作为一个优化示例，考虑图 5-5 中所示的代码，它使用某种运算，将一个向量中所有的元素合并成一个值。通过使用编译时常数 IDENT 和 OP 的不同定义，这段代码可以重编译成对数据执行不同的运算。特别地，使用声明：

```
#define IDENT 0
#define OP  +
```

它对向量的元素求和。使用声明：

```
#define IDENT 1
#define OP  *
```

它计算的是向量元素的乘积。

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       long i;
5
6       *dest = IDENT;
7       for (i = 0; i < vec_length(v); i++) {
8           data_t val;
9           get_vec_element(v, i, &val);
10          *dest = *dest OP val;
11      }
12  }
```

图 5-5　合并运算的初始实现。使用基本元素 IDENT 和合并运算 OP 的不同声明，我们可以测量该函数对不同运算的性能

在我们的讲述中，我们会对这段代码进行一系列的变化，写出这个合并函数的不同版本。为了评估性能变化，我们会在一个具有 Intel Core i7 Haswell 处理器的机器上测量这些函数的 CPE 性能，这个机器称为参考机。3.1 节中给出了一些有关这个处理器的特性。这些测量值刻画的是程序在某个特定的机器上的性能，所以在其他机器和编译器组合中不保证有同等的性能。不过，我们把这些结果与许多不同编译器/处理器组合上的结果做了比较，发现也非常相似。

我们会进行一组变换，发现有很多只能带来很小的性能提高，而其他的能带来更巨大的效果。确定该使用哪些变换组合确实是编写快速代码的“魔术(black art)”。有些不能提供可测量的好处的组合确实是无效的，然而有些组合是很重要的，它们使编译器能够进一步优化。根据我们的经验，最好的方法是实验加上分析：反复地尝试不同的方法，进行测量，并检查汇编代码表示以确定底层的性能瓶颈。

作为一个起点，下表给出的是 combine1 的 CPE 度量值，它运行在我们的参考机上，尝试了操作(加法或乘法)和数据类型(长整数和双精度浮点数)的不同组合。使用多个不同的程序，我们的实验显示 32 位整数操作和 64 位整数操作有相同的性能，除了涉及除法操作的代码之外。同样，对于操作单精度和双精度浮点数据的程序，其性能也是相同的。因此在表中，我们将只给出整数数据和浮点数据各自的结果。

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的未优化的	22.68	20.02	19.98	20.18
combine1	抽象的-O1	10.12	10.12	10.17	11.14