jagen 2014-07-23 14:22

微型语言系列之dc和bc

在Linux系统所提供的众多工具中，最容易让用户忘记它们真实能力应该当属dc和bc了。在人们的印象中，它们只不过是简单的命令行下任意精度计算器罢了，dc与bc最大区别就是dc要使用逆波兰式（后缀表达式）来表达算式。

dc和bc实际上是两个微型语言，它们都有选择分支和循环，拥有图灵完备的特性。其实dc和bc的通用性绝对不次于任何其它通用解释型语言，只是其数据类型的分类很有限，仅包括无限精度的数值和字符串。这使得它处在微型语言和通用解释型语言之间。它们可编程特性的目的是让它们不要作为计算器涉足普通计算，但由于dc和bc的操作界面简单又直观，不用掌握任何编程知识也能拿来就用，这导致大多数用户都没有认识到这些。

dc或bc可以做任意复杂的数学运算，只要把它们作为其它程序的从属进程进行调用，其它程序就可以轻易的获取这样的能力。这就是这两种微型语言的通用性和威力所在，让笔者不敢有丝毫吝啬之心，十分迫切的要将它们的种种与各位读者分享。

1.1 历史渊源

dc可以说是Unix世界中最为古老的语言，比C语言还早。dc是桌面计算器——deskcalculator的缩写，之所以使用逆波兰式来表达算式，是因为它诞生的年代没有足够强大的计算机可以容易的处理代数标记法。

bc要晚一些了，在1975年的Version 6Unix中第一次发行。bc是基础计算器——basic calculator的缩写。这个时候的计算机已经足够强大了，所以bc就使用了人们很容易理解的代数标记法。而且在这个时候C语言也被发明出来，而且还是个时髦的东西，所以bc的语法也弄的跟C很像。

笔者将dc和bc放在一起来将，是因为它们之间的渊源极深。

首先，它们有共同的作者——Robert Morris和Lorinda Cherry。Robert Morris是著名的密码学家、计算机科学家，普遍被认为是计算机安全的先驱人物。被Linux系统沿用至今的用于用户验证的密码加密体系就是这位爷爷发明的。他有一句名言：“确保计算机安全的三条黄金定律是：不要计算机，不开机，不用计算机”。非常遗憾的是，2011年6月26日，Robert Morris永远的离开了我们。另外一位作者Lorinda Cherry是极其稀罕的女性程序员。虽然名气不如其他技术牛人那样大，但是作为dc和bc的共同开发者，也足以让我们敬仰万分了。另外，Lorinda Cherry还与Brian W. Kernighan共同开发了Troff的预处理器eqn。

其次，由于dc使用逆波兰式来表达算式，可谓杀伤脑细胞的威力无边，这使得人们不得不尽快开发出对人更为有好的新型工具。但是dc的确已经非常强大了，从新造轮子实在是让人于心不忍。于是bc的开发者们使用了一个非常折中的方案，作为dc的前端出现。也就是说，bc只是将代数标记的表达式和类似C的程序语句翻译成dc能够处理的逆波兰式和命令。而bc和dc的通讯就是简单的管道。

当然，bc作为dc的前端只是在早期的版本是这样实现的。更为现代的，也是Linux系统所提供的GNU版本的dc和bc就不再是这样的结构了。现在的处理方法有点类似Java，dc或bc的代码会被编译成字节码，然后由所谓的虚拟机去解释执行。这样做的一个好处就是，其它程序可以直接调用这个虚拟机来完成任意精度的数学运算。但是到目前为止，笔者还没见到有谁这样用。

1.2 dc简介

dc有一个非常著名的例子，是一个用Perl实现的RSA公共密钥算法，广泛发布在签名档和T恤上，以示对美国1995年的限制密码学出口的抗议。见代码6-2所示：

```
print pack"C*",split/D+/,`echo"16iII*oU@{$/=$z;[(pop,pop,unpack "H*",
<>)]}EsMsKsN0[1N*11K[d2%Sa2/d0<x+d*lmLa^*1N%0]dsXx++1M1N/dsM0<J]dsJxp"|dc`
```

代码6-2 dc实现RSA公共密钥算法

不管你是否精通dc，看这段代码都如天书一般。一是由于作者为了让整个程序简练忽略了可读性；二是由于dc使用逆波兰式，完全不符合人的思维习惯；三是整个dc代码又融合了perl的正则表达式。也正因为如此，笔者不指望各位读者能够通过本书完全掌握dc并看懂这段代码。但是让各位读者跟随笔者穿越一下，看看C语言诞生之前的程序是啥样，还是一件有趣儿的事情。同时，这有助于更好的掌握bc。

1.2.1 逆波兰式

我们常用的类似“1+1”、“3*3”这样的表达式的标准称呼是代数标记法，也可以叫中缀表达式，因为运算符在中间。如果将它们转换成逆波兰式，则是：“1 1 +”和“3 3 *”。相对于中缀表达式，逆波兰式也叫后缀表达式。

计算机直接处理代数标记法的算式比较麻烦。因为加、减、乘、除的运算优先级是不同的，而其还有使用圆括号“()”强制优先级。如果都搞在一起，对于早期的计算机来说，简直是鸭梨山大。逆波兰式则不同，它不用考虑运算优先级的问题。比如：(1+2-3)*4/5可用逆波兰式表示为1 2 + 3 + 4 * 5 /，只要按顺序依次运算就可以得到最终的结果0。这可给计算机减轻了很多的负担。其实人在编写这样的算式时也很容易，不用去考虑优先级。只是由于人的思维惯性的原因，使得这个成了一个麻烦。

其实一直到现在，所有高级语言在处理数学表达式的时候，依然要将它们转换成逆波兰式来解决各种运算优先级的问题。具体的转换方法也就是使用堆栈这种数据结构。读者们随便找一本有关数据结构的书籍，在介绍堆栈的时候，基本上都会拿这个举例子。所以笔者就不在这里出丑了。

1.2.2 基本操作

dc与其它的Linux工具一样，既可以通过标准输入，也可以通过命令行或文件来获取输入。输入的内容就是dc程序。比如计算4*5：

```
$ dc
4 5 *
p
```

当读者们输入命令“p”之后，20会在下面出现了。当然，如果读者觉写多行有点别扭，也可以将这段程序写在一行，如：

```
$ echo "4 5 * p" | dc
```

稍微分析一下这段程序。dc会分别将4和5压入堆栈；当遇到运算符乘法运算符“*”的时候，就将4和5弹出堆栈并计算相乘的结果，再将结果压入堆栈；最后遇到命令“p”，输出堆栈顶端的内容。

从这种执行行为上可以看出，dc使用逆波兰式在处理用户输入的同时就开始进行计算，堆栈所需的内存空间一直被控制在很小的范围。将堆栈作为其工作基础，对于复杂的运算可以节省很大的内存开销。在dc诞生的那个年代，这绝对是一项符合经济利益的优化。付出的代价就是用户得改变一下习惯。当然，那个年代的程序员比现在还要“贱”。

堆栈是dc工作的基础，那么只有一个用于输出堆栈顶端内容的命令“p”是不够用的。dc还额外的提供“r”命令用于交换堆栈顶端的两个单元的内容、“c”命令用于清空堆栈，以及“d”命令复制堆栈顶端的内容在将其压入堆栈。输出堆栈内容的命令还有“f”用于输出整个堆栈，“n”命令在输出堆栈内容的时候还要将内容弹出堆栈。

dc还有一个十分有用的特性是能够用于各种进制之间的转换。比如要将16进制的数转为2进制输出，可以使用这样的代码：

```
$ echo "16 i 2 o ABCDEF p" | dc
```

输出结果是：101010111100110111101111。从代码中可以获悉，命令“i”是用于指定输入进制的，而命令“o”则是指定输出进制的，它们的命令参数保存在堆栈的顶端。

1.2.3 运算精度

既然dc号称是任意精度的计算器，那咱就做个不能整除的运算看看是个什么结果：

```
$ echo "2 3 / p" | dc
```

这个结果会是出乎意料的0，根本就谈不上精度。这是怎么回事呢？原来，dc使用命令“k”来指定具体精度的，这样就可以按需分配了。那么将程序改成：

```
$ echo " 10 k 2 3 / p" | dc
```

得到的结果就是：.6666666666。需要说明的是，虽然“k”可以指定精确到小数点后多少位，但是别指望付给它太大的值，比如1亿，这样的精度不但没意义，而且您那几个月工资买下来的电脑就别逞能了。

再来点刺激的例子，计算公式[attachment=57050]的值，程序是这样的：

```
$ echo "10 k 12 _3 4 ^ + 11 / v 22 - p" | dc
```

得到的结果是：-19.0923298925。dc算出正确的结果一点都不奇怪，奇怪的是“-3”的表示方式，它使用下划线“_”来代表符号。没别的，就因为减号“-”是运算符，dc为了简化处理就发明了新的负号。这个例子已经将dc所支持的运算符都列出来了。至于“v”的干什么，估计也能猜出来，就是求平方根的命令。

1.2.4 编程能力

dc作为一个能够编程的语言，只会算数和操作堆栈是不行的。可能那个时候还没有变量的概念，于是dc的变量被称为寄存器（register），跟汇编语言有一拼了。而且dc还拥有选择分支、循环和子程序，只是叫法依然有些怪。选择分支叫条件式；子程序叫宏；而循环实际上是递归。它们还都是基于寄存器实现的。这就是大师的杰作啊！

寄存器跟现代意义上的变量的作用是相同的，都是用于保存中间结果的。但是寄存器只能用单一字符来命名。先来看一个例子：

```
$ echo "3 sc 4 lc * p " | dc
```

这段程序的结果是：12，也就是4*3。使用了一个名为c的寄存器。命令“sc”就是将3从堆栈中弹出，并放入寄存器c；而命令“lc”则将寄存器c的内容取出并压入堆栈；由于之前已经将4压入堆栈，当遇到乘法运算符“*”时就会计算4*3的结果。注意，设置和提取寄存器内容的命令实际上就是“s”和“l”。笔者将“sc”说成是一个命令的原因在于，dc可以将任何单个字符看作是寄存器名，即使是空格或换行也会被认为是寄存器。如果真有人这么写，纯粹就是为了制造天书。

一个寄存器还可以被看作是另外的一个堆栈，可以使用“S”和“L”命令进行入栈和出栈操作。例如：

```
$ echo "3 Sc 4 Sc 5 Sc lc lc * lc * p" |dc
```

的结果是：125。

宏是使用一对中括号“[]”括起来的字符串，可以包含任何字符。宏既可以被放入堆栈，也可以放入寄存器。宏既可以跟我们现代语言中的字符串类似，能够使用“p”命令输出其内容，也可以理解为现代语言的子程序或函数，能够使用“x”命令计算其结果。例如：

```
$ echo "[1 + 2 *] p sm 3 lm x p" | dc
```

这段程序会现输出字符串：1+ 2 *，然后再输出结果：8。工作过程是：将宏[1 + 2*]压入堆栈，然后输出；将宏从堆栈顶端弹出并保存在寄存器m中；将3压入堆栈；从寄存器m中取出宏并压入堆栈；最后使用“x”命令执行宏与3的运算，并用“p”命令输出结果。

dc专门提供一个特殊大写的“P”命令，用于处理字符串的输出。它与小写的“p”不同的地方是：会将堆栈顶端的内容弹出，并试图将这些内容解为字符串，而且输出的内容不带有换行符。如果堆栈顶端是宏，则保持原样输出；如果是数字，则会取其整数部分的绝对值，然后将它拆分成字节，按照ASCII码所代表的符号输出。当然，它的拆分可能超过ASCII码所表示的字符范围（ASCII码只用了一个字节的7位），输出可能是完全不可见字符。

条件式的实现在今天看来相当难于理解，它是以寄存器和宏为基础的。工作方式是这样的：首先将条件为真时要执行的语句定义成一个宏，并保存在某个寄存器中；然后向堆栈压入两个要做比较的值；当堆栈顶端的两个值的逻辑运算值为真时，则实行指定寄存器内的宏。笔者在这里先给出一段用C描述的代码：

```
n = 5;
if ( n * 3 > 10 )
    printf( "OK" ）;
```

如果将这段代码使用dc来完成，则是：

```
$ echo "[[OK]p] sm 10 5 sn 3 ln * >m" | dc
```

这两段代码的输出结果都应该是：OK。dc支持的逻辑运算符与我们现在理解的也不太一样，包括：“>”、“!>”、“!<”、“<”、“=”和“!=”，跟汇编语言的逻辑运算是一致的。不过在笔者看来，从dc的程序代码上看，比汇编语言还要天书。需要注意的是，当条件式执行完毕后，堆栈顶端的两个数据会被丢弃。
如果说条件式你还觉得能够理解，也不算什么天书，那么遇到“循环”可就不一定了！不信？看看这段代码：

```
$ echo "5 [d 1 - d 1 <F *] d sF x p" | dc
```

不服气的读者可以现在思考一下，给你十分钟的时间，看看这段程序是干什么的。如果你能推算出来，说明你已经完全掌握dc了，笔者很欣慰；如果推算不出来，也不要紧，因为你是一个很正常的人。现在笔者来公布答案，结果是：120，这是一个用来计算5的阶乘的例子。

现在来分析一下这段程序：首先将5压入堆栈；然后将宏“[d 1 - d 1 <F *]”压入堆栈；使用“d”命令复制堆栈顶端的内容，也就是刚才定义的宏；将宏转移到寄存器F中保存起来，目前堆栈顶端依然是宏；使用命令“x”执行宏，就开始了循环。继续分析这个宏“[d 1 - d 1 <F *]”的执行过程：首先复制堆栈顶端的数据，也就是5；然后执行5-1的运算得到4；继续复制堆栈顶端的4，执行条件式，判断是否大于1；此时条件为真，执行寄存器“F”中的宏。这时就开始了递归调用，直到堆栈顶端的数据不大于1为止，然后开始递归返回，做逐一的乘法运算，直到堆栈为空。

dc实现循环就是利用这一套机制——递归，来模拟产生。本来理解递归就是一件非常要命的事情，现在又要用它来模拟循环，简直就是在扼杀我们金贵的脑细胞了。不管你怎么认为，反正笔者是这样认为的。

最后在说一下dc比较神奇的地方，就是它能够处理用户的输入，比如刚才的那个计算阶乘的程序，dc允许用户输入要计算数字几的阶乘。代码可以改成这样：

```
$ dc -e "? [d 1 - d 1 <F *] d sF x p"
```

需要注意，这样的程序必须使用选项“-e”给定或通过程序文件给定，不能象前面那些例子使用管道来给定。因为这会导致dc立即退出而报错。注意程序的改动只是将“5”换成了“?”，这是一个dc的命令，用于从标准输入读取内容。这段程序只需要输入一个数字并回车，就会给出阶乘结果并退出。

dc还有一个很不常用的特性，就是数组。使用“:”和“;”两个命令设置和提取。数组的命名规则与寄存器一致。与寄存器不同的是，“:”会一次将堆栈堆栈顶端的两个数据弹出，将最顶端的数字作为数组下标，而另外的数据会作为数组对应下标的值；而“;”命令会将堆栈顶端的数据弹出作为下标，将数组对应下标的数据压入堆栈。比如：

```
$ echo "100 1:s 1;s p" | dc
```

这段代码会输出100这个值。

在本节结束时，笔者提供一个比较完整的例子，给各位读者展示一下dc的超强能力。这是一个用于做长度单位转换的程序，代码超级短小而且用户界面还比较有好，只是有点难读：

```
dc -e '[[Enter a number (metres), or 0 to exit]psj]
sh[q]sz[lhx?d0=z10k39.370079*.5+0k12~1/rn[ feet ]Pn[ inches]P10Pdx]dx'
```

1.2.5 小结

不管是阅读还是编写dc程序，尤其是带有循环的程序，对于现代的程序员都是一件极其痛苦的事情。它很像是一种虚拟计算机的汇编语言，而且又十分别扭。但是dc依然能够存活在目前所有的类Unix系统中还是有它的道理的。至少可以用非常简短的代码来计算非常复杂的数学问题就无出其右者，而且这个世界从不缺乏变态的程序员，就喜欢用dc去虐待一下自己十分充盈的脑细胞。

如果有读者也觉得自己的脑细胞有富余，也可以去尝试一下。只是目前比较权威的dc文档少得可怜，联机帮助文档和info文档可能算是最为详尽的了。
最后要跟各位读者交代一下，笔者所提供的这些例子都是经过语法优化的，已经非常便于阅读了。实际上，dc除了数值需要使用空格或换行来区分，其它一切命令和寄存器都只需要单一的字符表示，这就允许在连续使用命令或寄存器、以及和它们交叉使用数值是，是不需要使用任何分隔符的。而且这样的实际代码非常多，从而使得这些代码非常难以理解。阅读现有代码的技巧就是将所有的独立命令或寄存器操作用空格分割一下，这样可以获得较好的阅读性。

1.3 更为常用的bc

作为更先进的bc，它的语法规则已经跟现代语言没有什么本质的差别了，因此bc的资料和代码也是非常丰富的。

到目前为止，除了Linux系统所附带的GNU的实现版本，其它版本的bc依然是基于dc实现的。而从前面对dc的介绍可以发现，dc很像某种虚拟计算机的汇编语言，那么bc就相当于这种虚拟计算机的高级语言了。而且这种将高级语言转换成对应机器的汇编语言方式，也是目前所有编译型语言的基本转换方式。

既然bc是属于现代意义的高级语言，那么它完整的提供变量、数组、子过程、选择分支和循环等这些基本特性是毋庸置疑的。而且这些特性都是参照C语言来实现的。需要注意的是，那些完全基于dc的非GNU版本的bc与C语言是有很大区别的，最重要的是变量、数组、子过程名只能是单一字符，而且if语句不能有else，有用这些限制的原因就是dc本身的限制。而GNU版本的bc，由于它和dc都是基于另外一种类似Java的虚拟机实现的，也就没有了这方面的限制。

由于本书是专门介绍Linux的，所以本书所讲述的有关bc的知识也是基于GNU版本验证的。但是为了能够让读者所获得的知识更具通用性，笔者特意忽略了那些GNU bc更具实力的特性。大家只要记住，GNU bc更像C就行了。

1.3.1 与dc和C的异同

bc在命令行的语法与dc基本是相同的，与其它大多数Linux命令也一样，所以这里就不做过多介绍了。
相对于dc，bc的程序语法更加易读。比如我们还是计算在介绍dc时给出的公式的值，bc代码是：

```
$ echo "scale=10; sqrt((12+-3^4)/11)-22;" | bc
```

这明显要比dc的代码清晰很多。如果你没有输入错误，这段代码的结果是：-19.0923298925。
同样的，bc也有控制精度的元素，只不过不再是具体的指令，而是使用了一个特殊内置变量“scale”；bc的运算表达式也是我们熟知的代数表示法，负数也不再使用不伦不类的下划线“_”来标注；而用于求平方根的“v”命令，也使用内置函数sqrt来取代。
虽然bc的语法号称是照搬C语言的，但是有时候也会有一些出入。bc的变量不用声明这种非常显著的差别就不用多说了。但是有些特性跟C很像确有着微妙的差别。比如取模运算的“%”，只有在scale为0的时候才与C的行为一致。如果scale大于0，那么会得到一个大于0的最小正值。例如：

```
$ echo "scale=1; 3 % 4;" | bc
```

这时候得到的结果是：.2。因为在精度允许的情况下，3/4的结果是0.7，会有0.2的余数。换句话说，可以有这样的等式：3 = 4 * 0.7 + 0.2。
此外，“^”运算符在C语言中是“位异或”运算符，而在bc中是整数幂运算符。由于bc并不支持位运算，所以C中的所有位运算符也都不被支持。另外，C语言非常著名的三目运算符“? :”也是不支持的，标明语句结束的分号“;”也不是必须的。
特别需要注意的bc中关系运算。比如在C中：

```
a = 3 < 2;
```

会使得变量a的值为0。但是在bc中就不是这样，a的值是2。有这样的行为是因为bc的赋值运算符优先及高于关系运算符。那么等价的代码应该是：

```
a = (3 < 2)
```

表6-3-1按优先级递减顺序列出了bc的各种运算符，与C是有一定区别的。

运算符	说明
++，--	与C的++和--相同，也有前后关系。
-	负数运算符。
^	整数幂运算符。
*，/，%	乘、除和取模。
=、+=、-=、*=,/=,^=	各种赋值运算符。除“^=”外，与C的行为一致
==,<=,>=,!=,<,>	各种关系运算符。

表6-3-1 bc的各种运算符

不知读者们是否注意到了，在所列举的这些例子中，并没有类似dc中“p”的用于“输出运算结果”的命令。bc采用一种按需输出的方式。如果一段表达式的计算结果或一个函数调用的返回值没有赋值给某个变量，那么bc就会将它们输出到标准输出。
依据这个特性，若要bc程序输出一段字符串，在程序中直接写出来就行了。当然，bc中的字符串与C的字符串是极其类似的，需要使用一对双引号“"”括起来。但是bc的字符串不支持C的转义字符。换句话说，如果需要让输出的字符串换行，必须手工输入一个换行符才行（在键盘上按“回车”：）。一般这样的程序都是以程序文件存在的，因为在命令行中很难构建。另外，bc的字符串长度受限制，取决于BC_STRING_MAX的定义，一般是1000。
bc并不具备类似dc的“-e”命令选项，所以bc要么从标准输入中获取程序，要么从程序文件中获取程序。而且这两种方式的执行行为还有些差别。从标准输入中获得的程序，一旦执行到程序末尾，就会立即退出；但是从程序文件中获取程序时，必须遇到“quit”语句才会退出。而且比较遗憾的是，bc程序并不能与用户进行交互，因为“标准”的bc不具备类似dc的“?”命令[1]。
bc程序允许有注释（dc其实也有，以井号“#”开头直到行尾的内容就是注释），这是极其值得让人欣慰的。bc的注释语法与C的完全相同的，是“/* */”。
上述的这些异同点并不完善，笔者仅是在这里做了一点汇总罢了。后面的内容还会遇到一些比较明显的差异，笔者会在实际的论述中给予提示。

1.3.2 标识符

bc程序中可以拥有有三种不同的标识符：普通标识符、数组标识符和函数标识符。所有这三种类型标识符都只能是单个的小写字母。拥有这样的限制是因为“标准”的bc是基于dc实现的。
所谓普通标识符，一般可以理解为变量。变量可分为全局变量和局部变量。函数的参数列表和函数内使用auto语句修饰的普通标识符属于局部变量，其它普通标识符则属于全局变量，可被所有函数使用。所有普通标识符，无论是本地还是全局的，初始值都是0，且都不能保存字符串。
数组标识符后跟一对方括号“[]”。除了在函数参数列表中或被auto语句修饰的时候，数组必须拥有下标。bc中的所有数组都是一维的，且最多可以包含BC_DIM_MAX指定数量的元素，这个值一般是2048。下标从0开始，且必须是整数，所以数组的最大下标值是BC_DIM_MAX-1。数组与变量一样，有全局和局部之分，区分的方法也与变量相同。数组各元素的初始值也是0，也不能保存字符串。
函数标识符必须后一对圆括号“()”，并且有可能包含参数列表。函数分为内置函数和自定义函数两种。只有自定义的函数名才受限于单个的小写字母。
由于普通标识符、数组标识符和函数标识符可以通过其所附带的后缀来区分，因此它们之间即便拥有相同的命名也不会冲突。
除了这些可以自由命名的标识符之外，bc还内置了几个标识符，有些被称为内置变量，有些被称为内置函数。内置变量有：scale、ibase和obase；内置函数有：sqrt、length和scale。
内置变量scale前面已经说过了，与dc的“k”命令拥有相同的作用，就是指定计算结果的精度；而ibase和obase实际上也是与dc的具体命令相关的，分别是“i”和“o”，用于控制输入和输出的进制的。默认情况下，scale的值是0，而ibase和obase则是10。scale不能小于0，最大值取决于BC_SCALE_MAX的定义，

一般是99（说bc是任意精度计算器，是不是有些扯淡：）。ibase和obase也有限制：2<=ibase<=16，2<=obase<=BC_BASE_MAX。BC_BASE_MAX的值一般是99。

内置函数sqrt在前面也介绍过了，就是计算平方根。length这个函数比较怪异。从命名上看是计算长的函数，但是确不能计算字符串的长度，而且计算数值长度的时候，也不会将浮点数的小数点“.”算在内。比如：length(3.14)的结果是3，而不是4。scale函数跟scale变量有些类似，用于获得计算精度。但是它很多时候不会与当前的scale变量值相同，因为它获得的结果是实际数值的小数位的长度。比如：scale(3.14)的结果是2，即便当前scale变量的值是99也没用。

1.3.3 基本语句

bc属于严格意义上的解释执行语言。当一条语句仅是一个表达式，执行该语句后就会将表达式的运算结果输出并换行，除非优先级最低的运算符是赋值。由于字符串不能赋值给任何变量或数组，所以字符串始终都会被直接输出，但是不会自动换行。例如：

```
scale=10;               /* 优先级最低的是赋值，没有输出。        */
"This is a example.       /* 字符串，由于不会自动换行，手工换行。   */
"
r=sqrt((12+-3^4)/11)-22;   /* 优先级最低的是赋值，没有输出。        */
r;                    /* 一个表达式，输出-19.0923298925并换行。*/
a=3<2;                  /* 优先级最低的是<关系运算符，输出0。     */
a;                    /* 一个表达式，输出3。                  */
```

bc语句的分隔符是分号“;”或换行符，按照语句出现的顺序执行。在bc命令的交互式调用中，每次读取一条完整语句且遇到一个换行符，该语句就会被立即执行。

bc只有一种选择分支语句——if。完整语句的语法是：

```
if ( 表达式 ) 语句
或
if ( 表达式 ) {
    语句列表
}
```

当if语句的表达式结果不等于0时，就会执行语句或语句列表。与C一样，当语句列表中只有一条语句时，可以省略花括号“{}”。但是这个时候必须将要被执行的语句和if写在同一行。

bc提供了两种循环语句——while和for。它们的语法是：

```
while( 表达式 ) 语句
或
while( 表达式 ) {
    语句列表
}

for(表达式1;表达式2;表达式3) 语句
或
for(表达式1;表达式2;表达式3) {
    语句列表
 }
```

它们的行为与C是完全一致的，只是需要注意一下在省略花括号时候，被将要被执行的语句必须与while或for在同一行。同样的，break语句提供了强制跳出当前循环的能力，但是continue语句不被支持，总感觉这是一件极其痛苦的事情[2]。

bc可以使用define语句来定义自定义函数。要注意是函数名只能是单个小写字符，所以能定义的函数数量相当有限，不能指望可以使用bc去编写一个多么了不起的程序。define语句的语法是：

```
define L( [参数列表] ) {
    [auto局部变量列表]
    语句列表
}
```

注意这个语法描述中的方括号“[]”并不是语句本身要求的，只是笔者要在这里表述它们是可选的。换句话说，“参数列表”和“auto局部变量列表”都是可选的内容，根据实际需要来使用。

在bc中的自定义函数是不存在名称冲突问题的。换句话说，即便你已经定义了函数a，而不小心又定义了一次函数a，bc是不会抱怨的。在这种情况下，新定义的函数会覆盖掉之前定义的。这种行为能够被允许的根本原因主要是dc是利用寄存器机制来实现宏而导致的。

bc函数的返回与C类似，使用return语句。return语句后面也可以带有一个表达式，表达式的值会作为函数的返回值。但是一个光秃秃的return语句与C就有了明显的差别。C会导致函数返回一个void值，而bc则返回的是0。

为了能够随时终止bc程序的执行，可以使用quit语句。在任何时候，只要执行了quit语句，程序会立即终止。可以将quit语句想象成为C的exit库函数，只是不能决定程序本身的退出状态值罢了。bc也有退出状态值。0依然是带代表成功完成工作；1是代表程序有语法错误；而其它值则没有严格定义。

1.3.4 数学库

为了用户能够比较容易的使用bc构建任意复杂的数学计算算法，bc还提供了一个数学库。数学库提供了很多最基本的数学函数，而且本身也是利用bc语言来编写的。要使用数学库可以使用bc的“-l”命令选项，这个选项不需要指定任何参数。例如代码：

```
$ echo "c(0)" | bc-l
```

会得到结果：1.00000000000000000000。因为函数c是数学库提供的余弦函数。表6-3-4列出了bc数学库中提供的全部函数。

函数	说明
s(x)	正弦函数，参数x的单位是弧度。
c(x)	余弦函数，参数x的单位是弧度。
a(x)	反正切函数。
l(x)	自然对数函数，ln(x)。
e(x)	自然数e的指数函数，ex。
j(n,x)	贝塞尔函数，n为函数的阶，x为参数值。

表6-3-4 bc数学库中的函数

虽然这些函数对于从来没遇到过“好”数学老师的笔者来说，只能做到一知半解。但是对于某些读者来说应该会是如鱼得水的。
当然，要注意的地方是这些函数能够被程序中同名的函数覆盖掉。这也是bc要提供“-l”选项单独去引用它们的一个根本原因。另外，读者们也可以将一些自定义的函数作为一个库给别人使用。比如设计一个能做任意乘方的函数，如代码6-3-4所示：

```
define p(x,y) {
       auto s, i
       s = scale
       scale = 0
       i = y / 1
       scale = s
       if( y == i ) {
          return ( x ^ y )
       }
       return ( e( y * l( x ) ) )
}
```

代码6-3-4 计算任意乘方

这份代码也展示了一些bc编程中的技巧。由于“^”运算符只能计算整数幂，不能满足我们的要求，但是使用数学函数e和l计算有时候效率又不够好，所以采用了这种折中的方法。但是bc中又没有判断整数的方法，也没有取整数的函数，所以利用了变换计算精度的方式，使得除1运算只保留整数位。
将这段代码保存成一个文件，并命名为“power.bc”。然后执行下面的命令：

```
$ echo "scale=10; p(1.2,3.4)" | bc -l power.bc
```

可以得到结果：1.8587296912。我们刚才定义的库函数工作了。从这个命令的表现形式上看，读者们是否能分析出bc在处理程序文件和标准输入的优先级呢？

1.4 总结

通过对两者的分别讲解，读者们是否发现，虽然bc较dc在语言层面上高级很多，但是bc的实际功能却不及dc。这个很难给出一个满意的答案，毕竟笔者跟它们的作者不是一个时代的人，即便是，人家也不一定理咱们不是？但是读者们非要深究的话，我可以做个比喻，就是C与汇编的比较。C比汇编高级，但是有些时候汇编能做的事情，C真的做不了[3]。
到这里，dc和bc的全部内容就算介绍完了。不管怎么样，dc和bc在计算机编程领域都是有非常重要的地位的。功能性、灵活性、乃至针对科学计算的通用性都是毋庸质疑的。至于怎么使用，就留给读者们自由发挥了。但是这么强大的两种语言，实现它们的代码确非常之简单。在最经典的dc的实现代码不超过2500行，这还要算上啰嗦的注释内容。基于dc开发的bc的总代码量依然少的可怜，甚至不足1000行。dc和bc有足够的说服力去说明简单并不等于弱小。这或许就是编程之美的终极表现吧！向发明它们的两位大师致敬。

相关文章

谁用的是windows 2008 换个角度理解正则表达式 独立云磁盘和普通云磁盘相比，磁盘性能会有提升

kideny 2014-07-23 15:15

大神的帖子慢慢看。

林林林林 2014-07-23 15:30

你懂的!

edison_du 2014-07-24 09:14

大牛，您好；大牛，再见。

村里一把手 2014-07-24 10:04

连载啊，牛逼，我记得好像有本书，楼主是不是作者啊！

xiaofanqie 2014-07-24 11:32

楼主就是作者本人，厉害吧 

china2058366 2014-07-24 18:05

占个位置先

西贝庄 2014-07-25 16:49

你是我偶像

净尘 2014-08-30 18:27

厉害，