Programming Style: Examples and Counterexamples

BRIAN W. KERNIGHAN and P. J. PLAUGER

Bell Laboratories, Murray Hill, N. J. 07974

Computer programs can be written many different ways and still achieve the same effect. Until recently, programmers have had little reason to favor one method of expressing code over another. We have come to learn, however, that functionally equivalent programs can have extremely important *stylistic* differences.

Good programming style cuts across application areas, technique and language. Programs written with good style are easier to read and understand, and often smaller and more efficient, than those written badly. Yet few programmers have ever been taught what style is, as we can see from even cursory inspection of their code. Even the techniques of structured programming do not ensure that code will be good; "structured" programs can be just as bad as their unstructured counterparts.

This paper is a survey of some aspects of programming style, primarily expression and structure, showing by example what happens when principles of style are violated, and what can be done to improve programs. To add the ring of truth to our discussion, the examples are all taken verbatim from programming textbooks.

Keywords and Phrases: programming style, structured programming, control-flow structures.

CR Categories: 2.49, 4.0, 4.6

I. INTRODUCTION

Five or ten years ago, if you had asked someone what good programming style was, you would likely have received (if you didn't get a blank stare) a lecture on

1) how to save microseconds.
2) how to save words of memory.
3) how to draw neat flowcharts.
4) how many comments to write per line of code.

But our outlook has changed in the last few years. E.W. Dijkstra [4] argues that programming is a job for skilled professionals, not clever puzzle solvers. While attempting to prove the correctness of programs, he found that some coding practices were so difficult to understand that they were best avoided. His now famous letter, "Go To Statement Considered Harmful" [5], began a debate, not yet completed, on how to structure programs properly. (See [10, 14], for instance.)

Harlan D. Mills [1], using chief programmer teams and programming with just a few well understood control structures (which did *not* include the GOTO), was able to report [12] the on-time delivery of a large application package with essentially no bugs. Clearly, if such results could be consistently reproduced, programming would be raised from the status of black art.

The final word is not yet in on how best to write code. G.M. Weinberg [13], approaching the problem as both a psychologist and a programmer, is studying what people do well, and what they do badly, so we can have a more objective basis for deciding what programming tools to use. Programming languages are still evolving as we learn which features encourage good programming [6]. We have learned that the way to make programs more efficient is usually by changing algorithms, not by writing very tight but incomprehensible code [8]. And people who continue to use GOTOs, out of preference or necessity, are at least thinking more carefully about how they use them [9].

CONTENTS

This paper was typeset by the authors using a Graphic Systems phototypesetter running under the UNIX operating system.

We feel, however, that programming style goes beyond even these considerations. While writing *The Elements of Programming Style* [7], we reviewed hundreds of published programs, in textbooks and in the recent literature. It is no secret that imperfect programming practice is common — we found plenty of evidence of that. There are even bug-infested and unreadable "structured" programs. Writing in teams, using proper structures, avoiding GOTOs — all are useful ingredients in the manufacture of good code. But they are not enough.

Today, if you asked someone what good programming style is, you would (or should!) get quite a different lecture, for we now know that neat flowcharts and lots of comments can't salvage bad code, and that all those microseconds and bytes saved don't help when the program doesn't work. Today's lecture on "What is good programming style?" would probably be more like this...

Expression:

At the lowest level of coding, individual statements and small groups of statements have to be expressed so they read clearly. Consider the analogy with English — if you can't write a coherent sentence, how will you put together paragraphs, let alone write a book? So if your individual program statements are incoherent and unintelligible, what will your subroutines and operating systems be like?

Structure:

The larger structure of the code should also read clearly — it should hang together the same way a paper or a book in English should. It should be written with only a handful of control-flow primitives, such as *if-then-else, loops, statement groups* (begin-end blocks, subroutines), and it probably shouldn't contain any GOTOs. This is one aspect of what we mean by structured programming. Coding in this set of well-behaved structures makes code readable, and thus more understandable, and thus more likely to be right (and incidentally easier to change and debug).

The data structure of a program should be chosen with the same care as the control flow. Choose a data representation that makes the job easy to program: the program shouldn't have to be convoluted just to get around its data.

Robustness:

A program should *work.* Not just on the easy cases, or on the well-exercised ones, but all the time. It should be written to defend itself against bad data from the outside world. "Garbage in, garbage out" is not a law of nature; it just means that a programmer shirked his responsibility for checking his input. Special cases should work — the program should behave at its boundaries. For instance, does the sorting program correctly sort a list with just one element? Does the table lookup routine work when the table is empty?

Efficiency and Instrumentation:

Only now should the lecture on style get around to "efficiency." Not that we don't care how fast a program runs or how much memory it takes, but until we have a working piece of code, we don't always know where it spends its time. And until we know that, talk of changing it "for efficiency" is foolish. Write the whole program clearly. If it is then too slow or too big, change to a better algorithm. (Since you wrote it clearly, change will be easy.) If the algorithm is already the "best," then *measure* the program, and improve the critical parts; leave the rest alone.

Documentation:

If you write code with care in the detailed expression, using the fundamental structures and choosing a good data representation, most of the code will be self-documenting — it can be *read.* Much of the need for detailed flowcharts and comments on every line will go away. And you will have less worry about the inevitable discrepancies between flowchart, comments, and code.

The approach in the "lecture" we just gave seems to lead to better (more reliable, more readable, and usually faster and shorter) code. In this paper we will talk mostly about expression and structure, with occasional digressions to robustness, efficiency, and documentation. Our presentation here is necessarily brief; all of these questions are more fully discussed in [7]. As we did there, to illustrate our points and to show what can go wrong when good style is forgotten, we have chosen a set of "real" programs. Each one is taken verbatim from some programming textbook. (We will not cite any texts by name, for we intend no criticism of textbook authors. We are all human, and it is all too easy to introduce shortcomings into programs.)

The examples are all in Fortran and PL/I; none contain particularly difficult constructions. If you have even a reading knowledge of some high level language you should be able to follow the examples without difficulty. The principles illustrated are applicable in all languages.

II. EXPRESSION

Being too Clever

Our favorite example, the one we feel best underlines the need for something that can only be called good style, is this three line Fortran program:

```
      DO 1 I=1,N
      DO 1 J=1,N
1     X(I,J)=(I/J)*(J/I)
```

It is an interesting experiment to ask a group of student Fortran programmers what this excerpt does. After thirty seconds or so, perhaps a third to a half of them will tentatively agree that they know what it does. Even after a minute of study, there are still puzzled looks. When the group is quizzed, one finds that only a few actually got the correct answer.

What does it do? It relies on Fortran's truncating integer division: if I is less than J, I/J is zero; conversely, if J is less than I, J/I is zero. Only when I equals J do we have a non-zero product, which happens to be one. Thus the code puts ones on the diagonal of X and zeros everywhere else.

Clever? Certainly, but it hardly qualifies as a *good* piece of code in any sense. It is neither short nor fast, despite its terse representation in Fortran. Worst of all, it is virtually unreadable, just because it is too clever for its importance. Even if the trick did happen to prove faster than more conventional methods, it should still be avoided, for initializing a matrix must surely be but a small part of the program that uses the matrix. It is far more important to make the code clear, so people can debug, maintain and modify it.

There is a principle of style in English that says, "Say what you mean, as simply and directly as you can." The same principle applies to programming. We mean that if I equals J, X(I,J) should be 1; if I is not equal to J, X(I,J) should be zero. So say it:

```
      DO 20 I = 1, N
         DO 10 J = 1, N
            IF( I .EQ. J ) X(I,J) = 1.0
            IF( I .NE. J ) X(I,J) = 0.0
10       CONTINUE
20    CONTINUE
```

If this proves to be too "inefficient", then it may be *refined* into a faster but somewhat less clear version:

```
      DO 20 I = 1, N
         DO 10 J = 1, N
10          X(I,J) = 0.0
20       X(I,I) = 1.0
```

It is arguable which of these is better, but both are better than the original. Don't make debugging harder than it already is — don't be too clever.

Being too Complicated

Here is another Fortran example, which is an interesting contrast with the previous one:

```
      IF(X .LT. Y) GO TO 30
      IF (Y .LT. Z) GO TO 50
      SMALL = Z
      GO TO 70
30    IF (X .LT. Z) GO TO 60
      SMALL = Z
      GO TO 70
50    SMALL = Y
      GO TO 70
60    SMALL = X
70    ...
```

Ten and a half lines of code are used, with four statement numbers and six GOTOs — surely something must be happening. Before reading further, test yourself. What does this program do?

The mnemonic SMALL is a giveaway — the sequence sets SMALL to the smallest of X, Y, and Z. Where the first example was too clever, this one is too wordy and simple-minded. Since this code was intended to show how to compute the minimum of three numbers, we should ask why it wasn't written like this:

```
      SMALL = X
      IF( Y .LT. SMALL ) SMALL = Y
      IF( Z .LT. SMALL ) SMALL = Z
```

No labels, no GOTO's, three statements, and clearly correct. And the generalization to computing the minimum of many elements is obvious.

Of course, if our goal is to get the job done, rather than teaching how to compute a minimum, we can write, much more readably than the original, the single statement:

```
          SMALL = AMINO(X, Y, Z)
```

One line replaces ten. How can a piece of code that is an order of magnitude too large be considered reliable? There is that much greater chance for confusion, and hence for the introduction of bugs. There is that much more that must be understood in order to make evolutionary changes.

Clarity versus "Efficiency"

It seems obvious that a program should be clear, yet clarity is often sacrificed needlessly in the name of efficiency or expediency.

```
      DO 10 I=1,M
      IF(BP(I)+1.0)19,11,10
   11 IBN1(I) = BLNK
      IBN2(I) = BLNK
      GO TO 10
   19 BP(I) = -1.0
      IBN1(I) = BLNK
      IBN2(I) = BLNK
   10 CONTINUE
```

If BP(I) is less than or equal to −1, this excerpt will set BP(I) to −1 and put blanks in IBN1(I) and IBN2(I). The code uses a hard-to-read Fortran arithmetic IF that branches three ways, two almost-duplicated pieces of code, two labels and an extra GOTO, all to avoid setting BP(I) to −1 if it is already −1.

There is no need to make a special case. Write the code so it can be read:

```
      DO 10 I = 1, M
         IF( BP(I) .GT. -1.0 ) GOTO 10
            BP(I) = -1.0
            IBN1(I) = BLNK
            IBN2(I) = BLNK
   10 CONTINUE
```

Interestingly enough, our version will be more "efficient" on most machines, both in space and in time: although we may reset BP(I) unnecessarily, we do less bookkeeping. What did concern with "efficiency" in the original version produce, besides a bigger, slower, and more obscure program?

Rewriting

These may seem like small things, taken one at a time. But look what happens when the need for clear expression is consistently overlooked, as in this PL/I program which computes a set of approximations to the integral of X**2 between zero and one, by adding up the areas of rectangles of various widths.

```
TRAPZ: PROCEDURE OPTIONS (MAIN);
     DECLARE MSSG1 CHARACTER (20);
       MSSG1 = 'AREA UNDER THE CURVE';
     DECLARE MSSG2 CHARACTER (23);
       MSSG2 = 'BY THE TRAPAZOIDAL RULE';
     DECLARE MSSG3 CHARACTER (16);
       MSSG3 = 'FOR DELTA X = 1/';
     DECLARE I FIXED DECIMAL (2);
     DECLARE J FIXED DECIMAL (2);
     DECLARE L FIXED DECIMAL (7,6);
     DECLARE M FIXED DECIMAL (7,6);
     DECLARE N FIXED DECIMAL (2);
     DECLARE AREA1 FIXED DECIMAL (8,6);
     DECLARE AREA FIXED DECIMAL  (8,6);
     DECLARE LMTS FIXED DECIMAL (5,4);
         PUT SKIP EDIT (MSSG1)  (X(9), A(20));
         PUT SKIP EDIT (MSSG2) (X(7), A(23));
         PUT SKIP EDIT (' ') (A(1));
      AREA = 0;
             DO K = 4 TO 10;
               M = 1 / K;
               N = K - 1;
       LMTS = .5 * M;
               I = 1;
             DO J = 1 TO N;
               L = (I / K) ** 2;
      AREA1 = .5 * M * (2 * L);
      AREA = AREA + AREA1;
         IF I = N THEN CALL OUT;
           ELSE I = I + 1;
       END;
     END;
 OUT: PROCEDURE;
         AREA = AREA + LMTS;
     PUT SKIP EDIT  (MSSG3,K,AREA) (X(2),A(16),F(2),X(6),
         F(9,6));
      AREA = 0;
     RETURN;
   END;
END;
```

Everything about this program is wordy. The output messages are declared and assigned unnecessarily. There are far too many temporary variables and their associated declarations. The structure sprawls.

Try going through the code, fixing just one thing at a time — put the error messages in the PUT statements where they belong. Eliminate the unnecessary intermediate variables. Combine the remaining declarations. Simplify the initializations. Delete the unnecessary procedure call. You will find that the code shrinks before your very eyes, revealing the simple underlying algorithm.

Here is our revised version:

```
TRAPZ: PROCEDURE OPTIONS(MAIN);
   DECLARE (J,K) FIXED DECIMAL (2),
           AREA  FIXED DECIMAL (8,6);

   PUT SKIP EDIT ('AREA UNDER THE CURVE',
                  'BY THE TRAPEZOIDAL RULE')
                  (X(9), A, SKIP, X(7), A);
   PUT SKIP;

   DO K = 4 TO 10;
      AREA = 0.5/K;

      DO J = 1 TO K-1;
         AREA = AREA + ((J/K)**2)/K;
      END;

      PUT SKIP EDIT ('FOR DELTA X=1/', K, AREA)
                    (X(2), A, F(2), X(6), F(9,6));
   END;
END TRAPZ;
```

Both versions give the same results, so this was not an exercise in debugging in the traditional sense. But if there were a bug localized to this part of a larger system, which version would you rather try to fix? Which would you give a higher mark to? Which would you rather be in charge of, when changes are necessary?

The original version reads like a hasty first draft which was later patched. Arriving at our "final draft" required no great ingenuity; just a series of almost mechanical steps much as we described. Applying the principles of good style, one at a time, gradually eliminates the features that make the original version so hard to read. The problem is, most programs never get past the "first draft" stage, possibly because the code appears too frightening when viewed all at once.

Programmers sometimes say that they haven't time to worry about niceties like style — they have to get the thing written fast so they can get on to the next one. (What actually happens is they get it written fast so they can get on to the tedious job of debugging it.) But you will soon find that, with practice, you spend less and less time revising, because you do a better and better job the first time.

Much more can be said about how to make code locally more readable (see [7]), but for now we will turn to a topic that has recently become popular — how to specify control flow with good style.

III. CONTROL FLOW STRUCTURE

One way to improve the apparently random control flow that several of our examples have demonstrated is to program consciously with just a small set of well-behaved control flow structures. One interpretation of "structured programming," in fact, is this way of coding. Although this is a narrow view, we will keep to just that aspect for the time being.

It has been shown [2] that programs can be written using just:

1) *Alternation,* such as IF-THEN-ELSE, where the ELSE part may be optional.
2) *Looping,* such as WHILE or the Fortran DO loop. Different flavors have the termination test at the beginning or end of the loop.

3) *Grouping*, such as subroutines and compound statements.

While these tools are sufficient, in the same sense that a Turing machine can perform any of a wide class of calculations, it is convenient to add:

4) *CASE switches*, which are essentially multi-way IF statements, and
5) *BREAK and ITERATE statements*, which exit from a loop or skip to the test portion of a loop, respectively.

Most languages have at best a subset of these forms, so the pragmatic programmer cannot hope to avoid the more primitive control statements carried over from earlier days. For example, the simplest way to implement a BREAK in PL/I is to use a GOTO. And in Fortran, of course, GOTOs and statement numbers must be sprinkled liberally throughout the best designed code. But the basic design of a program should be done in terms of the fundamental structures. GOTO's and other primitive language features should be used *only to implement the basic structures outlined above.*

While these are well tried and useful forms, there is a tendency to believe that just by using them (and only them) one can avoid all trouble. This is false — they are not panaceas. Good style, care and intelligence are still needed. We can see this just by studying the use and abuse of the IF-THEN-ELSE, certainly a simple and fundamental structure in any programming language.

Null THEN

The following routine is supposed to sort an array of eight numbers into increasing order of absolute value:

```
DCL A(8);
GET LIST (A);
DO I=1 TO 8;
  IF ABS(A(I))<ABS(A(I+1)) THEN;
    ELSE BEGIN;
      STORE=A(I);
      A(I)=A(I+1);
      A(I+1)=STORE;
      END;
END;
PUT LIST(A);
```

The heart of this sequence is a "DON'T" statement — if the specified condition is true, do nothing, otherwise do something. Anything so misleading should put us on guard; and indeed we see immediately that the sequence cannot possibly sort correctly because

1) only one pass is made over the array, and we know simple sorting takes about N passes.
2) a reference is made outside array bounds when A(I+1) is accessed on the last iteration with I equal to 8.

There are several ways of doing a simple sort correctly. We could make N−1 passes over the array, or we could set a flag every time it is necessary to exchange two elements, so we know that an additional pass over the array is needed. Applying this latter fix to the program above (and eliminating the subscript range error) should give us a working sort.

But there is still a lurking bug. Turning the test around so the IF-THEN is stated more naturally:

```
IF ABS(A(I)) >= ABS(A(I+1)) THEN DO;
   STORE = A(I);
   A(I) = A(I+1);
   A(I+1) = STORE;
   EXCH = '1'B;
END;
```

reveals that two elements will be exchanged even if they are equal. If A contains two equal elements, the program goes into an infinite loop exchanging them, because the flag EXCH will be set repeatedly. Using a null THEN may seem a small thing, until it adds a day of debugging time.

Even when code is correct, it can be very hard to read. Here's another sorting program, which sorts into descending order this time, with an almost-null THEN:

```
     DO  M = 1 TO N;
     K =  N-1;
     DO J = 1 TO K;
     IF ARAY(J) - ARAY(J+1) >= 0
              THEN GO TO RETRN;
                   ELSE;
     SAVE = ARAY(J);
     ARAY(J) = ARAY(J+1);
    ARAY(J+1) = SAVE;
RETRN:    END;
     END;
```

The construction THEN GOTO might be a BREAK statement in disguise, but often it is a tipoff that something is amiss. Here it branches around only three statements and not out of the loop. Why not turn the test around so no GOTO or label is needed? (The null ELSE has no function whatsoever; it only confuses the issue.) And why does the test subtract the two elements and then compare against zero, when a direct comparison would be far easier to understand and free of overflow problems? The program reads like a hasty translation from Fortran into PL/I. Revision is easy:

```
DO M = 1 TO N-1;
   DO J = 1 TO N-1;
      IF ARAY(J) < ARAY(J+1) THEN DO;
         SAVE = ARAY(J);
         ARAY(J) = ARAY(J+1);
         ARAY(J+1) = SAVE;
      END;
   END;
END;
```

The original program worked, but again we were able to improve it with little effort.

In Fortran, there are fewer options when using IFs, for there is no ELSE clause and no way to form compound groups of statements. But in the few cases where the language lets you write clearly, *do so*. Don't write like this:

```
     IF (A(I).GT.GRVAL) GO TO 30
     GO TO 25
30   GRVAL = A(I)
25   ...
```

A branch around the branch that branches around what we wanted to do in the first place! Say what you mean, as simply and directly as you can:

```
IF( A(I) .GT. GRVAL ) GRVAL = A(I)
```

There are now no labels, no GOTOs, and the code can be understood even when read aloud over a telephone. (This is always a good test to apply to your code — if you can't understand it when spoken aloud, how easy will it be to grasp when you read it quietly to yourself?)

ELSE BREAK

The BREAK statement has its uses, but it has to be used judiciously. Consider this sequence for finding the largest of a set of positive numbers:

```
        DCL NEWIN DEC FLOAT (4);
            LARGE DEC FLOAT (4) INIT (.0E1);
            /* .0 x 10**1 = .0 x 10 = 0.0                   */
NEXT_C: GET LIST (NEWIN);
        IF NEWIN >=0
           THEN IF NEWIN > LARGE
                   THEN LARGE = NEWIN;
                   ELSE GO TO NEXT_C;
           ELSE GO TO FINISH;
        GO TO NEXT_C;
FINISH: PUT LIST (LARGE);
```

Ignoring the curious zero in the INIT attribute, and the equally curious explanatory comment, we can see that this program does indeed use just the structures we mentioned above (the GOTOs implement BREAKs and ITERATEs). Therefore it should be readable. But tracing the tortuous flow of control is not a trivial exercise — how does one get to that last GOTO NEXT_C? Why, from the innermost THEN clause, of course.

The ELSE BREAK is just as confusing as the DON'T statement. It tells you where you went if you didn't do the THEN, leaving you momentarily at a loss in finding the successor to the THEN clause. And when ELSE BREAKs are used one after the other, as here, the mind boggles.

Such convolutions are almost never necessary, since an organized statement of the problem leads to a simple series of decisions:

```
        DECLARE (NEWIN, LARGE) DECIMAL FLOAT (4);
        LARGE = 0;
NEXT_C:    GET LIST (NEWIN);
           IF NEWIN > LARGE THEN LARGE = NEWIN;
           IF NEWIN >= 0 THEN GOTO NEXT_C;
        PUT LIST (LARGE);
```

What we have here is a simple DO-WHILE, done while the number read is not negative, controlling a simple IF-THEN. Of course we have rearranged the order of testing, but the end-of-data marker chosen was a convenient one and does not interfere with the principal work of the routine. True, our version makes one extra test, comparing the marker against LARGE, but that will hardly affect the overall efficiency of the sequence. Readability is certainly improved by avoiding the ELSE GOTOs.

THEN-IF

Now consider:

```
IF QTY > 10 THEN                                    /*A*/
   IF QTY > 200 THEN                                /*B*/
      IF QTY >= 500 THEN BILL_A = BILL_A + 1.00;    /*C*/
                    ELSE BILL_A = BILL_A + .50;     /*C*/
                ELSE;                               /*B*/
            ELSE BILL_A = .00;                      /*A*/
```

Those letters down the right hand side are designed to help you figure out what is going on, but as usual, no amount of commenting can rescue bad code. The code requires you to maintain a mental pushdown stack of what tests were made, so that at the appropriate point you can pop them until you determine the corresponding action (if you can still remember). You might time yourself as you determine what this code does when QTY equals 350. How about 150?

Since only one of a set of actions is ever called for here, a frequent occurrence, what we really want is some form of CASE statement. In PL/I, the most general CASE is implemented by a series of ELSE-IFs:

```
IF      cond1 THEN first case;
ELSE IF cond2 THEN second case;
...
ELSE IF condn THEN nth case;
ELSE               default;
```

If there is no default action, the last ELSE clause is omitted. We can rewrite the example as:

```
IF      QTY >= 500 THEN BILL_A = BILL_A + 1.00;
ELSE IF QTY > 200 THEN BILL_A = BILL_A + 0.50;
ELSE IF QTY <= 10 THEN BILL_A = 0.0;
```

Now all we need do is read down the list of tests until we find one that is met, read across to the corresponding action, and continue after the last ELSE. In Fortran, this can be rendered similarly as

```
IF(QTY .GE. 500.0) BILLA = BILLA + 1.0
IF(QTY .LT. 500.0 .AND. QTY .GT. 200.0) BILLA = BILLA + 0.5
IF(QTY .LE. 10.0) BILLA = 0.0
```

which is best if the relations and actions are simple enough to write one per line and the tests are mutually exclusive. Don't let anyone tell you this is not efficient — it doesn't take all that much time to make the whole set of tests, and you're more likely to get the code right the first time. If it does take too much time, and you have measurements that prove it, then and only then should you re-write it with GOTOs.

The THEN-IF was the culprit in this example, but we could have given the disease another name. Note the null ELSE clause, required to make the unstacking come out right when one of the conditions has no corresponding action. These seemingly useless statements cauterize the stumps of any ill-thought-out THEN-IFs buried in the code. A program containing null ELSE clauses is suspect, if for no other reason than that it was written by someone bitten by THEN-IFs often enough to sprinkle null ELSEs around for insurance.

The THEN-IF does have its uses. It is often the only way to ensure that tests with side effects are performed in the proper order, as in

```
IF I > 0 THEN
   IF A(I) = B(I) THEN ...
```

which ensures that I is in range before its use as an index. Some languages provide special Boolean connectives [11] which guarantee left-to-right evaluation and early exit as soon as the truth value of the expression is determined; but if you are not fortunate enough to be able to program with these useful tools, use THEN-IFs and don't forget to cauterize.

Bushy Trees

Most of the IF-THEN-ELSE examples we have shown so far have a characteristic in common, besides the unreadable practices we pointed out. Each approximates, as closely as the programmer could manage, a minimum depth decision tree for the problem at hand. If all outcomes have equal probability, such a tree arrives at the appropriate action with the minimum number of tests on the average, so we are all encouraged to lay out programs accordingly. But a program is a one-dimensional construct, which obscures any two-dimensional connectedness it may have. Perhaps the minimum depth tree is not the best structure for a reliable program.

Let us rewrite the minimum function in PL/I, adhering to the spirit of the original Fortran, but using only IF-THEN-ELSEs:

```
IF X >= Y THEN
   IF Y >= Z THEN SMALL = Z;
   ELSE           SMALL = Y;
ELSE
   IF X >= Z THEN SMALL = Z;
   ELSE           SMALL = X;
```

Even though neatly laid out and properly indented, it is still not easy to grasp. Not all the confusion of the original can be attributed to the welter of GOTOs and statement numbers. What we have here is a "bushy" tree, needlessly complex in 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.

Summary of IF-THEN-ELSE

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

```
IF ...
   THEN IF ...
   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. The resulting long thin tree is much easier to understand:

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

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. And an ELSE BREAK 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 block.

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.

WHILE

Looping is fundamental in programming. Yet explicit loop control in Fortran or PL/I can only be specified by a DO statement, which encourages the belief that all loops involve repeated incrementing of an integer variable until it exceeds some predetermined value. Fortran further insists that the loop body be obeyed once before testing to see whether the loop should have been entered at all.

Thinking in terms of DO statements, instead of loops, leads to programs like this sine routine:

```
      DOUBLE PRECISION FUNCTION SIN(X,E)
C     THIS DECLARATION COMPUTES SIN(X) TO ACCURACY E
      DOUBLE PRECISION E,TERM,SUM
      REAL X
      TERM=X
      DO 20 I=3,100,2
      TERM=TERM*X**2/(I*(I-1))
      IF(TERM.LT.E)GO TO 30
      SUM=SUM+(-1**(I/2))*TERM
   20 CONTINUE
   30 SIN=SUM
      RETURN
      END
```

The program consists entirely of a loop, which computes and sums the terms of a Maclaurin series until the terms get too small or a predetermined number have been included in the sum.

In its most general form, a loop should be laid out as:

```
initialize
while (reason for looping)
   body of loop
```

This way, the parts are clearly specified and kept separate. But this approach was evidently not taken here:

1) The program fails to initialize SUM along with TERM and I.
2) The program mis-states the convergence test, returning immediately on negative values of X.
3) The convergence test is misplaced, so the last TERM computed is not included in SUM. And TERM is computed unnecessarily when the convergence test is met right from the start.

These three bugs can be traced directly to poor structural design. There is also a fourth bug:

4) TERM is computed incorrectly because the "**" operator binds tighter than unary minus (another case of being too clever?).

We first write the code in an anonymous language that includes the WHILE.

```
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:

```
   SIN = X
   TERM = X
   DO 20 I = 3, 100, 2
   IF (DABS(TERM) .LT. E) GOTO 30
      TERM = -TERM * X**2 / FLOAT(I * (I - 1))
      SIN = SIN + TERM
20 CONTINUE
30 RETURN
```

In this case, the WHILE becomes a DO followed by an IF. The DO neatly summarizes the initialization, incrementing, and testing of I, and keeps the loop control separate from the computation. It is a useful statement. The important thing is to recognize its shortcomings and plan loops in terms of the more general WHILE.

In PL/I, the DO-WHILE and DO I=J TO K constructions make the test at the top of the loop, which is most often what is wanted. Fortran programs, on the other hand, frequently fail to "do nothing gracefully" because DO loops insist on being performed at least once, regardless of their limits, even when action is undesirable. For example, this function finds the smallest element in an array.

```
        FUNCTION SMALL(A,N)
        DIMENSION A(1)
        SMALL = A(1)
        DO 1 K = 2,N
        IF(A(K) - SMALL)2,1,1
2       SMALL = A(K)
1       CONTINUE
        RETURN
        END
```

Clearly it's more efficient to use the DO limits of "2,N" — it saves a useless comparison. But what if N is one? Don't kid yourself: N *will* be equal to one some day, and the program will surely fail when it looks at the undefined A(2). Had we first written this routine with a WHILE statement, we would have seen the need for an IF to protect the DO in the translated version. Or, we could have written directly:

```
      SMALL = A(1)
      DO 1 K = 1,N
         IF( A(K) .LT. SMALL ) SMALL = A(K)
    1 CONTINUE
```

This may be less "efficient" in the small, but the cost of finding the bug in the original, and repairing the damage it cost, will certainly outweigh the few microseconds more that our version takes. (You have to weigh for yourself the question of whether to test if N is less than one.)

IV. DATA STRUCTURE

Putting the hard parts of a program into an appropriate data structure is an art, but well worthwhile. (Imagine doing long division in Roman numerals.) This program converts the year and day of the year into the month and day of the month:

```
DATES:  PROC OPTIONS (MAIN);
READ:   GET DATA (IYEAR, IDATE);
        IF IDATE < 1 | IDATE > 366 | IYEAR < 0 THEN RETURN;
        IF IDATE <= 31 THEN GO TO JAN;
        L = 1;
        I = IYEAR/400; IF I = IYEAR/400 THEN GO TO LEAP;
        I = IYEAR/100; IF I = IYEAR/100 THEN GO TO NOLEAP;
        I = IYEAR/4; IF I = IYEAR/4 THEN GO TO LEAP;
NOLEAP: L = 0;
        IF IDATE > 365 THEN RETURN;
 LEAP:  IF IDATE > 181 + L THEN GO TO G181;
        IF IDATE > 90 + L THEN GO TO G90;
        IF IDATE > 59 + L THEN GO TO G59;
        MONTH = 2; IDAY = IDATE - 31; GO TO OUT;
  G59:  MONTH = 3; IDAY = IDATE - (59 + L); GO TO OUT;
  G90:  IF IDATE > 120 + L THEN GO TO G120;
        MONTH = 4; IDAY = IDATE - (90 + L); GO TO OUT;
 G120:  IF IDATE > 151 + L THEN GO TO G151;
        MONTH = 5; IDAY = IDATE - (120 + L); GO TO OUT;
 G151:  MONTH = 6; IDAY = IDATE - (151 + L); GO TO OUT;
 G181:  IF IDATE > 273 + L THEN GO TO G273;
        IF IDATE > 243 + L THEN GO TO G243;
        IF IDATE > 212 + L THEN GO TO G212;
        MONTH = 7; IDAY = IDATE - (181 + L); GO TO OUT;
 G212:  MONTH = 8; IDAY = IDATE - (212 + L); GO TO OUT;
 G243:  MONTH = 9; IDAY = IDATE - (243 + L); GO TO OUT;
 G273:  IF IDATE > 334 + L THEN GO TO G334;
        IF IDATE > 304 + L THEN GO TO G304;
        MONTH = 10; IDAY = IDATE - (273 + L); GO TO OUT;
 G304:  MONTH = 11; IDAY = IDATE - (304 + L); GO TO OUT;
 G334:  MONTH = 12; IDAY = IDATE - (334 + L);
OUT:    PUT DATA (MONTH,IDAY,IYEAR) SKIP;
        GO TO READ;
JAN:    MONTH=1; IDAY=IDATE; GO TO OUT;
        END DATES;
```

What we have here is a bushy tree to end all bushy trees. The rococo structure of the calendar is intimately intertwined with the control flow in an attempt to arrive at the proper answer with a minimum number of tests.

Clarity is certainly not worth sacrificing just to save three tests per access (on the average) — the irregularities must be brought under control. Most good programmers are accustomed to using subprocedures to achieve regularity. The procedure body shows what is common to each invocation, and the differences are neatly summarized in the parameter list for each call. Fewer programmers learn to use judiciously designed data layouts to capture the irregularities in a computation. But we can see that structured programming can also apply to the data declarations:

```
DATES: PROCEDURE OPTIONS (MAIN);
   DECLARE MONSIZE(0:1, 1:12) INITIAL(
      31,28,31,30,31,30,31,31,30,31,30,31,    /* NON-LEAP */
      31,29,31,30,31,30,31,31,30,31,30,31);   /* LEAP */

READ:
   GET LIST (IYEAR, IDATE) COPY;

   IF MOD(IYEAR,400)=0 |
      (MOD(IYEAR,100)¬=0 & MOD(IYEAR,4)=0)
         THEN LEAP = 1;
         ELSE LEAP = 0;

   IF IYEAR<1753 | IYEAR>3999 | IDATE<=0 | IDATE>365+LEAP THEN
      PUT SKIP LIST('BAD YEAR, DATE -', IYEAR, IDATE);

   ELSE DO;
      NDAYS = 0;
      DO MONTH = 1 TO 12
            WHILE ( IDATE > NDAYS + MONSIZE(LEAP, MONTH) ) ;
         NDAYS = NDAYS + MONSIZE(LEAP, MONTH);
      END;
      PUT SKIP LIST(MONTH, IDATE - NDAYS, IYEAR);
   END;

   GOTO READ;
END DATES;
```

Most people can recognize a table giving the lengths of the different months ("Thirty days hath September..."), so this version can be quickly checked for accuracy. The program may take a bit more time counting the number of days every time it is called, but it is more likely to get the right answer than you are, and even if the program is used a lot, I/O conversions are sure to use more time than the actual computation of the date. The double computation of MONSIZE(LEAP,MONTH) falls into the same category — write it clearly so it works; then measure to see if it's worth your while to rewrite parts of it.

Our revised date computation shows an aspect of modularity which is often overlooked. Most people equate modules with procedures, but our program has several distinct modules and only one procedure. A date is input, LEAP is computed, the date is validated, the conversion is made and the result is printed. Each of these pieces could be picked up as a unit and planted as needed in some other environment with a good chance of working unaltered, because there are no unnecessary labels or other cross references between pieces. (The label and GOTO implement a WHILE, done while there is still input.) The control flow structures we have described tend to split programs into *computational units* like these and thus lead to internal modularity.

V. CONCLUSION

Three topics we have hardly touched, which are usually associated with any discussion of style, are efficiency, documentation, and language design. We think these are straw men, almost always raised improperly in a consideration of only parochial issues.

Opponents of programming reform argue that anything that is readable must automatically be inefficient. This is the same attitude that says that assembly languages are preferable to high level languages. But as we have seen, good programming is not synonymous with GOTO-less programming, and it certainly does not have to be wasteful of time or space. Quite the contrary, we find that nearly all our revised programs take no more time and are about the same size as the originals. And in some cases the revised version is shorter and faster because unnecessary special cases have been eliminated.

We use few comments in our revisions — most of the programs are short enough to speak for themselves. And when a program cannot speak for itself, it is seldom the case that greater reliability or understanding will result by interposing yet another insulating layer of documentation between the code and the reader. Bad programming practice cannot be explained away; it must be rewritten.

Finally, many people try to excuse badly written programs by blaming inadequacies of the language that must be used. We have seen repeatedly that even Fortran can be tamed with proper discipline. The presence of bad features is not an invitation to use them, nor is the absence of good features an excuse to avoid simulating them as cleanly as possible. Good languages are nice, but not vital.

Our survey of programming style has been sketchy, for there are far too many details that must be covered to give a proper treatment here. But there is ample evidence for the existence of some discipline beyond a simple set of restrictions on what types of statements to use. It is called style.

REFERENCES

1. F. T. Baker, H. D. Mills, "Chief programmer teams", *Datamation* 19, 12 (December, 1973), 58-61.
2. C. Boehm, G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules", *CACM* 9 (May, 1966), 366-371.
3. O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*. Academic Press, 1972
4. E. W. Dijkstra, "The humble programmer", *CACM* 15 (October 1972), 859-866.
5. E. W. Dijkstra, "Go to statement considered harmful", *CACM* 11 (March, 1968), 147-148.
6. C. A. R. Hoare, "Hints for programming language design", Stanford Computer Science Technical Report CS-74-403 (January 1974).
7. B. W. Kernighan, P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, Inc. (1974).
8. D. E. Knuth, "An empirical study of Fortran programs", *Software - Practice and Experience* 1, (1971), 105-133.
9. D. E. Knuth, "Structured programming with goto statements", *Computing Surveys* 6, 4 (December, 1974).
10. B. M. Leavenworth, "Programming with(out) the GOTO", Proc. ACM National Conference (1972), 782-786.
11. J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *CACM* 3 (April, 1960), 184-195.
12. H. D. Mills, "Top down programming in large systems", in *Debugging Techniques in Large Systems*, ed. by R. Rustin. Prentice-Hall, Inc. (1971), 41-55.
13. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold (1971).
14. W. A. Wulf, "A case against the GOTO", Proc. ACM National Meeting (1972), 791-797.