show that the same program can be viewed from different perspectives, and that the job of critical reading doesn't end when you find a typo or even a poor coding practice. In the chapters to come we will explore the issues touched on here and several others that strongly affect programming style.

We begin, in Chapter 2, with a study of how to express individual statements clearly. Writing arithmetic expressions and conditional (IF) statements is usually the first aspect of computer programming that is taught. It is important to master these fundamentals before becoming too involved with other language features.

Chapter 3 treats the control-flow structure of computer programs, that is, how flow of control is specified through looping and decision-making statements. It also shows how data can be represented to make programming as easy as possible, and how data structure can be used to derive a clean control flow. Program structure is covered in Chapter 4, how to break up a program into manageable pieces. Considerable emphasis is given in these chapters to proper use of structured programming and sound design techniques.

Chapter 5 examines input and output: how to render programs less vulnerable to bad input data and what to output to obtain maximum benefit from a run. A number of common blunders are studied in Chapter 6, and tips are given on how to spot such errors and correct them.

Contrary to popular practice, efficiency and documentation are reserved for the last two chapters, 7 and 8. While both of these topics are important and warrant study, we feel they have received proportionately too much attention — particularly in introductory courses — at the expense of clarity and general good style.

A few words on the ground rules we have used in criticizing programs:

(1) Programs are presented in a form as close to the original as our typescript permits. Formatting, typographical errors, and syntax errors are as in the original. (Exception: three PL/I programs have been translated from the 48-character set into the 60-character set.)

(2) We regularly abstract parts of programs to focus better on the essential points. We believe that the failings we discuss are inherent in the code shown, and not caused or aggravated by abstracting. We have tried not to quote out of context. We have tried throughout to solve essentially the same problem as the original version did, so comparisons may be made fairly, even though this sometimes means that we do not make all possible improvements in programs.

(3) We will not fault an example for using non-standard language features (for example, mixed mode arithmetic in Fortran) unless the use is quite unusual or dangerous. Most compilers accept non-standard constructions, and standards themselves change with time. Remember, though, that unusual features are rarely portable, and are the least resistant to changes in their environment.

Our own Fortran hews closely to the 1966 American National Standards Institute (ANSI) version, except for our use of quoted Hollerith strings (we refuse to count characters). PL/I programs meet the standard set by IBM's checkout compiler, version 1, release 3.0. Although there are new versions of Fortran and PL/I in sight which will make better programming possible in both of these