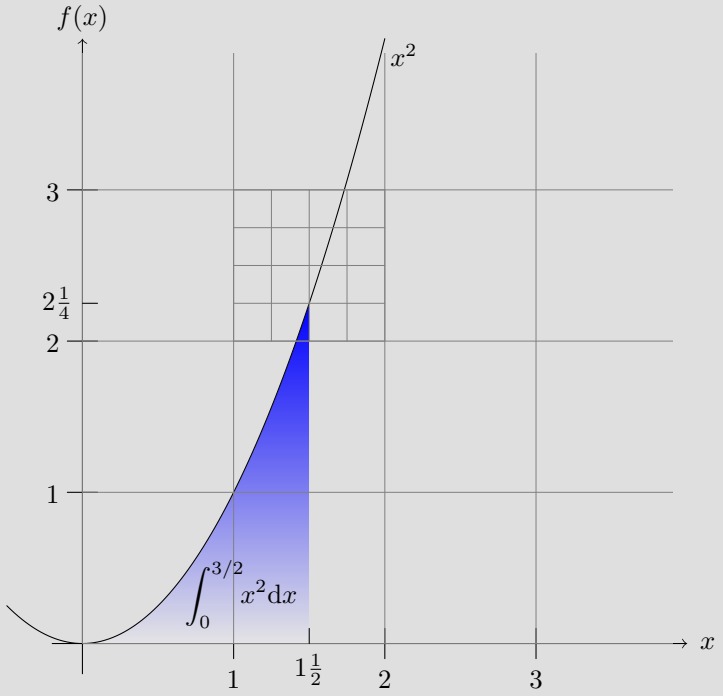
3. 研究函数图形的绘制。

(a) graph 模块提供大量函数可以用来帮助绘制二维函数图形。最基本的函数是 § 3.2 中提及的 graph 函数。请实现一个简化的 **guide** 函数版本，其函数原型为：

```
guide graph(real f(real), real a, real b);
```

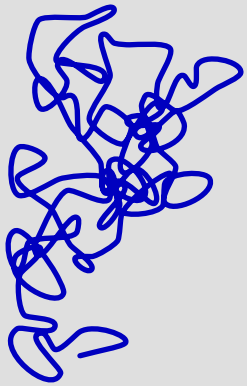
它使用多段直线连接函数 f 在区间 $[a, b]$ 的样本点。并用它画出正弦函数的图像。

(b) （较难）参考手册 [3] 中对 graph 模块 的用法和轴向渐变 函数 axialshade 的说明，画出下面的图形（尽量利用 graph 模块已经提供的功能）：



4. 随机游动模拟。

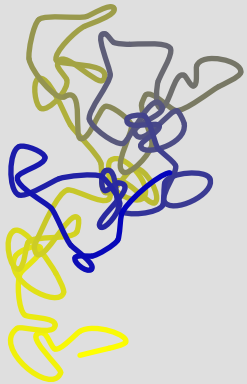
(a) 随机游动可以通过一列随机坐标 $z_1, z_2, \cdots$的轨迹模拟，其中 $z_{i+1} - z_i$ 在 $[-1,1] \times [-1,1]$ 均匀分布。试据此画出下面的随机游动模拟图：



(b) 函数

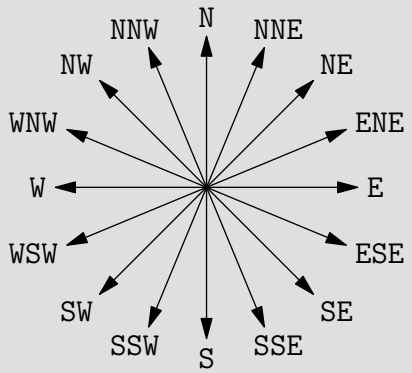
```
T interp(T a, T b, real t);
```

返回 a 与 b 的线性插值 (1-t)*a+t*b，其中 T 是数值、坐标或画笔。这个函数可以用来把两个颜色按比例混合，如 interp(white,red,0.2) 就是 20% 的红色。试利用混合颜色作出渐变效果的随机游动图：



（提示：先生成全路径，再利用 subpath 分别画出。）

5. 利用循环遍历一个字符串数组，画出 Asymptote 中 16 个罗盘方向的示意图：



6. 令狐庸在读 Roberts 教授写的一本 C 语言的教材时，发现里面使用了另外一种得到随机整数的办法（下面假定 $a < b$）：

```
int randomInteger(int a, int b)
{
    return ((double)rand() / (RAND_MAX+1)) * (b-a+1) + a;
}
```

其基本思路是：既然 rand() 返回从 0 到 RAND_MAX 的一个随机整数，那么

```
(double)rand() / (RAND_MAX+1)
```

就是一个 $[0,1)$ 区间上的随机实数，从而

```
(double)rand() / (RAND_MAX+1) * (b-a+1)
```

就是 $[0,b-a+1)$ 区间上的随机实数。当转换回整数时，对它截尾取整的结果正好就是 $[0,b-a]$ 上均匀分布的随机整数。

然而当令狐庸为此兴冲冲地写出下面的 Asymptote 代码时（与 C 语言相同，这里的 **(int)** 表示对后面的结果到整数进行强制类型转换）：

```
int randomInteger(int a, int b)
{
    return (int) (rand() / (randMax+1) * (b-a+1) + a);
}
```

Asymptote 却给出了一个整数计算溢出的错误（Integer overflow）。

请自己试一试上面的代码并分析，Asymptote 为什么会给出错误信息（也许你的 Asymptote 并不会产生错误，为什么？），而上面的 C 程序代码并不会产生程序运行错误。

假定 $a < b$，进一步分析，Roberts 教授给出的函数，在什么情况下会得到错误的结果？如何修改 Roberts 教授的 C 函数，得到正确的结果？如何把同样的方法应用于 Asymptote 代码中？

（提示：在 C 语言中整数除法的结果是整数，需要把操作数转换浮点数才能得到。C 语言中加法不检查溢出，但溢出时会因二进制补码表示得到一个负数。在上面错误的实现中，不仅有溢出的错误，对负数强制类型转换的取整方式也不对。）

本章的内容比较多。

首先是循环逐步构造曲线，同时借绘制星形的例子，说明了如何对图形进行数学分析，并把分析的结果使用 Asymptote 编写代码实现。

其次是函数利用。函数或子程序是对于各种程序设计语言都十分重要的过程抽象机制。本章极小的篇幅不可能尽举函数的众多应用，而 Asymptote 中的函数与 C/C++ 语法基本相同，熟悉任何编程的读者自然可以举一反三。需要注意的是，Asymptote 中函数是最高级别的数据元素，函数可以匿名定义（相当于 λ-表达式）、嵌套定义、随机改变函数变量的值，这与传统 C/C++ 的函数与函数指针有很大区别；而 Asymptote 函数的重载与键 -值参数调用也比 C++ 语言中更为灵活。有关 Asymptote 中函数语法的详细内容可以参看手册 [3]。

再次是随机数的使用。随机化图形在 METAPOST 中就十分普遍，而随机化算法在数值计算与现实模拟中也极为常用。Asymptote 的 stats 模块 给出了有关概率统计的一些实用函数，例如一维正态分布的随机数生成、最小二乘法和统计直方图的自动绘制，手册 [3] 对这个模块没有详细用法的介绍，读者需要自行参考 stats.asy 中的注释、函数原型和具体代码。

本章定义的 `random(real,real)` 函数和 `rand(int,int)` 函数是伪随机数比较通用的使用方式，而 `rand() % n` 是在 C/C++ 语言中最常用的生成数组随机下标的方式。通常电脑总是难以生成真正的随机数序列，而一个确定的算法也只能产生伪随机数序列。大多数电脑程序使用一种“线性模同余”的方式，从某一初始值生成一个周期很长的迭代序列作为伪随机数序列。只有在对随机性要求很高的场合（如网络密钥生成），才会使用记录键盘敲击时间、数字化设备电路噪声等硬件方式生成“真正”的随机数。有关伪随机数可参看 [8]。

数组是 Asymptote 乃至各种程序设计语言最重要的数据抽象类型之一，本章对数组的介绍是肤浅的。[3] 中有单独一节篇幅讲解数组的使用，读者可以详加参考。与 C/C++ 语言的数组相比，Asymptote 的数组是一种高级的数组结构，具有下标检查、自动增长等能力和丰富的操作，它更接近于 C++ 的 `vector` 模板类。

§ 3.3 一节除了随机数和数组外，也简单介绍了 **`while`** 和 **`do`** 循环。由于与 C/C++ 类似，手册 [3] 对于基本的控制程序流向的命令几乎没有介绍，对编程不熟悉的读者可以参考 C 或 C++ 语言的书籍，如 [7] 。

最后一部分是 Asymptote 的路径。Asymptote 中 **`guide`** 类型与 **`path`** 类型可以自动相互转换，用法也相互交织，二者的区别是初学者不易搞清的内容，本章试图把这个问题说清楚一些，并给出使用路径 **`path`** 的一个例子。路径的操作很多，§ 3.4 一节能给出的只是极简单的应用，一些高级的应用和涉及更深入数学运算的内容，将在后面一些章节示例讲解。

习题 3 选材自 [11]。`graph` 模块用来绘制各类函数图形，功能十分丰富，在官方手册 [3] 中已经有较详细的讲解和示例（但这部分手册尚没有中文翻译），故本文只做极简单的说明。

习题 6 选材自 [10]，原书给出了随机数函数使用的许多有用的技巧，并给出了 `randomInteger` 的一个正确的实现。

第四章　严教授的自动机

严宇教授是一位计算机语言专家。在他给学生讲授可计算性与自动机理论的课程时，需要制做含有大量图论图形和自动机。一个典型的例子是描述正则语言的有限自动机， 图 4.1 就是严教授手绘的一个有限自动机。

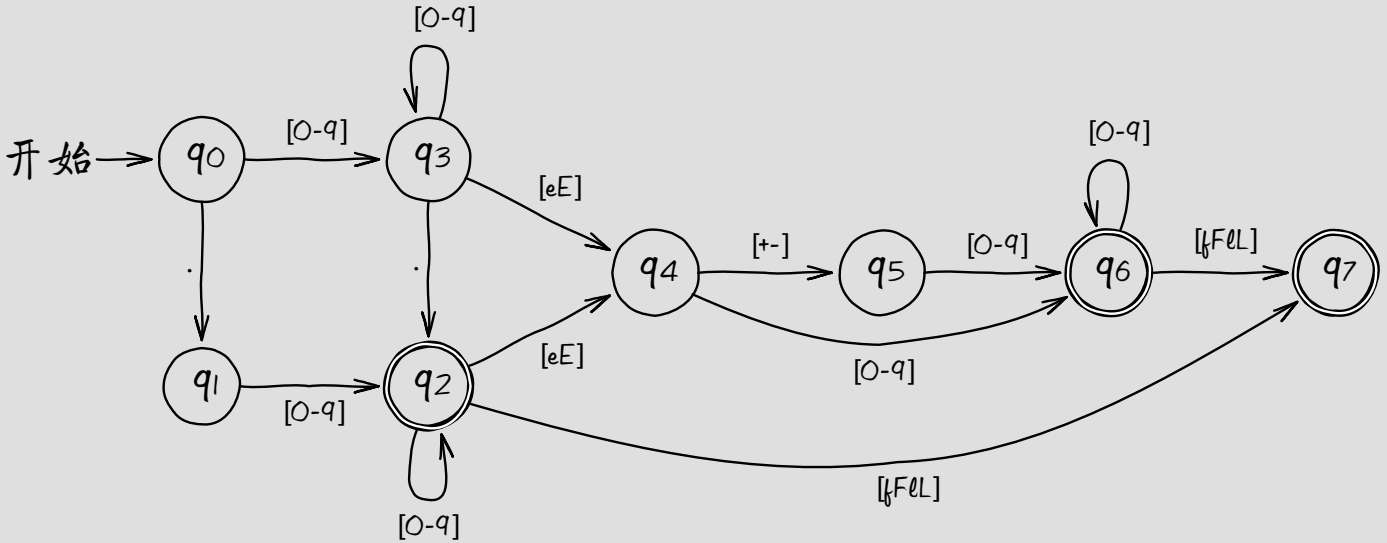


图 4.1: 严教授手画的有限自动机：它描述了 C 语言中浮点型字面值的词法。

然而，尽管图 4.1 中严教授的自动机画得很仔细，但要在他的课程幻灯片中使用，还是太粗疏了。严教授需要的是色彩鲜明、清晰准确的自动机图形，就像图 4.2 中的一样。

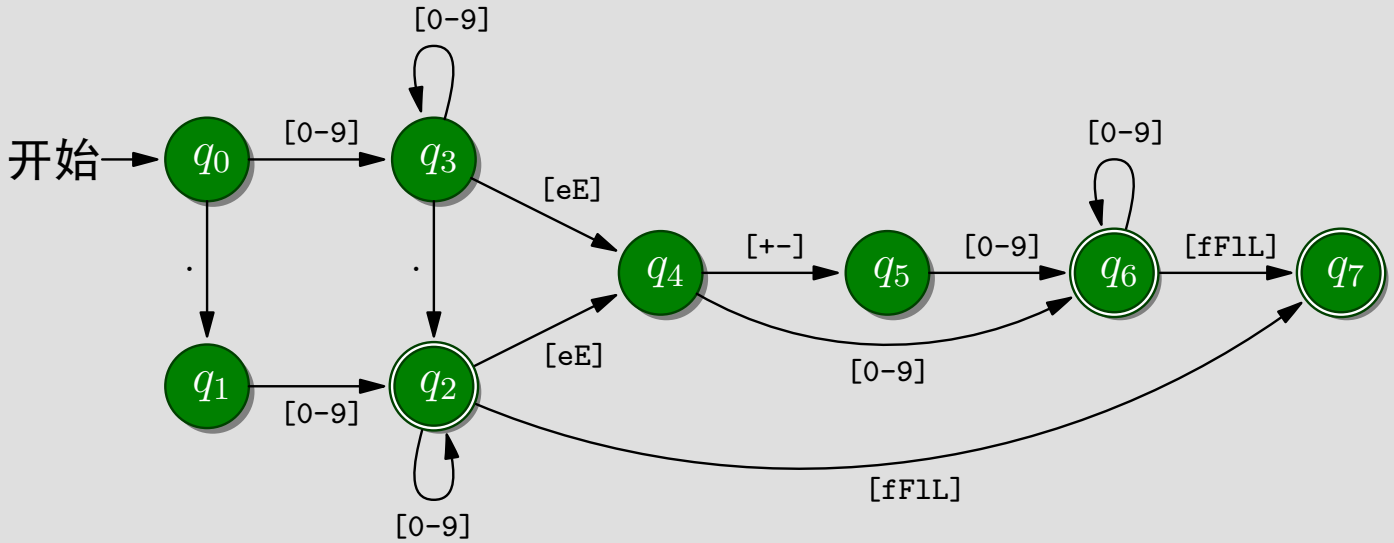


图 4.2: 严教授想要在幻灯片里使用的有限自动机

经过考虑，严教授决定用 Asymptote 语言来完成这样的绘图，因为这样可以画出最精确的图形，并能方便地对图形的所有细节进行控制，并且适合绘制大量形式相近的图形。然而要让图形和生成图形的代码都接近完美，也将是一个艰巨的任务。

4.1 标签和连线

把一些相似的小图形用线和箭头连接起来，是应用中非常广泛的一种作图类型。工作程序的流程图、电路设计图、网络结构图、计算机科学中的自动机、乃至古老的家谱，都可以看作是这类图形。这些相似的小图形往往是以

一个文字标签为中心、用某种形状的曲线围成的，再以各种线连接为一个整体——它可以抽象为数学中的“图”。这类图形可以统称为图论图形，这正是计算机科学中使用最多的一类图形，也是严宇教授工作的重点。

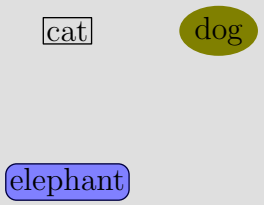
Asymptote 语言已经提供了一种给标签围上边框，再连上线条的方法。这依赖于一种特殊的数据类型：**object**（物件）。一个物件就是一个带有某种形状边框的文字标签。函数

```
object draw(picture pic=currentpicture, Label L, envelope e,
            pair position=(0,0), real xmargin=0, real ymargin=xmargin,
            pen p=currentpen, filltype filltype=NoFill);
```

将在图 pic 上的位置 **position** 画出一个以标签 L 为中心，边框形状为 e 的物件。参数 xmargin 和 ymargin 是边框与标签边沿的距离，画笔 p 用来绘制文字标签，而 **filltype** 控制边框如何绘制和填充。

Asymptote 的这个函数初看起来非常复杂，但排除默认的参数，使用起来是直接了当的：

```
object cat = draw("cat", box, (0,0), filltype=Draw),
       dog = draw("dog", ellipse, (2cm,0), filltype=Fill(olive)),
       elephant = draw("elephant", roundbox, (0,-2cm),
                       filltype=FillDraw(lightblue, darkblue));
```

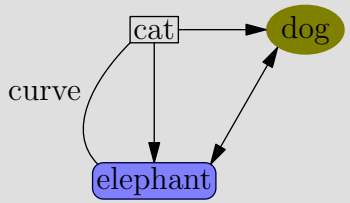


这里 box、**ellipse** 和 roundbox 分别是矩形、椭圆和圆角矩形三种可以使用的形状；Draw、Fill、FillDraw 则分别是对边框画线、填充的控制变量（参考 [3]）。调用 draw 函数时如果省略 **filltype** 参数，则这里的默认值 NoFill 效果和 Draw 一样，表示使用画笔 p 绘制边框。

下面的问题则是把这些物件用线连接起来。为此，Asymptote 提供了一种看起来颇为古怪的方式：

```
add(
    new void(picture pic, transform t) {
```

```
        draw(pic, point(cat,E,t) -- point(dog,W,t), Arrow);
        draw(pic, point(cat,S,t) -- point(elephant,N,t), Arrow);
        draw(pic, point(elephant,NE,t) -- point(dog,SW,t), Arrows);
        draw(pic, "curve", point(cat,SW,t) {SW} .. {SE} point(elephant,NW,t));
    }
);
```



在这里，实际只有一条语句 add，但 add 的参数则是一个匿名函数：

```
new void (picture pic, trasform t) {
    // 匿名函数体
}
```

意思是在 currentpicture（add 省略的第一个参数）实际进行绘图时，就使用这个匿名函数中的语句进行绘图[1]。

上面匿名函数的内容就是画出 cat、dog、elephant 之间的连线，其中用到了有关 **object** 物件类型的一个函数 **point**[2]：

```
pair point(object F, pair dir, transform t=identity());
```

这个函数输出物件 F 的边框在 dir 方向的坐标。这个坐标通过 t 作变换。注意这里的方向 dir 是经过边框形状矫正过的，因而对于矩形物件，NE 方向的坐标点就是矩形右上方向的直角点，而并非严格的 45° 角方向点。在匿名函数

[1]这种用法并没有在手册 [3] 中描述，它涉及 **picture** 图类型内部的实现方式。这里添加的匿名函数就相当于添加了一个等效的 drawer 类型的函数。参看 plain_picture.asy 中的代码。

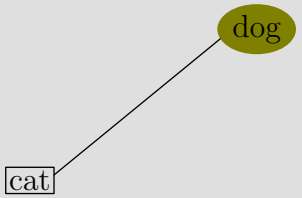
[2]函数 **point** 也没有在手册 [3] 中描述。关于此函数以及预置的物件形状 box、 roundbox 和 **ellipse** 的定义，参看 plain_boxes.asy 中的代码。

中，必须把匿名函数的参数 t 传给函数 **point**，并且绘图时在参数 pic 的图中进行，才能保证在正确的位置以正确的比例画出物件的坐标和连线。

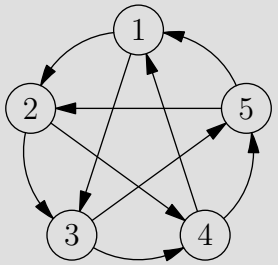
在匿名函数中，只要注意正确地使用 **point** 函数和 t、pic 参数，对于绘图的语句没有特别的限制。因此上面无论是连直线、曲线、箭头还是在连线上标注文字，都和普通的方式一样。

尽管语法多少有些难看和冗繁，但使用 Asymptote 中的物件类型和它的连线机制，就足够画出各种形式的图论图形了。严教授的自动机也可以这样逐步构造出来。然而，严教授最终却放弃了这种绘图的方式。不是因为这种方式的写法麻烦，而是因为这种方式很难表达出一些有用的功能。例如，仅指定物件而不使用 **point** 函数指定具体连线的位置来连接两个物件的能力，如在下面的代码中，我们能不能不去手工指定 cat 和 dog 的方向 ENE 和 WSW 呢？

```
object cat = draw("cat", box, (0,0), filltype=Draw),
       dog = draw("dog", ellipse, (3cm,2cm), filltype=Fill(olive));
add(
    new void(picture pic, transform t) {
        draw(point(cat,ENE,t) -- point(dog,WSW,t));
    }
);
```



确实，如果是复杂的图论图形，靠人力算出连点合适的方向来，一不留神就会出错；而如果想要通过一个循环自动化地连接五角星形呢？

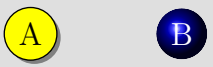


想要利用现有的 **object** 类型的连线机制画出上面的图形无疑就太麻烦了，因为定义分布在正五边形上的五个物件或进行连线并不难（参看第三章五角星的画法），但要精确地确定五个圆形上连线的位置就非常令人头疼了。而严教授更希望他能以一种更简单更自然的方式画连线，比如说：

```
draw(cat -- dog);
```

如果真的可以这样做连线的话，那么要画出前面的五角星形就容易得多了。

不仅如此，还有一些工作是超出目前的 **object** 类型能力的。例如，产生 **object** 的 draw 函数只能支持 box、**ellipse** 和 roundbox 三种形状，而使用的 **filltype** 参数也只有 FillDraw、Fill、Draw、NoFill、UnFill、RadialShade（参看 [3]）这么有限的几种。如果我们想要画出图形：



那么我们就要增加新的圆形，新的带阴影的和带球形立体效果的 **filltype**——但在仔细研究 Asymptote 本身实现 **filltype** 和 **object** 相关类型的代码之前，我们甚至不知道这种扩充能不能做到。在这种情况下，或者是通过研读 Asymptote 中 plain 等基本模块（Asymptote 在安装时就预定义好了许多模块，而其中最基本的 plain 模块则会在运行时自动导入，我们平时使用的 Asymptote 实际是建立在 plain 模块之上的）的代码，自行扩充；或者需要自己重新实现一个新类型。实际中很难说哪种方式更简单或更好用，但研究 Asymptote 中 plain 模块的实现，总是有教益的。

4.2 解析代码与扩充功能[3]

下面我们就考虑上一节最后提出的一个简单问题：增加新的圆形 envelope。为此，我们需要分析 Asymptote 中 plain 模块的部分实现，然后对它进行扩充和修改。

在 plain 模块的一个子模块 plain_boxes.asy 中，定义了有关生成 **object** 类型的 draw 函数和 box、roundbox、**ellipse** 三种 envelope 类型的形状，**point** 函数的定义也在其中。如果我们需要新定义一种圆形形状 **circle**，就可以模仿里面的代码。在研究相关的代码时，可能需要不断查阅手册 [3]。

首先在 plain_boxes.asy 的中间可以找到 evelope 类型的定义：

```
typedef path envelope(frame dest, frame src=dest, real xmargin=0,
                      real ymargin=xmargin, pen p=currentpen,
                      filltype filltype=NoFill, bool above=true);
```

这里使用了 **typedef** 命令定义类型的别名。**typedef** 命令的用法是

```
typedef 对标识符 foo 的声明;
```

表示把 foo 定义为它所声明的类型的一个别名。如

```
typedef int Integer;
```

就把 Integer 定义为整数类型 **int** 的别名，从而以后就可以用 Integer 来代替 **int**。上面就把 envelope 定义为一个函数类型的别名[4]，这个函数接受许多参数，而返回一条路径。不难猜想，这个函数返回的路径，正是将来生成 **object** 类型元素的形状。

此时再看 plain_boxes.asy 开头，就看到了 box、roundbox 和 **ellipse** 的定义，实际就是定义了三个函数。例如 **ellipse** 的定义（这里的代码来自 Asymptote 1.90 版本）：

```
1 path ellipse(frame dest, frame src=dest, real xmargin=0, real ymargin=xmargin,
2              pen p=currentpen, filltype filltype=NoFill, bool above=true)
3 {
```

[3]本节一些内容不易理解，初次阅读时可跳过。

[4]但在 Asymptote 中，函数类型的别名只能用来声明一个函数变量，不能用于定义一个有函数体的函数。

```
4  pair m=min(src);
5  pair M=max(src);
6  pair D=M-m;
7  static real factor=0.5*sqrt(2);
8  int sign=filltype == NoFill ? 1 : -1;
9  path g=ellipse(0.5*(M+m),factor*D.x+0.5*sign*max(p).x+xmargin,
10                 factor*D.y+0.5*sign*max(p).y+ymargin);
11 frame F;
12 if(above == false) {
13   filltype.fill(F,g,p);
14   prepend(dest,F);
15 } else filltype.fill(dest,g,p);
16 return g;
```

这个函数初看起来可能有点复杂。但如果只看它的返回值的话，那么函数主要的工作就是返回路径 g，也就是 9、10 两行定义的椭圆。现在来分析函数的其他部分，4、5 行定义的坐标 m 和 M 就是帧 src 的左下角和右下角点。其中帧 **frame** 是 Asymptote 中类似于图 **picture** 的一种功能更简单更基本的类型，不支持诸如自动放缩的功能；而函数 min 和 max 则接受帧、图、路径甚至画笔（表示笔尖的路径）作为参数，分别返回它们的左下角和右上角的坐标。于是，第 6 行定义的坐标 D 就是帧 src 的对角线向量。第 7 行定义了一个静态（**static**[5]）的实数因子 factor 为 $\sqrt{2}/2$。第 8 行判断填充类型 **filltype** 是否为空的 NoFill 而设置一个符号 sign。这样，9、10 两行定义的椭圆就清楚了，它的中心就是帧 src 的中心，坐标为 $(M+m)/2$；它的横轴半径是因子 factor 乘以对角线向量的横坐标 D.x，加上（或当 **filltype** 不为 NoFill 时减去）笔尖边框右上角位置的横坐标的一半[6]，再加上用户定义的距离 xmargin；纵轴的定义与长轴类似。后面的 11 至 15 行，实际就是根据 above 的具体取值，对边框的填充和画线（above 为假时在目标帧 dest 底部绘制）。

通过对比不难发现，box 与 roundbox 的实现与 **ellipse** 实际上大同小异。因此我们很容易模仿它们写出圆形

[5] 在这里表示一个常量，用静态类型保证只进行一次赋值。有关静态类型请参看手册 [3]。

[6] 不难发现其实这个实现是错误的，原来的代码试图给长轴加上笔尖宽度的一半，以保证即使画笔很粗，也可以让边框线与标签内容间有正确的距离；由于笔尖的路径相对于原点对称，这里计算笔尖边框右上角坐标的一半，其实只是笔尖宽度的 1/4。正确的实现应该是 0.5*sign*(max(p)-min(p))。在 1.91 以后版本的 Asymptote 中，这个错误已经修正。

的 envelope 函数[7]：

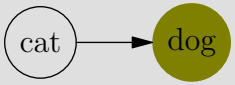
```
// circle 函数就是一个 envelope 类型的变量
path circle(frame dest, frame src=dest, real xmargin=0, real ymargin=xmargin,
            pen p=currentpen, filltype filltype=NoFill, bool above=true)
{
    pair m=min(src);
    pair M=max(src);
    pair D=M-m;
    static real factor=0.5*sqrt(2);
    int sign=filltype == NoFill ? 1 : -1;
    // 圆的半径 r 就是前面定义中椭圆的横轴与纵轴中的较大值
    real r = max(factor*D.x+sign*max(p).x+xmargin,
                 factor*D.y+sign*max(p).y+ymargin);
    path g=circle(0.5*(M+m), r);
    frame F;
    if(above == false) {
        filltype.fill(F,g,p);
        prepend(dest,F);
    } else filltype.fill(dest,g,p);
    return g;
}
```

把这段代码放在前面，马上就可以试验效果：

```
object cat = draw("cat", circle, (0,0)),
       dog = draw("dog", circle, (2cm,0), filltype=Fill(olive));
add(
    new void(picture pic, transform t) {
```

[7]修正了椭圆中计算轴长方法中的小错误。

```
        draw(point(cat,E,t) -- point(dog,W,t), Arrow);
    }
);
```



这样，通过分析并模仿 plain 模块原有的代码，就完成了一种新的 envelope 形状 **circle**，并能将其付诸使用。

但是，有关 envelope 类型的函数是比较复杂的，要想真正看懂它，像上面这样分析清楚，甚至可能需要了解一些手册 [3] 中没有提及的东西，也就是分析更多的源代码。例如上面 **filltype** 的内部结构在手册中并没有给出说明，而需要分析 plain_filldraw.asy 的代码；以画笔为参数的函数 max 也没有文档，更需要到 Asymptote 的 C++ 代码 runtime.in 和 pen.h 中才能找到定义。

上面分析 **ellipse** 函数的过程可以看做一般的分析 Asymptote 源代码的一个缩影。有时所分析的代码比较简单，可以像上面那样比较容易地看清楚代码的来龙去脉，并对它进行必要的扩充和修改；有时问题则会变得困难，需要缺少文档的情况下，在许多个源文件中搜索需要的定义（比如由 **ellipse** 的定义查找 **filltype** 的内部结构和 max 函数在 runtime.in 中的定义）。并非人人像严教授这样是这方面的专家，在很多情况下，分析源代码耗费的精力会比重写一遍相同功能的代码还要多，甚至会遇到看不懂源代码的情形。这时，就必须要考虑解决问题的其他途径了。

4.3 结构体与抽象机制

出于对功能的扩充和最终用法简洁性的要求，严教授还是决定自己来设计一个新的类型来完成他的自动机的绘制。

Asymptote 中的数据类型是十分丰富的，除了整数（**int**）、实数（**real**）、坐标（**pair**，即复数）、路向（**guide**）、路径（**path**）等基本类型，还有画笔（**pen** ）、图（**picture**）等预置的复杂类型，还有函数和数组类型。然而，更重要的是， Asymptote 还有把这些类型组合起来，建立新类型的一般方法，也就是对数据的抽象机制，这就是结构体。Asymptote 中的结构体与 C/C++ 语言的结构体在语法上十分接近，同时也有自己的特点。

一个结构体类型用下面的语法定义：

```
struct 结构体名 {
    类型 1 成员变量 1;
    类型 2 成员变量 2;
    ……
}
```

这样，就把若干个类型的变量结合为一个整体，形成一个新的类型。例如，可以把两个实数类型的变量复合为一种新的坐标类型：

```
struct newpair {
    real x, y;
}
```

于是，就可以使用这种新的类型 newpair 来声明一个复合类型的变量，并且像普通的 **pair** 类型一样，可以使用圆点运算符来分别得到它的两个分量：

```
newpair a;     // 定义 newpair 类型的变量 a
a.x = 1.0;      // 给 a 的 x 分量赋值
a.y = 2.5;      // 给 a 的 y 分量赋值
```

这样，就可以用一个变量名 a 表示相互紧密联系的两个量 a.x 和 a.y。这就是数据的抽象。

物件类型 **object** 实际上就是把一个表示边框的路径和边框中的内容结合组成的结构体[8]。严教授需要的也就是与 **object** 类型的一个类似的类型，使用相同的方法，把一个边框路径、它的内容（用图表示）以及它的位置结合为一个结构，并命名为“结点”node：

```
// 结点类型，第一个实现
struct node {
    path outline;   // 边框
    picture stuff;  // 内容
```

[8]参看 plain_Label.asy 的源代码。

```
    pair pos;       // 位置
}
```

于是，就可以写出这样的语句：

```
node cat;                                       // 定义 cat 结点
cat.outline = circle((0,0), 4mm);               // cat 的边框是一个圆
cat.pos = (0,0);                                // cat 的位置在原点
filldraw(cat.stuff, cat.outline, yellow);       // 把边框画进 cat 的内容
label(cat.stuff, "cat");                        // 把文字标进 cat 的内容
add(shift(cat.pos) * cat.stuff);                // 在指定位置画出 cat 的内容
```

这无疑是十分简陋的，还远远比不上 **object** 类型的精致。但这已经展示出了操作 node 类型的最重要的手段。下面要做的事就是，用函数把上面的一个个独立的操作包装起来。

首先是使用函数建立一个 node 类型的变量，它的内容是一个文字标签，边框是紧密围绕在标签之外的一个圆形，而位置由函数的参数指定。不仅如此，还要指定边框和标签分别怎么在 node 的内容中画出来，最简单的办法就是分别指定一些画笔：

```
// 圆形结点的构造函数，第一个实现
node Circle(Label text, pair pos, pen textpen=currentpen,
            pen fillpen=nullpen, pen drawpen=currentpen)
{
    node nd;        // 定义新结点
    nd.pos = pos;   // 设置位置
    label(nd.stuff, text, textpen);             // 在内容中标注文字
    pair M = max(nd.stuff), m = min(nd.stuff);  // 计算内容边界大小
    nd.outline = circle(0.5*(M+m), 0.5*length(M-m));    // 设置边框
```

```
    filldraw(nd.stuff, nd.outline, fillpen, drawpen);   // 绘制和填充边框
    return nd;      // 返回构造的新结点
}
```

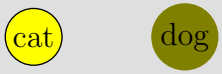
这里计算内容边界并设置圆形边框路径的办法，就是前面为 **object** 类型建立圆形 envelope 的算法的特别简化的版本：计算出左下、右上角的坐标，得到中心坐标和对角线长度，就是边框的圆心和直径。

然后是定义一个函数画出整个结点，方法和单独把结点的内容加进当前图一样：

```
// 画结点，简单实现
void draw(picture pic=currentpicture, node nd)
{
    add(pic, shift(nd.pos) * nd.stuff);
}
```

于是，利用上面的函数，就可以这样来画出结点了：

```
node cat = Circle("cat", (0,0), fillpen=yellow),
     dog = Circle("dog", (2cm,0), fillpen=olive, drawpen=nullpen);
draw(cat);
draw(dog);
```



现在来看怎么实现 **object** 类型中 **point** 函数的对应物。类似 **point**(cat,E,t) 的用法无疑是太蹩脚了，比如理想的方式是 cat.E。按前面 node 类型的定义，只要给 node 增加一组变量，分别保存边框在不同方向上的点的坐标。但更一般地，直接得到边框在某个特定方向（或角度）上的点也是非常有用的，因此最好还有一种类似 **point**.angle(30) 或 **point**.dir(ENE) 的语法，来表示边框上的特定方向的点。对此，也只要在结构体 node 中增加几个函数类型作为成员。因此，可以这样扩充 node 类型：

```
// 结点类型，第二个实现
```

```
struct node {
    path outline;
    picture stuff;
    pair pos;
    pair E, N, W, S;
    pair dir(pair v)
    {
        path g = shift(pos) * outline;
        pair M = max(g), m = min(g), c = 0.5*(M+m);
        path ray = c -- c + length(M-m)*unit(v);
        return intersectionpoint(g, ray);
    }
    pair angle(real ang)
    {
        return this.dir(dir(ang));
    }
}
```

这里需要解释的函数是新加进来的 dir 函数。就像上面做的，在结构体中可以直接定义函数，即这个结构体的成员函数。在 Asymptote 中，成员函数也是结构体的普通成员变量，结构体里面的函数定义只是给了这个变量一个初始值。在 dir 函数中，它接受一个表示方向的坐标变量 v，返回从结点边框的中心发出的在 v 方向的射线与边框路径的交点，也就是我们要的边框在 v 方向上的点。dir 函数的计算过程是简单而清晰的：首先通过平移得到正确的边框路径 g，而后计算边框的右上角坐标 M、左下角坐标 m、中心坐标 c，而后计算射线 ray，最后返回交点坐标。其中用到的 length 函数计算向量的长度（也就是对应复数的绝对值），unit 函数计算与向量方向相同的单位向量，而 intersectionpoint 正是 § 3.4 中的求两路径交点的函数。在成员函数 dir 中，可以直接使用结构体里面定义的变量，而不必再通过参数传递，这正是结构体成员函数的方便之处。

另一个成员函数 angle 只是简单地调用刚刚定义的 dir 函数，完成类似的功能。不过这里的语句有点令人疑惑：前面对 dir 的定义实际是重载了 Asymptote 本来就有一个函数 dir，因而 angle 要使用两个 dir 函数，一个是成员函数 dir(pair) 计算交点，另一个是系统的 dir(real) 计算一个角度方向上的向量。正像前面 § 3.3 中所

说的，重载的两个函数参数类型不同，所以可以直接用 dir(dir(ang))，并不会产生歧义，但为了避免读代码的人混淆，这里用了 **this**.dir 来表示结构体的成员函数。**this** 是一个特殊的关键字，可以用来代替这个结构体类型的实例本身（如已经定义了 node 类型的变量 a，那么调用 a.angle(0) 时，a.angle 函数中的 **this** 就代表 a 本身，从而 **this**.dir 就是 a.dir）。

此时，结点类型 node 增加了 E、N、W、S 几个坐标用来方便地访问边框上的点，这些坐标也必须在用函数构造结点类型时就进行初始化。这就需要在 Circle 函数中增加几个语句，调用成员函数 dir 函数完成这个工作：

```
// 圆形结点的构造函数，第二个实现
node Circle(Label text, pair pos, pen textpen=currentpen,
            pen fillpen=nullpen, pen drawpen=currentpen)
{
    node nd;
    nd.pos = pos;
    label(nd.stuff, text, textpen);
    pair M = max(nd.stuff), m = min(nd.stuff);
    nd.outline = circle(0.5*(M+m), 0.5*length(M-m));
    filldraw(nd.stuff, nd.outline, fillpen, drawpen);
    nd.E = nd.dir(E);   // 调用 nd.dir 函数对四个方向初始化
    nd.N = nd.dir(N);
    nd.W = nd.dir(W);
    nd.S = nd.dir(S);
    return nd;
}
```

对结点类型的 draw 函数不需要做改变。因此，下面就不仅可以画出结点，而且可以像使用原来的 **object** 类型一样，画出连线了：

```
node cat = Circle("cat", (0,0), fillpen=yellow),
     dog = Circle("dog", (2cm,0), fillpen=olive, drawpen=nullpen);
draw(cat);
draw(dog);
```

```
draw(cat.E -- dog.W, Arrow);
// 等价地，也可以用
// draw(cat.dir(E) -- cat.dir(W), Arrow);
// 或者
// draw(cat.angle(0) -- cat.angle(180), Arrow);
```

现在，利用结构体进行数据抽象，定义函数对抽象数据类型进行操作，我们就已经完成了对原来 Asymptote 中预定义的 **object** 类型一个简单的模拟。在这个过程中，我们失去了一些功能：新的 node 类型不能使用 size 函数正确地进行放缩，因为它并不像 **object** 类型在绘制时还小心翼翼地处理一个变换 t（见 § 4.1）；也没有像原来的 **object** 类型那样在计算边框线粗细等细节上锱铢必较（见 § 4.2）。但也得到了许多好处，那就是使用时的简洁、清晰和容易修改扩充。而要完成这些，只要二三十行的代码。对比代码

```
draw(cat.E -- dog.W);
```

和

```
add(
    new void(picture pic, transform t) {
        draw(point(cat,E,t) -- point(cat,W,t));
    }
;)
```

其实不难做出抉择。

4.4 运算符和记法

然而，严教授并不满足于让 node 重复 **object** 类型的功能，他需要的是更简单、更有效的写法:

```
draw(cat -- dog);
```

尽管看上去差别不大，但从 cat.E 到 cat 的转变是本质性的，这个区别比从 **point**(cat,E,t) 到 cat.E 的区别还要大。因为无论是 **point**(cat,E,t) 还是 cat.E，都是一个 **pair** 类型的量，用它来连线是顺理成章的；但 cat 则是 node 类型的变量，两个 node 类型的变量用 -- 连起来是什么东西？它们又怎么能连起来呢？

这就引发了一个新的问题，就是发明一种把两个 node 类型连接起来的新的记法。而这个语法是本来在 Asymptote 中是没有的。要完成这一点，就需要使用运算符的重载。

在 Asymptote 中，像 +、-、*、/ 以及 --、.. 这些运算符，都可以看成是一个以 **operator** 开头的函数[9]。如整数的加法运算符 + 就等价于函数

```
int operator+(int op1, int op2);
```

它接受两个整数参数，返回它们的和。因而 1+2 就等价于 **operator**+(1,2)，都得到结果 3。

既然运算符就是函数，那么就可以自己定义；即使已经有了定义，按照 § 3.3 中所说，也可以自己重载。因而，完全可以通过重载，定义一种不寻常的整数加法：

```
// 保存旧的加法定义
int oldadd(int op1, int op2) = operator+;
// 定义新的加法为模 5 意义下的加法
int operator+(int op1, int op2)
{
    return oldadd(op1, op2) % 5;
}
```

也就是把加号 + 定义为 $\mathbb{Z}_5$ 上的加法运算了。那么 1+2 的结果仍然是 3，但 2+3 的结果就成了 0，2+4 则得到 1。

这样，利用自定义运算符 --，严教授就顺理成章地完成了两个结点之间的连线定义：

```
path operator--(node nd1, node nd2)
{
    path g1 = shift(nd1.pos) * nd1.outline;
```

[9]在 Asymptote 中，绝大部分运算符都可以看做是函数，从而可以重载，甚至包括一些在基本语言中没有的运算符，如 @。但也有一些例外，包括赋值、圆点、中括号（下标）、圆括号（函数调用）和条件运算符。

```
    path g2 = shift(nd2.pos) * nd2.outline;
    pair c1 = (max(g1)+min(g1)) / 2;
    pair c2 = (max(g2)+min(g2)) / 2;
    path edge = c1 -- c2;
    edge = firstcut(edge, g1).after;
    edge = lastcut(edge, g2).before;
    return edge;
}
```

这里，**operator**-- 函数接受两个结点参数。它首先计算两个结点的边框路径 g1 和 g2，然后算出两个边框的中心 c1 和 c2，然后定义结点的连线 edge 就是 c1 -- c2，最后利用 firstcut 和 lastcut 函数截取 edge 在 g1 交点后面、在 g2 交点前面的中间一段，就是正确的连线。

在这里，函数 firstcut 和 lastcut 分别是函数 cut 的在 n 为 0 和 -1 时的特例：

```
slice cut(path p, path knife, int n);
```

cut 函数返回一个结构体 **slice**：

```
struct slice {
    path before,after;
}
```

它计算路径 p 与 knife 的第 n 个（负数表示倒数）交点，把 p 分成两半：其中 before 是把路径 p 用“刀” knife 切开的前半截，而 after 是后半截。

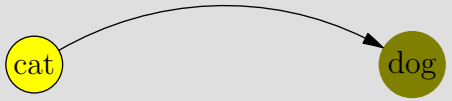
好了，现在严教授就可以这样画出两个结点及连线了：

```
node cat = Circle("cat", (0,0), fillpen=yellow),
     dog = Circle("dog", (2cm,0), fillpen=olive, drawpen=nullpen);
draw(cat);
draw(dog);
draw(cat -- dog, Arrow);
```

事情还没有完。上面只是定义了两个结点之间的直线连接，严教授还需要用曲线把结点连接起来。也就是说，严教授还希望有某种方便的 .. 运算符。

理想中的 .. 运算符应该是什么样的？回想第二章中曲线的绘制。把两个结点直接用 .. 连起来是没有什么用的，因为 cat..dog 就应该得到一条直线，那么 cat--dog 就足够了；而 cat{NE}..{SE}dog 这种记法合理一些，但也好不到哪里去，因为它并不比 cat.dir(NE){NE}..{SE}dog.dir(SW) 简单多少，同样需要手工去设想准确的连接方向。至于 **tension**、**curl** 等，更难看出实际的用处来。——说到底，严教授需要的是一种完全不同于以往的记法。考虑这个图形：

```
node cat = Circle("cat", (0,0), fillpen=yellow),
     dog = Circle("dog", (5cm,0), fillpen=olive, drawpen=nullpen);
draw(cat);
draw(dog);
draw(cat.angle(30){dir(30)} .. {dir(-30)}dog.angle(150), Arrow);
```



最后一行的曲线连线看起来十分复杂，出现了四个角度，但它们其实加起来只表示一个意思：就是从 cat 从左边偏 30° 连一条曲线拐到 dog，这个 30° 是相对 cat 和 dog 的直线连线而说的。因此，看起来合情合理的记法就呼之欲出了，它可以是这个样子的：

```
draw(cat .. bendleft .. dog, Arrow);
```

这个记法很像是定义路向控制点的 z0 .. **controls** c .. z1 记法，因而也很容易理解。而前面利用 angle 成员函数对 cat 和 dog 连线的复杂的写法，实际上揭示了这个记法的实现方法。

如果不考虑新记法的问题，首先可以实现把两个结点用曲线连起来的一个函数。不妨就叫它 bendleft：

```
// 用弯曲路径连接两个结点，第一个实现
path bendleft(node nd1, node nd2)
{
    path g1 = shift(nd1.pos) * nd1.outline;
    path g2 = shift(nd2.pos) * nd2.outline;
    pair c1 = (max(g1)+min(g1)) / 2;
    pair c2 = (max(g2)+min(g2)) / 2;
    real deg = degrees(c2 - c1);
    return nd1.angle(deg+30) {dir(deg+30)}
        .. {dir(deg-30)} nd2.angle(180+deg-30);
}
```

前面的计算就和直线连接一样，找出了两个结点的中心坐标 c1 和 c2。最后利用 degrees 函数（它计算向量的角度，正是 dir 函数的反函数），计算了直线 c1--c2 的角度 deg。最后返回的曲线路径就和前面直接连接 cat 和 dog 的过程差不多，只是考虑了 deg 的影响：所谓向左偏 30°，其实就是在角度 deg 的基础上加上 30°，从而得到 nd1 边框上点的角度和在这点的曲线切线方向，类似地可以得到 nd2 相关的角度。

要得到 nd1 .. bendleft .. nd2 这种新的记法，那大概就应该是一个接受三个参数的运算符函数：

```
? typedef path edgeconnector(node nd1, node nd2);
?
? path operator..(node nd1, edgeconnector con, node nd2)
? {
?     return con(nd1, nd2);
? }
```

这里利用 **typedef** 定义了一个连线的 edgeconnector 类型，其实就是从两个结点连成一条路径的函数类型的别名。然后就让 **operator..** 函数接受两个结点和一个连线函数，返回用这个函数计算得到的连线。

可惜，当我们用 cat .. bendleft .. dog 来测试这段代码的使用时，Asymptote 毫不留情地发出了一条错误信息：

```
no matching function 'operator ..(node, path(node nd1, node nd2))'
```

分析这条错误信息我们发现，Asymptote 并不把 cat .. bendleft .. dog 看成是一个整体，而是先按从左到右的优先级，去计算 cat .. bendleft，然后发现没有这种运算符的定义的错误。这无疑是说，想要用一个 operator.. 函数授受三个不同的参数的做法是成功不了的，前面的定义是错误的。

看上去问题陷入了僵局：我们已经定义了十分方便的 cat -- dog，却无法正确定义 cat .. bendleft .. dog，而不得不使用 bendleft(cat, dog)。后一种函数调用的记法并不复杂，但缺乏一致性。

然而严教授总有办法：既然 Asymptote 只允许 cat .. bendleft，那么何不就让它先算出一个正确的 cat .. bendleft 呢？可以让 cat .. bendleft 先算出一个中间值（比如说 f），它包含 cat 和 bendleft 的全部信息，然后再计算 f .. dog 以 bendleft(cat,dog) 作为结果。这个过程有点技巧性，关键是需要一个中间值，它仍然得是一个新的类型。

有两种实现中间值的办法，一种是使用结构体，一种是建立一个一元函数。严教授选择了后一种，因为它更简洁，也更能体现函数作为一种动态数据的作用：

```
typedef path edgeconnector(node nd1, node nd2);
typedef path edgemaker(node nd);

// nd1 .. con .. nd2 的前一半
edgemaker operator..(node nd, edgeconnector con)
{
    return new path(node nd2) {
        return con(nd, nd2);
    };
}

// nd1 .. con .. nd2 的后一半
path operator..(edgemaker maker, node nd)
{
    return maker(nd);
}
```

这时，使用

```
draw(cat .. bendleft .. dog, Arrow);
```

就能画出正确的图形了。注意，严教授在函数 **operator..**(node,edgeconnector) 中直接返回了一个新的函数，也就是一个 edgemaker 的对象。

类似地，还可以定义右弯的函数 bendright，这和 bendleft 函数的区别仅仅在于 30° 的符号。不过严教授使用了更一般的方式定义了一个向任意角度弯转的 bend 函数：

```
// 用弯曲路径连接两个结点，第二个实现
edgeconnector bend(real ang)
{
    return new path (node nd1, node nd2) {
        path g1 = shift(nd1.pos) * nd1.outline;
        path g2 = shift(nd2.pos) * nd2.outline;
        pair c1 = (max(g1)+min(g1)) / 2;
        pair c2 = (max(g2)+min(g2)) / 2;
        real deg = degrees(c2 - c1);
        return nd1.angle(deg+ang) {dir(deg+ang)}
            .. {dir(deg-ang)} nd2.angle(180+deg-ang);
    };
}

edgeconnector bendleft = bend(30);
edgeconnector bendright = bend(-30);
```

这里，edgeconnector 类型的函数被作为 bend 的返回值得到，因而 bendleft 就被定义为 bend(30)，而 bendright 被定义为 bend(-30)。如果需要更多的东西，甚至还可以加上构造曲线的 **tension** 值，扩充的余地还有很大。

更进一步，严教授还需要给他的自动机画循环的函数，来画出这样的图形来：

现在只要对 cat 一个结点向自己做连线，怎样的记法才是最自然的呢？直接用函数的形式 loop(cat) 就不错，如果需要确定循环连线的方向，那么 loop(cat,up) 也不错。不过严教授坚持在连线时使用统一的记法，那么比较好的写法就应该是

```
draw(cat .. loop(up), Arrow);
```

有了前面的东西，这个循环的记法并不难实现。而且严教授在前面定义的 edgemaker 类型在这里得到了应用：

```
path operator..(node nd, edgemaker maker)
{
    return maker(nd);
}

// 自动机循环
edgemaker loop(pair direction, real ratio=1.5)
{
    return new path(node nd) {
        real deg = degrees(direction);
        real angle1 = deg - 15, angle2 = deg + 15;
        pair mid = nd.angle(deg)
            + ratio*fontsize(currentpen)*unit(direction);
        return nd.angle(angle1) {dir(angle1)} .. mid
            .. {-dir(angle2)} nd.angle(angle2);
    };
}
```

最后，严教授发现，像 § 4.3 中那样每画一个结点就用一个 draw 函数的方法也不是什么好的记法。理想的办法应该是可以在一个 draw 函数中使用任意多个结点，就像

```
draw(cat, dog, elephant);
```

这种方式。然而，要使用“任意多个”参数，就必须依赖 Asymptote 的一种新的语法了，那就是可变长参数表。

带有可变长参数的函数原型为：

```
返回类型 function(前面的参数 ... T[] 数组)
```

其中使用 `...` 把前面的参数和最后的一个数组类型的参数分开（中间没有逗号），前面的参数和普通的函数参数一样，最后一个数组由在函数参数列表的末尾所有类型为 T 的那些参数构成。因为这些参数放在函数参数表的最后，所以在 Asymptote 中又被称为剩余参数（rest arguments）。

因而，如果画出一个结点数组的函数是

```
// 画出结点数组
void draw(picture pic=currentpicture, node[] nodearr)
{
    for (node nd: nodearr)
        add(pic, shift(nd.pos) * nd.stuff);
}
```

则画出一次多个结点的函数就是

```
// 画出一个或多个结点
void draw(picture pic=currentpicture ... node[] nodearr)
{
    draw(pic, nodearr);
}
```

在 `draw` 的代码中，严教授又使用了一种新的 **for** 循环记法：

```
for (elem : arr)
    循环语句体
```

表示让 `elem` 取遍数组 `arr` 中的元素进行循环。

最终，把所有这些东西都用在一起，就足够用相当简洁的语法画出一个完整的自动机的图形了（图 4.3）：

```
real u = 2cm;
pen StateText = white;
pen StateFill = deepgreen;
pen StateDraw = darkgreen+0.6;
pen AcceptDraw = darkgreen+1.8;
node q0 = Circle("$q_0$", (0,0), StateText, StateFill, StateDraw),
     q1 = Circle("$q_1$", q0.pos + u*S, StateText, StateFill, StateDraw),
     q2 = Circle("$q_2$", q1.pos + u*E, StateText, StateFill, AcceptDraw),
     q3 = Circle("$q_3$", q0.pos + u*E, StateText, StateFill, StateDraw),
     q4 = Circle("$q_4$", q3.pos + u*E + 0.5u*S, StateText, StateFill, StateDraw),
     q5 = Circle("$q_5$", q4.pos + u*E, StateText, StateFill, StateDraw),
     q6 = Circle("$q_6$", q5.pos + u*E, StateText, StateFill, AcceptDraw),
     q7 = Circle("$q_7$", q6.pos + u*E, StateText, StateFill, AcceptDraw);
draw(q0, q1, q2, q3, q4, q5, q6, q7);
pen edgepen = fontcommand("\scriptsize\ttfamily");
draw(".",     q0 -- q1, edgepen, Arrow);
draw("[0-9]", q1 -- q2, edgepen, Arrow);
draw(".",     q3 -- q2, edgepen, Arrow);
draw("[eE]",  q2 -- q4, edgepen, Arrow);
draw(Label("[0-9]", LeftSide),  q0 -- q3, edgepen, Arrow);
draw(Label("[eE]", LeftSide),   q3 -- q4, edgepen, Arrow);
draw(Label("[+-]", LeftSide),   q4 -- q5, edgepen, Arrow);
draw(Label("[0-9]", LeftSide),  q5 -- q6, edgepen, Arrow);
draw(Label("[fFlL]", LeftSide), q6 -- q7, edgepen, Arrow);
draw("[0-9]", q3 .. loop(N), edgepen, Arrow);
draw("[0-9]", q2 .. loop(S), edgepen, Arrow);
draw("[0-9]", q6 .. loop(N), edgepen, Arrow);
draw("[0-9]",  q4 .. bendright .. q6, edgepen, Arrow);
draw("[fFlL]", q2 .. bendright .. q7, edgepen, Arrow);
```

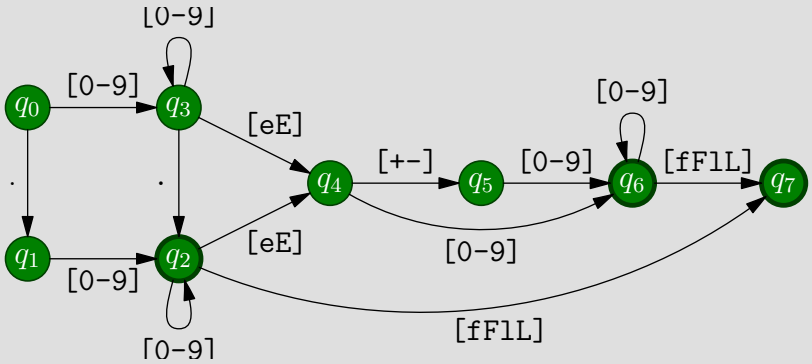


图 4.3: 严教授初步完成的有限自动机

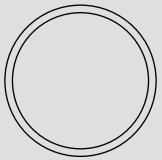
图 4.3 已经完成了整个有限自动机的整体框架，而且用来画出它的代码相当清晰，没有再玩弄看不清意义的变换 t，也完全免除了逐个考虑连线起末点的坐标方向。这正是新记法的威力。下面，严教授将逐步精化这个图形，按照理想中的要求，完善整个图形的细节。

4.5 完善图形细节

仔细审视图 4.3 的效果，严教授发现理想中的几点要求还没有做到：一是对于表示接受状态的结点 q_2, q_6, q_7，应该使用双线画出，而不是现在这样用一条粗线代替；二是为了最终效果的美观，结点应该加上灰色阴影的修饰；三是箭头显得太大，而且应该在末端处缩短一点，不与所指向的结点相接才好看。阴影效果只是为了修饰，但双线效果却是正确画出有限自动机图形所必需的特征，因而前面画出的似是而非的图形是不能拿到讲台上的。此外，原来手稿中的“开始”还没有画出来，它只是一个普通的文字标签，但又像图的结点一样有连线，一时还拿不准如何处理。

对于双线，首先想到的解决方案是画一大一小两条线。看上去只要把边框路径相对中心做一个小小的放缩：

```
path outline = circle((0,0), 1cm);
draw(outline ^^ scale(0.9)*outline);
```



然而这样做会带来许多问题：一是当放缩中心不在原点时，要进行比简单放缩更复杂的变换；二是放缩比例难以正确的确定；第三点是最主要的，就是当边框不再是前面定义的圆形时，其实任何放缩比例都不能得到正确的双线——距离中心远的地方双线距离也远，距离中心近的地方双线距离太近：

```
path outline = box((-3cm,-1cm), (3cm,1cm));
draw(outline ^^ scale(0.9)*outline, linewidth(1));
```

其实得到双线的合适办法画一粗一细两条线，粗的是线的颜色，细的则是背景色，粗线的宽度正好是细线的 3 倍：

```
path outline = box((-3cm,-1cm), (3cm,1cm));
draw(outline, black+3);
draw(outline, white+1);
```

再来看阴影。双线的解决办法是画两遍，其实阴影也是一样。要得到阴影效果，只要在原来图形的后面平移一小块位置用灰色再填充一遍：

```
path outline = circle((0,0), 1cm);
fill(shift(5SE) * outline, gray);
fill(outline, deepgreen);
```

不要这里要注意，前面的图形必须填充上颜色或者用 unfill 挖掉边框内的部分，否则灰色的阴影就露馅了。

综合来看，前两个问题其实是相同的：那就是结点的绘制不再能简单地用三个画笔（文字、填充、边框）描述清楚，而需要双线、阴影等更复杂的效果。这提示严教授必须修改结点构造时的绘制过程，也必须修改结点的构造函数。

要实现复杂的绘制过程，最有效的办法莫过于使用函数。用根据边框进行复杂画图的函数代替原来的画笔，就可以得到新的结点构造函数。首先给这种函数类型定义一个新的别名：

```
typedef void draw_t(picture pic, path g);
```

于是圆形结点就变成：

```
// 圆形结点的构造函数，第三个实现
node Circle(Label text, pair pos, pen textpen=currentpen,
            draw_t drawfn)
{
    node nd;
    nd.pos = pos;
    label(nd.stuff, text, textpen);
    pair M = max(nd.stuff), m = min(nd.stuff);
```

```
    nd.outline = circle(0.5*(M+m), 0.5*length(M-m));
    drawfn(nd.stuff, nd.outline);
    nd.E = nd.dir(E);
    nd.N = nd.dir(N);
    nd.W = nd.dir(W);
    nd.S = nd.dir(S);
    return nd;
}
```

形式上的改动是微小的：第二个实现中函数原型的 fillpen 和 drawpen 参数都代以函数参数 drawfn，而原来绘制的 filldraw 语句也换成了 drawfn 语句。

不过这样一来，暂时就要使用这样的语句来构造结点了：

```
node cat = Circle("cat", (0,0),
                  new void(picture pic, path g) {
                      filldraw(pic, g, yellow, black);
                  }
                 );
```

为了能让使用普通方法画结点还像原来一样方便，就得定义一些与 fill、draw、 filldraw 等对应的 draw_t 类型的函数：

```
// 空函数，不做边框的绘制
draw_t none = new void(picture pic, path g){};
// 画边框线
draw_t drawer(pen p)
{
    return new void(picture pic, path g) {
        draw(pic, g, p);
    };
}
```

```
draw_t drawer=drawer(currentpen);
// 填充边框
draw_t filler(pen p)
{
    return new void(picture pic, path g) {
        fill(pic, g, p);
    };
}
draw_t filler=filler(currentpen);
// 对边框填充并画线
draw_t filldrawer(pen fillpen, pen drawpen=currentpen)
{
    return new void(picture pic, path g) {
        filldraw(pic, g, fillpen, drawpen);
    };
}
```

这里，drawer(pen) 函数返回一个用给定画笔画边框线的 draw_t 函数，但严教授同时又定义 drawer 本身也是一个 draw_t 函数，使用当前画笔画线。类似地，填充的 filler 也有两种形式。这实际上是利用了 Asymptote 的函数重载功能（两个 drawer、两个 filler 其实都是参数不同的函数类型）。

于是，现在又可以像原来一样方便地画结点了：

```
node cat = Circle("cat", (0,0), filldrawer(yellow, black)),
     dog = Circle("dog", (2cm,0), drawer);  //drawer 相当于 drawer(currentpen)
draw(cat, dog);
```

当然，问题的关键是完成画双线和阴影的复杂函数。有了前面的铺垫，画双线边框是轻而易举的：

```
// 画双线边框，第一个实现
draw_t doubledrawer(pen p)
{
    return new void(picture pic, path g) {
        draw(pic, g, p+3*linewidth(p));
        draw(pic, g, white+linewidth(p));
    };
}
draw_t doubledrawer = doubledrawer(currentpen);

node cat = Circle("cat", (0,0), doubledrawer),
     dog = Circle("dog", (2cm,0), doubledrawer(red+1));
draw(cat, dog);
```

这个双线的实现考虑到了使用不同的边框画笔，并且自动计算画笔和白色背景的宽度，看上去是完美无缺了。不过，严教授仍然不满意：这种实现方式还是不够一般，不仅背景色不一定是白色的，而且结点可能不仅需要画双线，还需要填充、画阴影或是做别的什么操作——有无数种组合的可能性，却必须为这每种可能性单独写一个 draw_t 类型的函数。既然前面已经定义了画线、填充这些基本操作，那何不把它们组合起来呢？

严教授意识到，组合起多个 draw_t 函数，形成一个新的画结点边框的函数，这个功能是的应用可能十分广泛。因此，他写出了把多个 draw_t 函数复合起来的函数，也就是在复合函数中依次调用这些函数：

```
// 多个边框绘制函数的复合
draw_t compose(... draw_t[] drawfns)
{
    return new void(picture pic, path g) {
        for (draw_t f : drawfns)
```

```
            f(pic, g);
        };
    }
```

这样，实现双线的函数就成为两个 drawer 的复合了：

```
// 画双线边框，第二个实现
draw_t doubledrawer(pen p, pen bg=white)
{
    return compose(drawer(p+linewidth(p)*3), drawer(bg+linewidth(p)));
}
draw_t doubledrawer = doubledrawer(currentpen);
```

更重要的是，可以更自由地组合不同的函数功能了：

```
node cat = Circle("cat", (0,0), compose(filler(yellow), doubledrawer)),
     dog = Circle("dog", (2cm,0), compose(filler(olive), doubledrawer(darkgreen+1)));
draw(cat, dog);
```

下面对于阴影，只要考虑阴影本身，阴影之外的功能都可以通过复合函数得到了。因而画阴影的函数就很平常了：

```
// 画阴影
draw_t shadow(pair shift=2SE, real scale=1, pen color=gray)
{
    return new void(picture pic, path g) {
        fill(pic, shift(shift)*scale(scale)*g, color);
    };
```

```
}
draw_t shadow=shadow();
```

使用阴影时，因为阴影在图形的最下方，因而应该在复合函数的最前面：

```
node cat = Circle("cat", (0,0), compose(shadow, filldrawer(yellow))),
     dog = Circle("dog", (2cm,0), compose(shadow, filler(olive)));
draw(cat, dog);
```

最后来看箭头的问题。在 [3] 中，详细说明了箭头类型 **arrowhead** 的各种参数，如使用 Arrow(6) 就表示使用 6 bp 大小的箭头，修改也起来十分容易，例如：

```
Arrow = Arrow(6);
draw((0,0) -- (2cm,0), darkblue+2, Arrow);
```

然而要将箭头缩短一点，就没有现成的参数可用了。但可以把画箭头的曲线路径截短。可以使用 § 3.4 中提及的 subpath 函数取路径的一段。于是只要知道所截路径首尾的时间参数。严教授查了 [3]，得知可以使用

```
real arclength(path p);
real arctime(path p, real L);
```

来分别得到路径的总长度和长度 L 处的时刻。于是，把路径 g 缩短 2 bp 就可以写成：

```
subpath(g, 0, arctime(g, arclength(g) - 2bp));
```

这正是严教授需要的。

缩短路径长度的表达式长而复杂，当然可以写成一个函数使用。但追求形式美的严教授甚至给缩短路径长度的函数调用一个新的记法：

```
typedef path fpath(path);
path operator@(path p, fpath t)
{
    return t(p);
}

fpath shorten(real pre=0, real post=2)
{
    return new path(path p) {
        return subpath(p, arctime(p, pre), arctime(p, arclength(p)-post));
    };
}
fpath shorten=shorten(0,2);
```

这种记法几乎就是前面用曲线连接结点表达式的翻版。使用这种记法，严教授就可以用

```
g @ shorten
```

或是

```
g @ shorten(2)
```

来表示把 g 缩短 2 bp 的功能。事实上，这种记法用在实际的绘图命令中非常简洁明快：

```
node cat = Circle("cat", (0,0), compose(shadow, filldrawer(yellow))),
     dog = Circle("dog", (2cm,0), compose(shadow, filler(olive)));
draw(cat, dog);
draw(cat -- dog @ shorten, Arrow);
```

至此，所有的准备工作就全部完成了，每一个细节都被严教授仔细实现出来。如果读者一直跟着严教授的思路一步步做下来的话，那么已经可以完整地画出严教授一开始设想的有限自动机（图 4.2）了。

4.6 自定义模块

可是，严教授还不满意（他总是对现状不够满足）。最大的问题是，现在一点一点拼凑起来的代码已经凌乱得难以看清，尤其是大量自定义类型和功能函数实现的代码已经完全淹没了实际描述并画出有限自动机主体的代码——好像写了很多东西，到头来却忘了最初的目的。而如果严教授想在他的幻灯片中画出类似的图形，是否还要从这些零乱的代码中挑选可用的部分呢？

当然不。解决的办法仍然与之前一样：既然绘图与计算的过程可以抽象为函数、数据与函数的组合可以抽象为结构体，那么一样可以用抽象的办法把一组类型和函数封装 起来，这就成为一个模块。

就好像 § 1.3 中的 math 模块、§ 1.5 中的 roundedpath 模块、§ 3.5 中的 graph 模块这些标准模块一样，严教授为了画出有限自动机而做出的所有这些类型、函数、运算符，也可以定义为一个模块，供以后导入使用。建立新模块的办法十分简单，就是把模块中的所有函数都放进一个 Asymptote 文件中。因此，严教授就把所有这些与结点 node 类型相关的代码整理了一下，都装进了一个 simplenode.asy 的文件中：

```
1  //#-*- coding: utf-8 -*-
2  // simplenode.asy
3  // 关于绘制图论图形、自动机、流程图的简单模块
4
5  // 结点类型，第二个实现
6  struct node {
7      path outline;
8      picture stuff;
9      pair pos;
10     pair E, N, W, S;
11     pair dir(pair v)
12     {
```

```
13          path g = shift(pos) * outline;
14          pair M = max(g), m = min(g), c = 0.5*(M+m);
15          path ray = c -- c + length(M-m)*unit(v);
16          return intersectionpoint(g, ray);
17      }
18      pair angle(real ang)
19      {
20          return this.dir(dir(ang));
21      }
22  }
23
24  // 画出结点数组
25  void draw(picture pic=currentpicture, node[] nodearr)
26  {
27      for (node nd: nodearr)
28          add(pic, shift(nd.pos) * nd.stuff);
29  }
30
31  // 画出一个或多个结点
32  void draw(picture pic=currentpicture ... node[] nodearr)
33  {
34      draw(pic, nodearr);
35  }
36
37  // 边框绘制函数类型
38  typedef void draw_t(picture pic, path g);
39
40  // 多个边框绘制函数的复合
```

```
41 draw_t compose(... draw_t[] drawfns)
42 {
43     return new void(picture pic, path g) {
44         for (draw_t f : drawfns)
45             f(pic, g);
46     };
47 }
48
49 // 空函数，不做边框的绘制
50 draw_t none = new void(picture pic, path g){};
51 // 画边框线
52 draw_t drawer(pen p)
53 {
54     return new void(picture pic, path g) {
55         draw(pic, g, p);
56     };
57 }
58 draw_t drawer=drawer(currentpen);
59 // 填充边框
60 draw_t filler(pen p)
61 {
62     return new void(picture pic, path g) {
63         fill(pic, g, p);
64     };
65 }
66 draw_t filler=filler(currentpen);
67 // 对边框填充并画线
68 draw_t filldrawer(pen fillpen, pen drawpen=currentpen)
```

```
{
    return new void(picture pic, path g) {
        filldraw(pic, g, fillpen, drawpen);
    };
}

// 画双线边框，第二个实现
draw_t doubledrawer(pen p, pen bg=white)
{
    return compose(drawer(p+linewidth(p)*3), drawer(bg+linewidth(p)));
}
draw_t doubledrawer = doubledrawer(currentpen);

// 画阴影
draw_t shadow(pair shift=2SE, real scale=1, pen color=gray)
{
    return new void(picture pic, path g) {
        fill(pic, shift(shift)*scale(scale)*g, color);
    };
}
draw_t shadow=shadow();

// 圆形结点的构造函数，第三个实现
node Circle(Label text, pair pos, pen textpen=currentpen,
            draw_t drawfn)
{
    node nd;
    nd.pos = pos;
```

```
97      label(nd.stuff, text, textpen);
98      pair M = max(nd.stuff), m = min(nd.stuff);
99      nd.outline = circle(0.5*(M+m), 0.5*length(M-m));
100     drawfn(nd.stuff, nd.outline);
101     nd.E = nd.dir(E);
102     nd.N = nd.dir(N);
103     nd.W = nd.dir(W);
104     nd.S = nd.dir(S);
105     return nd;
106 }
107
108 // 直线连接结点
109 path operator--(node nd1, node nd2)
110 {
111     path g1 = shift(nd1.pos) * nd1.outline;
112     path g2 = shift(nd2.pos) * nd2.outline;
113     pair c1 = (max(g1)+min(g1)) / 2;
114     pair c2 = (max(g2)+min(g2)) / 2;
115     path edge = c1 -- c2;
116     edge = firstcut(edge, g1).after;
117     edge = lastcut(edge, g2).before;
118     return edge;
119 }
120
121 // 结点连线函数类型
122 typedef path edgeconnector(node nd1, node nd2);
123 // 由单结点生成边的函数类型
124 typedef path edgemaker(node nd);
125
```

```
126 // nd1 .. con .. nd2 的前一半
127 edgemaker operator..(node nd, edgeconnector con)
128 {
129     return new path(node nd2) {
130         return con(nd, nd2);
131     };
132 }
133
134 // nd1 .. con .. nd2 的后一半
135 path operator..(edgemaker maker, node nd)
136 {
137     return maker(nd);
138 }
139
140 path operator..(node nd, edgemaker maker)
141 {
142     return maker(nd);
143 }
144
145 // 以 ang 角弯曲连曲线
146 edgeconnector bend(real ang)
147 {
148     return new path (node nd1, node nd2) {
149         path g1 = shift(nd1.pos) * nd1.outline;
150         path g2 = shift(nd2.pos) * nd2.outline;
151         pair c1 = (max(g1)+min(g1)) / 2;
152         pair c2 = (max(g2)+min(g2)) / 2;
153         real deg = degrees(c2 - c1);
```

```
154         return nd1.angle(deg+ang) {dir(deg+ang)}
155             .. {dir(deg-ang)} nd2.angle(180+deg-ang);
156     };
157 }
158 
159 // 左弯与右弯
160 edgeconnector bendleft = bend(30);
161 edgeconnector bendright = bend(-30);
162 
163 // 自动机循环
164 edgemaker loop(pair direction, real ratio=1.5)
165 {
166     return new path(node nd) {
167         real deg = degrees(direction);
168         real angle1 = deg - 15, angle2 = deg + 15;
169         pair mid = nd.angle(deg)
170             + ratio*fontsize(currentpen)*unit(direction);
171         return nd.angle(angle1) {dir(angle1)} .. mid
172             .. {-dir(angle2)} nd.angle(angle2);
173     };
174 }
175 
176 // 对路径作任意变换的函数类型
177 typedef path fpath(path);
178 
179 // 用函数 t 作用于路径 p
180 path operator@(path p, fpath t)
181 {
```

```
182     return t(p);
183 }
184 
185 // 缩短路径。默认为末端缩短 2bp
186 fpath shorten(real pre=0, real post=2)
187 {
188     return new path(path p) {
189         return subpath(p, arctime(p, pre), arctime(p, arclength(p)-post));
190     };
191 }
192 fpath shorten=shorten(0,2);
```

把自定义的模块与使用模块的 Asymptote 文件放在一起，只要使用一个 **import** 命令，就能在一个单独的文件中画图，使用自定义模块的全部功能了：

```
1  //#-*- coding: utf-8 -*-
2  // automata.asy
3  // 描述 C 语言浮点数词法有限自动机（需要 simplenode 模块）
4  import simplenode;
5  
6  settings.tex="xelatex";
7  usepackage("xeCJK");
8  texpreamble("\setCJKmainfont{SimHei}");
9  
10 real u = 2cm;
11 Arrow = Arrow(6);
12 pen text = white;
13 pen starttext = black;
14 currentpen = linewidth(0.6) + fontcommand("\scriptsize\ttfamily");
15 
```

```
draw_t Initial = none;
draw_t State = compose(shadow, filldrawer(deepgreen, darkgreen+0.6));
draw_t Accepting = compose(shadow, filler(deepgreen),
                           drawer(darkgreen+1.8), drawer(white+0.6));

node q0 = Circle("$q_0$", (0,0), text, State),
     q1 = Circle("$q_1$", q0.pos + u*S, text, State),
     q2 = Circle("$q_2$", q1.pos + u*E, text, Accepting),
     q3 = Circle("$q_3$", q0.pos + u*E, text, State),
     q4 = Circle("$q_4$", q3.pos + u*E + 0.5u*S, text, State),
     q5 = Circle("$q_5$", q4.pos + u*E, text, State),
     q6 = Circle("$q_6$", q5.pos + u*E, text, Accepting),
     q7 = Circle("$q_7$", q6.pos + u*E, text, Accepting);
node start = Circle("开始", q0.pos + 0.7u*W, starttext, Initial);
draw(start, q0, q1, q2, q3, q4, q5, q6, q7);

draw(start -- q0 @ shorten, Arrow);
draw(".",     q0 -- q1 @ shorten, Arrow);
draw("[0-9]", q1 -- q2 @ shorten, Arrow);
draw(".",     q3 -- q2 @ shorten, Arrow);
draw("[eE]",  q2 -- q4 @ shorten(1,2), Arrow);
draw(Label("[0-9]", LeftSide),  q0 -- q3 @ shorten, Arrow);
draw(Label("[eE]", LeftSide),   q3 -- q4 @ shorten, Arrow);
draw(Label("[+-]", LeftSide),   q4 -- q5 @ shorten, Arrow);
draw(Label("[0-9]", LeftSide),  q5 -- q6 @ shorten, Arrow);
draw(Label("[fFlL]", LeftSide), q6 -- q7 @ shorten(1,2), Arrow);
draw("[0-9]", q3 .. loop(N) @ shorten, Arrow);
draw("[0-9]", q2 .. loop(S) @ shorten(1,2), Arrow);
draw("[0-9]", q6 .. loop(N) @ shorten(1,2), Arrow);
```

```
45 draw("[0-9]",  q4 .. bendright .. q6 @ shorten, Arrow);
46 draw("[fFlL]", q2 .. bendright .. q7 @ shorten(1,2), Arrow);
```

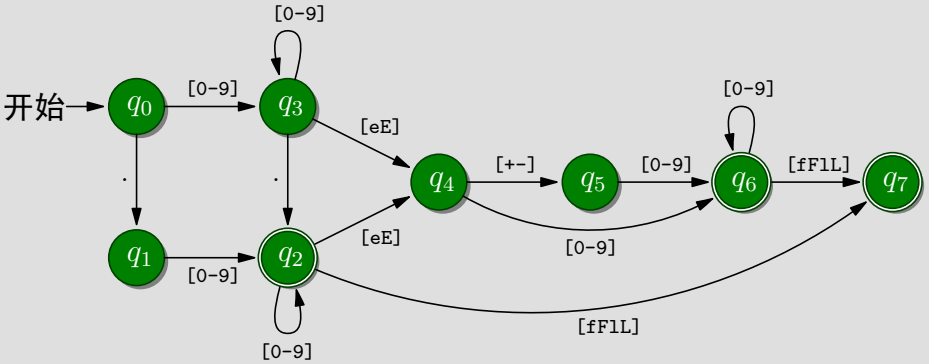
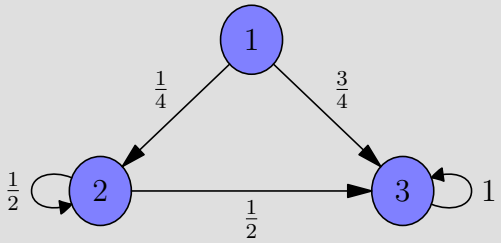


图 4.4: 最终完成的有限自动机，同图 4.2

以上就是严教授画出图 4.4 的全部代码。而且，更重要的是，他为了画出这个有限自动机所做出的各种努力有着比画出这一幅图更多的回报：他可以继续使用已经完成的 simplenode 模块方便地画出类似的自动机图形，并且可以扩充这个模块，以完成更丰富的功能。

4.7 习题和评注

1. 使用 Asymptote 本身的 **object** 标签及连线机制，画出下面的状态转移图：



注意其中自环箭头的绘制。

利用 simplenode 重画此图，比较两种方式有什么不同。

2. （此题较难）查看 Asymptote 标准模块 plain_filldraw.asy 中对 **filltype** 类型的实现，参考 [3] 的说明，分析其机制。在 Asymptote 1.91 中， **filltype** 被定义为：

```
struct filltype
{
  typedef void fill2(frame f, path[] g, pen fillpen);
  fill2 fill2;
  pen fillpen;
  pen drawpen;

  int type;
  static int Fill=1;
  static int FillDraw=2;
  static int Draw=3;
  static int NoFill=4;
  static int UnFill=5;

  void operator init(int type=0, pen fillpen=nullpen, pen drawpen=nullpen,
                     fill2 fill2) {
```

```
    this.type=type;
    this.fillpen=fillpen;
    this.drawpen=drawpen;
    this.fill2=fill2;
  }
  void fill(frame f, path[] g, pen p) {fill2(f,g,p);}
}
```

搞清楚：fill2 是什么类型？参考 Fill 等 **filltype** 填充类型的实现，尤其是其中 RadialShade 的实现，写一个 Shadow 的填充类型，完成前面严教授给结点画阴影函数相同的效果。在 Asymptote 1.90 中 RadialShade 被实现为：

```
filltype RadialShade(pen penc, pen penr)
{
  return filltype(new void(frame f, path[] g, pen) {
    pair c=(min(g)+max(g))/2;
    radialshade(f,g,penc,c,0,penr,c,abs(max(g)-min(g))/2);
    });
}
```

其效果是：

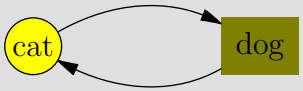
```
object cat
    = draw("cat", ellipse, (0,0), xmargin=2mm, RadialShade(yellow, red));
```

3. 仿照前面 Circle 函数的实现，为 simplenode 模块增加矩形、椭圆、菱形或是其他基本形状结点的构造函数，以便画出更丰富的图形。你可以需要一些额外的参数，如文字与边框的距离 margin。矩形结点的构造函数原型可以是这样的（你可以有不同的做法）：

```
node Rectangle(Label text, pair pos, pen textpen=currentpen,
               real margin=fontsize(textpen)/3, draw_t drawfn);
```

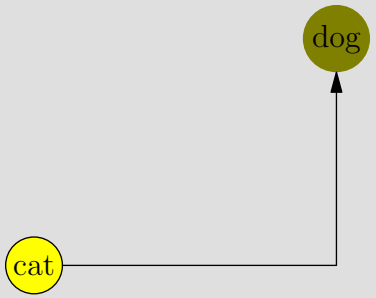
应该能和 Circle 一样方便地使用不同的形状，如：

```
node cat = Circle("cat", (0,0), filldrawer(yellow)),
     dog = Rectangle("dog", (3cm,0), filler(olive));
draw(cat, dog);
draw(cat .. bendleft .. dog, Arrow);
draw(dog .. bendleft .. cat, Arrow);
```



4. 模仿 bendleft、bendright 的定义，为 simplenode 模块扩充 edgeconnector 类型的连线方式。例如，可以增加水平、垂直方向的折线连接 HV，它的用法类似于：

```
node cat = Circle("cat", (0,0), filldrawer(yellow)),
     dog = Circle("dog", (4cm,3cm), filler(olive));
draw(cat, dog);
draw(cat .. HV .. dog, Arrow);
```



再定义类似的 VH，它先画垂直的线，再画水平的线。（提示：HV 和 VH 的定义可能与 **operator--** 非常相似。）

5. 在前面实现 `cat .. bendleft .. dog` 这种语法时，严教授使用一个函数作为中间类型。事实上也可以使用一个结构体。请补全下面的代码：

```
typedef path edgeconnector(node nd1, node nd2);

// 中间类型
struct node_con {
    node nd;
    edgeconnector con;
}

// nd1 .. con .. nd2 的前一半
node_con operator..(node nd, edgeconnector con)
{
    // ...
}

// nd1 .. con .. nd2 的后一半
path operator..(node_con nc, node nd)
{
    // ...
}
```

6. 研究记法。

   (a) 在把多个 `draw_t` 类型的函数复合为一个函数时，严宇教授定义了 `compose` 函数。但还可以进一步赋予函数复合一种新的记法，如使用加法运算符表示复合。试补全下面的实现：

```
draw_t operator+(draw_t f1, draw_t f2)
{
    // ...
}
```

并动手试验，能否把加法运算符函数的原型定义这个样子：

```
? draw_t operator+(... draw_t[] fns);
```

并说明事实上前面定义的二元运算就已经足够了。

(b) 新记法的创立需要符合直观性。前面用加法运算符表示函数的复合函数的方式是否符合直观？把加法换成乘法怎么样？

(c) 严教授定义的 shorten 函数之所以能方便地使用，一个前提条件是在 Asymptote 中 @ 运算符的优先级比 -- 和 .. 要低，因而在变换路径时不需要加括号。查阅资料或推测，Asymptote 中不同运算符的优先级。设计实验并动手在 Asymptote 环境中验证。并考虑在 Asymptote 中所有可重载的运算符中，各自有什么语法特点或限制（能带几个参数、参数类型有没有限制等）。

7. 严教授已经给出了几种 draw_t 的类型：none、drawer、filler、 filldrawer、shadow 等等。请扩充 simplenode 中的定义，增加类似放射渐变填充类型 RadialShade 的效果。想想还有什么可以扩充的，比如说， § 4.1 中球形立体效果的图形。

8. Asymptote 的 flowchart 是用来画流程图的一个模块。它定义了一种特殊的记法，看起来很像严教授 simplenode 模块的用法，也有 § 4.1 中 **object** 连线的影子。

   flowchart

本章的有限自动机及其表示的正则语言，可参看 [1]，C 语言词法的形式化描述参看 [4]。

在 Asymptote 中绘制图论图形一直没有很完善的官方模块，以往多是采用 § 4.1 一节 **object** 类型的机制绘制的。随 Asymptote 安装的基本模块 flowchart 是一个专用于绘制流程图的模块，里面定义了 **block** 类型，及包括矩形、菱形、圆形、圆角矩形、斜多边形（bevel）在内的多种预定义形状，在安装目录下可以找到例子。1.90 版以前的 flowchart 因为没有采用像本章所采用的方便的记法，使用的方法与 **object** 类型类似，所以很不方便；自 1.91

版以后的 flowchart 模块开始支持相对方便的记法了，实现的方法与本章中介绍的类似。习题 8 给出了 flowchart 的相关例子。使用 flowchart 并不足以完成本章带有阴影等修饰和曲线连线的图形，但有可能扩充这个模板完成类似的功能。

加拿大 Waterloo 大学的博士生 Hubert Chan 曾给出过一个图论模块的框架 graphtheory2.asy（在 http://asymptote.sourceforge.net/links.html 有介绍），也完成了类似的功能，并与本章给出的 simplenode 模块有相似，都使用了一些新的记法。

本章的 simplenode 模块是在作者在以前写的一个 node 模块的基础上简化改编得到的，源代码可以在 http://asy4cn.googlecode.com/ 找到。

本章中 simplenode 的一些设计受 Ti*k*Z 宏包的影响，参看 [11]。

附录 A　GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

A.1　Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general–purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non–free.

The precise terms and conditions for copying, distribution and modification follow.

A.2 Terms and Conditions

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright–like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non–source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general–purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those

works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti–Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non–permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

(a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

(b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

(c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non–Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine–readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer–to–peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which

is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non–consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non–permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non–permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer–to–peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross–claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non–exclusive, worldwide, royalty–free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non–exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING

BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

A.3 End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

索引

参考文献

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974. 这是算法设计与分析的经典文献，机械工业出版社有中文译本：《计算机算法的设计与分析》.

[2] Hans Hagen. *METAFUN*, preliminary edition, January 2002.

[3] Andy Hammerlindl, John Bowman, and Tom Prince. *Asymptote: the Vector Graphics Language*, 1.91 edition, 2009. 这是 Asymptote 的官方手册，刘海洋给出了文档的一个翻译，见 `http://code.google.com/p/asy4cn/`.

[4] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual*. Prentice–Hall, fifth edition, February 2002. 机械工业出版社有中译本《C 语言参考手册》.

[5] John D. Hobby and the MetaPost development team. *METAPOST: A User's Manual*, 1.005 edition, May 2008.

[6] Philippe Ivaldi. *geometry.asy: Euclidean geometry with Asymptote*, 1.0 edition, May 2009.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C programming Language*. Prentice–Hall, second edition, 1988. 这是 ANSI C 语言的经典文献，机械工业出版社有中译本《C 程序设计语言》.

[8] Donald Ervin Knuth. *Seminumerical Algorithm*, volume 2 of *The Art Of Computer Programming*. Addison–Wesley, second edition, 1981. 这是算法分析的经典文献，国防工业出版社有中文译本《计算机程序设计艺术：半数值算法》（旧译《计算机程序设计技巧：半数值算法》）.

[9] Donald Ervin Knuth. *The METAFONTbook*, volume C of *Computers & Typesetting*. Addison Wesley, 1986.

[10] Eric S. Roberts. *The Art and Science of C: A Library Based Introduction to Computer Science*. Addison Wesley, 1995. 机械工业出版社有中文译本.

[11] Till Tantau. *The TikZ &* PGF *Packages*. Institut für Theoretische Informatik Universität zu Lübeck, 2.00 edition, February 2008.

[12] Timothy Van Zandt. *PSTricks: PostScript macros for Generic TEX*, 97 edition, July 2003.

[13] 曲安京. 商高、趙爽與劉徽關於勾股定理的證明. 數學傳播, 20(3), 1998.