Asymptote 作图指南

莫图 (gzmfig@gmail.com)

2007 年 1 月 25 日

内容提要:

本书向你介绍一个功能强大的作图语言: Asymptote.

设计 Asymptote 的灵感来自有名的 MetaPost. 但由于 Asymptote 的语法格式与 C/C++ 很接近, 使得它更容易被接受, 而且易学易懂不容易忘记. 你可以轻而易举地用它作出达到出版要求的非常精美的数学图形, 例如几何图、示意图、函数和数据曲线或曲面图等. Asymptote 具有极大的可扩展性, 稍加学习就可以写出自己的模块来.

Asymptote 能够与 LaTeX 完美地结合. 可以把 Asymptote 代码直接嵌入到 LaTeX 文件, 也可以使用 Asymptote 制作出高质量的 EPS 图形, 然后插入到 LaTeX 文件中. Asymptote 利用 LaTeX 对图形中的文字和数学公式排版, 这保证你的文章中的插图和正文具有完全相同的风格. 就象 LaTeX 是数学排版的标准工具一样, Asymptote 的作者们也正在试图把它做成科技作图的标准工具!

本书就是关于 Asymptote 的介绍. 读完之后你将学会怎样绘制出非常“专业”的数学图形. 相信从中学到大学, 从学生到教师, 都会有它的受益者的.

Asymptote 作图指南

本书由作者使用 LaTeX 及其 CCT 中文宏包排版. 全部插图由作者使用 Asymptote 绘制.

目 录

第一章　概述

1.1　Asymptote 是什么?

Asymptote 是一款按照 GNU GPL[1] 授权的自由软件, 你可以免费地得到并使用它. 它的作者是 A. Hammerlindl, J. Bowman, 和 T. Prince. Asymptote 最初是为 Unix 和 Linux 操作系统设计的, 但后来也移植到了其它主流操作系统, 例如 MacOS 和 MS Windows 等.

Asymptote 是专门用来画高质量的精确数学图的. 你可以轻而易举地用它作出符合出版要求的各种数学图形, 例如几何图、数学示意图、函数和数据曲线或曲面图等等. 因此, Asymptote 是一个作图软件而不是图像处理软件. 把 Asymptote 与 LaTeX 结合使用将能够排版出完美的科技文章或者书籍. 这也正是 Asymptote 作者们的初衷.

Asymptote 可以多种格式输出图形, 默认的情况下产生 Postscript 输出. 它也可以输出 ImageMagic 所支持的任何格式, 例如 pdf, jpg, gif, png 等等. 你当然可以把这些格式的图形插入到网页以及其它文档中. Asymptote 调用 LaTeX 来处理文本标签和数学公式排版, 因此图形中的文字质量也是绝对一流.

其实, 与其说 Asymptote 是一款软件, 不如说它是一种语言, 一种专门为作数学图而设计的高级计算机编程语言. 这里应该提到著名的 MetaPost 语言. Asymptote 的作者们正是受到 MetaPost 的启发才开发了 Asymptote. MetaPost 的设计目的几乎与 Asymptote 相同, 但它的语法对于大多数用户来说显得很陌生, 学习起来难度很大. 鉴于这一点, Asymptote 吸收了 MetaPost 的一些非常好的特点, 但使用了一种类似 C/C++ 语言的语法规则. 这使得它更加易于接受, 从而减小了学习和使用的难度.

有人可能一听到“编程”就感到头疼. 因此 Asymptote 不会令所有的人满意. 但不管怎么说, 至少我认为, 仅仅因为这一点就离开 Asymptote 的确非常遗憾. 用 Asymptote 编程可能远不像你想象的那样可怕. 如果你仅仅是拿它来作一些图（这正是大多数人的目的）, 它可以是非常简单的. 打个比方, 比如你想算个简单的数学—就象 $\sin(\pi/5)$ 之类的—你所需要做的并不是系统地学习 C 语言然后再编个程序, 而仅仅是去找个计算器按按键就行了. Asymptote 语言也是这个意思. 当然, 如果你属于那种对编程不头疼的人, 那么你一定会在 Asymptote 中发现更多的乐趣, 并会使你自己以及他人受益的.

1.2　Asymptote 的特点

有很多自由的和商业的科技作图软件, 相信读者曾经或者正在使用着它们. 那么为什么还需要 Asymptote 呢? 其实很简单, 每个软件都有各自的优缺点, 没有一个万能的软件可以做好每件事情. 每个软件的特点决定了什么人会使用它以及使用它做什么. Asymptote 的主要特点包括:

- 是一种计算机高级程序语言, 语法类似于 C/C++;

[1] GNU General Public License （GNU 通用公共许可证）, 按照这一许可证授权的软件是开放源代码的自由软件. 顾名思义, 你可以得到它的源代码, 并在一定条件下具有修改甚至发行它的权力. 具体内容见 http://www.gnu.org/licenses/gpl.txt. 自由软件不见得一定是免费的, 但 Asymptote 是免费的.

- 使用精确的坐标系统, 可以输出高质量的向量图;
- 用 LaTeX 排版图中的文字和数学公式;
- 具有很大的灵活性, 用户通常可以找到办法作出满足自己意愿的图;
- 具有很强的可扩展性, 对于常用的功能可以写出通用的模块, 这类似于 LaTeX 的宏包和 C/C++ 的库.

这些特点令 Asymptote 区别于其它软件. 特点本身谈不上好坏, 任何特点都有可能, 在某些情况下, 给你带来方便或者麻烦.

不同于 Origin, SigmaPlot, Grace, XFig 等作图软件, Asymptote 是编程语言. 也就是说, 它是基于代码的. 你要用它作图, 就要写一段程序. 这也表明, 它不是“所见即所得”的. 对那些喜欢使用鼠标点来点去的读者来说, 这无疑是一个缺点（希望你不觉得这个缺点很严重）. 而考虑到 Asymptote 的用户多半也会使用 LaTeX, 那么写写代码应该也算不上绝对不可接受的.

但也恰恰是这一点给我们带来了很多好处. 我们可以完全按照我们自己的意愿来精确地安排作图方式和内容, 我们可以重复使用代码, 我们还可以做很多鼠标完成不了的其它工作. 基本上, Asymptote 的所有其它特点都是基于这一点的. 所以, 我把它当作 Asymptote 的优点来看.

同样是基于代码的, 还有一些其它的科技作图软件, 例如前面提到的 MetaPost, 以及 gnuplot, Gri, Pyx 等. 它们也都是各具特色. 有时候结合起来使用可能更好. 比如作数据图时, 我就喜欢先用 gnuplot 大致看看, 最后定稿时再用 Asymptote 输出.

1.3 互联网上的资源

Asymptote 的官方网站位于 http://asymptote.sourceforge.net, 在那里可以找到它的最新版本下载链接以及最新的文档和相关信息等.

第二章　安装和配置

2.1　安装前的准备

Asymptote 依赖一些其它软件完成任务, 因此在安装 Asymptote 前应该先将这些软件安装好.

首先是 TEX 系统. 由于 Asymptote 使用 LATEX 来对图中的文字和数学公式排版, 因此你的系统中必须有一个可用的 TEX 系统. 对于 Unix 或 Linux 用户, 通常在操作系统中已经配置好了 teTEX, 否则请你的系统管理员安装它. 如果需要使用中文, 还需安装 CJK 宏包, 并配置好中文字体. 对于 MS Windows 用户, 建议安装中文 CTEX 套装, 它可以在 http://www.ctex.org 下载. 这些软件都是免费的.

其次是免费的 GhostScript 和 GhostView. 这是用来处理和显示 PostScript 文件的软件. 同样地, 在 Unix 或者 Linux 操作系统下, 它们通常已经安装好了. 而在 MS Windows 下你需要自己安装它们, 在上面提到的 CTEX 中文套装中已经包含了它们.

Asymptote 提供了一个简单的图形界面程序 xasy, 用来对 Asymptote 作出的图形对象作简单的修改. 如果你需要使用这个工具, 那么还需要一个 Python (http://www.python.org) 的解释程序. 你可以免费得到它. (在 Unix 或者 Linux 操作系统下, 它们通常已经安装好了.)

ImageMagick 是可选的. 它也是免费的自由软件, 用于处理和显示多达数十种格式的图像. 它有用于各种操作系统的版本.

2.2　安装 Asymptote

2.2.1　从源代码安装

Asymptote 是开放源代码的软件, 你可以免费地得到全部源代码, 然后在本地电脑上编译并安装. 例如在 Unix 或 Linux 操作系统下, 从源代码安装 Asymptote 的方法如下:

1. 首先通过 Asymptote 主页上的链接下载它的源代码. 它是一个打包的压缩文件, 文件名为 asymptote-x.xx.tar.gz, 其中 x.xx 表示版本, 例如 1.04 表示这是 Asymptote 的第 1.04 版的源代码.

2. 使用下面的命令把它解压缩:

   ```
   tar -zxf asymptote-x.xx.tar.gz
   cd asymptote-x.xx
   ```

   然后下载 http://www.hpl.hp.com/personal/Hans_Boehm/gc_source/gc6.7.tar.gz 并把它放在当前目录中.

3. 执行下面的命令来编译 Asymptote:

   ```
   ./configure
   make all
   ```

4. 以 root 的身份执行:

```
make install
```

至此就完成了 Asymptote 的安装.

按照上面的步骤从源代码安装 Asymptote 时, 需要注意以下几点:

1. 你可能需要先安装 FFTW (Fastest Fourier Transform in the West, http://www.fftw.org). 这是用于离散傅立叶变换的程序库(对于非商业应用, 它是免费的自由软件). 如果你不需要在 Asymptote 中进行傅立叶变换的运算, 那么你不需要它.

2. 你可能需要先安装 GSL (GNU Scientific Library, http://www.gnu.org/software/gsl). 这是一个用于科学计算的免费的自由函数库. 如果你事先安装了 GSL, 就可以在 Asymptote 中直接计算很多特殊函数.

3. 系统默认安装位置为: 可执行文件 asy 和 xasy 位于 /usr/local/bin, 基本模块位于 /usr/local/share/asymptote, 例子和文档位于 /usr/local/share/doc/asymptote .

4. 如果想改变默认编译和安装设置, 可以给 configure 命令加上相应选项. 执行下面的命令可以列出所有选项:

```
./configure --help
```

5. 如果你不具有 root 权限, 你仍然可以在自己的个人目录下安装 Asymptote. 这时, 你需要执行下面的命令来编译和安装:

```
./configure --prefix=\$HOME/asymptote
make all
make install
```

但你要确认, 编译好的可执行文件 asy 和 xasy 位于你的 $PATH 路径中, 并需要适当配置 Asymptote, 参见第 2.3 节.

2.2.2 编译好的安装包

对于不熟悉或者不方便编译源代码的用户, 可以选择使用已经预先编译好的二进制安装包来安装 Asymptote.

Linux 系统: Asymptote 的官方网站上提供了 i386 系统的安装包. 实际上它只是一个压缩文件, 文件名是 asymptote-x.xx.i386.tar.gz. 以 root 身份执行:

```
tar -C / -zxf asymptote-x.xx.i386.tar.gz
texhash
```

就会把 Asymptote 安装到默认目录下 (/usr/local).

Debian 用户可以可以从以下网址得到安装包:

http://www.uhoreg.ca/programing/debian/pool/main/a/asymptote

http://packages.debian.org/asymptote

MacOS X 系统: 安装包可以从以下网址下载:

http://www.hmug.org/pub/MacOS_X/BSD/Applications/Publishing/asymptote

MS Windows 系统: Asymptote 的官方网站上提供了 MS Windows 下的安装程序直接运行它就可以安装 Asymptote 到系统中.

默认的安装位置为 C:\Program Files\Asymptote. 如果你需要安装到其它位置, 那么在安装完成后必须作适当的配置, 参见第 2.3 节.

这个安装程序实际上仅仅是个自解压程序, 它不会修改你的注册表和环境变量. 为了在安装后能够让操作系统找到可执行文件 asy.exe, 你需要手工修改环境变量 PATH 的值, 使得它包含 asy.exe 所在的目录.

2.3 配置 Asymptote

2.3.1 配置文件

在某些情况下, 你需要对 Asymptote 进行适当的配置才能够使用它. 这包括 Asymptote 的路径设置和其它辅助软件的路径设置.

在你的个人主目录 (Unix 或 Linux 下为 $HOME, MS Windows 下为 %USERPROFILE%) 下建立一个名为 .asy 的子目录, 在这个目录中建立一个名为 config.asy 的文本文件. 然后将你的配置内容输入这个文件中即可.

config.asy 的第一行是:

```
import settings;
```

以下各行都具有下面这样的形式:

```
配置变量="字符串";
```

例如在 Linux 下, 你的配置文件的内容可能是:

```
import settings;
psviewer="kghostview";
pdfviewer="acroread";
```

而在 MS Windows 下, 你的配置文件的内容可能是:

```
import settings;
dir="d:\asymptote";
psviewer="C:\Ghostgum\gsview\gsview32.exe";
pdfviewer="C:\Adobe\Acrobat\Reader\AcroRd32.exe";
```

当然, 具体内容取决于你的系统.

通常你可能要在配置文件中设定以下配置变量或者其中的一部分:

- dir — 这一变量用于设置 Asymptote 的搜索目录, 参见第 2.3.2 节;
- psviewer — PostScript 文件查看程序, 例如 GhostView;
- pdfviewer — PDF 文件查看程序, 例如 Acrobat Reader;
- gs — PostScript 解释程序, 例如 GhostScript;

- `python` — `python` 解释程序;
- `latex` — TEX 系统提供的 LATEX 编译程序;
- `dvips` — TEX 系统提供的 `dvips` 程序;
- `convert` — ImageMagick 提供的图像格式转换程序;
- `display` — ImageMagick 提供的图像显示程序;
- `animate` — ImageMagick 提供的动画显示程序;
- `xasy` — `Asymptote` 的图形界面工具.

在配置文件中还可以设置其它一些配置变量, 在此不详细介绍了, 请读者参考 `Asymptote` 的文档.

2.3.2 搜索目录

在运行 `Asymptote` 的时候, 它按照下面的顺序搜索 `Asymptote` 的系统文件:

1. 当前目录;
2. 用户个人主目录(Unix 或 Linux 下为 `$HOME`, MS Windows 下为 `%USERPROFILE%`)下的 `.asy` 子目录;
3. 配置文件中由配置变量 `dir` 指定的目录, 如指定多个目录, 在 Unix 或 Linux 下用冒号分隔, 在 MS Windows 下用分号分隔;
4. 系统目录, 在 Unix 或 Linux 下默认为 `/usr/local/share/asymptote/`, 在 MS Windows 下默认为 `C:\Program Files\Asymptote`.

2.4 运行 Asymptote

我们现在已经准备好 `Asymptote` 了. 照例, 我们先用它来向世界问好.

2.4.1 交互模式

打开终端窗口(在 MS Windows 下, 打开一个 MSDOS 命令行窗口), 在提示符后面键入命令 `asy` 并按下回车键. 如果一切正常, 你将看到下面的欢迎信息:

```
Welcome to Asymptote version 1.04 (to view the manual, type help)
>
```

这表示我们已经进入了 `Asymptote` 的交互模式了. 第二行开头的 `>` 是 `Asymptote` 的提示符, 它提示我们输入 `Asymptote` 命令.

在提示符后面输入以下命令(请你暂时不要去管这些命令的含义):

```
draw((-3cm,-1cm)--(3cm,-1cm)--(3cm,1cm)--(-3cm,1cm)--cycle);
```

如果你已经配置好了 psviewer 配置变量, 那么你会发现有一个 PostScript 查看程序 (例如 GhostView) 被打开. 先不管它, 我们回到终端窗口. 在这里, 你看到了另一个提示符, 表示它正在等待你的下一个命令. 这时继续输入下面的命令:

```
label("Hello, World!");
```

现在去看看刚才打开的 PostScript 查看程序（如果需要, 请刷新它的显示）, 你会看到在一个长方形的中央有一串文字 “Hello World!”, 就象图 2.1 中的那样.

图 2.1: 用 Asymptote 作的第一幅图.

接下来, 在终端窗口的提示符后面输入:

```
quit;
```

我们就退出了 Asymptote 的交互模式回到系统.

请你这时查看以下当前目录下的文件, 你会发现多出了一个 out.eps. 它正是我们刚才用 Asymptote 作图的结果.

2.4.2 批处理模式

除了交互模式, Asymptote 也可以在批处理模式下运行. 这更像我们通常编程的做法. 你简单地把全部命令保存到一个纯文本文件中, 并以 .asy 作为它的扩展名. 例如, 建立一个纯文本文件 helloworld.asy, 其中包含下面的内容（每行前面的数字表示行号, 它们并不是文件内容的一部分）:

```
1 draw((-3cm,-1cm)--(3cm,-1cm)--(3cm,1cm)--(-3cm,1cm)--cycle);
2 label("Hello, World!");
```

在终端窗口中执行:

```
asy helloworld
```

你就会在当前目录下得到一个名为 helloworld.eps 的文件, 它与刚才的 out.eps 完全一样.

2.4.3 两种模式的比较

Asymptote 提供的这两种运行模式各有特点. 在交互模式下我们可以逐条地执行命令, 即时看到结果. 而在批处理模式下, 我们可以快速地完成作图任务.

在多数情况下, 我们会使用批处理模式. 这也正是用其它计算机语言编写和执行程序的一般做法. 而且它允许我们把整个程序保存起来, 便于将来修改和使用.

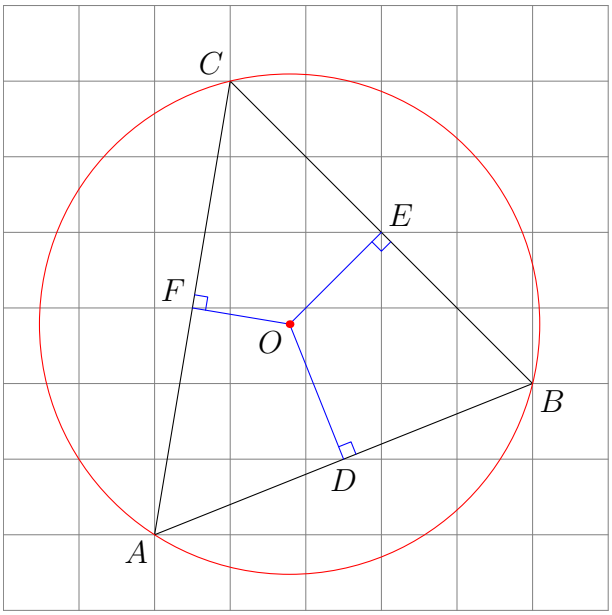
交互模式则可以很方便地用来调试程序的片段, 观察程序的局部效果等.

第三章　一个实例

本章将通过一个例子来使你对用 Asymptote 语言编程有一个初步的印象, 并对 Asymptote 的语法有一个大体上的了解. 通过本章的学习, 你将能够使用 Asymptote 编写出简单的程序, 作出不太复杂的图形. 后续各章将给出 Asymptote 语言的细节.

3.1　作图任务

假设现在你是一个初中几何教师, 正准备讲“三角形外接圆”这一内容. 在准备教案和课件的时候, 你觉得应该画一幅图来帮助你. 于是你边在纸上画草图, 边在心中确定了整个作图方案, 它看起来就像图 3.1 的样子.



$\triangle ABC$ 的外接圆 $\odot O$.
三角形的外心是三角形三条边的垂直平分线的交点.

图 3.1: 最终的作图计划.

在这幅图中需要分别作出几种不同的图形元素. 大体来说, 图中包括了一个背景网格、一个三角形、一个外接圆、几条中垂线、和一些文字.

3.2　背景网格

我们这里的背景网格是由一些横线和竖线组成. 每一条线可以由其两个端点来确定. 因此, 只要给定了两个点的坐标, 就可以画出一条线段了.

在 Asymptote 中, 点的坐标用 (x,y) 来表示, 其中 x 和 y 是实数, 分别表示该点的横坐标和纵坐标. 这种表示方法跟我们的习惯是相符的 (实际上, 在 Asymptote 的内部点的坐标是用复数表示的, 见第 4.4 节). 可以把点的坐标赋值给一个变量, 它的数据类型是对型 (pair), 例如:

```
pair z = (1cm,5cm);
```

声明一个变量 z 并把它的值初始化为 (1cm,5cm), 在以后就可以用 z 来引用这个点了.

> 现在你可能会有一个疑惑, 我们在图 3.1 中并没有指出每个点的坐标分别是多少, 那么, "我应该把这个直角坐标系的坐标原点建在哪里呢?" 其实这并不重要, 随便你建在哪里.

现在我们不妨把坐标原点建在背景网格的左下角. Asymptote 认为, 横轴的正方向是水平向右的, 而纵轴的正方向是竖直向上的. 这也恰好符合我们的习惯.

把两个点的坐标用两个减号 -- 连接起来就是一条线段. 用 Asymptote 的术语来说, 线段是一种称为路径 (path) 的数据类型. Asymptote 提供了一个函数 draw 来画路径. 例如, 要画一条端点坐标分别为 (3cm,1cm) 和 (4.5cm,3.2cm) 的线段, 可以用命令:

```
draw((3cm,1cm)--(4.5cm,3.2cm));
```

请注意行尾的分号. 你可以在交互模式下试试看.

这里出现了一些诸如 8cm 之类的记号来表示 8 厘米的长度. 前面说过, 在一个点的坐标 (x,y) 中, x 和 y 是实数. 那么 8cm 是实数吗? 答案是肯定的, 你可以在交互模式下用下面的命令来验证这一点:

```
write(8cm);
```

这里的 write 是 Asymptote 提供的一个函数, 用来向终端屏幕或者文件输出数值等信息. 执行这个命令你将真的看到一个实数 226.771207143293. 它的意思是说, 在 Asymptote 看来, 8cm 就等于 226.771207143293.

> 有点迷惑吗? 其实很简单, Asymptote 预先定义了一些变量, 其中包括 cm. 而按照 Asymptote 的语法规则, 当一个数字和一个变量相乘并且数字在前时, 可以省略乘号. 因此 8cm 实际上就是 8 * cm. 这个规定可以使我们的程序看起来更容易一些 (详见第 4.5 节, 第 29 页).
> 当用实数来表示长度或者距离时, Asymptote 使用 PostScript 的"大点" (big point, bp) 作单位, 一个 bp 等于 1/72 英寸.

好了, 我猜你可能已经知道该怎样画背景网格了. 没错, 象这样就行:

```
draw((0cm,0cm)--(8cm,0cm));
draw((0cm,1cm)--(8cm,1cm));
draw((0cm,2cm)--(8cm,2cm));
......
draw((0cm,0cm)--(0cm,8cm));
draw((1cm,0cm)--(1cm,8cm));
draw((2cm,0cm)--(2cm,8cm));
......
```

这里我们总共有 18 条线要画, 你需要写 18 行 draw, 还好, 任务还不算很重. 但如果要画一个更大的网格怎么办?

Asymptote 中有类似于 C/C++ 中的循环结构来帮助我们做这种重复性工作. 请在交互模式下尝试:

```
for (int i = 0; i <= 2; ++i) write(i);
```

你会得到如下的输出结果:

```
0
1
2
```

这个例子演示了 `for` 循环结构. 我们首先定义了一个整型循环变量 `i`, 并将它的值初始化为 0; 然后设定循环条件 `i <= 2`, 即只要这个条件仍然满足, 就重复执行循环体; 接下来的 `++i` 是说每次执行完循环体后, 将变量 `i` 的值增加 1. 本例的循环体只有一句话, 就是 `write(i);`, 在终端屏幕上显示变量 `i` 的值.

下面是我们使用循环结构来画背景网格的程序清单:

```
1 // 背景网格
2 for (int i = 0; i <= 8; ++i) {
3   real x = i * cm;
4   // 横线
5   draw((0,x)--(8cm,x));
6   // 竖线
7   draw((x,0)--(x,8cm));
8 }
```

你发现一些行是由 `//` 开头的, 它们是注释. 当 `Asymptote` 读到 `//` 时, 它会忽略从这个地方开始直到这一行结束的全部内容. 在程序中写注释的目的是为了让人更容易读.

这个程序仅仅包含了一个循环结构. 花括号 `{}` 包围的部分是循环体. 由于循环体多于一条语句, 因此花括号是必需的. 循环体中定义了一个实型 (`real`) 变量 `x`, 后面调用了两次 `draw` 函数来分别画横线和竖线. 程序的执行结果见图 3.2.

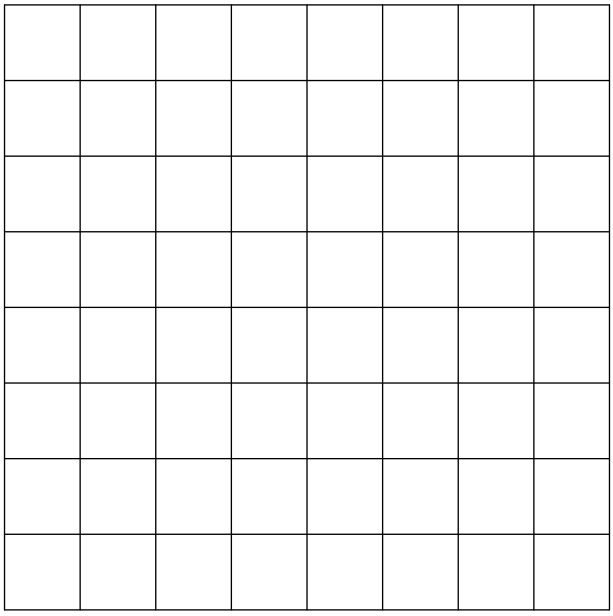


图 3.2: 背景网格的初步实现.

比较一下图 3.2 和我们的最终目标图 3.1 就会发现不同. 由于网格是属于背景性质的, 它看起来不应该很醒目. 因此我们希望用较细的线条和较浅的颜色来画它.

Asymptote 的数据类型中有一种叫做笔 (pen). 用这种数据可以定义所画线条的颜色、粗细、线型等性质. 在默认的情况下, Asymptote 用黑色、0.5bp、实线来画线条. 下面我们把程序作一些修改, 来用灰色和 0.2bp 的笔画背景网格. 为此我们定义一个 pen 类型的变量 helpline, 并在两个 draw 函数中使用它. 现在我们的程序清单为（输出结果见图 3.3）:

```
// 辅助线
pen helpline = linewidth(0.2bp) + gray(0.5);

// 背景网格
for (int i = 0; i <= 8; ++i) {
  real x = i * cm;
  // 横线
  draw((0,x)--(8cm,x), helpline);
  // 竖线
  draw((x,0)--(x,8cm), helpline);
}
```

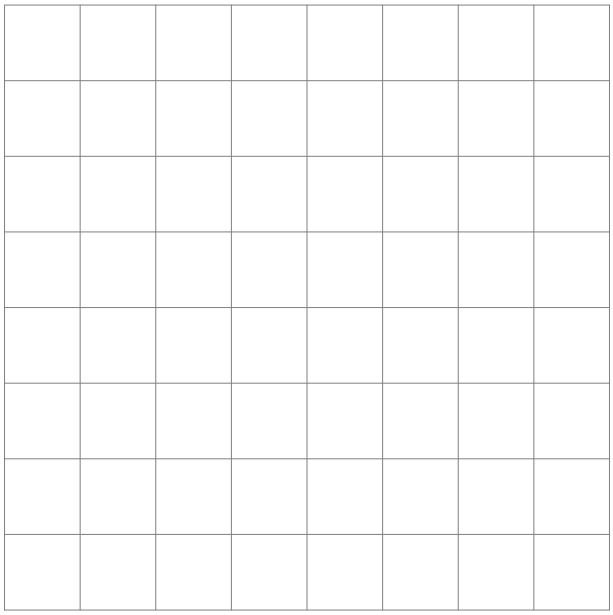


图 3.3: 修改过的背景网格.

3.3 三角形

我们把三角形 ABC 的三个顶点分别放在 (2cm,1cm), (7cm,3cm), 和 (3cm, 7cm) 处, 见图 3.1.

我们没必要一段一段地画出三角形的三条边, 而可以一次就完整地画出整个三角形的封闭路径. 程序如下（作图结果见图 3.4）:

```
1  // 辅助线
2  pen helpline = linewidth(0.2bp) + gray(0.5);
3
4  // 背景网格
5  for (int i = 0; i <= 8; ++i) {
6    real x = i * cm;
7    // 横线
8    draw((0,x)--(8cm,x), helpline);
9    // 竖线
10   draw((x,0)--(x,8cm), helpline);
11 }
12
13 // 三角形 ABC
14 pair a = (2cm,1cm), b = (7cm,3cm), c = (3cm, 7cm);
15 draw(a--b--c--cycle);
```

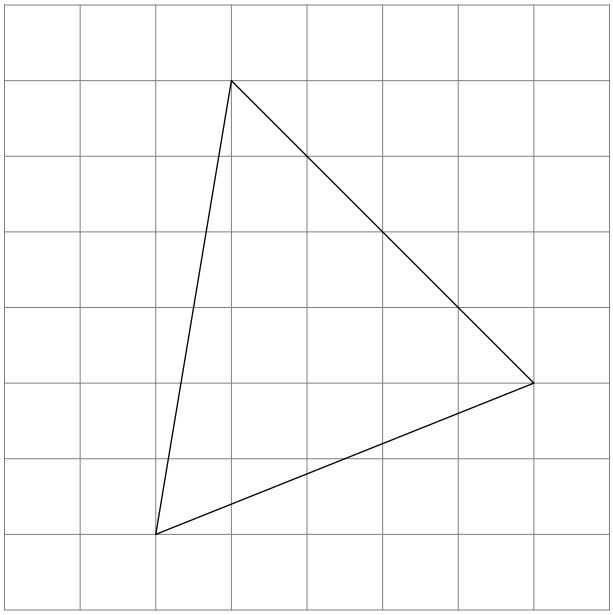


图 3.4: 在背景网格上添加三角形.

在程序的第 14 行, 我们定义了三个 `pair` 类型的变量用来记录三角形的三个顶点坐标; 第 15 行则直接调用 `draw` 函数用默认的笔画出了整个三角形. 注意, 三角形的路径是 `a--b--c--cycle`, 这里最后的 `cycle` 将把终点 `c` 和起点 `a` 连接起来形成一个封闭的路径. 封闭的路径和开放的路径不同, 差别之一是, 前者可以用 `fill` 函数填充颜色而后者不行. 注意, `a--b--c--a` 不是封闭路径, 虽然它的终点和起点是同一点.

路径不见得都是由直线段组成, 它可以是曲线. 请在交互模式下试验如下命令的结果并比较封闭和开放路径的区别:

```
draw((0,0)--(4cm,0)--(5cm,1cm)--(4cm,2cm)--(0,0));
```

```
draw((0,0)..(4cm,0)..(5cm,1cm)..(4cm,2cm)..(0,0), red);
draw((0,4cm)--(4cm,4cm)--(5cm,5cm)--(4cm,6cm)--cycle);
draw((0,4cm)..(4cm,4cm)..(5cm,5cm)..(4cm,6cm)..cycle, red);
```

3.4 外接圆

由于圆既常见又常用, Asymptote 专门提供了一个函数来产生圆形路径, circle(c, r), 其中 c 是 pair 类型, 表示圆心的坐标, r 是实数, 表示半径. 因此, 只要我们找到圆心和半径就可以用它来画外接圆了.

我们知道, 三角形的外心是三条边的垂直平分线的交点. 所以, 我们必须有至少两条垂直平分线, 并计算它们的交点. 计算两条直线的交点通常意味着要求解一个二元一次方程组, 不过不要担心, Asymptote 已经为我们准备好了这样一个函数, extension.

Asymptote 的内核其实只提供了一些基本的功能, 而大量的实用功能都能用这些基本的功能在 Asymptote 语言下实现. 这样, 人们可以把一些具有相关的通用功能的 Asymptote 代码组织到一起, 称为一个模块. 通过这种方式, 任何人都可以对 Asymptote 加以扩展. 几乎任何计算机程序设计语言都有类似的扩展方式, 例如 C/C++ 的库以及 TEX/LATEX 的宏包等. 在 Asymptote 的发行中提供了一些包括科技作图和 3 维作图等功能的基本模块.

计算两条直线交点的函数 extension 就定义在 math 模块中. 这个函数有 4 个参数, 前两个是一条直线上的两个点, 后两个是另一条直线上的两个点, 函数的返回值就是它们的交点.

现在, 我们需要确定一条边（例如 AB 边）的中垂线上的两个点. 显然其中之一可以取为 AB 的中点 D, 它的坐标为 0.5(a + b)（根据前面讲过的规则, 我们省略了一个乘号）. 而另外一点则可通过把 B 点绕 D 逆时针旋转 90° 得到. Asymptote 有一种称为变换 (transform) 的数据类型专门做这种平移、旋转、缩放等变换工作, 例如函数 rotate 就可以用来把一个点或者路径等图形元素绕某一特定点旋转一定的角度.

至此, 我们的程序清单如下（结果如图 3.5 所示）:

```
1  import math;
2
3  // 辅助线
4  pen helpline = linewidth(0.2bp) + gray(0.5);
5
6  // 背景网格
7  for (int i = 0; i <= 8; ++i) {
8    real x = i * cm;
9    // 横线
10   draw((0,x)--(8cm,x), helpline);
11   // 竖线
12   draw((x,0)--(x,8cm), helpline);
13 }
14
15 // 三角形 ABC
```

```
16 pair a = (2cm,1cm), b = (7cm,3cm), c = (3cm, 7cm);
17 draw(a--b--c--cycle);
18
19 // 中垂线、外心、外接圆
20 pair d = 0.5(a + b), e = 0.5(b + c), f = 0.5(c + a);
21 pair o = extension(d, rotate(90, d) * b, e, rotate(90, e) * b);
22
23 draw(circle(o, abs(o - a)), red);
24
25 draw(o--d, blue);
26 draw(o--e, blue);
27 draw(o--f, blue);
28
29 dot(o, red);
```

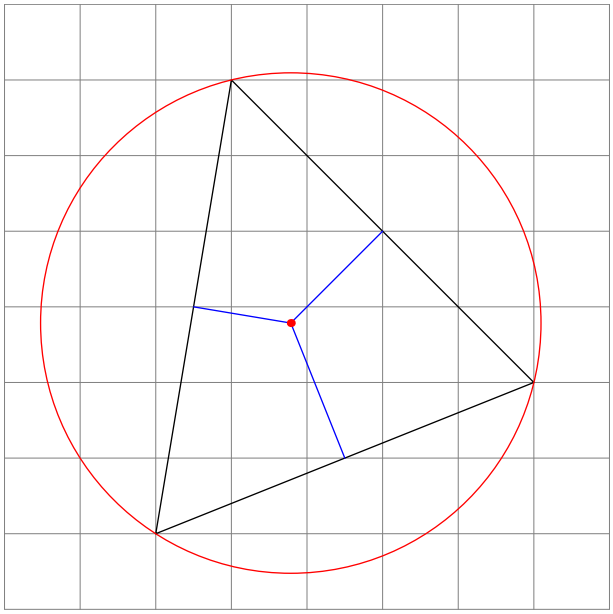


图 3.5: 添加了三条边的中垂线、外心、和外接圆.

第 1 行的 `import math;` 装入 `math` 模块, 我们要用到其中的 `extension` 函数. 第 20 行计算三条边的中点. 第 21 行计算 AB 和 BC 边中垂线的交点, 即三角形的外心 O, 其中 `rotate(90, d) * b` 得到“将 B 点绕 D 点逆时针旋转 90° 后的坐标”. 第 23 行用红色的笔画出了外接圆, 其中 `abs(o - a)` 的结果是 O 点到 A 点的距离, 即外接圆的半径. 第 29 行在圆心 O 处画一个红色的点.

3.5 直角标记

在前面的程序中, 你已经看到 `Asymptote` 中定义了很多函数帮助我们完成一些特定工作. 这一节里我们准备亲自定义一个函数来画出中垂线和相应边之间的直角标记. 虽然在 `geometry`

模块中定义了 perpendicular 函数来画直角标记, 但为了了解函数的定义方法, 我们还是决定不用这个现成的, 而由自己来定义.

我们把这个函数起名为 rightangle, 并且希望它返回一个描述直角标记的路径, 这样就可以用 draw 函数来画它. 当我们描述一个角时, 我们需要 3 个点, 其中两个点位于这个角的两条边上, 而另一个则是它的顶点, 例如 ∠ODB. 因此我们的函数应该接受 3 个参数, 分别为这 3 个点. 另一方面, 我们还需要知道该把这个标记画多大, 因此还需要另外一个实型参数.

根据上面的思路, 我们可以把这个函数定义如下:

```
path rightangle(pair a, pair b, pair c, real size = 5bp) {
  pair ba = size * unit(a - b);
  pair bc = size * unit(c - b);
  pair bb = ba + bc;
  return shift(b) * (ba--bb--bc);
}
```

这个函数总共有 4 个参数, 其中最后一个 size 表示直角符号的大小. 注意, 我们为 size 指定了一个默认值, 就是说, 当调用这个函数时, 可以不给出最后这个参数 size 的大小, 在这种情况下, 它将取其默认值 5bp.

前 3 个参数是描述角的点, 我们用 a, b, 和 c 表示. 虽然这 3 个变量的名字和三角形的三个顶点的变量名相同, 但它们是不同的变量. 这里的 a, b, 和 c 是以函数参数的形式出现的, 它们仅仅在函数体内（两个花括号{}之间）才有意义, 因此是局部变量. 当函数被调用时, 例如 rightangle((1cm,1cm), (2cm,2cm), (1cm,3cm)) , 参数 a, b, 和 c 将分别被赋值为(1cm,1cm), (2cm,2cm), 和 (1cm,3cm). 但这并不会改变三角形三个顶点的坐标, 这正是由于函数的参数与函数之外定义的三角形顶点是不同的变量. 函数内局部变量的生存期仅限于函数体内部, 当函数调用结束并返回时, 局部变量所占用的内存将被系统收回.

> 三角形的顶点是在函数之外定义的, 变量名也是 a, b, 和 c. 它们是全局变量. 但由于它们的名字刚好跟我们函数中的局部变量重名, 因此在函数体内将不能被访问, 或者说, 它们被相应的同名局部变量屏蔽了. 见第 4.3 节.

在函数体中, 我们定义了 3 个点. 象参数变量一样, 它们也是局部变量. 其中 ba 是沿着 a - b 方向并且与坐标原点的距离为 size 的点. 这里调用 unit(a - b) 函数的结果为在 a - b 方向上的单位向量（一个点也可以理解为从坐标原点指向该点的向量）. 而 ba + bc 则可以理解为向量加法. 因此, 路径 ba--bb--bc 就是一个直角标记, 只不过它的位置在坐标原点. 于是, 我们在最后不直接返回它, 而是把这个路径平移到 b 点之后返回, shift 函数帮我们做了平移的工作.

现在, 我们程序的最新版本变为（输出结果如图 3.6 所示）:

```
1 import math;
2
3 // 辅助线
4 pen helpline = linewidth(0.2bp) + gray(0.5);
5
6 // 背景网格
7 for (int i = 0; i <= 8; ++i) {
```

```
  real x = i * cm;
  // 横线
  draw((0,x)--(8cm,x), helpline);
  // 竖线
  draw((x,0)--(x,8cm), helpline);
}

// 三角形 ABC
pair a = (2cm,1cm), b = (7cm,3cm), c = (3cm, 7cm);
draw(a--b--c--cycle);

// 中垂线、外心、外接圆
pair d = 0.5(a + b), e = 0.5(b + c), f = 0.5(c + a);
pair o = extension(d, rotate(90, d) * b, e, rotate(90, e) * b);

draw(circle(o, abs(o - a)), red);

draw(o--d, blue);
draw(o--e, blue);
draw(o--f, blue);

dot(o, red);

// 直角
path rightangle(pair a, pair b, pair c, real size = 5bp) {
  pair ba = size * unit(a - b);
  pair bc = size * unit(c - b);
  pair bb = ba + bc;
  return shift(b) * (ba--bb--bc);
}

draw(rightangle(b, d, o), blue);
draw(rightangle(b, e, o), blue);
draw(rightangle(c, f, o), blue);
```

3.6 文字标签

在图中添加文字标签非常简单, `label` 专门做这个. 你可以按照如下的方式简单地调用它:

```
label("字符串", 点);
label("字符串", 点, 方向);
```

这里, 双引号中的*字符串*就是要添加的文字内容, *点* 表示添加文字的位置, *方向*则是文字相对

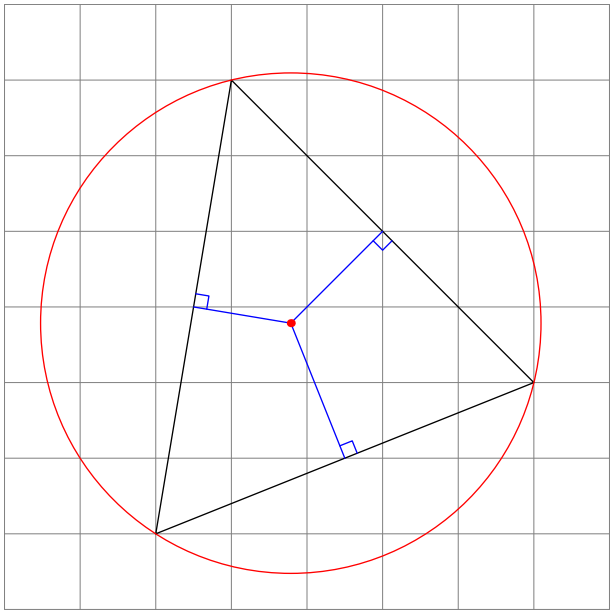


图 3.6: 添加直角标记.

于点的方向. Asymptote 已经定义好了一些方向常数, 例如 E, S, W, N 分别表示东、南、西、北, 而 SE 则是东南, 等等.

对于较多的文字, 可以使用 minipage 函数, 它实际上就是 LaTeX 中的小页环境. minipage 函数接收两个参数, 第一个是文字内容, 第二个是小页的排版宽度.

需要注意的是, 所有的文字都会被送往 *LaTeX* 进行排版. 因此, 文字中可以包含 LaTeX 格式的数学公式. 当然也可以使用中文, 但要记得使用 CJK 宏包.

最终的程序列在下面, 它的输出就是图 3.1.

```
1  import math;
2  texpreamble("\usepackage{CJK}\AtBeginDocument{\begin{CJK*}{GBK}{song}}
3    \AtEndDocument{\clearpage\end{CJK*}}");
4
5  // 辅助线
6  pen helpline = linewidth(0.2bp) + gray(0.5);
7
8  // 背景网格
9  for (int i = 0; i <= 8; ++i) {
10   real x = i * cm;
11   // 横线
12   draw((0,x)--(8cm,x), helpline);
13   // 竖线
14   draw((x,0)--(x,8cm), helpline);
15 }
16
17 // 三角形 ABC
```

```
pair a = (2cm,1cm), b = (7cm,3cm), c = (3cm, 7cm);
draw(a--b--c--cycle);

// 中垂线、外心、外接圆
pair d = 0.5(a + b), e = 0.5(b + c), f = 0.5(c + a);
pair o = extension(d, rotate(90, d) * b, e, rotate(90, e) * b);

draw(circle(o, abs(o - a)), red);

draw(o--d, blue);
draw(o--e, blue);
draw(o--f, blue);

dot(o, red);

// 直角
path rightangle(pair a, pair b, pair c, real size = 5bp) {
  pair ba = size * unit(a - b);
  pair bc = size * unit(c - b);
  pair bb = ba + bc;
  return shift(b) * (ba--bb--bc);
}

draw(rightangle(b, d, o), blue);
draw(rightangle(b, e, o), blue);
draw(rightangle(c, f, o), blue);

// 标签
label("$A$", a, SW);
label("$B$", b, SE);
label("$C$", c, NW);
label("$D$", d, S);
label("$E$", e, NE);
label("$F$", f, NW);
label("$O$", o, SW);

// 文字
label(minipage("$\triangle ABC$ 的外接圆\ $\odot O$.\\
三角形的外心是三角形三条边的垂直平分线的交点.", 2.5cm), (8.2cm,5cm), E);
```

第四章　基本数据类型和程序流程

用任何语言编写程序的目的都是为了对各种特定的数据进行处理或运算. 在 Asymptote 语言中, “数据”的概念是很广泛的, 不仅包含了逻辑、数值、字符串等一般程序设计语言所支持的类型, 更包含了众多的与作图有关的数据类型. 除此之外, 用户还可以自己定义新的扩展数据类型以实现复杂的数据结构.

Asymptote 的目的是精确作图. 作图不外乎画线条、填充区域、写文字等任务. 你可以（也必须）为任何一个图形元素指定精确的坐标. 所以, 如果能够知道每个图形元素的坐标, 那么的绘图是简单的事情. 坐标需要计算, 这往往是一个 Asymptote 程序的核心任务. Asymptote 的类似 C/C++ 的语法规则和比较强大数值计算能力, 再配以日渐完善的扩展模块, 使得你能够非常方便地计算坐标和作图.

本章将介绍 Asymptote 的基本数据类型和程序流程, 它们属于程序设计的基础知识, 但不涉及具体作图内容.

4.1　语句

一个 Asymptote 程序由若干条语句组成, 每条语句都以分号 ; 结束. 一条语句完成一个功能, 例如定义、赋值、输出、调用函数、判断、循环等.

跟 C/C++ 语言一样, 一条语句不见得写在一行上. 在 Asymptote 看来, 空格、制表符、回车符都是同样的, 统称为空白. 连续的多个空白等同于一个空白. 例如, 下面的语句

```
int i = 3;
```

完全等价于

```
int
  i  =
    3;
```

采用什么样的写法取决于你的编程习惯, 而 Asymptote 则没有意见.

> 但是在字符串中, 不同的空白字符是不等同的, 多个空白也不等同于一个空白. 简而言之, 字符串中的每个空白字符都保留它的本来含义.

可以把多条语句放在一对花括号 {} 之间组成一个语句块. 例如:

```
{
    int a=1;
    write(a);
}
```

语句块通常用于判断、循环、函数定义、结构定义等.

语句块可以嵌套, 例如:

```
{
    int a=1;
```

```
    write(a);
    {
        int b=2;
        write(a+b);
    }
    write(a);
}
```

4.2 注释语句

可以在 Asymptote 程序中添加注释. 与 C++ 一样, 有两种形式的注释:

```
// 注释内容
/* 注释内容 */
```

对于第一种形式, 从 // 开始, 直到本行的回车符之前的全部字符都是注释内容; 而对于第二种形式, 从 /* 组合开始, 直到下一个 */ 组合为止的全部字符都是注释内容. 在程序编译的过程中, 所有的注释内容都会被一个空白所取代.

出现在字符串中的 // 以及 /* 和 */ 不被看作注释的标记.

在程序中使用注释一般有两个目的. 一是为了增加一些对程序所作的说明或解释, 以便于阅读. 二是在编写或调试程序的过程中, 只是想临时让一些语句不起作用, 但又不愿意完全删掉它们.

4.3 变量

4.3.1 定义

变量用于在程序执行的过程中存储数据. 变量的名称由英文字母 (A–Z, a–z)、下划线 (_)、和数字 (0–9) 组成, 但必须以英文字母或者下划线开头. Asymptote 对变量名的长度没有限制, 因此你可以使用完整的单词作为变量名以增加程序的可读性. 需要注意的是, Asymptote 区分大小写字母. 下面给出的都是合法的变量名:

```
Var           a_var      name365
WidthOfBox2   nCount     sqrt_1_over_2
```

变量在使用前必须定义, 定义语句的语法格式为:

```
<类型说明符> <变量名> [ = <表达式> ] ;
```

其中方括号 [] 中的内容为可选的. <类型说明符> 指的是 Asymptote 的基本数据类型或者已经定义的扩展数据类型, 关于 <表达式> 见第 4.5 节. 如果在定义语句中给出了可选的 = <表达式>, 则在定义变量的同时为它赋了初始值. 例如下面的定义语句

```
int i=3;
```

定义了一个整型 (int) 变量, 变量名为 i, 且它的初始值为 3.

也可以用一个定义语句定义多个具有相同数据类型的变量, 各个变量之间用逗号分隔. 定义语句完整的语法格式为:

```
<数据类型> <变量名> [ = <表达式> ] [, <变量名> [ = <表达式> ] ... ] ;
```

例如下面的语句

```
int a=3, b, c=a+4;
```

定义了 3 个整型变量, 其中 a 和 c 分别赋了初值 3 和 7, 而 b 则没有赋初值.

定义语句可以出现在程序的任何地方, 但只有定义之后才能使用这个变量. 你可以为它赋值或者把它用于计算等.

4.3.2 作用域

每个变量都有一定的作用域, 即该变量能够被直接访问的程序范围. 在一个语句块内部定义的变量, 其作用域局限于它所在的语句块, 即从它被定义开始, 直到该语句块结束. 这种变量可以称为局部变量, 因为你只能在程序的局部范围内访问它. 在所有语句块之外定义的变量称为全局变量, 它的作用域从它被定义开始直到程序结束.

我们来看下面的程序:

```
1  int a=2;
2  {
3      int b=3;
4      {
5          int c=a+b;
6          write(c);
7          a=5;
8      }
9      // write(c); // 错误! 此处已经超出 c 的作用域
10 }
11 write(a);
```

这个程序包含了两个嵌套的语句块. 在程序的第 1 行, 我们定义了一个全局变量 a, 它的作用域直到程序结束. 第 3 行定义了作用域为外层语句块的局部变量 b. 而第 5 行则定义的局部变量 c 的作用域为内层语句块. 由于变量 a 和 b 的作用域涵盖了内层语句块, 因此我们可以在内层语句块中访问它们（第 5 行）, 甚至修改它们（第 7 行）. 我们把第 9 行注释起来, 是因为这里已经超出了变量 c 的作用域, 你不能直接访问它. 程序的运行结果为:

```
5
5
```

分别是第 6 行和第 11 行的输出.

如果两个变量有相同的变量名, Asymptote 并不会报告错误, 而是将它们视为不同的变量. 但由于它们的名字相同, 你将只能访问最后一个变量, 而无法直接访问前面的变量, 我们称前一个变量被后一个同名变量屏蔽了. 但若后一个变量的作用域首先结束, 则我们就又可以重新访问前一个变量了.

请看下面的程序:

```
1 int a=2, b=3;
2 write(a,b);
3 {
4     real a=3.5;
5     write(a,b);
6 }
7 write(a,b);
8 real b=4.2;
9 write(a,b);
```

在语句块 3–6 行中, 全局变量 int a 被 real a 暂时屏蔽, 到第 7 行之后, 这个屏蔽就不存在了. 而从第 8 行开始, 全局变量 int b 被 real b 屏蔽. 程序的运行结果为:

```
2       3
3.5     3
2       3
2       4.2
```

> 需要注意的是, 被屏蔽的变量并没有消失. 在第 5.5 节我会再次讲到这个话题.

4.4 基本数据类型

Asymptote 语言中的基本数据类型包括:

- int — 整型;
- real — 实型;
- pair — 对型;
- triple — 三元型;
- string — 字符串;
- bool — 逻辑型.

一、 整型

整型的类型说明符为 int. 整型变量只能用来存储整数. 使用整数的好处是它是精确的, 即整数运算不会产生任何误差. 因此它们经常被用来作为循环控制变量以及数组下标等.

在定义整型变量时, 可以为它指定一个初始值. 如果没有指定, 则默认设为 0. 因此下面的两行定义语句具有完全相同的作用:

```
int a;
int a=0;
```

在 Asymptote 中定义了一个常数 intMax, 它是 Asymptote 所能表示的最大正整数, 而 −intMax 则是最小的负整数. 你的程序中所使用的整数必须在这个区间之内. intMax 的具体

数值取决于你的计算机, 例如在 32 位机上, intMax = 2 147 483 647, 这是一个相当大的数, 一般来说足够应付通常的任务. 你可以在交互模式下使用下面的命令来查看这个常数:

```
write(intMax);
```

二、 实型

实型又称浮点型, 用来表示实数. 其类型说明符为 real. 实型变量的默认初始值为 0.0.

在 Asymptote 语言中, 实型常数有两种表示方法. 你可以直接写出它们, 如 3.1415926 和 -2.71828 等. 但需注意, "3" 是一个整型常量, 而 "3." 或者 "3.0" 则是实型常量. 当一个实数的整数部分是 0 时, 也可以把这部分省略, 例如 .5. 除此之外, 还可以用科学计数法的格式, 如 3.05e8 或者 3.05e+8 表示 3.05×10^8, 以及 -2.6e-5 表示 -2.6×10^{-5} 等. 可见, 在使用科学计数法形式时, 用小写字母 e 后面跟一个整数来表示指数.

Asymptote 为作图方便, 预定义了一些实型的长度单位常量: PostScript 的长度单位 "大点" (big point) bp, 它的大小是 1/72 英寸, Asymptote 默认以这个单位作图; TEX 的长度单位 "点" pt, 它比 bp 略小, 等于 1/72.27 英寸; 英寸可以写为 inches 或 inch; 厘米 cm; 毫米 mm.

在 Asymptote 中定义了常数 realMax 和 realMin. 你的程序中不能使用绝对值大于 realMax 的实数. 而绝对值小于 realMin 的实数将被当作 0.0 处理. 这两个常数的大小依赖于具体的计算机, 例如在 32 位机上, realMax = $1.79769313486232 \times 10^{308}$, 和 realMax = $2.225073858507 \times 10^{-308}$. 这个范围不算小了. Asymptote 预定义了一个常量 infinity 作为无穷大, 它等于 $\sqrt{0.25 \times \texttt{realMax}}$.

需要特别注意的是, 实数的表示是不精确的! (在其它有浮点数运算的计算机程序设计语言里同样存在这个问题.) 这是因为在计算机内部用二进制来存储数据, 而对于大多数十进制小数来说, 当把它转化为二进制的时候会成为循环节不为 0 的无限循环小数, 因此只能在一定位数上被截断, 而这样必然会带来截断误差.

请看下面的例子:

```
real x=1cm+1cm+1cm+1cm+1cm+1cm+1cm;
write(x-7cm);
```

结果会是什么? 按照理论来说, x 是 7 个 1cm 相加的和, 它应该严格等于 7cm, 因此输出结果应该是 0. 但很不幸, 在我的 32 位计算机上, 上面的程序给出的结果为 -2.8421709430404e-14, 即 $-2.8421709430404 \times 10^{-14}$. 这个误差很小, 在通常的情况下我们用不着去过分担心它. 但有时即便是这么小的一个数也会给我们带来足够大的麻烦, 比如当我们比较两个实数是否相等时, 哪怕是非常接近于 0 的微小差别, 只要它不严格是 0, 就会给出 "不相等" 的结论. 请注意, "相等" 和 "不相等" 是有本质差别的. 所以, 永远不要在程序中比较两个实数是否相等!

再看一个例子:

```
write(1e+50 + 0.1 - 1e+50);
write(1e+50 - 1e+50 + 0.1);
```

根据加法交换律, 上面两条语句都是在计算 $1 \times 10^{50} + 0.1 - 1 \times 10^{50}$ 的值, 所以单纯从数学上看, 它们的结果是一样的, 都是 0.1. 但我的计算机给出的结果却是:

```
0
0.1
```

你能解释它吗? 所以, 必要时请适当调整实数的运算顺序. 当然, 这种情况在用 Asymptote 作图时很少遇到, 不过一旦遇到, 你应该心里有数.

三、 对型

对型的类型说明符为 pair. 一个对型量是形式为 (x,y) 的有序实数对, 其中 x 和 y 分别是实数. 对型变量的默认初始值是 (0.0,0.0).

在 Asymptote 语言中, 对型数据其实就是数学中的复数, 而 (x,y) 中的 x 和 y 分别是实部和虚部. 若定义了对型变量 z:

```
pair z;
```

则它的实部和虚部可以直接用 z.x 和 z.y 取得, 但不能直接修改, 例如下面的语句是非法的:

```
z.x=2.5; // 非法!
z.y=3.8; // 非法!
```

要想改变一个对型量的值, 可以直接对它赋值, 例如下面的赋值语句都是合法的:

```
z=(2.5,3.8);
z=(z.x,3.8);
z=(2.5,-z.y);
```

Asymptote 预先定义了一个对型常量 I, 它的值是数学上的虚数单位 $\mathrm{i}=\sqrt{-1}$, 即 (0,1). 因此你也可以按下面这样对 z 赋值:

```
z=2.5+3.8I;
z=z.x+3.8I;
z=2.5-z.y*I; // 这里的乘号不能省略
```

在 Asymptote 程序中, 通常用对型数据表示一个点的 2 维坐标, 或者表示一个特定的方向. Asymptote 预定义了一些表示方向的常量, 如 up=(0,1), down=(0,-1), left=(-1,0), 和 right=(1,0) 分别表示上、下、左、右 4 个方向; E=(1,0), S=(0,-1), W=(-1,0), 和 N=(0,1) 分别表示东、南、西、北 4 个方向. 其它方向见图 4.1, 它们都是单位圆上的点 (或者说是长度为 1 的向量).

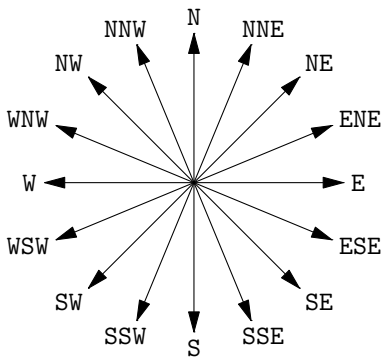


图 4.1: 表示方向的对型常量.

四、 三元型

三元型数据由三个实数组成, 形式为 (x,y,z), 其类型说明符为 triple. 三元型变量的默认初始值为 (0.0,0.0,0.0). 若 v 是三元型变量, 则它的 3 个分量可以分别用 v.x, v.y, v.z 得到, 但不可以直接修改. 三元型数据通常用来描述 3 维空间中点的坐标和方向.

在 three 模块中定义了 4 个三元型常量: X=(1,0,0), Y=(0,1,0), Z=(0,0,1), 和 O=(0,0,0).

五、 字符串

所谓字符串数据就是简单的一串字符, 其类型说明符为 string. 字符串常量有两种表示方法, 一是用两个半角双引号 " 包围一串字符, 二是用两个半角单引号 ’ 包围一串字符. 但字符串的值并不包含作为限制符的引号. 例如 "Einstein" 和 ’Einstein’ 一样, 它们的值都是 Einstein.

当使用双引号包围字符串时, 如果又想在字符串中使用双引号这个字符本身, 可以用换码序列 \", 如 "Schr\\"odinger" 的值为 Schr\"odinger.

当使用单引号作为字符串常量的限制符时, 在字符串内可以使用 ANSI C 中定义的换码序列, 见表 4.1. 但在 Asymptote 中, 你可能很少有机会用到它们.

表 4.1: 使用单引号作为字符串常量的限制符时的特殊字符序列.

换码序列	对应的字符	换码序列	对应的字符
\’	单引号 ’	\"	双引号 "
\?	?	\\	\
\a	报警	b	退格
\f	进纸	\n	换行
\r	回车	\t	水平制表符
\v	竖直制表符		
\0–\377	八进制编码相应的字符	\x0–\xFF	十六进制编码相应的字符

在 Asymptote 的字符串常量内, 所有空白字符（包括空格、制表符、和回车符）都被原样保留. 所以你可以在字符串内直接换行以得到多行文字. 例如下面的程序:

```
string a;
a="这是一个字符串
这是第二行";
write(a);
```

的执行结果为:

这是一个字符串
这是第二行

字符串有两个主要用途, 一是用于输入输出的文件名, 二是用于作图时插入文字. Asymptote 预定义了字符串型常量 defaultfilename, 它的值等于你正在编译的 Asymptote 文件的文件名, 但不包含扩展名 “.asy”.

六、 逻辑型

逻辑型是 Asymptote 语言中最简单的数据类型, 它的类型说明符是 bool. 逻辑型常量只有两个: true 和 false, 分别表示“真”和“假”. 逻辑型变量的默认初始值是 false.

逻辑型数据用来比较两个数据的关系, 例如大于、小于、相等.

4.5 运算符和表达式

计算离不开表达式. 所谓表达式就是把数据和变量用运算符和括号连接起来. 任何表达式都有一个确定的值. 单独一个常量或者变量是最简单的表达式.

Asymptote 语言中的运算符见表 4.2. 本节介绍除路径运算符之外的其它运算符.

表 4.2: 运算符.

类别	运算符	运算	运算符	运算
算术运算符	+	加	-	减
	*	乘	/	除
	%	模	^ 或 **	乘方
前缀运算符	++	增加 1	--	减小 1
比较运算符	==	相等	!=	不等
	<	小于	<=	小于或等于
	>	大于	>=	大于或等于
逻辑运算符	&&	与	||	或
	!	非	^	异或
条件运算符	?:			
赋值运算符	= += -= *= /= %= ^=			
路径运算符	& -- .. :: --- ^^			

一、 算术运算符

+, -, *, / 分别代表加、减、乘、除运算. * 和 / 的优先级高于 + 和 -, 即先算乘除后算加减, 与我们的习惯相符. 若要改变运算顺序, 可以用一对括号 () 把需要先计算的部分包围起来. 括号可以嵌套, 例如表达式 4+(3+4*(1+2))*2 的值是 34.

int, real, 和 pair 类型的数据都可以进行加减乘除运算. 一般地来说, 同种类型的两个数据作运算的结果也是该种类型的数据, 例如两个实数相乘也得到一个实数. 但有一个例外, 当两个整型量作除法运算时, 结果是一个实数而不是整数, 例如 5/2 的结果为 2.5.

triple 类型的数据只能作加减运算, 而不能作乘除运算. 但 triple 可以跟一个整数或者实数相乘, 相当于它的每个分量都乘以这个整数或实数. triple 也可以除以一个整数或实数, 即每个分量都除以这个数.

% 是取模运算符, 结果是两个数相除所得的余数. 如 8%3 的结果是 2. % 的优先级与 * 和 / 相同, 如 5*8%3 得 1, 而 8%3*5 得 10. 取模运算一般用于 int 数据, 但也可以用于 real. 例如 5.2%2.3 等于 0.6.

^ 和 ** 完全相同, 都是乘方运算符, 前者是 TEX 习惯, 而后者是 Fortran 习惯. 例如 a^b 和 a**b 的结果都是 a^b. ^ 和 ** 的优先级高于 * 和 /. int, real, 和 pair 类型的数据都可以进行乘方运算.

当计算一个整数或实数的数值常数和一个表达式的乘积, 并且数值常数在前时, 可以省略乘号 * 不写. 这个规则在有些时候会很方便. Asymptote 定义了一些变量作为长度单位, 如厘米 cm, 毫米 mm 等, 这样我们就可以用 3cm 或者 20mm 来表示 3 厘米或 20 毫米了. 再比如, 用 0.5(a+b) 来表示 a 和 b 的平均值或中点也恰好符合我们的习惯.

但省略乘号和明确写出乘号还是有一点细微差别的. 比如 a^2*b 等于 a^2b, 而 a^2b 却等于 a^{2b}. 规则是, 省略了乘号的乘法运算总是比它前一个运算符的优先级高. 但对于后一个运算符而言, 省略乘号和不省略乘号具有相同的优先级, 例如 2*cm^2 和 2cm^2 都是 2cm^2.

二、 前缀运算符

++ 和 -- 只能用于一个int, real, 或者 pair 变量的前面, 其作用为使得这个变量的值增加或者减小 1. 它们可以用在表达式中, 这时它们具有最高的优先级. 例如:

```
b=3*++a;
```

在功能上完全等价于:

```
a=a+1;
b=3*a;
```

++ 和 -- 主要用于控制循环变量的增量.

从上面的例子可以看出, 这两个运算符是完全可以用其它方法代替的. Asymptote 之所以提供它们, 是为了迎合一些 C/C++ 程序员的习惯. 请不要过度使用这两个运算符.

> Asymptote 并没有提供 C/C++ 的后缀运算符, 因此你不能写 i--. 这是由于 Asymptote 把 -- 作为路径运算符了.

三、 比较运算符

比较运算符用于比较两个值之间的关系, 运算结果是 bool 型的 true) 或者 false.

两个等号 == 比较两边的数值是否相等. 若相等则计算结果为 true, 否则为 false.

!= 刚好相反, 它比较两边的数值是否不等. 若不等则计算结果为 true, 否则为 false.

<, <=, >, 和 >= 分别比较左边的量是否小于、小于或等于、大于、大于或等于右边的量. 若是, 则结果为 true, 否则为 false.

== 和 != 的优先级低于其它比较运算符. 全部比较运算符的优先级都低于算术运算符.

<, <=, >, 和 >= 只可以用于 int 和 real 型数据的比较; 而 == 和 != 则可以用于全部基本类型数据的比较.

四、 逻辑运算符

逻辑运算符只用于 bool 型表达式的计算.

&& 作逻辑与运算. 若两边的逻辑表达式都是 true, 则计算结果为 true, 否则为 false.

|| 作逻辑或运算. 若两边的逻辑表达式中任意一个是 true, 则计算结果为 true, 否则为 false.

^ 作逻辑异或运算. 若两边的逻辑表达式结果不同, 即一个是 true, 而另一个是 false, 则计算结果为 true, 否则为 false.

! 作逻辑非运算. 若其后面的逻辑表达式是 true, 则计算结果为 false, 否则为 true.

这 4 个运算符中, ! 的优先级最高, 其次是 ^, 再次是 &&, 最低的是 ||. 除 ! 外, 其它的优先级都低于比较运算符.

五、 条件运算符

条件运算符 ?: 的用法为:

```
<bool 表达式> ? <表达式1> : <表达式2>
```

若 <bool 表达式> 为 true, 则运算结果为 <表达式1>, 否则为 <表达式2>. 例如:

```
b = a<0 ? -a : a;
```

将把 a 的绝对值赋给 b.

条件运算符的优先级低于逻辑运算符.

六、 赋值运算符

我们已经见过多次赋值运算符 = 了. 它很简单, 就是把它右边表达式的值计算出来并赋给左边的变量. 但既然是运算符, 它也有运算结果, 那就是等号左边变量的新值. 赋值运算符的运算顺序是从右至左的. 因此, 我们可以为多个变量连续赋值. 例如下面的语句:

```
a=b=3;
```

把变量 a 和 b 同时赋值为 3.

+=, -=, *=, /=, %=, 和 ^= 是来自于 C/C++ 的复合赋值运算符. 例如:

```
a+=b;  // 等价于 a=a+b;
a-=b;  // 等价于 a=a-b;
a*=b;  // 等价于 a=a*b;
a/=b;  // 等价于 a=a/b;
a%=b;  // 等价于 a=a%b;
a^=b;  // 等价于 a=a^b;
```

赋值运算符的优先级是所有运算符中最低的.

4.6 类型转换

当在表达式中对不同类型的数据作计算时, Asymptote 默认的转换规则是 int 转换为 real, real 转换为 pair. 例如 3+3.0 得到实型常数 6.0, 而 5+(3,4.5) 的结果为对型的 (8.0,4.5).

要想把一个实型数据转换为整型, 必须使用明确的类型转换, 这是由于这种转换会损失精度. 当把实数转换为整数时, 仅仅简单地去掉了它的小数部分. 例如:

```
real a=3.5;
int b=(int)a;
int c=(int)(-a);
```

则 b 的值是 3, 而 c 的值是 -3.

4.7 程序流程

如果程序只能从开头一成不变地顺序执行到结尾, 那就太枯燥了, 而且功能也会很有限. Asymptote 提供了判断和循环语句来控制程序的流程.

4.7.1 判断

判断语句的目的是选择性地执行, 即在某种情况下执行一些语句, 而在另外的情况下执行另一些语句. 它有两种格式.

第一种格式是:

```
if ( <bool 表达式> ) <true 语句 或 语句块>
```

表示当 <bool 表达式> 的值为 true 时, 执行后面的 <true 语句或语句块>.

第二种格式是:

```
if ( <bool 表达式> )
    <true 语句 或 语句块>
else
    <false 语句 或 语句块>
```

当 <bool 表达式> 的值为 true 时, 执行后面的 <true 语句或语句块>; 否则执行 else 后面的 <false 语句或语句块>. 可见, 在每次执行到判断语句时, 总是只在 <true 语句或语句块> 和 <false 语句或语句块> 之中选择一个来执行.

下面举一个例子. 假设 year 是一个整型量, 表示一个年份; 而 leapyear 是一个逻辑型量, 表示 year 这一年是否闰年. 我们可以利用 if 判断语句从 year 计算出 leapyear 来:

```
int year=2008;
bool leapyear;

if (year % 100 == 0) {
    if (year % 400 == 0)
        leapyear = true;
    else
        leapyear = false;
}
else {
    if (year % 4 == 0)
        leapyear = true;
    else
        leapyear=false;
}

write(leapyear);
```

从这个例子可以看出, if 判断语句可以嵌套, 来完成更加复杂的判断. 实际上, 上面程序中的两对花括号 {} 都可以不写, 但写出它们会使程序更具有可读性.

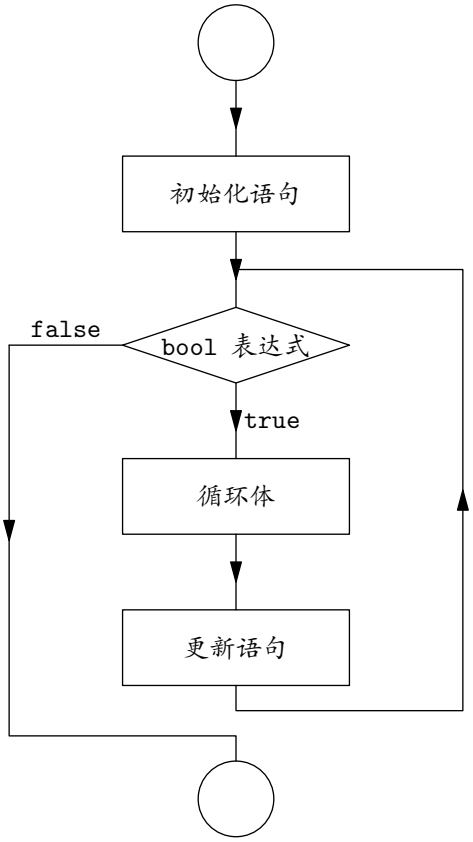


图 4.2: for 循环的流程图.

4.7.2 循环

循环语句用于重复执行一些代码. Asymptote 有 3 种形式的循环语句, 分别为 for 循环、while 循环、和 do–while 循环.

使用循环结构时需要注意:

- 无论哪种循环, 都可以在循环体内随时使用 break; 语句退出循环体外, 并继续执行接下来的语句;
- 无论哪种循环, 都可以在循环体内随时使用 continue; 语句直接跳转到循环体的结尾处进行下一轮循环.

一、 for 循环

for 循环语句的格式为:

```
for ( <初始化语句> ; <bool 表达式> ; <更新语句> ) <循环体>
```

其中 <循环体> 可以是一条语句或者一个语句块. for 循环中, 在 <初始化语句> 中定义的变量跟 <循环体> 语句块中的一样, 也是局部变量. for 循环的流程见图 4.2.

下面的例子利用 for 循环计算 $1+2+\cdots+100$ 的值:

```
1 int sum=0;
```

```
2 for (int i=1; i<=100; ++i) sum += i;
3 write(sum);
```

请你自己分析这个程序的流程.

二、 while 循环

while 循环语句的格式为:

while (<bool 表达式>) <循环体>

它的流程见图 4.3.

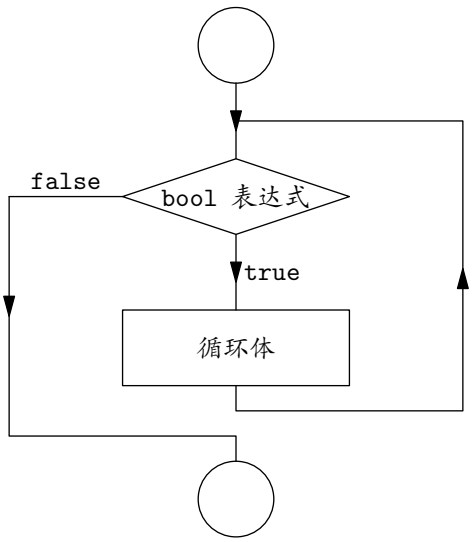


图 4.3: while 循环的流程图.

下面的例子用 while 循环计算 $1 \times 2 \times \cdots \times 9$:

```
1 int factor=1;
2 int i=0;
3 while (i<10) {
4     ++i;
5     factor *= i;
6 }
7 write(factor);
```

> 请不要用这个程序计算 100!, 因为那个数太大了, 远远超过了 intMax. 实际上它等于 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 210 916 864 000 000 000 000 000 000 000 000.

三、 do–while 循环

do–while 循环语句的格式为:

do <循环体> while (<bool 表达式>);

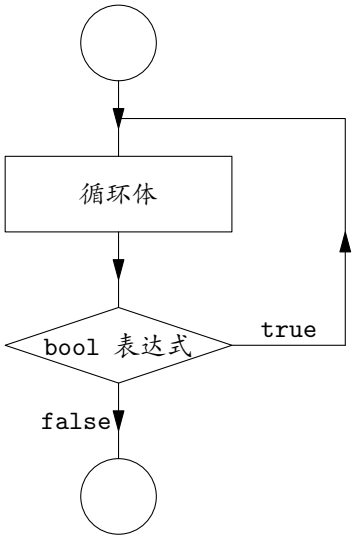


图 4.4: do–while 循环的流程图.

请注意最后的分号. 它的流程见图 4.4.

do–while 循环与 while 循环的区别在于, while 循环先进行判断, 若条件不满足则不执行循环体的语句; 而 do–while 循环则至少执行一遍循环体.

前面计算 $1 \times 2 \times \cdots \times 9$ 的例子也可以写为:

```
1 int factor=1;
2 int i=0;
3 do factor *= ++i; while (i<10);
4 write(factor);
```

> 这里我要说, 这个程序的风格不如前面的好. 当然, 使用 do–while 循环本身没问题. 但是, 那个 factor *= ++i; 就已经很难令人理解了, 又把整个循环写在一行上, 使得这行程序的可读性非常差.

第五章　函数

我们已经多次见识过函数了. 使用函数可以把需要多次重复使用的一段代码集中起来, 这段代码完成一个特定的功能. 这样就无需复制来复制去的, 不但节省人力, 而且使得整个程序干净整洁, 提高了可读性的同时又便于维护. 我们也可以把通用的函数和数据组成模块, 达到代码共享的目的. Asymptote 的基本模块中就提供了很多这样的通用函数来让我们使用. 正因为这样, 在很多作图任务中, 你可能只需要写非常简单的函数, 甚至根本用不着自己定义函数, 因此这些现成的函数大大简化了编程的工作量. 但对函数有足够的了解仍然是有帮助的.

本章将教你怎样写 Asymptote 函数, 以及怎样使用这些函数.

5.1　函数的定义和调用（一）

我们先来看一个简单的例子:

```
1 real f(real x)
2 {
3     return 4x^2-4x-1;
4 }
5 real y=f(2.0);
6 write(y);
```

这段程序的前 4 行定义了一个数学函数 f(x). 但到此为止, 还仅仅是定义, 而没有作任何计算. 不过以后就可以直接使用这个函数来计算 $3x^2+4x-1$ 了. 当程序执行到第 5 行的赋值语句时, 首先调用函数 f(2.0). 或者说, 程序将改变它的流程, 进入函数部分, 这时, f 的参数变量 x 被赋值为 2.0. 函数中只有一条语句, 即第 3 行的 return 语句, 它的意思是, 计算 $3x^2-4x-1$（得到 7）, 然后返回到调用函数的位置（第 5 行）, 并用计算结果 7 替代 f(2.0). 于是, 第 5 行的赋值语句最终将结果 7 赋值给变量 y. 第 6 行会在终端屏幕上输出 7.

5.1.1　函数定义

函数的定义语句包括两个部分, 函数原型 和函数体. 在上面的例子中, real f(real x) 就是函数原型, 而 { return 3x^2+4x-1; } 就是函数体. 一般来说, 函数定义语句的格式为:

<函数原型> <函数体>

一、函数原型

函数原型的基本格式为:

<返回值类型> <函数名> (<参数变量列表>)

每个函数都有一个名字. 函数的命名方法与一般变量相同, 例如上面例子中的函数被命名为 f.

在 <函数名> 前面的 <返回值类型> 指的是, 调用这个函数后, 你将得到一个什么类型的数据. 对于上面的例子, 调用函数 f 将得到一个实数 (real). <返回值类型> 可以是任何基本数据类型或者已经定义好的扩展数据类型. 然而一个函数也不见得一定要返回一个数值, 它也可以

只完成一定的任务但什么数据都不返回. 例如 write 函数, 每次调用它都向屏幕终端或者文件输出一些数据, 但没有任何返回数据, 当然我们也不需要它返回什么. 这时, 这个函数的 <返回值类型> 为 void, 表示没有返回值.

<函数名> 之后是由一对括号包围的 <参数变量列表>. 它指出了在调用这个函数时, 你需要提供什么样的参数给它. 在上面的例子中, 函数 f 只需要一个参数 x, 它的类型是 real. <参数变量列表> 的基本格式为:

```
<参数类型> <变量> [ , <参数类型> <变量> ...]
```

即, 如果有多个参数变量, 则它们之间用逗号分隔, 并且对每个参数变量都要明确指出它的数据类型. 在特殊情况下, 一个函数也有可能根本不需要参数. 这时 <参数变量列表> 应该写为 void, 或者干脆什么都不写（但是包围它的括号不能省略, 因为那是函数的标志）.

二、 函数体

<函数体> 是一个语句块, 它是真正完成函数功能的部分. 你可以把函数体理解为一段相对独立的小程序. 在完成计算和处理后, 你一般需要使用 return 语句从函数返回.

> 需要提醒你的是, 函数体只能是语句块. 哪怕只有一条语句, 也要用花括号包围起来.

如果函数要求有一个返回值, 或者说函数的返回值类型不是 void, 则函数最终一定要通过 return 语句返回一个适当类型的数值, 格式如下:

```
return <表达式> ;
```

其中 <表达式> 的结果的数据类型应该与 <返回值类型> 一致.

如果函数并不返回任何数据, 或者说函数的返回值类型是 void, 则 return 语句的格式为:

```
return;
```

如果这个 return 语句位于整个函数体语句块的最后, 那么也可以省略, 即, 当程序执行完函数体语句块的最后一句之后会自动返回.

下面我们来试着写几个简单的函数.

1. 写一个函数, 来判断一个实数是否大于 0.

我们首先给这个函数起一个名字, 不妨叫它 positive. 显然这个函数仅需要一个 real 参数, 而应该返回一个 bool 型的数据. 如果参数大于 1, 就返回 true, 否则就返回 false. 于是我们的函数可以写成下面的样子:

```
bool positive(real x)
{
    if (x>0.0)
        return true;
    else
        return false;
}
```

当然, 如果你记得我们有一个 ?: 运算符, 你可以利用它简化一下程序:

```
bool positive(real x)
```

```
{
    return x>0.0 ? true : false;
}
```

其实你可以写得更简单一些:

```
bool positive(real x)
{
    return x>0.0;
}
```

2. 写一个函数, 以两点连线为直径画一个圆.

这个问题稍微麻烦一些. 我们为函数命名为 drawcircle. 它应该接受两个 pair 参数, 但不需要返回任何值. 在函数体中我们将利用这两个参数, 找到它们的中点作圆心, 求出半径, 最后完成作圆. 函数如下:

```
void drawcircle(pair a, pair b)
{
    pair o = 0.5(a+b); // 圆心就是 a 和 b 的中点
    real r = abs(o-a); // 半径就是 o 到 a 的距离
    draw(circle(o,r)); //画圆
    return; // 这个语句也可以不写
}
```

根据第三章的经验, 你能写一个过给定三点画圆的函数吗?

5.1.2 函数调用

定义好一个函数之后, 我们就可以使用它了, 这称为函数调用. 函数调用的方法很简单, 格式如下:

```
<函数名> ( <实际参数列表> )
```

其中 <实际参数列表> 是一些用逗号分隔的表达式. 如果函数没有参数, 那么 <实际参数列表> 就是空的, 但 <函数名> 后面的一对括号仍然要写. 本节开头的例子中, 第 5 行调用了函数 f, <实际参数列表> 中只有一个实数 2.0.

调用函数时, Asymptote 会把 <实际参数列表> 中的表达式分别求值, 然后按照顺序把结果一一赋值给函数原型中的参数变量. 因此, 实际参数必须跟函数原型中的参数变量一一对应. 这有两层含义, 首先参数个数应该一样, 其次对应位置的参数应具有相同的数据类型. 例如:

```
void f(bool b, real x)  // 函数原型
{ }                     // 函数体
f(true, 2.3);      // 正确. true 赋值给 b, 2.3 赋值给 x
f(true);           // 错误! 缺少一个 real 数据
f(true, 2.3, 4);   // 错误! 最后的 4 不对应任何参数变量
f(2.3, true);      // 错误! 2.3 不能赋值给 b, true 也不能赋值给 x
```

当然, Asymptote 还有更灵活的函数调用方法, 参见第 5.2 节.

我们知道, 一个函数可以有一个返回值. 在这种情况下, 函数调用本身就是一个表达式, 它的值就是函数的返回值, 因此它可以直接参与到各种运算中. 例如:

```
real square(real x) { return x^2; }  // 定义函数 square
real a=2, b;
b=3square(a);  // 调用函数 square(2), 得到 4, 把 3 × 4 = 12 赋值给 b
```

函数也可能没有返回值, 显然这样的函数调用不能参与任何运算. 你只能在函数调用的后面加上一个分号, 来把它写成一个语句. 通常这样的函数都是用来完成一定的功能. 例如:

```
void drawcircle(pair a, pair b) // 函数定义
{
    pair o = 0.5(a+b);
    real r = abs(o-a);
    draw(circle(o,r));
    return;
}
pair p1=(0,0), p2=(3cm,2cm);
drawcircle(p1,p2); // 调用函数 drawcircle 以 p1--p2 为直径画圆
```

> 在 Asymptote 中, 你可以在任何一个表达式的后面加上一个分号, 来得到一条语句. 这不奇怪, 比如 "a=3" 就是一个表达式 (它有一个值的, 等于 3, 见第 30 页), 而 "a=3;" 就成了一条赋值语句. 其实你也可以写出 "3+5;" 这样的语句, 只不过它没有实际作用, 因为计算完之后 Asymptote 就扔掉了它的结果. 但像 "++i;" 这样的语句就有用了, 它把变量 i 增加了 1.
> 事实上, Asymptote 把 void 也看成是一种基本数据类型, 只不过这种类型的数据 (或者表达式) 没有值. 所以, "drawcircle(p1,p2)" 也是一个表达式, 只不过它没有值而已, 在它后面加上分号就是一条语句了.

5.1.3 函数名重载和屏蔽

我们知道, 两个普通变量可以使用相同的变量名. 但是如果它们出现在同一个语句块中, 则后一个变量会屏蔽前一个同名变量, 被屏蔽的变量暂时不能被直接访问 (见第 4.3.2 节). 这是由于普通变量只用一个名字来表征它的身份. 例如:

```
int a;
real a;
write(a);
```

在这段代码中, 最后一行试图向终端屏幕输出变量 a 的值. 这里你唯一能提供的就是一个变量名, 而无法添加额外的信息来区分它到底应该是 int a 还是 real a. 因此 Asymptote 规定, 在这种情况下a 指的是后定义的那一个.

你同样可以给两个或者多个函数起相同的函数名. 但函数跟普通变量不同, 调用函数的时候, 除了要给出函数名之外, 还必须要给出实际参数列表. 因此区分两个函数不仅看函数名, 还可以看参数. 例如:

```
real f(real x) { return x+1; }
```

```
void f(bool b) { write( b ? "Yes" : "No" ); }
```

在这段代码中, 我们定义了两个函数, 它们都叫 f, 但参数变量列表不同, 前一个 f 接受一个 real 参数, 而后一个函数 f 则接受一个 bool 参数. 这使得它们的区别非常明显. 当你使用 f(true) 的形式调用时, 很明显你指的不是第一个 f, 而是第二个; 相反, 当你写出 f(3.5) 时, 毫无疑问这时你是在调用第一个函数而不是第二个. Asymptote 能够作出这种区分, 它正像我们一样认为上面的两个函数是完全不同的, 因此不会用后面的函数屏蔽前面的函数.

用同一个函数名给两个具有不同参数变量列表的函数命名称为函数名重载. 恰当地使用函数名重载是一个好习惯, 它可以让函数的使用者省掉很多记忆的麻烦. 例如, 我们前面已经写了一个函数 void drawcircle(pair a, pair b), 它会以两个给定点的连线为直径作一个圆. 现在我们又想写一个函数, 它通过三个给定的点作圆. 把这个函数也命名为 drawcircle 显然是合理的, 只不过它的参数变量列表中应该包含三个 pair 参数. 当你调用 drawcircle(p1,p2) 时就用两个点作圆, 而调用 drawcircle(p1,p2,p3) 时就用三个点作圆, Asymptote 不会搞混, 你也不会搞混, 而且好记.

函数名重载的关键在于两个同名函数具有不同的参数变量列表. 这种不同指的是下面的两种情况之一:

1. 两个函数的参数个数不同. 例如上面提到的两个 drawcircle 函数.
2. 参数个数虽然相同, 但对应位置的参数变量类型不同.

如果两个函数有相同的名字, 相同个数的参数变量, 对应位置的参数变量的数据类型也分别相同时, Asymptote 就会让后定义的函数屏蔽 先定义的函数, 就像同名变量的屏蔽一样.

需要注意的是, 函数的返回值类型并不能用来区分两个函数. 同样, 参数变量的变量名也不行. 道理很简单, 在调用函数的时候, 例如 g(3.5, 6.4), 你能认出这个函数 g 会返回一个什么类型的数据吗? 你能认出与 3.5 和 6.4 对应的参数变量叫什么名字吗? 你认不出来, Asymptote 也认不出来. 下面举个例子来说明函数的重载和屏蔽:

```
int f(int x) { return x; }  // 定义第 1 个 f
real f(real x) { return x^2; }  // 第 2 个 f, 重载函数
pair f(int a) { return (a,a); }  // 第 3 个 f, 屏蔽第 1 个 f
write(f(3));  // 调用第 3 个 f, 输出 (3,3)
write(f(2.0));  // 调用第 2 个 f, 输出 4
```

*5.1.4 歧义

如果你不打算写通用模块, 你可以跳过本小节不读.

技术是把双刃剑, 用得好它会带来方便, 用不好则带来的就是麻烦. 使用函数名重载要小心, 因为这样做很容易产生歧义. 歧义的一个来源是数据类型的自动转换 (见第 4.6 节).

我们知道, int 数据会自动转换为 real, real 会自动转换为 pair. 因此当我们把一个 int 数据赋值给一个 real 变量时, Asymptote 不会认为这有什么错. 例如:

```
real x=2;  // 整数 2 首先被自动转换为实数 2.0, 然后再赋值给 x
```

当调用函数时, Asymptote 首先也要做一些赋值的事情, 它要把实际参数赋值给函数的参数变量. 所以, 如果这个参数变量是 real 类型的, 你传给它一个 int 类型的数据 Asymptote 也不会抱怨. 但是, 如果函数被不恰当地重载了, 而又被不恰当地调用了, 就可能会出问题. 例如:

```
real f(int x, real y) { return x+y; } // 第 1 个 f
real f(real x, int y) { return x-y; } // 第 2 个 f
write(f(2,  1.0));  // 正确, 调用第 1 个 f, 输出 3
write(f(2.0, 1));  // 正确, 调用第 2 个 f, 输出 1
write(f(2, 1));  // 错误, Asymptote 不知道该调用哪个 f
```

在上面的代码中, 第 3 行调用了函数 f, Asymptote 能够根据参数类型判断出来调用第 1 个 f. 第 4 行的调用类似. 代码的最后一行中, 用两个 int 调用 f. 根据规则, 这种形式可以调用两个 f 中的任何一个, 而且说不上哪一个更合适一些. 我们说这个调用发生了歧义, Asymptote 会给出错误信息并终止执行.

发生歧义是不好的事情, 但这不应该责怪函数的使用者, 而应该责怪函数的定义者. 定义函数的程序员应该尽可能地保证他所定义的函数不导致歧义. 一个解决办法是不用函数名重载, 把两个函数起成不同的名字. 这是根本的解决办法, 但有时也许不是最好的. 在某些特殊的情况下, 函数的定义者可能认为这两个函数确实应该使用相同的名字, 而用户也应该能够正确区分这两个函数, 那么可以使用修饰符 explicit. 在定义函数的时候, 如果在一个参数变量的数据类型前面有 explicit 修饰符, 那么在调用这个函数的时候, 该参数变量所对应的实际参数必须是严格类型匹配的, Asymptote 将不对它进行自动类型转换. 例如:

```
real f(explicit int x, explicit real y) { return x+y; } // 第 1 个 f
real f(explicit real x, explicit int y) { return x-y; } // 第 2 个 f
write(f(2,  1.0));  // 正确, 调用第 1 个 f, 输出 3
write(f(2.0, 1));  // 正确, 调用第 2 个 f, 输出 1
write(f(2, 1));  // 错误, 没有定义函数 f(int, int)
```

虽然最后一行的函数调用仍然出错, 但这个错误在函数的使用者而不是定义者.

> 即便没有重载函数, 你也可以在定义函数时使用 explicit 以明确指出此处不进行自动类型转换.

5.2 函数的定义和调用（二）

上一节所介绍的只是函数定义和调用的基本方法. 本节将学习一些更加灵活的方法, 在很多时候它们会给你使用函数带来方便.

5.2.1 参数变量的默认值

在定义函数时, 可以为参数指定默认值, 方法是在函数原型的 <参数变量列表> 中, 直接指定它的值. 例如下面的函数:

```
void f(real x=3.0) { write(x); }
```

我们为这个函数的参数变量 x 指定了一个默认值 3.0.

当调用一个包含默认参数值的函数时, 你可以像通常一样调用, 此时那个参数的默认值完

全不起作用; 也可以在 实际参数列表 中省略相应的实际参数, 这时该参数变量自动取其默认值. 例如:

```
void f(real x=3.0) { write(x); }  // 定义函数 f，其参数变量 x 有默认值
f(2.0);  // 通常的调用方式，输出 2
f();  // x 自动取默认值，相当于调用 f(3.0)，输出 3
```

对于有多个参数的函数, 你可以为任意一个或几个参数指定默认值.

5.2.2 命名参数调用

*5.2.3 递归调用

5.3 内建函数

*5.4 函数也是变量

> 本节涉及一些高级内容, 初学编程的读者可以跳过本节不读. 这样做并不会影响你下面的学习.

*5.5 函数中变量的使用

> 本节涉及一些高级内容, 初学编程的读者可以跳过本节不读. 这样做并不会影响你下面的学习.

一、 作用域和生存期

在第 4.3.2 节, 我们讲了变量的作用域. 这里我们有必要针对函数中所使用的变量再来看一看这个问题.

在函数体语句块内部定义的变量是局部变量. 它们只在函数体内才能被直接访问, 因此当函数返回后, 你将不能访问这些变量. 函数定义中的参数变量也是局部变量, 虽然它们形式上位于函数体语句块之外. 这一点跟 `for` 循环中 <初始化语句> 里面定义的循环变量很相似（见第 4.7.2 节）.

一般来说, 当函数返回时, 在函数体语句块内部定义的局部变量将被系统完全抛弃, 先前为它们分配的内存空间也将被系统回收. 换言之, 在函数调用返回之后, 那些变量你就再也找不到了. 当下一次调用这个函数时, 系统又会分配一些新的内存空间给它们. 因此, 我们说这些局部变量的生存期仅限于函数内部.

这种处理方式在大多数情况下正是我们想要的. 但有时我们也希望保留一个变量的值, 以便在下次调用时能够继续参加运算. 例如, 你可能希望知道现在到底是第几次调用这个函数. 这时你就需要一个整型变量, 每次调用时让它的值加 1, 而调用结束后, 这个变量仍然保留住而不被清除.

显然, 为了完成这个任务, 你必须让这个变量的生存期长一些. 一个可行的做法是, 在函数体之外定义这个变量, 例如:

```
int count;
void f()
```

```
{
    ++count;
    write(count);
}
f(); // 第 1 次调用，输出 1
f(); // 第 2 次调用，输出 2
```

这里我们在函数外定义了变量 count, 因此每次调用结束时它并不会被系统回收.

但这个做法有个缺点. 由于 count 的作用域超出了函数范围, 因此你有可能在函数之外改变它. 例如:

```
int count;
void f()
{
    ++count;
    write(count);
}
f(); // 第 1 次调用，输出 1
count=-5; // 可以在函数之外改变它的值
f(); // 第 2 次调用，但却输出 -4
```

对于简单的程序, 这不是问题, 你只要自己注意一下就行了. 但如果你的程序较长, 这会是一个隐患, 而且很难调试. 另外, 这样的程序也显得不够完美. 因为从逻辑上看, 变量 count 是完全为函数 f 而定义的, 因此让它隶属于函数 f 才更合理一些.

产生这个问题的根源在于, 虽然 count 的生存期长了, 但它的作用域也大了. 所以一个理想的解决办法应该是只延长它的生存期, 但仍然把它的作用域限制在函数之内. Asymptote 提供了一个办法解决这个问题. 为了限制作用域, count 仍然必须在函数体内定义; 为了延长生存期, 你只需在定义变量时, 在定义语句的最前面加上修饰符 static, 这种变量称为静态变量. 相应的, 没有 static 修饰符的变量称为动态变量. 函数体内定义的静态变量的生存期跟函数本身的生存期一样大. 这样, 我们的程序可以写为:

```
void f()
{
    static int count;
    ++count;
    write(count);
}
f(); // 第 1 次调用，输出 1
f(); // 第 2 次调用，输出 2
// count=-5; // 不可以在函数之外改变它的值!
```

二、 函数体外定义的变量

如果一个变量是在函数之前定义的, 那么函数体内部也属于它的作用域之内, 因此你也可以在函数中访问它（当然, 要没有局部变量屏蔽它才行）. 在函数体内访问一个外部变量时, 它会取函数调用时该变量的值. 我们看下面的例子:

```
int m=3; // 定义一个外部变量 m
void f() { write(m); } // 定义函数 f，它仅仅输出 m 的值
f(); // 调用 f，输出 3
m=5; //改变 m
f(); // 再次调用 f，输出 5
```

在第 4.3.2 节我讲过:“如果两个变量有相同的变量名, Asymptote 并不会报告错误, 而是将它们视为不同的变量”. 这时我们说后一个变量屏蔽了前一个同名变量. 虽然在一般的情况下, 我们无法直接访问被屏蔽的变量, 但仍然可以“间接地”访问它们. 我们来看下面的例子:

```
int m=0;
void f() { ++m; }  // 这个 m 就是前面定义的 m
void g() { write(m); }  // 这里的 m 也是前面定义的 m

f();  // 调用函数 f 使得 m 变为 1
g();  // 输出 1

int m=-5;  // 这第二个 m 是一个新的变量，它屏蔽了第一个 m
write(m);  // 输出 -5，这是第二个 m 的值

f();  // 但函数 f 改变的仍然是第一个 m 的值，它现在变为 2
g();  // 函数 g 输出的也仍然是第一个 m，输出 2
```

从这个例子可以清楚地看出, 被屏蔽的变量仍然在那里, 并没有被新的同名变量所替代.

Asymptote 语言的这一特点对于模块作者是有用的. 当你写一个通用模块时, 有时免不了定义一些“全局变量”以便在若干个函数之间共享数据. 但你用不着担心模块的用户会由于不小心定义了同名变量而使你的函数功能受到影响.