China-pub.com
下载

第9章　记　　法

在人的所有造物中，语言或许是最奇妙的东西了。

Giles Lytton Strachey，《词和诗》

采用正确的语言有可能使某个程序的书写变得容易许多。正是由于这种情况，在实际程序员的武器库里都有许多东西，不仅有像C这一类的通用语言，还有可编程的shell、脚本语言以及许多面向特定用途的语言。

好记法的威力不仅表现在传统的程序设计中，也体现在各种特定问题领域。正则表达式使我们能以非常紧凑的形式(或许还有点像密码)定义一个字符串类；HTML使我们能定义交互式文档的编排格式，其中还常常嵌入其他语言，如JavaScript程序；PostScript能把整个文档——例如这本书——表示为一个格式程序。电子表格和文字处理器也常常包含某种程序语言，例如Visual Basic，用以计算其中的表达式，访问有关信息，或者做格式编排的控制等。

如果你发现自己为某些平庸的事情写了太多代码，或者你需要表述某些过程却遇到了很大麻烦，那么你正在使用的很可能是一种不适当的语言。如果合适的语言不存在，那么这很可能就是个机会，需要你自己来建立一种。发明语言并不意味着是建立某种像Java那样复杂的东西，许多棘手的问题常常可以通过改变描述方式的办法来解决。请考虑一下`printf`一类函数的格式描述串，那就可以看作是一种控制数据打印方式的、紧凑的、描述能力很强的语言。

在本章中我们要讨论怎样通过记法去解决问题。这里还要展示一些技术，这些技术可以用于实现你自己的专用语言。我们还要探索用程序来写其他程序的可能性，这是记法使用的一种极端形式，也是很常见的。实际上，这种技术并不难使用，远不像许多程序员所认为的那样。

9.1　数据格式

在我们最希望对计算机说的东西(“请解决我的问题”)与为了使一个工作能够完成而必须说的东西之间，永远存在着一条鸿沟。能把这条鸿沟填得越窄，当然就越好。好的记法使我们能更容易说出自己想说的东西，又不太容易因为不当心而说出错误的东西。确实也有这样的情况，好的记法能给人提供新的见识，帮助我们解决看起来非常困难的问题，甚至引导我们得出新的发现。

小语言是指那些针对较窄的领域而使用的特定记法，它们不仅提供了某种好界面，还能够帮助人组织实现它们的程序。`printf`的格式控制序列是一个很好的例子：

```
printf("%d %6.2f %-10.10s\n", i, f, s);
```

格式串里的每个%标记着一个位置，要求在这里插入`printf`下一个参数的值。在一些可省缺的标志或域宽说明之后，最后一个字符指明了所要求的参数类型。这种记法非常紧凑，既直观又容易书写，实现起来也很方便。作为其替代物的C++的`iostream`和Java的

China-pub.com 下载

java.io看起来更笨拙，究其原因，虽然它们扩充到了用户定义类型，提供了类型检查，但却没能提供一种特殊的记法。

也有些非标准的printf实现，它们允许人们在内部功能之外增加自己的方式。如果你使用其他数据类型，经常要做它们的输出转换，有这种功能就非常方便。例如，一个编译程序可能想用%L输出行号和文件名；一个图形系统可能用 %P表示点，而用%R表示矩形。在第4章里我们看到为提取股票价格行情而设的字母数字的神秘序列，采用的也是类似想法，是为编排股票数据的组合而提供一种紧凑记法。

现在我们在C和C++里做些类似的例子。假设我们需要从一个系统向另一个系统传送一种包含各种数据类型的组合数据包。在第 8章我们已经了解到，最清晰的一种解法可能就是把数据包转换为正文表示。当然，标准网络规程所使用的多半是二进制格式，为的是效率或者数据规模。现在的问题是：应该如何去写处理数据包的代码，使它能够可移植、高效率，而且又很容易使用？

为使这个讨论更实在些，设想我们在系统间传递的是包含 8位、16位和32位数据项的包。ANSI C告诉我们，8位数据总可以存储在char里，16位可以放进short，而32位放进long，因此我们就用这些数据类型表示有关的值。实际中可能存在许多不同种类的包。例如，在第一种包里有一个字节的类型描述，2字节的计数值，1字节的值，以及一个4字节的数据项：

0x01	cnt_1	cnt_0	val	$data_3$	$data_2$	$data_1$	$data_0$

第二种包类型包含有一个短的和两个长的数据字：

0x02	cnt_1	cnt_0	$dw1_3$	$dw1_2$	$dw1_1$	$dw1_0$	$dw2_3$	$dw2_2$	$dw2_1$	$dw2_0$

可采用的一种方式就是为每种类型的包写一对打包和开包函数：

```
int pack_type1(unsigned char *buf, unsigned short count,
        unsigned char val, unsigned long data)
{
    unsigned char *bp;

    bp = buf;
    *bp++ = 0x01;
    *bp++ = count >> 8;
    *bp++ = count;
    *bp++ = val;
    *bp++ = data >> 24;
    *bp++ = data >> 16;
    *bp++ = data >> 8;
    *bp++ = data;
    return bp - buf;
}
```

对于实际规程，我们将需要成打的这种例程，它们都是某个东西的一种变形。利用处理基本类型(如short、long等)的宏或者函数可以简化这些例程。但是，即使是按这种方式去做，这么多重复性代码也是很容易写错的。它们既难阅读，也难进行维护。

这些代码的内在重复性实际上提醒了我们，记法可能会有所帮助。我们可以从 printf借用有关的想法，定义一个小语言，使每种包在这里都可以用一个简洁的串描述，用这个串刻画数据包的编排形式。包中连续的各个元素用编码表示，用 c表示8位字符，s表示16位短整

China-pub.com
下载

数，l表示32位的长整数。例如，上面提出的第一个包（包括它的类型描述）就可以用格式串cscl表示。这样，我们只需要写一个打包函数就可以应付所有不同类型的数据包了。要建立上面的包，需要写的不过是：

```
pack(buf, "cscl", 0x01, count, val, data);
```

这里的格式串只包含数据定义，因此不必像printf那样使用%一类的字符。

在实践中，数据包开始处的信息可以告诉接受方怎样对其他部分进行解码。我们将假定数据包的第一个字节能用于确定有关的编排格式。发送方按这种格式对数据编码并发送出来，接受方读这种包，提取其第一个字节，并依据它对包的其余部分解码。

下面是pack的一个实现，它按照格式串确定的方式，将参数的编码表示填入buf中。我们把所有数据都看成无符号的，包括位于包缓冲区里的字节数据，这样可以避免符号扩展问题。在这里还用了一些typedef，以便使声明更简短些：

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
```

就像sprintf、strcpy及其他类似函数一样，pack假定它所用的缓冲区对于要存放的结果而言是足够大的，调用它的程序必须保证这个条件。这里也不企图去检查格式与参数表之间不匹配的情况。

```
#include <stdarg.h>

/* pack: pack binary items into buf, return length */
int pack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp;
    ushort s;
    ulong l;

    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c':   /* char */
            *bp++ = va_arg(args, int);
            break;
        case 's':   /* short */
            s = va_arg(args, int);
            *bp++ = s >> 8;
            *bp++ = s;
            break;
        case 'l':   /* long */
            l = va_arg(args, ulong);
            *bp++ = l >> 24;
            *bp++ = l >> 16;
            *bp++ = l >> 8;
            *bp++ = l;
            break;
        default:    /* illegal type character */
            va_end(args);
            return -1;
        }
    }
```

China-pub.com 下载

```
    va_end(args);
    return bp - buf;
}
```

这个pack函数使用了stdarg.h头文件的功能，比第4章的eprintf用得更多。利用va_arg顺序地提取各个参数：va_arg的第一个参数是va_list类型的变量，必须调用va_start对它预先做设置；va_arg的第二个参数是函数参数的类型(这也是为什么va_arg是宏，而不是函数的根本原因)。在处理完成后，应该调用va_end。虽然‘c’和‘s’对应的参数分别是char和short值，这里必须用int方式提取它们，原因在于char和short是用函数参数表最后的...表示的，在这种情况下，C语言将自动把它们提升到int。

现在每个打包函数都只有一行了，不过是把它的参数排列在一个pack调用里：

```
/* pack_type1: pack format 1 packet */
int pack_type1(uchar *buf, ushort count, uchar val, ulong data)
{
    return pack(buf, "cscl", 0x01, count, val, data);
}
```

开启数据包的工作也可以同样处理：不是为敲开每种数据包格式写一段单独代码，而是写一个带有格式描述的unpack。这种做法把所有数据变换都集中到了一个地方：

```
/* unpack: unpack packed items from buf, return length */
int unpack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp, *pc;
    ushort *ps;
    ulong *pl;

    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c':   /* char */
            pc = va_arg(args, uchar*);
            *pc = *bp++;
            break;
        case 's':   /* short */
            ps = va_arg(args, ushort*);
            *ps  = *bp++ << 8;
            *ps |= *bp++;
            break;
        case 'l':   /* long */
            pl = va_arg(args, ulong*);
            *pl  = *bp++ << 24;
            *pl |= *bp++ << 16;
            *pl |= *bp++ << 8;
            *pl |= *bp++;
            break;
        default:    /* illegal type character */
            va_end(args);
            return -1;
        }
    }
    va_end(args);
    return bp - buf;
}
```

China-pub.com
下载

像scanf一样，unpack也必须给它的调用者返回多个值，因此它的参数是那些指向准备存储结果的变量的指针。函数返回数据包的字节数，可以用于错误检查。

由于所有值都是无符号的，而且我们坚持不超出 ANSI C对各种数据类型定义的大小，上面这些代码总能以可移植的方式传递数据，甚至在那些具有不同大小的 short和long的机器之间。例如，只要使用 pack的程序不试图把32位无法表示的值当作 long传出去，传递的值就一定能正确接收。如果它这样做了，那么实际传送的将是数据的低 32位。如果真的需要传送更大的值，我们也很容易增加其他格式定义。

通过调用unpack，特定类型开包函数的定义都变得非常简单：

```
/* unpack_type2: unpack and process type 2 packet */
int unpack_type2(int n, uchar *buf)
{
    uchar c;
    ushort count;
    ulong dw1, dw2;

    if (unpack(buf, "csll", &c, &count, &dw1, &dw2) != n)
        return -1;
    assert(c == 0x02);
    return process_type2(count, dw1, dw2);
}
```

要调用unpack_type2，必须先确认遇到的正是一个类型 2的数据包，这意味着在接收程序里应该有一个下面形式的循环：

```
while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
    switch (buf[0]) {
    default:
        eprintf("bad packet type 0x%x", buf[0]);
        break;
    case 1:
        unpack_type1(n, buf);
        break;
    case 2:
        unpack_type2(n, buf);
        break;
    ...
    }
}
```

以这种方式做程序设计也可能拖得很长。实际上有一种紧凑写法，那就是定义一个函数指针的表，其中各个项是所有的开包函数，以数据包的类型编号作为下标：

```
int (*unpackfn[])(int, uchar *) = {
    unpack_type0,
    unpack_type1,
    unpack_type2,
};
```

表中的每个函数处理一种数据包，检查结果，并启动对包的下一步处理过程。有了这个表之后，接收程序的工作就非常简单了：

```
/* receive: read packets from network, process them */
void receive(int network)
{
    uchar type, buf[BUFSIZ];
    int n;
```

China-pub.com 下载

```
        while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
            type = buf[0];
            if (type >= NELEMS(unpackfn))
                eprintf("bad packet type 0x%x", type);
            if ((*unpackfn[type])(n, buf) < 0)
                eprintf("protocol error, type %x length %d",
                    type, n);
        }
    }
```

这样，每种数据包的处理代码都非常紧凑，而且只写在一个地方，很容易维护。这也使接收程序基本上独立于传输规程，同时又是清晰和快速的。

上面的例子来源于为实现某个产品网络规程而写的实际代码。当作者认识到这种方式能工作之后，数千行重复的、充满错误的代码被缩减成几百行非常容易维护的代码。记法消解混乱的力量实在太大了。

练习9-1 修改pack和unpack，使它能正确传递带符号的值，甚至在具有不同大小的short和long的机器之间。你应该如何修改格式串以描述带符号数据？你将怎样去测试这些代码，例如，怎样确定程序能正确地把-1从一台具有32位long数据的机器传送给另一台具有64位long的机器？

练习9-2 扩展pack和unpack，使它们能处理字符串。一个可能的方式是在格式串里包含有关字符串长度的描述。扩充函数，使它们能利用计数值处理重复的数据项。这与字符串的格式编码方式又会有什么相互影响？

练习9-3 上面C程序里的函数指针表是 C++ 虚函数机制的核心。在 C++ 里重写pack、unpack和receive，利用C++ 在这方面记法上的优点。

练习9-4 写一个printf的命令行版本，它按第一个参数指明的格式，打印其第二个及后面的参数。有些操作系统外壳已经提供了这种内部功能。

练习9-5 写一个程序，它应能实现电子表格程序或者 Java的DecimalFormat格式规范，能按照模式显示数值。模式里指明必须的或可选的数字、小数点和逗号的位置等等。作为说明，下面的格式：

```
##,##0.00
```

描述的是有两个小数位的数，小数点左边至少有一个数字，在千位数字后面有一逗号，这里还要求用空格填充，直到万位数字的位置。这样， 12345.67将被表示为12,345.67，而 .4将表示为_ _ _ _0.40(这里的下划线表示的是实际空格)。完全的规范请参看DecimalFormat的定义，或者某种电子表格程序。

9.2 正则表达式

pack和unpack的格式描述串是非常简单的记法，它们可用于定义数据包的编排形式。我们的下一论题是一种稍微复杂一点，但是更具表达能力的记法，正则表达式，它能够描述正文的各种模式。我们已经在本书中许多地方零星地用过正则表达式，但还没有给它准确的定义。正则表达式是我们很熟悉的东西，不需要多少解释也能理解。虽然正则表达式在 Unix 程序设计环境里随处可见，但在其他的系统里使用得却没有这么广泛，因此，在这一节里，我们想显示正则表达式的某些威力。可能你手头没有正则表达式函数库，我们也要在这里给

China-pub.com
下载

出一个初步的实现。

正则表达式有多种不同的风格，但它们在本质上都是一样的，是一种描述文字字符模式的方法，其中包括重复出现、不同选择以及为某些字符类 (如数字、字母)而提供的缩写形式等等。人们最熟悉的一个例子就是所谓“ 通配符 ”，它用在命令处理器或者外壳中，用于描述被匹配文件名的模式。典型的使用方式是令 *表示“ 字符的任意序列 ”，这样，下面的命令：

```
C:\> del *.exe
```

用的是一个模式，该模式将与一些文件相匹配，只要文件名是以“ .exe ”结尾的某个字符串。对于这种模式的作用，不同系统之间也可能有细微差别，程序与程序间也可能这样，这也不足为奇。

对于正则表达式，不同程序的处理确实存在差别，这可能使人认为它不过是某种信手拈来的东西。实际上，正则表达式也是一种语言，在语言中所有能说的东西都有严格的意义。进一步说，正则表达式的正确实现的运行速度很快。理论和工程实践的结合方式不同有可能造成很大的差异，第2章里提过这方面的例子，讲过特殊算法的作用。

一个正则表达式本身也是一个字符序列，它定义了一集能与之匹配的字符串。大部分字符只是简单地与相同字符匹配，例如正则表达式 abc将匹配同样的字符序列，无论它出现在什么地方。在这里还有几个**元字符**(metacharacter)，它们分别表示重复、成组或者位置。在 Unix 通行的正则表达式中，字符 ^表示字符串开始， $表示字符串结束。这样， ^x只能与位于字符串开始处的x匹配， x$只能匹配结尾的 x， ^x$只能匹配单个字符的串里的 x，而^$只能匹配空串。

字符“ . ”能与任意字符匹配。所以，模式 x.y能匹配xay、x2y等等，但它不能匹配 xy或xaby。显然 ^.$能够与任何单个字符的串匹配。

写在方括号[]里的一组字符能与这组字符中的任一个相匹配。这样 [0123456789] 能与任何数字匹配。这个模式也可以简写为 [0-9]。

这些就是基本构件，对它们可以用括号结成组，用 |表示两个里面选一个，用*表示零次或者多次重复出现， +表示一次或多次出现，而?表示零或一次出现。最后， \作为一种前缀，用于引出一个元字符，关闭它的特殊意义。例如*表示的就是*本身，而\\表示的就是字符\。

最有名的正则表达式工具就是前面已经多次提到过的 grep程序。这个程序是一个极好的例子，它确实显示出记法的价值。 grep将一个正则表达式作用于输入的每一行，打印出所有包含匹配字符串的行。这是一个非常简单的规范，但是，借助于正则表达式的威力，它能够完成许多我们天天都能遇到的工作。在下面例子里，请注意作为 grep参数的那些正则表达式，其语法形式与用于表示文件名集合的通配符差别很大，这种差异正反映出它们的不同用途。

哪些文件里用到类Regexp？

```
% grep Regexp *.java
```

这个类的实现在哪里？

```
% grep 'class.*Regexp' *.java
```

我在哪里保存着Bob发来的电子邮件？

```
% grep '^From:.* bob@' mail/*
```

在这个程序里有多少空行？

```
% grep '.' *.c++ | wc
```

China-pub.com 下载

加上各种标志开关，如打印匹配行的行号、对匹配计数、做区分大小写的匹配、在相反的意义下工作(选择那些不匹配的行)以及这些基本想法的许多其他变形，grep被使用得如此广泛，以至它已经成为基于工具的程序设计的典型实例。

不幸的是，并不是每个环境里都提供了grep或者它的等价物。有的系统提供了一个正则表达式库，通常称为regex或regexp，你可以用它写一个自己的grep版本。如果这些都没有，要实现正则表达式的一个适当子集也不是件难事。下面我们要给出正则表达式的一个实现，同时给出一个grep。为了简单起见，这里采用的元字符只包括 ^ $.和*，用*表示位于它前面的单个圆点或一个字符的重复出现。这个子集已经提供了一般正则表达式的大部分威力，但程序的复杂性却小得多。

让我们从匹配函数本身入手。这个函数的工作就是确定一个正文串的什么地方与一个正则表达式匹配：

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

如果正则表达式的开头是^，那么正文必须从起始处与表达式的其余部分匹配。否则，我们就沿着串走下去，用matchhere看正文是否能在某个位置上匹配。一旦发现了匹配，工作就完成了。注意这里do-while的使用，有些表达式能与空字符串匹配(例如：$ 能够在字符串的末尾与空字符串匹配，.*能匹配任意个数的字符，包括0个)。所以，即使遇到了空字符串，我们也还需要调用matchhere。

递归函数matchhere完成大部分匹配工作：

```
/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

如果正则表达式为空，就说明我们已经到达了它的末端，因此已经发现了一个匹配；如果表达式的最后是$，匹配成功的条件是正文也到达了末尾；如果表达式以一个圆点开始，那么它能与任何字符匹配。否则表达式一定是以某个普通字符开头，它只能与同样的字符匹配。这里把出现在正则表达式中间的^或$都当作普通字符看待，不再作为元字符。

注意，当matchhere完成模式与字符串中一个字符的匹配之后，就会递归地调用自己。

China-pub.com
下载

所以，函数递归的深度将与模式的长度相当。

在这里，惟一不容易处理的情况就是在表达式开始处遇到了带星号字符，例如 x*。这时我们调用matchstar，其第一个参数是星号的参数 (在上面的例子里是x)，随后的参数是位于星号之后的模式，以及对应的正文串。

```
/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

在这里又遇到了do-while。之所以这样写，原因同样是x*也能与0个字符匹配。在这个循环里，检查正文能否与表达式的剩余部分匹配，只要被检查正文的第一个字符与星号的参数匹配，程序就会继续检查随后的位置。

这是一段完全可以接受的、并不太复杂的实现程序。它也说明正则表达式不需要高级的技术就可以投入使用。

我们不久将会展示一些扩充这个代码的想法。现在，让我们先写一个 grep的版本，它调用match函数。这里是主函数：

```
/* grep main: search for regexp in files */
int main(int argc, char *argv[])
{
    int i, nmatch;
    FILE *f;

    setprogname("grep");
    if (argc < 2)
        eprintf("usage: grep regexp [file ...]");
    nmatch = 0;
    if (argc == 2) {
        if (grep(argv[1], stdin, NULL))
            nmatch++;
    } else {
        for (i = 2; i < argc; i++) {
            f = fopen(argv[i], "r");
            if (f == NULL) {
                weprintf("can't open %s:", argv[i]);
                continue;
            }
            if (grep(argv[1], f, argc>3 ? argv[i] : NULL) > 0)
                nmatch++;
            fclose(f);
        }
    }
    return nmatch == 0;
}
```

令C程序在成功结束时返回一个0值，对各种失败都返回非0值，这也是一种规矩。在我们的程序grep里，就像一般的Unix版本一样，成功被定义为至少发现了一个匹配行，如果存在匹配就返回0值；找不到匹配时返回值为1；如果出现错误，程序返回2(通过eprintf)。这些返回

China-pub.com 下载

值可以在其他程序里检查，例如在操作系统外壳里。

函数grep扫描一个文件，对其中的每个行调用 match：

```
/* grep: search for regexp in file */
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

主程序在无法打开文件时并不直接结束。这个设计是一种选择，考虑到人们经常会做的某些事，如：

```
% grep herpolhode *.*
```

既而发现目录下的某个文件无法打开。让grep在报告了问题后继续工作下去可能是更合适的，这里不应该立即放弃，因为那样做就会迫使用户一个个地输入文件名，以回避那些出了问题的文件。另外还请注意，grep在一般情况下打印出文件名和匹配的行，但当它读标准输入文件时就不输出文件名。这种设计看起来好像很奇怪，其实它反映的是实际使用中的一种习惯性方式。在只提供一个文件名时，grep的工作常常是做选择，附加上的文件名就将成为赘物。如果用它检索许多文件，人们要做的通常是发现某些东西的所有出现，这时文件名就成为很有帮助的东西了。请比较：

```
% strings markov.exe | grep 'DOS mode'
```

和

```
% grep grammer chapter*.txt
```

这些想法已经接触到grep为什么变得如此大众化的基础，它也说明了另一个问题：好的记法只有与人们的工程经验相结合，才能产生出自然而又有效的工具。

我们的match在发现了一种匹配后就立即返回。对于grep而言，这是一种很好的默认行为方式。但是如果要实现的是文本编辑器中的那种代换操作 (搜索和替换)，实现**最左最长**的匹配就更是合适的。例如，给定的文本是“ aaaaa”，模式a*能与文本开始的空字符串匹配，但是，看起来最好还是让它匹配所有的五个 a。要使match能找到最左最长的匹配串，matchstar必须重新写成最贪婪的：它不应该从左向右一个个地查看字符，而应该跳过最长的能与带星号字符匹配的串，只有在串的剩余部分不能与模式的剩余部分匹配时才往后退。换句话说，它应该从右向左工作。下面是实现最左最长匹配的 matchstar版本：

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
```

China-pub.com
下载


```
{
    char *t;
    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do {    /* * matches zero or more */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

对于grep而言，发现的匹配是什么并不重要，因为它只需要检查匹配的存在性，并打印出相应的整个行。由于最左最长匹配需要做许多附加工作，这对 grep而言并不是必须的，但是对于实现一个替换操作而言，这种功能就是最基本的了。

如果不考虑正则表达式的形式，我们的 grep还是能和系统提供的东西相比美的。不过，这里也存在病态表达式，它们可能导致指数式的行为。例如在用模式 a*a*a*a*a*b去匹配输入串aaaaaaaaac的时候。实际上，指数式行为也出现在某些商品的实现中。在 Unix里有一个grep的变体，名字是egrep，它使用一种非常复杂的匹配算法，在部分匹配失败后它能够避免进行回溯，这样就能保证线性的性能。

怎样使match能处理所有的正则表达式？为此就必须包括：像 [a-zA-Z]那样的字符组与各个字母的匹配，能够把元字符引起来(例如需要查找正文中的一个圆点)，形成分组的括号，以及选择(abc或def)等。要做的第一步，应该是把模式翻译到某种表示形式，使它很容易查看。在与一个字符做比较时，如果每次都要检查整个字符组，是非常费时间的，用一个基于位向量的预先做出的表示形式，就能很有效地实现字符组。对于完全的正则表达式，带有括号和选择，实现起来必定更加复杂。在本章后面部分要讲到的某些技术可以用在这里。

练习9-6 如果做普通正文的查找，match的执行性能与strstr比较起来怎么样？

练习9-7 写一个非递归的matchhere版本，将它与递归的版本做性能比较。

练习9-8 给grep增加几个选项。常见的包括：-v表示颠倒匹配的含义，-i表示对字母做区分大小写的匹配，-n表示在输出中指明行号。行号应该如何输出？它们是否应该与被匹配的正文输出在同一行里？

练习9-9 扩充match，增加 +(一个或多个)和?(0个或一个)。模式a+bb?应该匹配一个或多个a后跟一个或两个b。

练习9-10 当^和$不出现在表达式的开头或结尾，或者 *不是紧跟在普通文字字符或圆点的后面时，目前的match实现都不使用它们作为元字符的特殊意义。一种更习惯的实现是在元字符前放一个反斜线符号，表示将它引起来。修改 match，使之能以这种方式处理反斜线符号。

练习9-11 给match增加字符组功能。字符组表示的是一个模式，它能与方括号里任一个字符匹配。要使字符组使用更方便，还应该再加上范围描述，例如 [a-z]能匹配所有的小写字母；翻转匹配的含义，例如[^0-9]能匹配所有不是数字的字符。

练习9-12 改造match，使用最左最长匹配的matchstar版本；再修改它，使它能返回被匹配字符串的开始和结束位置。用这个 match写一个程序gres，它与grep类似，但打印的是用一个新字符串替代了被匹配串之后的字符行。

China-pub.com 下载

```
% gres 'homoiousian' 'homoousian' mission.stmt
```

练习9-13 修改match和grep，使它们能处理Unicode字符的UTF-8串。由于UTF-8和Unicode都是ASCII的超集，这种修改是向上兼容的。除了被检索的正文，正则表达式本身也需要改造，使之能正确使用UTF-8。现在字符组又该如何实现呢？

练习9-14 写一个正则表达式的自动测试程序，它应能生成测试表达式和被检索的测试串。如果你能用某个现成的库作为实现时的参考，或许还能发现其中的程序错误。

9.3 可编程工具

许多工具都是围绕着一个专用语言构造起来的。grep程序不过是常见工具集合中的一个，这些工具使用正则表达式或者其他什么语言来解决程序设计中的问题。

这方面最早的例子是命令解释器或作业控制语言。人们很早就认识到，应该能把常用的命令序列写在一个文件里，这种文件可以由命令解释器的一个实例，或者说是一个外壳作为输入来执行。从这里出发，向前迈出一小步，就是加上参数、条件、循环、变量以及其他东西，所有这些都是从常规程序设计语言里搬来的。最主要的差别是在这里只有一种数据类型，即字符串，而外壳程序的基本运算通常都是完整的程序，它们能完成具有实质意义的计算工作。虽然外壳程序设计的辉煌时代已经过去了，它的领地也丢给了其他替代品，比如命令环境中的Perl语言，图形用户界面上的按钮。但是，它仍然是从简单片段构造出复杂操作的一种非常有效的手段。

Awk是另一种可编程工具，它带有一种很小的、特定的模式匹配语言，主要用于对输入流进行选择和变换。正如我们在第3章已经看到的，Awk程序能自动读入输入文件，把每个行分解为一些域，分别称作 $1到 $NF，这里的NF是一行中域的个数。通过对许多常见工作提供相关的默认行为方式，使人可以用 Awk做出许多很有用的单行程序。例如，下面就是一个完整的Awk程序，

```
# split.awk: split input into one word per line
{ for (i = 1; i <= NF; i++) print $i }
```

它打印各个输入行里的词，一个词一行。再看看另一方向的东西。下面是 fmt的一个实现，该程序用词填起每个输出行，每行不超过60个字符。空行将导致分段。

```
# fmt.awk: format into 60-character lines

/./  { for (i = 1; i <= NF; i++) addword($i) } # nonblank line
/^$/ { printline(); print "" }                 # blank line
END  { printline() }

function addword(w) {
    if (length(line) + 1 + length(w) > 60)
        printline()
    if (length(line) == 0)
        line = w
    else
        line = line " " w
}

function printline() {
    if (length(line) > 0) {
        print line
        line = ""
    }
}
```

China-pub.com
下载

我们常用fmt对电子邮件或其他短文件重新做分段整理。我们也用它对第 3章Markov程序的输出做格式化。

可编程工具常常起源于一个小语言，该语言可能是为在某个较窄领域里自然地表达问题的解而设计的。Unix工具eqn是个很好的例子，它完成数学表达式的显示排版。 eqn的输入语言接近数学家读数学公式的表达方式： $\frac{\pi}{2}$ 写为pi over 2。TEX采用了类似形式，它对上面公式的记法是\pi\over 2。对于你要去解决的问题，如果存在一种自然的或人们熟悉的记法，那么就应该使用它，或者是修改它，不要总是从头开始。

Awk是由另一个程序获得的灵感，该程序利用正则表达式在电话流通记录中标记出反常的数据。当然，由于 Awk包含了变量、表达式和循环等等，这使它成了一个真正的程序设计语言。 Perl和Tcl在开始设计时，也是为了将小语言的方便性和表达能力与大语言的威力结合到一起。它们都是真正的通用程序设计语言，虽然最常见的应用是处理正文。

这类工具的通用名称是**脚本语言**，因为它们是从早期的命令解释器发展起来的，其可编程能力受到一定限制，只能执行包装起来的程序脚本。脚本语言创造性地使用正则表达式，不仅仅是做模式匹配 —— 即识别某种特定模式的存在 —— 也用于标识需要做变换的正文区域。在下面的Tcl程序里，两个regsub(regular expression substitution，正则表达式代换)做的就是这种事情。这个程序是我们在第 4章给出的股票行情提取程序的一个扩充，它按给定的第一个参数值取得URL。其中的第一个替换用于去掉 http://(如果存在的话)；第二个替换把遇到的第一个/换成空格，这就把参数分成了两个域。lindex命令从串里取出第一个域(用下标0)。括在[]里面的正文将被作为Tcl命令执行，并用结果正文串替代；$x用变量x的值替代。

```
# geturl.tcl: retrieve document from URL
# input has form [http://]abc.def.com[/whatever...]

regsub "http://" $argv "" argv  ;# remove http:// if present
regsub "/" $argv " " argv       ;# replace leading / with blank

set so [socket [lindex $argv 0] 80] ;# make network connection
set q "/[lindex $argv 1]"

puts $so "GET $q HTTP/1.0\n\n"  ;# send request
flush $so
while {[gets $so line] >= 0 && $line != ""} {} ;# skip header
puts [read $so]                 ;# read and print entire reply
```

在典型的情况下，这个脚本将产生大量输出，其中许多是由 <和>括起来的HTML标志。用Perl做正文替换非常方便，所以我们的下一个工具是个 Perl脚本，它利用正则表达式和替换，去掉文本里的HTML标志。

```
# unhtml.pl: delete HTML tags

while (<>) {            # collect all input into single string
    $str .= $_;         # by concatenating input lines
}

$str =~ s/<[^>]*>//g;   # delete <...>
$str =~ s/ / /g;   # replace   by blank
$str =~ s/\s+/\n/g;     # compress white space
print $str;
```

如果不懂Perl，这段东西看起来就像是密码。结构

```
$str =~ s/regexp/repl/g
```

在字符串str的正文里，把与正则表达式 regexp匹配的(最左最长)串替换为repl，最后的g表示要全局性地做，也就是说，对串中所有匹配做这件事，而不是仅处理第一个匹配。元字

China-pub.com 下载

符序列\s是一个缩写形式，表示所有的空白字符 (空格、制表符、换行一类东西)；\n表示换行字符。串“   ”是HTML里的一个字符，就像第2章说明的那些，它定义一个不允许断开的空白字符。

把所有这些东西放到一起，就构成了一个能力不强但也可以工作的网络浏览器，实现它只要写一个单行的外壳程序：

```
# web: retrieve web page and format its text, ignoring HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

它能提取网页，丢掉其中所有的控制和格式信息，然后按照自己的规矩重新进行格式化。这是从网络上快速获取页面的一种途径。

注意，在这里我们以串联方式同时使用了多种语言： Tcl、Perl和Awk，每种语言都有特别适合的工作，在每种语言里都有正则表达式。记法的威力就在于对每个问题都可能有最合适的工具。用Tcl完成通过网络获取正文的工作十分方便； Perl和Awk适合做正文的编辑和格式化；当然，还有正则表达式，它最适合用来刻画作为查找和修改对象的正文片段。这些语言放在一起，其威力远大于它们中的任何一个。把工作分解成片段有时是很值得做的，如果这样做有可能使你得益于某些正确的记法。

9.4 解释器、编译器和虚拟机

一个程序怎样才能从它的源程序代码形式进入执行？如果这个语言足够简单，就像出现在printf里的或者是前面最简单的正则表达式那样，我们可以直接对照着源代码执行。这非常容易，可以立即就开始做。

在设置方面的时间开销和执行深度之间有一种此消彼长的关系。如果语言更复杂些，人们通常就会希望做一些转换，把源代码转换为一种对执行而言既方便又有效的内部表示形式。处理原来的源代码需要用一些时间，但这可以从随后的快速执行中得到回报。有些程序里综合了转换和执行的功能，它们能读入源代码正文，对其做转换后运行之，这种程序称作“ 解释器 ”。Awk和Perl是解释性的，许多其他脚本语言和专用语言也是这样。

第三种可能性是为特定种类的计算机 (程序要在这里执行)生成指令代码，编译器做的就是这件事。这种做法事先要付出极大的努力，但能导致随后最快速的执行。

也存在着另一种组合方式，这就是我们在本节里准备研究的：把程序编译成某种假的计算机(虚拟机)的指令，而这种虚拟机可以用任何实际计算机进行模拟。虚拟机综合了普通解释和编译的许多优点。

如果语言很简单，要弄清程序的结构并把它转换为某种内部形式，并不需要做多少处理工作。当然，如果语言里存在着一些复杂性 —— 有说明、嵌套结构、递归定义的语句或表达式、带优先级的运算符以及其他一些类似东西 —— 要正确地剖析输入，弄清楚它的结构就麻烦多了。

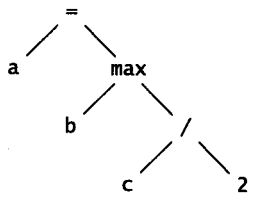
分析程序(parser)常常是借助于某个分析程序自动生成器写出来的，这种工具也被称为编译程序的编译程序，例如yacc或bison等。这种工具能从语言的描述 (语言的语法)出发，转换出(典型情况是)一个C或C++ 程序。这个程序经过编译后，就能把该语言的语句转换到对应的内部形式。当然，由语法描述出发能生成一个剖析程序，这也是好记法威力的一个例证。

由分析程序生成的表示形式通常是树，其内部结点包含着运算符号，叶结点包含的是运算对象。下面这个语句：

China-pub.com
下载


```
a = max(b, c/2);
```

可能产生下面这个分析树(或称语法树)：



在第2章描述的许多树算法可以用来构造或者处理分析树。

一旦这种树构造好了，我们就有许多可能的方法来处理它。最直接的方法（也是Awk所采用的)是直接在树中运动，并在这个过程中求出各个结点的值。要完成对这种整数表达式语言的求值过程，下面给出一个简化的程序版本，它采用后序遍历方式：

```
typedef struct Symbol Symbol;
typedef struct Tree Tree;

struct Symbol {
    int     value;
    char    *name;
};
struct Tree {
    int     op;         /* operation code */
    int     value;      /* value if number */
    Symbol  *symbol;    /* Symbol entry if variable */
    Tree    *left;
    Tree    *right;
};

/* eval: version 1: evaluate tree expression */
int eval(Tree *t)
{
    int left, right;

    switch (t->op) {
    case NUMBER:
        return t->value;
    case VARIABLE:
        return t->symbol->value;
    case ADD:
        return eval(t->left) + eval(t->right);
    case DIVIDE:
        left = eval(t->left);
        right = eval(t->right);
        if (right == 0)
            eprintf("divide %d by zero", left);
        return left / right;
    case MAX:
        left = eval(t->left);
        right = eval(t->right);
        return left>right ? left : right;
    case ASSIGN:
        t->left->symbol->value = eval(t->right);
        return t->left->symbol->value;
    /* ... */
    }
}
```

China-pub.com 下载

前几个分情况完成的是对最简单的表达式求值，例如那些常量和值等；随后的情况对算术表达式求值；其他的可能是做另一些特殊处理，如条件语句和循环等。要实现这些控制结构，树中还需要附加其他信息，以表示控制流。这里都没有给出来。

就像在前面处理pack和unpack那样，我们也可以用一个函数指针的表取代这里显示的开关语句。各运算符对应的函数和上面开关语句里的差不多：

```
/* addop: return sum of two tree expressions */
int addop(Tree *t)
{
    return eval(t->left) + eval(t->right);
}
```

一个函数指针的表将运算符关联到那些实际执行操作的函数：

```
enum {  /* operation codes, Tree.op */
    NUMBER,
    VARIABLE,
    ADD,
    DIVIDE,
    /* ... */
};

/* optab: operator function table */
int (*optab[])(Tree *) = {
    pushop,     /* NUMBER */
    pushsymop,  /* VARIABLE */
    addop,      /* ADD */
    divop,      /* DIVIDE */
    /* ... */
};
```

求值过程以运算符为指标，从表里得到函数指针，随之去调用正确的函数。下面的版本将递归地调用其他函数。

```
/* eval: version 2: evaluate tree from operator table */
int eval(Tree *t)
{
    return (*optab[t->op])(t);
}
```

上面这两个eval版本都是递归的。存在着一些去除递归的方法，包括一种称为**线索化代码**的巧妙技术，它能使调用栈完全瘪下来。要去掉所有递归，最巧妙的方法是把有关函数存入一个数组，然后线性地穿过这个数组，执行对应的程序。这个数组实际上已经变成了由一个小的特殊机器执行的指令序列。

我们仍然需要用一个栈来表示计算过程中部分计算出的结果。这时函数的形式也要做些改变，不过这种转换是很容易的。这样，我们已经发明了一种堆栈机器，它的指令是一些小函数，而运算对象都存在一个独立的运算对象栈里。这并不是一个真正的机器，但我们可以像使用真的机器一样为它编程，同时也很容易把它实现为一个解释器。

我们将不再需要通过在树上的运动，对树进行求值，而是生成一个执行这个程序的函数数组。数组里也应该包含指令所使用的数据值，例如常数和变量 (符号)等，因此这个数组的元素类型应该是一个联合：

```
typedef union Code Code;
```

China-pub.com
下载


```
union Code {
    void    (*op)(void); /* function if operator */
    int     value;       /* value if number */
    Symbol  *symbol;     /* Symbol entry if variable */
};
```

下面是函数generate，它生成一些函数指针，并把它们放入以这些东西为内容的数组 code 里。generate的返回值不是表达式的值——这种值在所生成代码被真正执行的时候才产生——而是一个code下标，随后再生成的操作将被放在这里：

```
/* generate: generate instructions by walking tree */
int generate(int codep, Tree *t)
{

    switch (t->op) {
    case NUMBER:
        code[codep++].op = pushop;
        code[codep++].value = t->value;
        return codep;
    case VARIABLE:
        code[codep++].op = pushsymop;
        code[codep++].symbol = t->symbol;
        return codep;
    case ADD:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = addop;
        return codep;
    case DIVIDE:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = divop;
        return codep;
    case MAX:
        /* ... */
    }
}
```

处理语句a= max(b,c/2)生成的代码看起来是下面的样子：

```
pushsymop
b
pushsymop
c
pushop
2
divop
maxop
storesymop
a
```

有关运算符函数将对堆栈进行操作，弹出运算对象并压入结果。

解释器本身就是一个循环，它用一个程序计数器，在函数指针数组里移动：

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */

/* eval: version 3: evaluate expression from generated code */
int eval(Tree *t)
```

China-pub.com 下载

```
{
    pc = generate(0, t);
    code[pc].op = NULL;
    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
}
```

在我们发明的这个堆栈机器里，上述循环实际上是以软件方式模拟着真实计算机的硬件所做的事情。下面是几个有代表性的运算函数：

```
/* pushop: push number; value is next word in code stream */
void pushop(void)
{
    stack[stackp++] = code[pc++].value;
}

/* divop: compute ratio of two expressions */
void divop(void)
{
    int left, right;

    right = stack[--stackp];
    left = stack[--stackp];
    if (right == 0)
        eprintf("divide %d by zero\n", left);
    stack[stackp++] = left / right;
}
```

注意，对除数为0的检查出现在divop中，而不放在generate里。

检查条件、分支和循环的执行等，在运算符函数里表现为直接修改程序计数器，这样做就产生了分叉，到达函数数组中的另一个地方。例如，一个goto运算符就是直接设置pc变量，而一个条件分支只在条件为真的情况下设置 pc。

code数组是解释器内部的东西。当然，我们也可以设想把生成出来的程序存入一个文件。如果直接把函数的地址写出去，得到的结果将是不可移植的和脆弱的。我们应该换一种方式，写出去一些代表函数的常数，例如用 1 000表示addop，用1001表示pushop等等。在重新读入程序准备执行时，再把这些常数翻译回函数指针。

如果我们查看由上面的过程产生的文件，它们就像是虚拟机的指令流，虚拟机的指令实现的是我们小语言的基本运算符，而 generate函数实际上就是个编译程序，它把我们的语言翻译到对应的机器。虚拟机是一个可爱的历史悠久的想法，最近，由于 Java和Java虚拟机(JVM)的出现，这种想法又变得时髦起来。要从高级语言程序产生出一种可移植并且高效的表示形式，虚拟机方法实际上指出了一条康庄大道。

9.5 写程序的程序

函数generate最值得注意的地方，或许就在于它是一个写程序的程序：它的输出是另一个(虚拟)机器可以执行的指令序列。编译程序每时每刻都在做这件事：把源代码翻译成机器指令，所以这种想法是人人皆知的。在实践里，写程序的程序可能以许多不同的形式出现。

一个最常见的例子是网页 HTML的动态生成。HTML是一种语言，其功能当然非常有限，

China-pub.com
下载

它还可以包含JavaScript代码。网页常常是由Perl或者C程序在运行中生成的，其具体内容(例如，某些查询结果或者是定向广告)根据外来需求确定。我们对这本书里的各种图形、表格、数学表达式以及索引等使用了多种特殊的语言。作为另一个例子， PostScript也是一种程序设计语言，它可以由文字处理程序、画图程序和许许多多其他程序生成。在处理的最后阶段，本书就被表示为一个大约有57 000行的PostScript程序。

一个文档是一个静态的程序，使用某个程序语言作为记法，表达各种问题领域是一种非常有力量的思想。许多年以前，程序员就梦想着有朝一日计算机能帮他们写所有的程序，当然这可能永远都只是一个梦。但是，今天的计算机已经在日复一日地为我们写程序，经常是在写那些我们原来从未想到还可能表达为程序的东西。

最常见的写程序的程序是编译器，它能把高级语言程序翻译成机器代码。然而，把代码翻译到主流程序设计语言常常也很有用。在前面的章节里，我们曾提到的分析程序生成器能把一个语言的语法定义转换为一个能分析这个语言的C程序。C常常被作为一种“高级的汇编语言”使用。有些通用的高级语言(例如Modula-3和C++)的第一个编译程序产生的就是C代码，这种代码随后用标准C编译系统编译处理。这种方式有不少优点，包括效率——因为这些程序原则上可以运行得像C程序一样快，可移植性——因为该编译系统可以搬到任何有C编译器的系统上。这大大促进了这些语言的早期传播。

作为另一个例子， Visual Basic的图形界面生成一组Visual Basic赋值语句，完成用户通过鼠标在屏幕上选择的那些对象的初始化工作。许多其他语言也有“可视化的”开发环境和“巫师(wizard)”，它们都能从鼠标的点击中综合产生出各种用户界面的代码。

尽管各种程序生成器的威力如此强大，尽管有这么多极好的例子，记法问题还是没有得到它应有的尊重，仍然很少被程序员们使用。实际中确实存在着大量的用程序生成小规模代码的机会，所以，你自己应该学会从这里获得一些益处。下面是生成C或者C++代码的几个例子。

Plan 9操作系统由一个头文件生成它的错误信息，该文件里包含一些名字和注释；这些注释被机械性地转换成带引号的字符串，放进一个数组里，而该数组由枚举值做为下标。下面这段代码显示了头文件的结构：

```
/* errors.h: standard error messages */

enum {
    Eperm,      /* Permission denied */
    Eio,        /* I/O error */
    Efile,      /* File does not exist */
    Emem,       /* Memory limit reached */
    Espace,     /* Out of file space */
    Egreg       /* It's all Greg's fault */
};
```

有了这个输入，一个简单的程序就能产生下面这段错误信息声明：

```
/* machine-generated; do not edit. */

char *errs[] = {
    "Permission denied", /* Eperm */
    "I/O error", /* Eio */
    "File does not exist", /* Efile */
    "Memory limit reached", /* Emem */
    "Out of file space", /* Espace */
    "It's all Greg's fault", /* Egreg */
};
```

China-pub.com 下载

这种方式有许多优点。首先，enum值和它们所表示的字符串之间的关系在书面上自成文档，很容易做到对自然语言的独立性。此外，信息只出现一次，只有“一个真理点”，所有其他代码都由此生成。这样，如果需要做更新，那么就只有一个地方需要考虑。如果这些信息出现在许多地方，很难避免在某些时候它们之间失去同步性。最后，我们很容易对 .c文件做出适当安排，使得在这个头文件改变时，所有有关的文件都重新建立和重新编译。当某个错误信息必须改变时，需要做的所有事情就是修改这个头文件，并重新编译整个操作系统。这样所有信息都能得到更新。

生成程序可以用任何语言来写。用Perl一类字符串处理语言写起来更容易：

```
# enum.pl: generate error strings from enum+comments

print "/* machine-generated; do not edit. */\n\n";
print "char *errs[] = {\n";

while (<>) {
    chop;                           # remove newline
    if (/^\s*(E[a-z0-9]+),?/) {     # first word is E...
        $name = $1;                 # save name
        s/.*\/\* *//;               # remove up to /*
        s/ *\*\///;                 # remove */
        print "\t\"$_\", /* $name */\n";
    }
}

print "};\n";
```

正则表达式又参与了这里的行动。所有第一个域是标识符后跟着逗号的东西都被选中[⊖]，第一个代换删掉直到注释里第一个非空字符之前的所有东西，第二个代换删去注释结尾和它前面的所有空格。

作为编译系统测试工作的一部分，Andy Koenig开发了一种写C++ 代码的方便方法，以检查编译系统能否捕捉到各种程序错误。他给所有能导致编译诊断的代码段都加上一个奇妙的注释，描述应该出现的错误信息。每个这种注释行都以///开头(以便区别于普通的注释)，还包含一个能与相关诊断信息做匹配的正则表达式。例如，下面这两个代码段都应该产生诊断信息：

```
int f() {}
    /// warning.* non-void function .* should return a value

void g() {return 1;}
    /// error.* void function may not return a value
```

如果用我们的C++ 编译系统处理到第二个测试实例，它打印出的正是我们所期望的，能与对应正则表达式匹配的信息：

```
% CC x.c
"x.c", line 1: error(321): void function may not return a value
```

每个这种代码片段都提交给编译系统，然后将产生的输出与所期望的诊断信息做比较，这个过程通过shell和Awk程序的结合来管理，出现失误时将指明是哪个测试实例使编译系统的输出与期望的不同。由于在注释里用的是正则表达式，输出可以有些自由度，根据情况不同，可以把它们写成或多或少具有某些宽容性。

使用带语义的注释不是什么新想法，PostScript里就有这种东西。在PostScript里，注释由

⊖ 实际上是以E开始的标识符，后跟逗号。——译者

China-pub.com
下载

%开头，而那些由%%开始的注释，按照习惯，带有一些附加信息，可以描述关于页号、限界盒(bounding box)、字体名以及其他类似的东西：

```
%%PageBoundingBox: 126 307 492 768
%%Pages: 14
%%DocumentFonts: Helvetica Times-Italic Times-Roman
            LucidaSans-Typewriter
```

在Java里，以/**开始**/结束的注释被用于建立跟随其后的类定义的文档。对自带文档代码做大范围的推广，就成为所谓的带文档程序设计 (literate programming)，就是说，把一个程序和它的文档集成在一起。这样，通过某种过程，其中的文档可以按自然顺序打印出来供人阅读，另一个过程则以正确顺序安排它去进行编译。

对于上面的所有实例，仔细研究其中记法的作用、语言的混合方式以及工具的使用等，都是很有价值的。这几个方面的组合能够放大各个成分的威力。

练习9-15 有一个历史很久远的有关计算的故事，要求写出一个程序，执行后恰好产生其自身原来的形式。这是用程序写程序的一个非常特殊而又很巧妙的例子。请在你自己最喜欢的语言里试一试。

9.6 用宏生成代码

现在向下降几个层次，我们同样也能写这样的宏，它们在编译时能够产生代码。在这本书里我们始终在告诫读者，应该反对使用宏和条件编译，因为它们鼓励的是一种破绽百出的程序设计风格。但是，宏也确实有它自己的位置，有时正文替换恰恰就是问题的正确解决方法。这里有一个例子，是关于如何利用C/C++ 的宏预处理装配起同样风格的重复出现的代码片段。

在第7章里我们提到过估计语言基本结构速度的程序，它就利用了 C的预处理功能，通过把具体代码装入一段框架，组装起一大批测试例子。有关测试的基本构架是封装起来的一段代码，其中有一个循环。在循环开始时设置时钟，随后将代码段运行许多次，最后停下时钟并报告结果。所有重复出现的代码段都被包裹在几个宏里面，实际被计时的代码则被作为参数传递给宏。基本的宏具有如下形式：

```
#define LOOP(CODE) {                          \
    t0 = clock();                             \
    for (i = 0; i < n; i++) { CODE; }         \
    printf("%7d ", clock() - t0);             \
}
```

写在最后的反斜线符号使宏可以延伸到多行。这个宏将被用在“语句”里，应用的形式大致是下面的样子：

```
LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
LOOP(f1 = f2 - f3)
```

有时需要用另一些语句完成初始化工作，但最基本的计时部分都表示在这种一行一个的程序段里，它们展开后将形成很长的一段代码。

宏处理功能也可以用于生成产品代码。有一次， Bart Locanthi要写一个高效的两维图形操作程序。这种操作被称为bitblt或rasterop。要想把它们做得速度很快是十分困难的，因为在这里有许多参数，它们可能以很复杂的方式互相组合。经过认真的分析之后， Locanthi把

China-pub.com 下载

所有组合情况归结到一个循环，它可以独立地进行优化。在此之后，他用宏替换构造出每种情况，就像前面性能测试的例子那样，然后把各种变形安排在一个大的开关语句里面。原始的源代码只有几百行，宏处理的结果就变成了数千行。这样通过宏展开得到的代码并不是最优的，这是由于问题本身的复杂性。但是这种做法又非常实际，也很容易完成。另一方面，作为一段高性能代码，它也具有相对的可移植性。

练习9-16 练习7-7要求写一个程序去度量C++ 里各种操作的代价。使用本节提出的想法重新写这个程序。

练习9-17 练习7-8要求做一个Java的代价模型，在Java中没有宏的功能。我们可以通过另外写一个程序的方式来解决这个问题，在这里可以采用你所选定的任何语言 (或几个语言)。令这个程序写出所需要的Java程序，并自动完成计时运行。

9.7 运行中编译

这几节我们一直在谈论写程序的程序。在前面的所有例子里，生成出来的程序都具有源代码的形式，它还需要经过编译或者解释才能运行。实际上也可能产生出能够立即投入执行的代码，只要我们生成的不是源代码而是机器指令。这种做法通常被称为“运行中”的编译，或者“即时”编译。前一个词出现得更早，而现在后一个词却更流行，包括该词 (Just In Time)的首字母缩写词JIT。

虽然这样编译产生代码必定是不可移植的 —— 只能运行在一种类型的处理器上 —— 但它却可能非常快。考虑表达式：

```
max(b, c/2)
```

计算过程中必须求出c，将它除以2，用得到的结果与b比较，选出其中较大的一个。如果用本章前面给出了轮廓的虚拟机计算这个具体表达式，我们可以删去在 divop里对除零的检查，因为2不是0，有关的检查毫无意义。但是，无论确定采用什么样的设计，我们在对虚拟机的实现做安排时，都不可能去掉其中的检查，在任何除法运算的实现里都必须做除数与 0的比较。

这也就是动态代码生成可以起作用的地方。如果直接从表达式出发构造代码，而不只是简单地串起预先定义的一些操作，在那些已知除数不是零的地方，是完全可以避免再做除零检查的。实际上还可以再前进一步，如果整个表达式就是常数，例如 max(3*3,4/2)，我们可以在产生代码的过程中对它求一次值，然后直接用常数 9取代它。如果表达式出现在一个循环里，那么循环的每次执行都将节省时间。只要循环的次数足够多，我们就能把为弄清表达式和生成代码所花费的额外时间都节省回来。

这里的关键思想还是记法。记法给了我们一种表达问题的一般性方法，而这种记法的编译程序可以针对特定计算的细节生成专门代码。例如，在处理正则表达式的虚拟机里，我们最好有一个匹配文字型字符的运算：

```
int matchchar(int literal, char *text)
{
    return *text == literal;
}
```

当我们为特定模式生成代码时，由于 literal的值是固定的，例如就是‘ x ’，那么我们最好能改用像下面这样的运算：

China-pub.com
下载


```
int matchx(char *text)
{
    return *text == 'x';
}
```

与此同时，我们不希望为每个文字字符值预先定义一个特殊运算，而是想把问题弄得更简单些，只有在表达式实际需要的时候才生成这种代码。把这个思想推广到整个的运算集合，就可以写出一个运行时的编译器，它把当时处理的正则表达式翻译成一段为此表达式而特别优化了的代码。

1967年，Ken Thompson在IBM7094机器上做正则表达式的实现时，所做的实际上就是这件事。他的程序对表达式里的各种操作生成一些小块的二进制 7094代码，再把它们串联在一起，最后通过调用来运行结果程序，就像调用普通函数一样。类似技术也可以用到图形系统里，创建一些屏幕更新的特定指令序列。由于在图形处理中存在太多的特殊情况，与其事先把它们都写出来，或在更一般的代码中加入大量条件测试，还不如对每种实际发生的情况动态地构造相关代码。后面这种方法更有效。

如果要展示实际的运行时编译系统的构造过程，就有可能使我们的讨论涉及到过多的特定指令集合的细节。但是，显示一下这种系统如何工作还是很值得的。在阅读本节余下的部分时，你主要应该了解其中的思想和见解，而不是具体的实现细节。

请回忆一下，前面我们把虚拟机写成具有下面的结构：

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */
    ...
    Tree *t;

    t = parse();
    pc = generate(0, t);
    code[pc].op = NULL;

    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
```

要将这些代码弄成运行时编译，我们必须做一些修改。首先， code数组将不再是函数指针的数组，而应该是可执行指令的数组。到底这些指令的类型是 char或int或long，要看我们的编译所面对的处理器的情况，这里假定它是 int。在代码生成之后，我们准备像函数一样调用它。这里将不再需要虚拟的程序计数器，因为处理器本身有它的执行循环，它将为我们遍历这些代码，一旦计算完成，它就会返回，就像正常的函数一样。此外，我们还可以有些选择，是自己为机器维护一个运算对象栈，还是直接使用处理器的堆栈。这两种方法各有长短。在这里我们仍然使用一个独立的栈，以便把注意力集中到代码本身的细节方面。现在的实现大致是这个样子：

```
typedef int Code;
Code code[NCODE];
int codep;
int stack[NSTACK];
```

China-pub.com 下载

```
int stackp;
    ...
    Tree *t;
    void (*fn)(void);
    int pc;

    t = parse();
    pc = generate(0, t);
    genreturn(pc);      /* generate function return sequence */
    stackp = 0;
    flushcaches();      /* synchronize memory with processor */
    fn = (void(*)(void)) code;  /* cast array to ptr to func */
    (*fn)();            /* call function */
    return stack[0];
```

在generate结束后，genreturn将安排一些指令，使生成的代码能把控制返回到eval。

函数flushcaches要求完成一些动作，迫使处理器为执行新生成的代码做好准备，这是必须做的。现代计算机的速度非常快，部分原因就在于它们拥有为指令和数据而使用的各种缓冲存储器，而且还有内部的流水线，这使许多顺序指令的执行可以互相重叠。缓存和流水线都期望遇到的指令流是静态的。如果我们要求它执行刚刚生成的那些代码，处理器很可能会被搞糊涂。因此，在执行新生成的代码之前， CPU需要腾空其流水线，刷新其缓存。这些都是对具体机器有高度依赖性的操作，对各种特定类型的计算机， flushcaches的实现肯定是不同的。

最值得注意的表达式是(void(*)(void)) code，这是一个模子(Cast)[⊖]，它把包含着新生成的指令序列的数组地址转换成一个函数指针，通过它就可以像函数一样调用这些代码。

从技术上看，生成代码本身并没有太大困难，但要想有效地完成这个工作还有许多工程性的事项。让我们从某些基本构件开始。和前面一样，在编译过程中需要维护 code数组和它的一个下标。为了简单起见，这里把它们都定义为全局的，这也是前面一直采用的方式。此后，我们就可以写一个编排指令的函数了：

```
/* emit: append instruction to code stream */
void emit(Code inst)
{
    code[codep++] = inst;
}
```

指令本身可以通过与具体处理器相关的宏或者小函数来定义，它们填充指令字中各个域，装配起各种指令。假设我们有一个名为popreg的函数，它生成的代码是从堆栈里弹出一个值，并将它存入处理器的寄存器；另一个函数叫 pushreg，它生成的代码从一个寄存器取出值，并将这个值压进堆栈。我们修改后的 addop使用这些基本功能，它看起来大致是下面的样子，使用某些预先定义的常数，这些常数描述指令本身 (像ADDINST)或者其编排形式(定义有关格式的各种SHIFT位置)：

```
/* addop: generate ADD instruction */
void addop(void)
{
    Code inst;

    popreg(2);      /* pop stack into register 2 */
    popreg(1);      /* pop stack into register 1 */
    inst = ADDINST << INSTSHIFT;
```

⊖ cast，也就是类型强制转换运算符。 —— 译者

China-pub.com
下载


```
        inst |= (R1) << OP1SHIFT;
        inst |= (R2) << OP2SHIFT;
        emit(inst);     /* emit ADD R1, R2 */
        pushreg(2);     /* push val of register 2 onto stack */
    }
```

这些只不过是个开始。如果要写一个实际的运行中编译器，我们还必须做许多优化方面的工作。例如，如果被加的是个常数，那么就不必先把它压入堆栈，然后再弹出来做加的操作，而是可以直接做加法。通过这类考虑可以清除大量多余的开销。当然，即使是写成上面的样子，这个addop也会比我们前面写的东西快得多，因为各种操作不是通过大量函数调用串联起来的。在这里，执行操作的所有代码都放在存储器里，形成一个指令块，真实处理器的程序计数器为我们完成所有的顺序处理。

与我们为虚拟机实现的那个generate相比，这个函数看上去要大得多。因为现在它编排的是真实的机器指令，而不是预先定义的函数指针。为了生成高效代码，还需要花很多功夫，去查看应该去掉的常量计算，以及做其他可能的优化。

在这里，对代码生成的讨论完全是走马观花式的，只是很初步地显示了一些真实编译系统所使用的技术，更多的东西完全被忽略了。我们还回避了许多由于现代 CPU的复杂性带来的问题。但是，即使如此，这些讨论仍然显示出一个程序可以如何去分析问题的描述，生成能够高效解决问题的特殊代码。你可以利用这些思想写一个快如闪电的 grep程序，或去实现你自己发明的某个小语言，或设计和实现一个能完成特定计算的虚拟机，或者甚至是 (在不多的其他东西的帮助下)为某种有趣的语言写一个编译程序。

从正则表达式到C++ 语言之间有一段很长很长的路，但是，这两者都是为了解决问题而设计出的记法。有了正确记法的辅佐，许多问题的解决将变得容易得多。而设计和实现这些记法本身也是趣味无穷的。

练习9-18 如果运行时编译能把仅仅包含常数的表达式，例如 max(3*3, 4/2)用对应的值取代，它将能产生更快的代码。一旦识别出这种表达式，它怎样才能计算出有关的值？

练习9-19 你怎么测试一个运行时的编译器？

补充阅读

由Brian Kernighan和Rob Pike合写的《Unix程序设计环境》(The Unix Programming Environment，Prentice Hall，1984)里对于计算的基于工具途径做了广泛讨论， Unix系统对这些有很好的支持。该书第 8章给出了一个简单程序设计语言的完整实现，从 yacc语法描述直到可执行代码。

Don Knuth的《TEX: 程序》(TEX: The Program，Addison-Wesley，1986)描述了复杂文档的格式化问题，给出了一个完整的程序，大约有 13 000行具有“带文档程序设计”风格的Pascal代码。在这些程序的正文里结合进有关文档，可以用程序对文档做格式化，或者从中提取可编译代码。Chris Fraser和David Hanson在《一个目标可重定位的C编译器：设计和实现》(A Retargetable C Compiler: Design and Implementation，Addison-Wesley，1995)里对ANSI C做了同样的事情。

Java虚拟机在 Tim Lindholm和Frank Yellin的《Java虚拟机规范》(The Java Virtual

China-pub.com 下载

Machine Specification，第2版，Addison-Wesley，1999)中描述。

有关Ken Thompson算法(它是最早的软件专利之一)的讨论出现在《正则表达式搜索算法》(Regular Expression Search Algorithm，Communications of The ACM，卷11，第6期，pp419~422，1968)。Jeffrey E. F. Friedl的《掌握正则表达式》(Mastering Regular Expression，O'Reilly，1997)包含了对这个问题的范围非常广泛的论述。

Rob Pike、Bart Locanthi和John Reiser的文章《Blit上点阵图形的硬件/软件权衡》(Hardware/Software Tradeoffs for Bitmap Graphics on the Blit，Software-Practice and Experience，卷15，第2期，pp131~152，1985年1月)描述了一个针对二维图形操作的运行时编译器。