在二进制的科学计数法中，规范化式的有效数应该大于或等于 1 且小于 10（即十进制的 2）。因此，在二进制的科学计数法中，下面这个数字：

$$101.1101$$

其规范化式应该是：

$$1.011101\times 2^2$$

这个规则暗示了这样一个有趣的现象：在规范化二进制浮点数中，小数点的左边通常只有一个 1，除此之外没有其他数字。

当代大部分计算机和计算机程序在处理浮点数时所遵循的标准是由 IEEE（Institute of Electrical and Electronics Engineers，美国电气和电子工程师协会）于 1985 年制定的，ANSI（American National Standards Institute，美国国家标准局）也认可该标准。ANSI/IEEE Std 754-1985 称作 IEEE 二进制浮点数算术运算标准（IEEE Standard for Binary Floating-Point Arithmetic）——它只有 18 页——相对于其他标准来说是非常简短了，但却奠定了以简便方式编码二进制浮点数的基石。

IEEE 浮点数标准定义了两种基本的格式：以 4 个字节表示的单精度格式和以 8 个字节表示的双精度格式。

让我们首先来了解一下单精度格式。它的 4 个字节可以分为三个部分：1 位的符号位（0 代表正数，1 代表负数），8 位用做指数，最后的 23 位用做有效数。下表给出了单精度格式的三部分的划分方式，其中有效数的最低位在最右边。

s = 1 位符号	e = 8 位指数	f = 23 位有效数

三部分共 32 位，也就是 4 个字节。我们刚才提到过，对于二进制科学计数法的规范化式，其有效数的小数点左边有且仅有一个 1，因此在 IEEE 浮点数标准中，这一位没有分配存储空间。在该标准中，仅存储有效数的 23 位小数部分，尽管存储的只有 23 位，但仍然称其精度为 24 位。我们将在下面的内容里体会 24 位精度的含义。

8 位指数部分的取值范围是 0～255，称为偏移（biased）指数，它的意思是：对于有符号指数，为了确定其实际所代表的值必须从指数中减去一个值——称做偏移量（bias）。对于单精度浮点数，其偏移量为 127。

指数 0 和 255 用于特殊的目的，稍后将简单介绍。如果指数的取值范围是 1～254，那么对于一个特定的数，可以用 s（符号位），e（指数）以及 f（有效数）来描述它：

$$(-1)^s \times 1.f \times 2^{e-127}$$

–1 的 s 次幂是数学上所采用的一种巧妙的方法，它的含义是：如果 $s=0$，则该数是正的（因为任何数的 0 次幂都是 1）；如果 $s=1$，则该数是负的（因为–1 的 1 次幂等于–1）。

表达式的中间部分是 $1.f$，其含义是：1 的后面是小数点，小数点后面跟着 23 位的有效数。$1.f$ 与 2 的幂相乘，其中指数等于内存中的 8 位的偏移指数减去 127。

注意，目前为止我们还没有学习如何表达那个经常遇到却又总被遗忘的一个数字：那就是“0”。这是一种特殊的情况，下面我们对其进行说明。

- 如果 $e=0$ 且 $f=0$，则该数为 0。在这种情况下，通常把 32 位都设置为 0 以表示该数为 0。但是符号位可以设置为 1，这种数可以解释为负 0。负 0 可以用来表示非常小的数，这些数极小以至于不能在单精度格式下用数字和指数来表示，但它们仍然小于 0。
- 如果 $e=0$ 且 $f\neq 0$，则该数是合法的，但不是规范化的。这类数可以表示为：

$$(-1)^s \times 0.f \times 2^{-127}$$

注意，在有效数中，小数点的左边是 0。

- 如果 $e=255$ 且 $f=0$，则该数被解释为无穷大或无穷小，这取决于符号位 s 的值。
- 如果 $e=255$ 且 $f\neq 0$，则该值被解释为“不是一个数”，通常被缩写为 NaN（not a number）。NaN 用来表示未知的数或非法操作的结果。

单精度浮点格式下，可以表示的规格化的最小正、负二进制数是：

$$1.00000000000000000000000_2 \times 2^{-126}$$

小数点后面跟着 23 个二进制 0。单精度浮点格式下，可以表示的规格化的最大正、负二进制数是：

$$1.11111111111111111111111_2 \times 2^{127}$$

在十进制下，这两个数近似地等于 $1.175494351\times10^{-38}$ 和 3.402823466×10^{38}，这也就是

单精度浮点数的有效表示范围。

如前所述，10 位二进制数可以近似地用 3 位十进制数来表示。其含义是，如果把 10 位都置为 1，即十六进制的 3FFh 或十进制的 1023，它近似等于把十进制数的 3 位都置为 9，即 999，可以表示为下面的约等式：

$$2^{10} \approx 10^3$$

两者之间的这种关系意味着：单精度浮点数格式存放的 24 位二进制数大体上与 7 位的十进制数相等。因此，可以说单精度浮点格式提供 24 位的二进制精度或者 7 位的十进制精度。其深层的含义是什么呢？

当我们查看定点数时，其精确度是很明显的。例如，当我们表示钱款时，采用两位定点小数就可以精确到美分。但是对于采用浮点格式的数，就不能如此肯定了。其精确度依赖于指数的值，有时候浮点数可以精确到比美分还小的单位，但有时候其精确度甚至达不到美元。

这样说可能更合适：单精度浮点数的精度为 $1/2^{24}$，或 1/16777216，或百万分之六，但其真正的含义是什么呢？

首先，这意味着在单精度浮点格式下，16,777,216 和 16,777,217 将表示成同一个数。不仅如此，处于这两个数之间的所有的数（例如，16,777,216.5）也将被表示成同一个数。所以上面提到的 3 个十进制数都按 32 位单精度浮点数：

4B800000h

来存放。将该数按符号位、指数位和有效数位划分，可以表示为：

0 10010111 00000000000000000000000

也就是：

$$1.00000000000000000000000_2 \times 2^{24}$$

下一个二进制浮点数可表示的最大有效数是 16,777,218，即：

$$1.00000000000000000000001_2 \times 2^{24}$$

以同一个浮点数来表示两个不同的十进制数，有时可能成为一个问题，也可能不会。

但如果你为银行编写程序，用单精度浮点数来存放以美元、美分为单位的数字时，就会发现 262144.00 美元和 262144.01 美元在计算机中存储为同一个数：

$$1.00000000000000000000000_2 \times 2^{18}$$

这也是为什么人们在处理以美元、美分表示的钱款数目时更愿意使用定点数的一个原因。当使用浮点数时，你会发现它还存在着一些让人崩溃的小问题。你的程序进行了一系列计算，应该得到的结果为 3.50 的，但由于使用浮点数，你得到的可能是 3.499999999999。这种问题在浮点数运算中经常发生，而且没有一套完整的解决方案。

如果想在程序中使用浮点格式数，但使用单精度格式又会出现各种问题，这时你可以考虑使用双精度浮点数（double-precision floating-point format）。这种类型的数需要用 8 个字节来表示，它的结构如下表所示。

s = 1 位符号位	e = 11 位指数位	f = 52 位有效数

双精度浮点数的指数偏移量是 1023，或十六进制的 3FFh，因此以该格式存储的数可以表示为：

$$(-1)^s \times 1.f \times 2^{e-1023}$$

上面提到的关于单精度浮点格式下的 0，无穷大（小）和 NaN 的判断规则同样适用于双精度浮点格式。

双精度浮点格式下可以表示的最小正数或负数为：

$$1.0000000000000000000000000000000000000000000000000000_2 \times 2^{-1022}$$

注意，小数点的后面共有 52 个 0。同样的，可以表示的最大数为：

$$1.1111111111111111111111111111111111111111111111111111_2 \times 2^{1023}$$

其所能表示的范围，用十进制可以近似记为：

$$2.2250738585072014\times10^{-308}\sim1.7976931348623158\times10^{308}$$

10 的 308 次幂是一个非常巨大的数，在 1 的后面跟着 308 个 0。

双精度浮点格式的有效数有 53 位（包括前面没有列出的那一位），大致相当于十进制的 16 位。与单精度浮点格式相比，这已经有了很大的改进了，但仍然不能避免两个不同的数存储为同一个结果的情况。例如，140,737,488,355,328.00 和 140,737,488,355,328.01 在内存中存放时，会被当做同一个数来处理，它们的双精度浮点格式表示为：

42E0000000000000h

即：

$$1.0000000000000000000000000000000000000000000000000000_2 \times 2^{47}$$

当然，为浮点数发明一种在内存中的存储方式，这只是在汇编程序使用浮点数所涉及的工作的一小部分。如果你决定闭门造车，完全独立地开发一台计算机，则必须要独立编写用于浮点数加、减、乘、除运算的函数集。幸运的是，有了前面关于整数四则运算的学习，这些关于浮点数的运算就可以分解成许多小的关于整数的加、减、乘、除运算，这样就能将问题大大简化。

例如，浮点数加法中最重要一点的就是如何对有效数相加，为了能使它们的有效位匹配，需要利用指数来确定对其如何移位。假设要进行下面的加法运算：

$$(1.1101\times 2^5)+(1.0010\times 2^2)$$

你需要把有效数部分的 11101 和 10010 相加，但并不是简单地直接相加。两个数的指数部分的不同决定了第二个数必须相对于第一个数右移。实际上我们要做的整数加法应该是 11101000 加 10010。最后的运算结果是：

$$1.1111010\times 2^5$$

前面我们曾列出过太阳与地球的距离以及氢原子的半径，如果把这两个数相加会是怎样的结果呢？显而易见，因为它们两者的指数相差太大，因此较小的数对结果甚至没有影响。

两个浮点数的乘法意味着要把有效数当作整数相乘，并且把指数部分相加。为了使结果规范化，一般需要对指数调整一到两次。

浮点数运算另一层次的复杂性体现在处理一些较为繁杂的函数运算，例如平方根、

指数、对数和三角函数。但所有的这些运算都可以通过加、减、乘、除这四种基本的浮点数运算来实现。

例如，三角函数中的 sin 函数可以通过下面的一系列展开式近似计算：

$$\sin(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots$$

参数 x 的值必须是弧度，360° 对应的弧度范围是 2π。上式中的感叹号表示阶乘运算符，它的意义是把从 1 到该数之间的所有整数相乘，例如 $5!=1\times2\times3\times4\times5$。这只是简单的乘法运算，每一项的指数部分也是乘法运算。其余的部分也只是简单的除法，加法或减法运算，这些都是容易实现的。上面的算式中，唯一让人感到棘手的地方是最后的省略部分，这意味着计算会一直继续下去。然而事情并没有想象中的那么糟糕，在实际运算中，如果把弧度的取值限制在 $0\sim\pi/2$ 的范围内（从这个范围就可以推导出所有的正弦值），你根本不需要进行多少运算，因为大约展开 12 项后，就可以使结果精确到双精度浮点数要求的 53 位。

当然，使用计算机的目的就是帮助人们更加方便地解决问题，而编写程序来进行浮点数运算这一繁杂工作似乎和这个目的背道而驰。但这正是软件的优势所在：一旦某个人为特定的计算机编写了浮点数运算的程序，那么其他的人都可以使用它。浮点数运算在科学和工程类程序中极为重要，因此常常被赋予很高的优先级。在计算机发展的早期，为新制造的计算机做的第一项工作就是为其编写浮点数运算程序。

实际上，甚至可以直接利用计算机机器码指令来实现浮点数的运算。当然，实际做起来要比“动动嘴皮子”困难得多，但这也从另一个方面说明了浮点数运算的重要性。如果可以在硬件上实现浮点数算术运算——类似在 16 位微处理器上进行乘法和除法运算——则该机器上的所有的浮点数运算都会变得更快。

IBM 公司在 1954 年发布了 IBM 704，它是第一台将浮点数运算硬件作为可选配件的商用计算机，该机器以 36 位空间来存储所有的数。对于浮点数而言，其 36 位被分成 27 位的有效数，8 位的指数和 1 位的符号位。浮点运算硬件可以直接进行加法、减法、乘法和除法运算。其他的浮点运算则必须通过软件来实现。

从 1980 年开始，浮点运算硬件开始应用于桌面计算机，这起始于英特尔当年发布的

8087 数字协同处理（Numeric Data Coprocessor）芯片，当时这种集成电路被称做数学协同处理器（math coprocessor）或浮点运算单元（floating-point unit，FPU）。8087 不能独立工作，它只能与 8086 或 8088（Intel 的首个 16 位微处理器）芯片一起工作，因此被称做协处理器。

8087 拥有 40 个管脚，它使用的很多信号与 8086 或 8088 完全相同。微处理器和数学协处理器通过这些信号相连。当 CPU 读取到一条特殊指令 ESC（Escape）时——协同处理器开始接管控制权并执行下一条机器指令，该指令可以是三角函数、指数和对数等 68 条指令中的任一条。它所处理的数据类型遵循 IEEE 标准。在当时，8087 被认为是市面上最高水平的集成电路。

可以把协处理器当做一个小型的自包含计算机。当响应某个特定的浮点运算机器码指令时（例如 FSQRT 指令，它用来计算平方根），协处理器会以固有的方式执行存放在 ROM 中属于自己的指令序列。这些内部指令称做微代码（microcode）。通常，这些指令都是循环的，因此不能立即得到最终的结果。虽然如此，但数学协处理器在运算速度方面仍然表现优异，与软件方法相比，其速度至少是后者的 10 倍。

在最初版本的 IBM PC 主板上，位于 8080 芯片的右边有 1 个 40 个管脚的插槽供 8087 芯片接入。但令人失望的是，这个插槽是空的，用户如果需要进行浮点数运算就必须单独购置一块 8087 芯片，并将其插入主板后才能使用。安装数学协处理器并不能提高所有应用程序的运行速度，因为有些应用程序——比如文字处理程序——几乎用不到浮点数运算。其他应用程序，比如电子表格处理程序，对浮点数运算依赖程度很高，在安装了数学协处理器之后，它们的执行速度有很大的提高，但并非所有的程序都是如此。

可以看到，在安装了数学协处理器后，程序员必须使用协处理器机器码指令来编写特殊的代码，因为数学协处理器不是标准硬件，因此它只能执行这些特殊的代码。而这些工作让程序员烦不胜烦。尽管他们不愿意，他们仍不得不编写自己的浮点数运算子程序（因为大多数人并没有安装数学协处理器），因此这就多了一个额外的工作——一个并不轻松的工作——在程序中支持 8087 芯片。最后就出现了这样的局面：如果机器上安装了数学协处理器，程序员就要学会编写相应的应用程序以支持它的运行；如果没有安装，程序员就要通过编程来模拟它进行浮点数的运算。

在随后的几年内，英特尔还发布了与 286 配合工作的 287 数学协处理器，与 386 配

合工作的 387 数学协处理器。但是，在 1989 年发布的 486DX 芯片中，FPU 已经内建在 CPU 的结构里，它不再作为一个配件供选择安装了。令人失望的是，在 1991 年发布的一款低端芯片 486SX 中，英特尔没有为该其内建 FPU，而是提供了一块可选的 487SX 数学协处理芯片。但 1993 年发布的奔腾芯片中，CPU 内置 FPU 再次成为标准，也许这是永远的标准。在 1990 年发布的 68040 芯片中，摩托罗拉首次将 FPU 集成在 CPU 中，在此之前，摩托罗拉发布了 68881 和 68882 数学协处理器来支持 68000 家族早期的微处理器。PowerPC 芯片同样使用了内置浮点数运算硬件的技术。

浮点数运算硬件对于困惑的汇编程序员来说无疑是个惊喜的礼物，但相对于 20 世纪 50 年代开始的某些其他工作来言，这只是历史所迈出的一小步。接下来，我们的探索之旅即将到达下一站：计算机语言。

高级语言与低级语言

使用机器码编写程序就如同用牙签吃东西，伸出手臂使出较大的力气刺向食物，但每次都只获取到小小的一块，这个过程是辛苦且漫长的。同样的，每个机器码字节所能完成的工作，是你能想象到的最微小且最简单的工作——从内存获取一个数，之后加载到处理器，再把它与另一个数相加，最终将运算结果保存到内存等——正因如此，很难想象如何使用这些机器码构成一个完整的程序。

目前为止，至少对于在第 22 章讨论的原始模型阶段来说，我们已经取得了一定的进步，在那个阶段，我们使用过控制面板上的开关将二进制数据输入到存储器。在第 22 章中，介绍了如何编写一段简单的程序，让我们可以利用键盘将十六进制机器码输入计算机，以及通过视频显示设备来检查这些代码。这种改进固然可取，但仍不是我们的终极目标。

前面的章节介绍过，可以使用某些较短的助记符来关联机器码字节，这些助记符包括 MOV，ADD，CALL，HLT 等，通过这些类似英语的符号我们可以较方便地引用机器

码。通常这些助记符的后面会跟着操作数，这可以进一步指明它所关联的机器码指令的功能。例如 8080 机器码字节 46h，它的功能是令处理器将存储在内存特定地址的字节转移至寄存器 B，而该地址由寄存器对 HL 中的 16 位数寻址。这个操作可以简单地写做：

```
MOV B,[HL]
```

显然，使用汇编语言编写程序要比使用机器语言简单得多，但微处理器并不能解释汇编语言。在前面的章节中我们已经学习了如何在纸上编写汇编程序，但只有当你确实准备在微处理器上运行汇编程序，才会手工对其汇编，这样就可以将汇编语言程序的语句转换成了机器语言代码，并把它们输入内存。

当然，我们希望最好由计算机能独自完成语言转换的工作。如果你的 8080 计算机正在运行 CP/M 操作系统，而且你已经拥有了所有必需的工具，那就再好不过了，因为下面我们将介绍其工作原理。

第一步，建立一个文本文件，并将汇编语言程序输入到该文本文件中。这项工作可以使用 CP/M 的应用程序 ED.COM 来完成。该程序是一个可以用来创建、修改文本文件的编辑器。假设你把该文本文件命名为 PROGRAM1.ASM，其中 ASM 是文件类型，用来指明该文本文件的内容是由汇编语言程序组成。这个文件的内容如下：

```
        ORG 0100h
        LXI DE, Text
        MVI C,9
        CALL 5
        RET
Text:   DB 'Hello!$'
        END
```

这个文件中有两条语句我们从未接触过。第一条语句是 ORG（origin），它不与任何 8080 指令对应，其功能是用来指明下面语句的地址从 0100h 地址处开始。如前所述，该地址是 CP/M 将程序装入内存的起始地址。

第二条语句是 LXI（Load Extended Immediate）指令，其功能是将一个 16 位数加载到寄存器对 DE。在本例中，该 16 位数是由标记 Text 提供的。该标记在程序底端的附近，位于 DB（Data Byte）语句之前。DB 语句我们也是第一次遇到，其后可以跟着一些字节，这些字节以逗号分隔或者用单引号括起来（如本例）。

MVI（Move Immediate）语句将数值 9 转移到寄存器 C。CALL 5 语句实现 CP/M 的函数调用功能。函数 5 的作用是：显示以寄存器对 DE 给出的地址为起始处的字符串，直到遇到$结束（可以看到，在程序的结尾处使用了美元符号“$”作为文本的结束标志，这种方式看起来很奇怪，但 CP/M 就是采取的这种方式）。最后的 RET 语句用来结束程序，并把控制权交还给 CP/M（实际上，这只是结束 CP/M 程序的方法之一）。END 语句用来指明汇编语言文件已经结束。

现在我们已经有了一个包含 7 行语句的文本文件，下一步要做的就是对其进行汇编，即将其转换成机器语言代码。以前这项工作是通过手工完成的，但现在我们的机器运行的是 CP/M 系统，可以利用 CP/M 中一个叫做 ASM.COM 的模块来完成这项工作。该模块是 CP/M 的汇编器（assembler）。可以在 CP/M 的命令行中使用下面的语句运行 ASM.COM 文件：

```
ASM PROGRAM1.ASM
```

ASM 对 PROGRAM1.ASM 文件进行汇编，产生一个名为 PROGRAM1.COM 的新文件，PROGRAM1.COM 包含了与我们编写的汇编程序相对应的机器码（实际上，该过程还包含另一个步骤，但在该操作中并不重要）。现在就可以使用 CP/M 的命令行来运行 PROGRAM1.COM 文件，程序运行的结果是显示字符串“Hello!”然后结束。

PROGRAM1.COM 文件包含以下 16 个字节：

```
11  09  01  0E  09  CD  05  00  C9  48  65  6C  6C  6F  21  24
```

开始的 3 个字节是 LXI 指令，其后的两个字节是 MVI 指令，接下来的三个字节是 CALL 指令，紧随其后的一个字节是 RET 指令，最后的 7 个字节是 ASCII 码，包括 5 个字母“Hello”，感叹号“!”以及美元符号“$”。

像 ASM.COM 这样的汇编器程序所做的工作是：读取一个汇编语言文件（source-code，通常称做源代码文件），将其转换得到一个包含机器码的文件——可执行文件（executable file）。从宏观的角度来看，汇编器是非常简单的，因为构成汇编语言的助记符和机器码之间是一一对应的。汇编器拥有一张包括所有可能助记符及其参数的表，它逐行读取汇编语言程序，把每一行都分解成为助记符和参数，然后把这些短小的单词和字符与表中的内容匹配。通过这种匹配的过程，每一个语句都会找到与其对应的机器码指令。

注意，汇编器如何知道 LXI 指令必须将寄存器 DE 的值设置为地址 0109h（Text 的地址）。如果 LXI 指令本身被存放在地址 0100h 处（CP/M 将程序加载至内存开始运行时的起始地址），而 0109h 则是 Text 字符串的起始地址。一般来说，程序员在使用汇编器时有很多方便之处，其中一点就是不需要关心汇编程序各部分在内存中的存放地址。

第一个编写汇编器的人需要手工对程序汇编。如果要为机器写一个新的汇编器（或者对其修改），则可以使用汇编语言编写该程序，然后使用原有的汇编器对其汇编。一旦新的汇编器通过了汇编，则它也就可以对自身进行汇编。

每当一种新的微处理器面世，就需要为其编写新的汇编器。然而，新的汇编器可以在已有的计算机上编写，并利用其汇编器进行汇编。这种方式称为交叉汇编（cross-assembler），即利用计算机 A 的汇编器对运行在计算机 B 上的程序汇编。

虽然汇编器的引入消除了汇编语言编程中重复性的劳动部分（即手动汇编部分），但汇编语言仍然存在两个主要问题。第一个问题（也许你已经意识到了），使用汇编语言编程非常乏味，因为这是在微处理器芯片级的编程，因此不得不考虑每一个微小的细节。

汇编语言存在的第二个问题是不可“移植”（portable）。如果你为 Intel 8080 写了一个汇编语言程序，则该程序不能在 Motorola 6800 上运行，你必须在 6800 上重写一个相同功能的汇编语言程序。编写类似程序的过程也许没有编写第一个程序那么困难，因为你应经解决了程序的组织和算法问题，但仍然还有很多工作要做。

上一章介绍了现代微处理器集成浮点运算机器码指令的原理。不可否认，这已经为我们带来了很大的便利，但仍不能令人特别满意。一种更好的方式是：完全放弃那些实现每个基本操作的机器码指令，这些指令与处理器相关，因而导致程序缺乏移植性。我们采用的替代策略是使用一些经典的数学表达式来描述复杂的数学运算。下面是一个表达式的例子：

$$A \times Sin(2 \times \mathrm{PI} + B) / C$$

上式中的 A，B，C 代表数字，而 PI = 3.14159。

这看起来不错，为什么不动手尝试一下呢？假设在某个文本文件中有这样一个表达式，那么我们可以尝试编写一个汇编语言程序来读取该文本文件，并将其中的数学表达

式转换为机器码。

如果只需要计算一次该表达式，那么可以手工计算或借助计算器来完成。如果需要对 *A*、*B*、*C* 取不同的值多次计算该表达式，那么你可能要考虑使用计算机来完成这些计算。因此，代数表达式不会孤立地出现，必须考虑其前后的语句，这些语句使表达式对不同的值进行运算。

现在你所创建的东西已经触及所谓的高级程序设计语言（high-level programming language）。我们一直在介绍的汇编语言称做低级语言（low-level programming language），因为它与计算机硬件的关系相当紧密。尽管除了汇编语言以外的其他程序设计语言都可以称为“高级语言”，但它们之间还是有高低之分的，一些语言通常被认为比别的语言更高级。如果你是一家公司的总裁，坐在计算机前输入这些命令（也可能做得更轻松：口头发布这个命令），“计算出本年度的收益和损耗，生成年度报表，最后打印出 2000 份送至每个股东”，你所使用的才真正是一种非常高级的语言！但在实际工作中，程序设计语言还达不到这种理想化的水平。

人类语言通常都是经历了千百年复杂的互相影响、偶然演变以及不断吐故纳新才形成的，就算一些人工语言如世界语（Esperanto），也处处显露出与现实语言的渊源。但高级程序设计语言是经过深思熟虑的设计的，更加概念化的语言。设计程序设计语言所面临的一大挑战就是：如何让语言更具吸引力。因为语言定义了人们向计算机传送指令的方式，只有更易用的方式才能让人们对语言产生兴趣。据 1993 年的一项估算，从 1950 年到 1993 年大约有 1000 多种高级程序设计语言被发明出来并被应用。

然而，仅仅定义（define）高级语言，包括定义语言的语法（syntax）来表达该语言可以描述的一切事物，还远远不够；我们还需要为其编写一个编译器（compiler），编译器可以将高级语言的程序语句转换成机器码指令。同汇编器类似，编译器也是逐字逐句地读取源文件并将其分解成为短语、符号和数字的，但实现过程要比汇编器更加复杂。从某些方面来看，汇编器相对简单，因为汇编语言的语句和机器码是一一对应的。而一般的高级语言却不具备这种对应关系，编译器通常必须把一条语句转换多个机器码指令。编译器的编写非常复杂，许多书都是用全部的篇幅来讲解如何设计和构造编译器。

当然，任何事物都是具有两面性的，高级语言也不例外，它有很多优势但也存在不少缺陷。高级语言最基本的优点在于它比汇编语言易于学习并且更容易编写程序，用高

级语言编写的程序通常更加清晰简明——与汇编语言不同，高级语言通常不依赖于特定的处理器，因此它们通常具有良好的可移植性。因为这种特点，使用高级语言的程序员不再需要关心最终运行程序的计算机的底层结构。当然，如果要在不同类型的处理器上运行程序，则需要用处理器对应的编译器将程序转换成对应的机器码。因此，最后生成的可执行文件仍然只适用于特定的处理器。

另一方面，有一种普遍现象：一个优秀的汇编程序员所编写的程序比编译器所产生的代码更加有效率。也就是说，从高级语言程序生成的可执行程序比相同功能的汇编语言程序更大，并且运行速度更慢（但从近年的发展来看，这种差别已变得不再明显，因为微处理器变得更加复杂，而且编译器在优化代码方面也更加成熟）。

此外，虽然高级语言提高了处理器的易用性，但并没有让其变得更强大。微处理器的任何一个功能都可以通过汇编语言实现，因此汇编语言可以高度利用处理器的功能。因为高级语言必须转化成机器码，所以它只会降低微处理器的能力。事实上，如果某种高级语言具有真正意义的可移植性，那么它将不能使用某些处理器的特有功能。

例如，许多微处理器都有移位指令。如前所述，这些指令能将累加器中的字节的每一位向左或向右移动。但事实上，几乎没有哪一种高级语言包含这种操作。如果在程序中需要进行移位操作，则必须通过乘 2 或除 2 来模拟该过程（这并不是什么坏事：事实上，许多现代编译器都是利用处理器的移位指令来实现乘以或除以 2 的幂的）。除此之外，许多高级语言也不包括按位逻辑运算。

在早期的家用计算机中，大部分应用程序都是用汇编语言写的，而现在除了一些特殊的应用场合之外，汇编语言已经很少使用了。而今处理器引入了一些新的硬件，可以实现流水线技术——同时有若干个指令码渐次执行——这使得汇编语言变得更加复杂且不易处理。与此同时，编译器却变得更加成熟，越来越多的程序开始使用高级语言来编写。现代计算机大容量的存储器也作为一个重要的角色，推动了这种趋势：程序员不再局限于编写运行在小内存和小磁盘上的程序。

早期的计算机设计者都曾尝试用数学符号来描述问题，但公认的第一个真正可以工作的编译器是 A-0，它是为 UNIVAC 开发的编译器，于 1952 年由雷明顿兰德公司（Remington-Rand）的格瑞斯·穆雷·霍珀（Grace Murray Hopper，1906–1992）开发完成。霍珀博士的早期计算机研究工作始于 1944 年，那时她效力于霍华德·艾肯（Howard

Aiken），主要研究 Mark Ⅰ。在她八十多岁的时候，仍然孜孜不倦地在计算机界工作，当时她在 DEC（Digital Equipment Corporation）公司从事公关事务。

FORTRAN 语言是目前仍在使用的最古老的高级语言（虽然这些年来人们对其进行了大量修改）。你可能注意到了，很多计算机语言都是以大写字母命名的，这是因为它们的名字大都是由几个单词的首字母组成。FORTRAN 这个名字来源于 FORmula 的前三个字母和 TRANslation 的前四个字母的组合，它由 IBM 在 20 世纪 50 年代中期开发，主要应用于 704 系列计算机。自其发布的几十年来，FORTRAN 一直被认为是科学和工程应用程序开发的首选语言。它广泛地支持浮点运算，甚至支持非常复杂的数的运算（即我们上一章讲到的由实数和虚数构成的复数）。

任何一种计算机程序设计语言都有其支持者和批评者，而且人们通常只对自己喜欢的语言有热情。本书尽量以一种客观的态度来讨论某种语言，这里选取了一种语言作为原型，通过它来解释那些几乎已经销声匿迹的程序设计概念。我们的选择是 ALGOL（即 ALGOrithmic 的缩写，有趣的是，ALGOL 也是仙女座第二亮的恒星的名字）。ALGOL 作为过去 40 年中许多曾经流行一时的通用高级语言的直接鼻祖，也非常适合用来研究高级程序设计语言的本质，该语言可看做是一粒种子，它的成长最终形成了高级语言这棵大树。直到今天，人们仍然在使用“类 ALGOL”程序设计语言的概念。

ALGOL 语言的原版由某国际委员会在 1957 至 1958 年间设计，它被称做 ALGOL 58。两年后，也就是在 1960 年，ALGOL 58 的改进版 ALGOL 60 面世，其最终版本是 ALGOL 68。本章所采用的版本在 *Revised Report on the Algorithmic Language ALGOL 60* 说明文档中有具体描述，该文档于 1962 年完成并在 1963 年首次发行。

让我们开始写第一个 ALGOL 程序。假设我们使用的操作系统平台是 CP/M 或 MS-DOS，并且安装了一个名为 ALGOL.COM 的编译器。该程序是一个文本文件，命名为 FIRST.ALG。注意，文件类型名是 ALG。

ALGOL 程序以 begin 开始，以 end 作为结尾，程序的主要内容被包括在这两个语句之间。下面的程序用来显示一行文本：

```
begin
     print ('This is my fist ALGOL program!');
ende
```