和加法有相同的优先级以及乘法和除法有相同的优先级而将减法和除法添加到指令集中去。需要注意的是，表达式 a－b－c 应转换成 ab－c－而不是 abc－－。我们的算法进行了正确的操作，因为这些操作符是从左到右结合的。一般情况未必如此，比如下面的表达式就是从右到左结合的：$2^{2^3}=2^8=256$，而不是 $4^3=64$。我们将把取幂运算添加到操作符指令集中的问题留作练习。

方法调用

检测平衡符号的算法提出一种在编译的过程语言和面向对象语言中实现方法调用的方式[⊖]。这里的问题是，当调用一个新方法时，主调例程的所有局部变量需要由系统存储起来，否则被调用的新方法将会重写由主调例程的变量所使用的内存。不仅如此，该主调例程的当前位置也必须要存储，以便在新方法运行完后知道向哪里转移。这些变量一般由编译器指派给机器的寄存器，但存在某些冲突(通常所有的方法都是获取指定给 1 号寄存器的某些变量)，特别是涉及递归的时候。该问题类似于平衡符号的原因在于，方法调用和方法返回基本上类似于开括号和闭括号，二者相同的想法应该都是行得通的。

当存在方法调用的时候，需要存储的所有重要信息，诸如寄存器的值(对应变量的名字)和返回地址(它可从程序计数器得到，一般情况是在一个寄存器中)等，都要以抽象的方式存在“一张纸上”并被置于一个堆(pile)的顶部。然后控制转移到新方法，该方法自由地用它的一些值代替这些寄存器。如果它又进行其他的方法调用，那么它也遵循相同的过程。当该方法要返回时，它查看堆顶部的那张“纸”并复原所有的寄存器，然后进行返回转移。

显然，所有全部工作均可由一个栈来完成，而这正是在实现递归的每一种程序设计语言中实际发生的事实。所存储的信息或称为**活动记录**(activation record)，或叫做**栈帧**(stack frame)。在典型情况下，需要做些微调整：当前环境是由栈顶描述的。因此，一条返回语句就可给出前面的环境(不用复制)。在实际计算机中的栈常常是从内存分区的高端向下增长，而在许多非 Java 系统中是不检测溢出的。由于有太多的同时在运行着的方法，因此栈空间用尽的情况总是可能发生的。显而易见，栈空间用尽常是致命的错误。

在不进行栈溢出检测的语言和系统中，程序将会崩溃而没有明显的说明；而在 Java 中则抛出一个异常。

在正常情况下我们不应该越出栈空间，发生这种情况通常是由失控递归(忽视基准情形)的指向引起。另一方面，某些完全合法并且表面上无问题的程序也可以越出栈空间。图 3-35 中的例程打印一个集合，该例程完全合法，实际上是正确的。它正常地处理空集合的基准情形，并且递归也没有问题。可以证明这个程序是正确的。但是不幸的是，如果这个集合含有 20 000 个元素要打印，那么就要有表示第 10 行嵌套调用的 20 000 个活动记录的栈。一般这些活动记录由于它们包含了全部信息而特别庞大，因此这个程序很可能要越出栈空间。(如果 20 000 个元素还不足以使程序崩溃，那么可用更大的元素个数代替它。)

这个程序是称为**尾递归**(tail recursion)的使用极端不当的例子。尾递归涉及在最后一行的递归调用。尾递归可以通过将代码放到一个 while 循环中并用每个方法参数的一次赋值代替递归调用而被手工消除。它模拟了递归调用，因为它什么也不需要存储；在递归调用结束之后，实际上没有必要知道存储的值。因此，我们就可以带着在一次递归调用中已经用过的那些值转移到方法的顶部。图 3-36 中的方法显示手工改进后的程序。尾递归的去除是如此的简单，以至于某

⊖ 由于 Java 是解释而不是编译执行的，因此本节有些细节不可用到 Java 上，但是一般的概念仍然可以在 Java 和许多其他语言上使用。