It would be well to test this routine further, for it is still incorrect. Suppose X is negative upon entry, as it may often be. Then TERM is negative, and remains so after it is recomputed just inside the DO loop. So it will be less than E, and an immediate exit will take place. The "sine" computed, needless to say, is worthless. We replace the test by

```
IF (DABS(TERM) .LT. E) GOTO 30
```

and now we have a working sine routine.

Or do we?

Where there are two bugs, there is likely to be a third. Look at the clever expression used to provide alternating signs for successive terms:

```
(-1**(I/2))
```

Is this expression "minus one to an integer power," as desired, or "one to an integer power, with a minus sign in front," which is always minus one? We encountered a similar expression in Chapter 2 that happened to work right. This one happens to work wrong.

Additional parentheses should be used in either case to remove all ambiguity. Even better, the computation inside the DO loop should be rewritten as

```
IF (DABS(TERM) .LT. E) GOTO 30
TERM = -TERM * X**2 / FLOAT(I*(I-1))
SUM = SUM + TERM
```

and the whole issue is avoided. Notice that the new TERM is added to SUM as soon as it is computed, thus ensuring that each TERM computed is accumulated before the loop is exited. This corrects the fourth bug in the routine. We also test for convergence before computing TERM, to eliminate any possibility of underflow should X already be very small upon entrance. This corrects a *fifth* bug.

Surely there is a more orderly way to write a program in the first place. As we suggested in Chapter 3, pseudo-code is a great help, particularly when the target language (Fortran in this case) is less than ideally expressive. We write in our anonymous language

```
sin = x
term = x
i = 3
WHILE (i < 100 & abs(term) >= e)
   term = -term * x**2 / (i*(i-1))
   sin = sin + term
   i = i + 2
return
```

and then translate into Fortran: