到循环的前面。

优化编译器会试着进行代码移动。不幸的是，就像前面讨论过的那样，对于会改变在哪里调用函数或调用多少次的变换，编译器通常会非常小心。它们不能可靠地发现一个函数是否会有副作用，因而假设函数会有副作用。例如，如果 vec_length 有某种副作用，那么 combine1 和 combine2 可能就会有不同的行为。为了改进代码，程序员必须经常帮助编译器显式地完成代码的移动。

举一个 combine1 中看到的循环低效率的极端例子，考虑图 5-7 中所示的过程 lower1。这个过程模仿几个学生的函数设计，他们的函数是作为一个网络编程项目的一部分交上来的。这个过程的目的是将一个字符串中所有大写字母转换成小写字母。这个大小写转换涉及将“A”到“Z”范围内的字符转换成“a”到“z”范围内的字符。

```
1   /* Convert string to lowercase: slow */
2   void lower1(char *s)
3   {
4       long i;
5
6       for (i = 0; i < strlen(s); i++)
7           if (s[i] >= 'A' && s[i] <= 'Z')
8               s[i] -= ('A' - 'a');
9   }
10
11  /* Convert string to lowercase: faster */
12  void lower2(char *s)
13  {
14      long i;
15      long len = strlen(s);
16
17      for (i = 0; i < len; i++)
18          if (s[i] >= 'A' && s[i] <= 'Z')
19              s[i] -= ('A' - 'a');
20  }
21
22  /* Sample implementation of library function strlen */
23  /* Compute length of string */
24  size_t strlen(const char *s)
25  {
26      long length = 0;
27      while (*s != '\0') {
28          s++;
29          length++;
30      }
31      return length;
32  }
```

图 5-7　小写字母转换函数。两个过程的性能差别很大

对库函数 strlen 的调用是 lower1 的循环测试的一部分。虽然 strlen 通常是用特殊的 x86 字符串处理指令来实现的，但是它的整体执行也类似于图 5-7 中给出的这个简单版本。因为 C 语言中的字符串是以 null 结尾的字符序列，strlen 必须一步一步地检查这