any event, but still hard to read simply because it is conceptually short and fat.

The ELSE-IF sequence, on the other hand, is long and skinny as trees go; it seems to more closely reflect how we think. (Note that our revised minimum function was also linear.) It is easier to read down a list of items, considering them one at a time, than to remember the complete path to some interior part of a tree, even if the path has only two or three links. Seldom is it actually necessary to repeat tests in the process of stringing out a tree into a list; often it is just a matter of performing the tests in a judicious order. Yet too often programmers tend to build a thicket of logic where a series of signposts are called for.

Let us summarize our discussion of IF-ELSE. The most important principle is to avoid bushy decision trees like

```
IF ... THEN
   IF ... THEN
   ELSE ...
ELSE
   IF ... THEN
   ELSE ...
```

The bushy tree should almost always be reorganized into a CASE statement, which is implemented as a string of ELSE-IF's in PL/I and as a series of IF's linked with GOTO's in Fortran. The resulting long thin tree is much easier to understand.

A THEN-IF is an early warning that a decision tree is growing the wrong way. A null ELSE indicates that the programmer knows that trouble lies ahead and is trying to defend against it. An ELSE GOTO from such a structure may leave the reader at a loss to understand how the following statement is reached. A null THEN or (more commonly) THEN GOTO usually indicates that a relational test needs to be turned around, and some set of statements made into a group with DO-END.

The general rule is: after you make a decision, *do something*. Don't just go somewhere or make another decision. If you follow each decision by the action that goes with it, you can see at a glance what each decision implies.

Follow each decision as closely as possible with its associated action.

We turn now to the general area of data structure and representation. A program frequently turns out ill-formed or hard to work with because its data representation is inappropriate for the job at hand. We have already seen one small example of this: the change from LOGICAL to INTEGER variables clarified the dating service program of Chapter 2.

Choosing the right data types is usually pretty simple in Fortran, for there are few choices — basically just INTEGER, LOGICAL, and REAL. In PL/I, however, there are many more choices, and accordingly more ways to go wrong. Here is an example where data structure makes all the difference. It is from a program which makes change for an amount DIFF dollars, using $20 bills and smaller denominations. The example is supposed to handle positive values of DIFF up to $10,000;