GOOD PROGRAMS IN BAD LANGUAGES

by: Irving N. Rabinowitz

DCS-TR-#40

To be presented at:
The Third Annual New Jersey Conference on
Use of Computers in Higher Education

March 22, 23, 1976
Rutgers University --

Dept. Computer Science

February 1976

The construction of programs to solve problems of a general kind has been recognized as being an intellectual endeavour of some magnitude, perhaps not yet on a level with the discovery of deep mathematical results, but certainly as taxing as the normal mathematical manipulations of an engineer or statistician, for example. The thought processes involved in the creation of a program, at least on the abstract level, run parallel in some respects to those involved in the solution of a differential equation, say, or the derivation of the properties of an assemblage described by a distribution. In both cases the investigator is concerned with more than simply moving from one step to the logically following one. Indeed, the "logically following one" is so ill-known that its discovery is the center of the intellectual battle. This is true of both programming and mathematics (and nearly all other kinds of work that may involve what can be loosely called "problem-solving").

When we come to compare the weapons that are made available for this intellectual battle, we find a great disparity between those used by the mathematician and those of the programmer. Where the mathematician has notations, theorems, and concepts developed over the entire history of the mathematical sciences, the programmer is working with a relatively slim armamentarium. He has popguns at his disposal, with elephants as his targets. The number of deep and widely applicable theorems in the programming area is still very small, and the notation at the programmer's disposal ranges from the barely adequate to the ridiculous. In the former class we may place such highly-praised super-languages as APL, Algol, Pascal, Lisp, and Snobol. In the latter, the most widely-used of all languages: Basic, Fortran, and Cobol. All of these claim to be programming languages, yet all of them fail as useful tools in various ways.

A language, to be useful, must have several components, all working harmoniously. The two parts of programming languages that have been studied with any effect so far have been their syntax, the rules governing how sentences of the language can be written, and their semantics, the rules which state what such sentences mean. The study of the semantics of programming languages is still in a rather rudimentary state, but the syntactic structure of such languages is much better understood, even though such understanding may not be well applied. In this note we shall concentrate more on the syntactic aspects of programs than on semantic questions. It should be noted, however, that in dealing with what has come to be known as structured programming, the syntactic part is relatively unimportant compared to the meaning that a program must have. It is largely to this end that we deal with the syntactic questions: Paradoxically, it is only by paying attention to the form in which programs are expressed that we can dispel the aura of importance that syntax seems to have. By showing that the syntactic expression is relatively unimportant, we are freed to concentrate on the deeper and more important questions that have to be dealt with in writing programs. In this we see an analogy with mathematics, where good notation may have an importance in helping the mathematician in his work, but where the notation is really secondary to the concepts being represented and manipulated. Similarly, in the construction of programs we would like to be able to concentrate on the concepts, writing "abstract programs", which are then turned into syntactically acceptable programs, preferably by a compiler, but more usually by hand.

One of the difficulties in this approach of going from an abstract program to a concrete one is the question of machine efficiency. A major drawback to the construction of programs is one which does not afflict the

mathematician, namely that where the mathematician can properly use existence arguments, the programmer cannot. Such arguments generally correspond with grossly inefficient searches through large data spaces to find items whose existence we are sure of, but whose locations we do not know. Thus one of the creative acts of the programmer is to find representations of abstract objects which avoid such inefficiencies. For example, the programmer who must deal with a set of objects over which many searches will be performed will choose to represent the set as a hash table, say, so that the searches can be done efficiently, rather than relying upon some abstractly defined operator. Such an operator might be abstractly thought of by considering the set of objects as a mapping from the objects to their attributes, and using the phrase "attribute (object)" to deal with the object in the set. Of course this is not done in practice, partly because of the inherent inefficiency, but also partly because no programming language for which a compiler exists allows such locutions.* Thus the programmer must make a mental leap from the abstraction to the realization, and in this transition can come much grief. Not only is the problem an inherently difficult one, but in complicated cases, there is no easy way to show that a representation has the same properties as an abstract structure.

Such problems are among the most difficult in the programming area, and we shall not even attempt to deal with them here. We limit ourselves to a much simpler and less problematical area, namely the expression of certain methods of program construction in languages which do not have the

---

* Languages like SETL (Schwartz 1974) come close to this ideal, although the problem of efficiency of representation still is a fundamental difficulty.

syntax available for them. We shall concentrate particularly on Fortran, since it is perhaps the syntactically most impoverished language of all those in wide use. The specific rules for conversion of "structured statements" into lower-level groups of statements can be applied in any language. Section 1 discusses the relatively simple translations of the control statements used in structured programs. Such translations are so straightforward that they are easily handled by a compiler. Indeed, the production of Fortran pre-processors, which convert a "structured Fortran" program into a standard one, has become a flood (Meissner 1975). Section 2 exhibits a scheme whereby a recursive routine may be manually rewritten as a standard compilable stacking program.

1. Structured statements

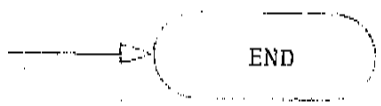
The usual kind of flowchart includes five elementary building blocks, out of which any flowchart may be constructed.* There are the following:

The start box, of which there may be only one in a program. It is common in drawing the chart of a multiple-entry subroutine to have several of these, but this should be thought of as an abbreviation for several different flowcharts which happen to share common code.

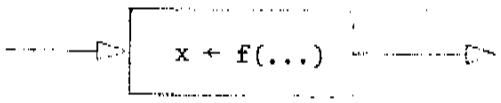
The end box, of which there may be more than one. This indicates the logical end of the program, not necessarily its physical end. Thus in Fortran it corresponds to RETURN (in a subprogram) or STOP (in a main program), rather than END, which is more an end-of-file indicator than a statement of the program.

---

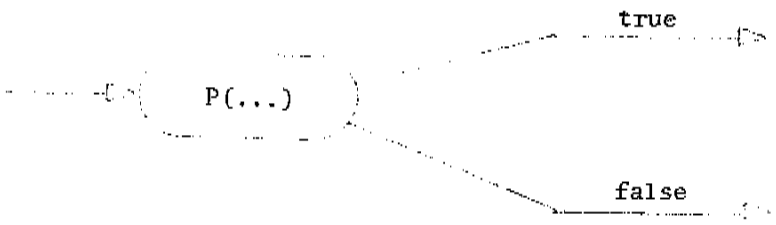
* This does not mean that any program can be represented by such a flowchart. For example, recursive programs and parallel programs cannot be so drawn.



The function box, in which things happen. We indicate it the notation "x ← f(...)", to denote that an arbitrary function of the program variables is computed, and a value assigned to some variable. Of course the f(...) part may be quite complex, and involve a fair piece of program, but the net effect is to act like an assignment statement. Input statements, in which the function is hidden from the programmer, have the same effect in that a value is assigned to a variable. Note that the function box, unlike the start and end boxes, is both entered and exited.

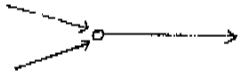


The test box, in which an arbitrary predicate depending upon the program variables is evaluated, and one of two paths is chosen depending on its truth value. (It should be noted that in both the function box and the test box there are no "side-effects". Thus the test box changes no program variable, and the function box changes only the program variable on the left. This incidentally is quite unlike most programming languages, in which side-effects are rampant. However, if we were to draw the flowchart of a program with side-effects, we would have to expand such boxes into sequences of our basic ones.)



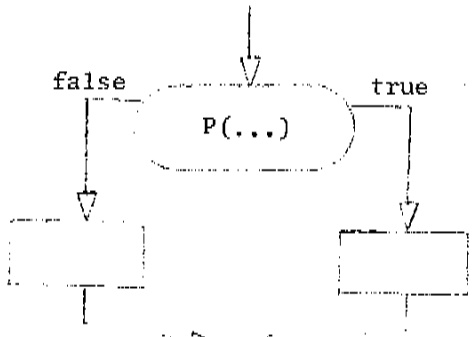
Finally there is the join, in which two arrows of the flowchart come

together. This is the analogue in pictoral form of the existence of a GO TO statement on one or both of the incoming branches.

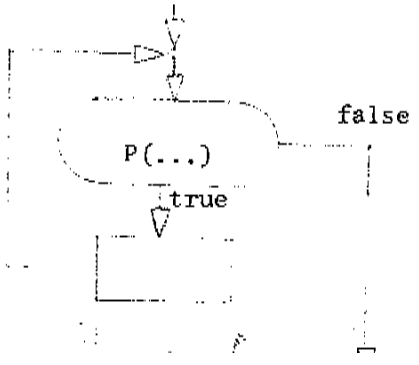


One of the interesting theoretical results about programs (Bohm and Jacopini 1966) which has had a great deal of influence in the practical area of structured programming is that the programs flowchartable by these figures can be simulated by flowcharts in which there exist much more restricted-looking control structures. Thus, the join of control lines is not needed in order to compute what can be done by a program which involves GO TO statements. In addition to the start, end, and function boxes, we need the if-then-else and the do-while, which can be characterized in flowchart form as follows:

if-then-else:



do-while:



In these two charts, the rectangular boxes stand for either a function box, an if-then-else, or a do-while. That is, any one-in, one-out construct can be placed where the rectangle is. This is one of the primary points to be noted about this result, namely that a flowchart (or quivalently, a program) always "goes in one direction", and does not loop back on itself, except through the use of the loop which is part of the do-while. As a consequence, a program in which only these two constructs appear (or in which certain other disciplined control constructs are allowed) can be read from the top of a page to the bottom, without ever having to follow a GO TO from a lower to a higher point on the page. Although this does not appear, on the face of it, to make much difference, it can have far-reaching effects in the practical construction of programs, as well as in the theoretical study of them. It has been claimed, sometimes quite shrilly, that such program structure is a panacea for everything from the software crisis (whatever that is) to dandruff. The truth is somewhat less dramatic. When used in a rational, systematic, disciplined way, these limited control structures can contribute heavily to ease of writing, debugging, and maintaining programs.

Assuming that structured programming is a worthwhile technique, we may properly ask how it can be applied when writing programs in a language in which the two basic structures do not exist. The answer, of course, is to be found in simulation. When a program is conceived in a more-or-less abstract way, the language in which it is to be ultimately expressed tends to have less impact upon its structure than if it is originally to be written in this final form. This is one of the major difficulties that

inexperienced programmers have in constructing programs. Being conversant with (say) Fortran, the overall structure of the program tends to be overlooked in favor of its detailed expression in Fortran terms. This is, in an exaggerated and obvious form, an example of how our language shapes our thought processes. It would be a boon if programmers whose introduction to programming is through the vehicle of a "bad" language were taught to create programs in the abstract form, and then to hand-translate them into the compilable language. (It would be even better if we were to have a very high level language that allows us to construct our abstract programs in an already compilable form, but this is at present only a patch of blue sky, far away.)

Let us consider the simulation of these basic structures. Since Fortran has an IF statement of sorts, there is no problem in writing skeleton programs to realize the two basic figures. For the do-while statement, for example, which in an Algol-like form is

while condition **do** statement

and in PL/I is

```
DO WHILE ( condition ) ;  statement; END;
```

we may write

```
10  IF ( .NOT. ( condition ) )  GO TO 20
    statement
    GO TO 10
20  . . .
```

Although this skeleton realizes the structure, it should be noted that there is a stylistic element in programming which can have as great an effect on the readability of programs as does an author's prose style. Bad programming style can be deadly to the effective construction and use of programs, as a novel whose prose style is recognized to be bad will usually sink without a trace. Some stylistic considerations in our use of structured programming are incorporation of comments in appropriate places, use of indentation to indicate blocks of code, and the clean separation of such blocks from each other. In the present case, the above form would be effectively as unreadable as any Fortran program written from scratch. Although we are simulating a do-while statement, there is no indication that this block of code is indeed such a construction. It would be halpful to the reader if there were some commentary to the effect that statement 10 opens such a group, and that statement 20 is its end. This is easy to do with even minimal commentary. The indication of the body of the loop is best done by indentation, which may quite generally be used to indicate subordination of a group of statements to some control word. As a general rule, the group of statements to be executed under control of a **while** or **if** should be indented, whereas statements to be executed sequentially should share the same margin. Finally, the question of separation of blocks from each other can be done (in Fortran) by the liberal use of the CONTINUE statement. Using these conventions, we can give Fortran a do-while statement as

```
C       WHILE
   10   IF ( .NOT. ( condition ) )  GO TO 20
            statement
            GO TO 10
   20       CONTINUE
C       END WHILE
```

It may seem that all that we are doing here is to write Fortran programs in a particularly stereotyped way, but without making any fundamental change in its structure. In a trivial way this is true, since it is Fortran. In a much deeper sense it is no longer true that it's Fortran, in that the program has been written as an abstract program (or at the very least has been conceived and organized as such), and has simply been translated, quite mechanically, into Fortran. In a sense, we can think of the Fortran realization as the analogue of what a compiler does, namely the creation of unreadable machine code from a program written in a higher-level language. In our case, the programmer is acting as his own compiler, using his abstract program as the higher-level version, and the Fortran as the object code.

The translation of if-then-else is quite as simple as the above. Where the Algol programmer writes

if condition **then** statement-1 **else** statement-2

and where the PL/I programmer writes

IF condition THEN statement-1; ELSE statement-2

the Fortran programmer can write (with no introductory comment needed)

```
      IF ( .NOT. ( condition ) )  GO TO 10
         statement-1
         GO TO 20
C     ELSE
10       CONTINUE
         statement-2
20       CONTINUE
C     END IF
```

It may appear that there are an awful lot of CONTINUE statements here, but this is how we separate blocks. Any later change to the program, as for example a complete rewrite of statement-2, can be isolated from the rest of the program by the CONTINUEs, which effectively serve the purpose of holding nothing but statement numbers. The pieces of program that really do something are kept as blocks that interact as little as possible with each other.*

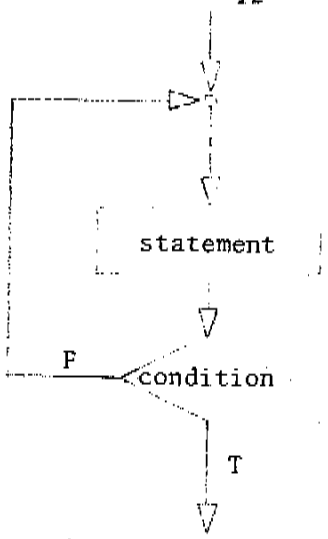
While the do-while and if-then-else statement structures are adequate to write any program, and in some shops are even the only control structures allowed, I feel that a wider vocabulary is useful. The repeat-until statement, which may have the syntactic form

repeat statement **until** condition

has the graphical form

---

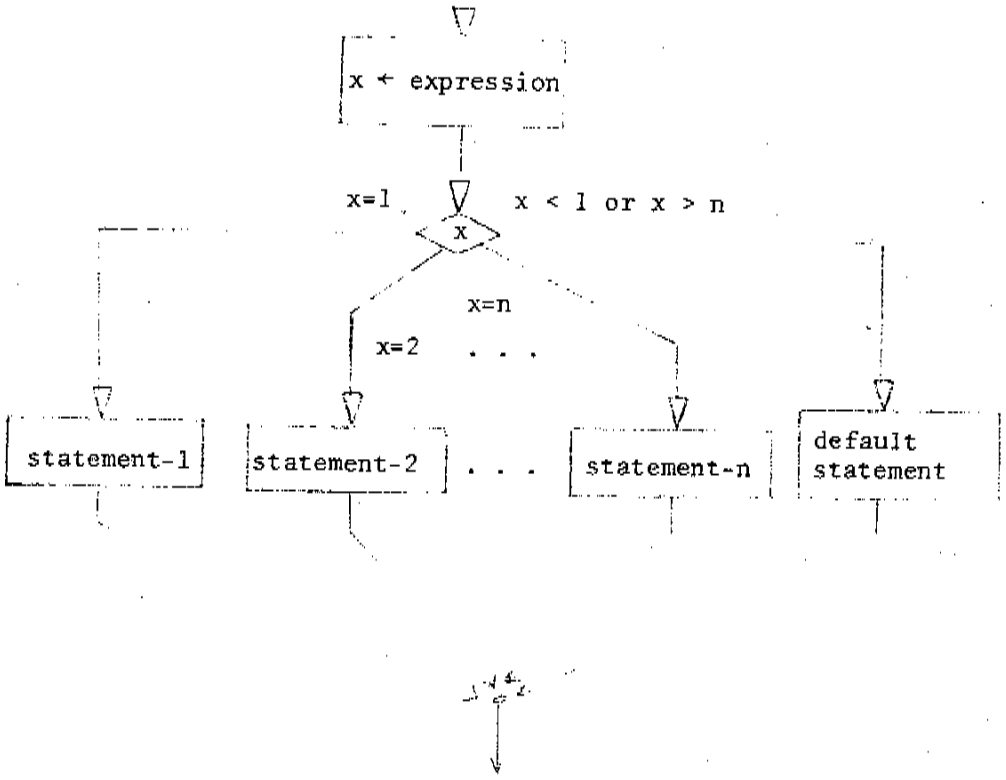
* As an aside, I have found it very beneficial to close every do-loop, including those nested within others, with an explicit and separate labelled CONTINUE. The ability to close several different loops at the same statement has been a cause of grief to Fortran programmers since the beginning of time.



The case statement is rendered differently in almost every language which allows it, but one of its useful forms is

```
select case expression of
        1:  statement-1 ;
        2:  statement-2 ;
             . . .
        n:  statement-n ;
        otherwise
            statement for default situation
        end
```

The meaning of such a statement is described as the evaluation of the expression to yield an integer between 1 and n, and the execution of the appropriately labelled statement, followed by flow of control to the statement following the **end**. If the value of the expression is not in the range 1 to n, the default statement is executed, and control flows past the **end**. Diagramatically,



Many other control structures have been proposed, and some have even been implemented in one language or another. The important point to realize is that when a program is to be constructed, any useful control structure is worth using, and to realize it in Fortran we need simply use the skeleton corresponding to it, in a perfectly mechanical way.

2. Recursion in Fortran

Although recursion is ordinarily not taught in introductory courses in programming, this is more a result of the fact that languages like Fortran and Cobol have no provision for recursive routines than of the inherent difficulty of the concept. Indeed, definition by recursion is an easily understood idea, and few students have any objections to defining things recursively. For example, binary trees may be defined simply as

> A binary tree consists of
>
> either nothing (i.e., no nodes whatever), and is called a null tree,
>
> or it has a node called the root, from which are hung two binary trees, called the left subtree and the right subtree.

When presented this way, it has been my experience that students will swallow it without a murmur, and will be able to deal with such structures in a natural way. For example, traversals of such trees are easily expressed recursively, with an inorder traversal being defined as

> Perform an inorder traversal of the left subtree.
>
> Visit the root (i.e., perform at the root node whatever operations we are to do at nodes of the tree).
>
> Perform an inorder traversal of the right subtree.

Again, the definition seems to cause no trouble to the student.

Once the notion of recursive processes has been introduced, the recursive program can be introduced, and with only a little sleight-of-hand can be made a natural programming technique. The tree traversal, written in an abstract form, is

```
Inorder (T) =
    if T is not null  then
       call  Inorder (Left(T))
       call  Visit(T)
       call  Inorder(Right(T))
```

This description eliminates all the messy programming details of a real program, and we can discuss the implementation questions in a relatively uncluttered atmosphere, introducing the stack implementation quite easily, once the student has grasped the basic concept of a recursive routine. Assuming, then, that we have introduced the ideas of recursion to the student, let us proceed to let him use them by converting his recursive program, which cannot be handled by the Fortran compiler, into a stacking program, which can.

We illustrate the method by transforming a program for the Ackermann function.* This is defined as

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m \neq 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{otherwise} \end{cases}$$

*Although this has no earthly use for a student programmer except to illustrate recursion, it is probably the only non-primitive recursive function he will ever see, and has been a great favorite for many years. Incidentally, it should not be evaluated for any old values of its arguments. The value of $A(4,2)$ is $2^{65,536} - 3$, a number with about 20,000 digits in its decimal representation!

Given a language in which recursion can be expressed, it is straightforward to write a program to evaluate A(m,n) for any values of m and n (except for the overflow problem). For example, PL/I allows us to write

```
A:PROCEDURE(M,N) RECURSIVE RETURNS(FIXED)  ;
  DECLARE (M,N) FIXED;
  IF M = 0 THEN RETURN (N+1);
  IF N = 0 THEN RETURN (A(M-1,1));
           ELSE RETURN(A(M-1,A(M,N-1));
  END A;
```

If Fortran allowed it, we could write a similar program. As it is, we shall write a pseudo-Fortran program, and convert it, by the mechanical application of certain straightforward rules, into a program like that which the compiler produces for the PL/I. In an obvious notational extension of the Fortran language, we get the (noncompilable) program.*

```
00      INTEGER FUNCTION A(M,N)
10      IF (M .NE. 0) GO TO 30
20          A = N + 1 ; RETURN
30      IF (N .NE. 0) GO TO 50
40          A = A(M-1,1) ; RETURN
50      A = A(M-1,A(M,N-1)) ; RETURN
60      END
```

In order to cleanly separate questions of flow of control from those of parameter passing, let us rewrite line 50 so that the two recursive calls to A are explicitly displayed. This calls for a new temporary variable, whose lifetime is obviously very short, and which will be gotten rid of in the sequel:

*With apologies for not following the formatting advice of Section 1.

```
50      I = A(M,N-1)
51      A = A(M-1,1)        ;       RETURN
```

The steps to be done to convert this "recursive" Fortran program, which can't be compiled by an ordinary Fortran compiler, into a Fortran program that can be so handled, are the following:

(1) Bring the values of the parameters into the body of the procedure.

(2) Replace recursive calls which immediately precede RETURNs by assignment of parameter values, followed by a GO TO the beginning of the routine.

(3) Replace other recursive calls by a "push" of all local variables and return address onto a stack, followed by a GO TO to the start.

(4) Combine all RETURN statements into a single one by simply GOing TO one of them from the others. Then replace the surviving RETURN by

   (a) an inspection of the stack to determine whether it is empty. If so, perform a RETURN from the subroutine to the original caller.

   (b) If not, "pop" the local variables and return address from the top of the stack, and GO TO the return address.

(5) Finally, declare the stack as an array of the appropriate size, and define the push and pop operations and the empty predicate.

In detail, the steps are as follows:

Step (1) Because it is often necessary to make assignments to the parameters of the routine in the course of the recursion, we avoid any changes to their outside values by making a copy of these values inside the routine. This is a consequence of the fact that ordinary Fortran uses

the "call-by-reference" method of passing parameters, in which the subroutine is given the address of the parameter, so that any appearance of a parameter on the left side of an assignment causes the value of the actual parameter (in the calling program) to change. A simple way to achieve this is to replace the header of the routine by

```
00      INTEGER FUNCTION A(MN,MN)
01      M = MM ;    N = NN ;
```

Step (2) Recursive calls which immediately precede a RETURN statement don't have to be implemented as calls which cause stacking of arguments. That this is so can be seen by considering the sequence of events that occur in such a call-return combination. Suppose that we are $L \geq 0$ levels deep into the recursion, and that we want to execute a recursive call. Then there occur the following actions:

1. We push locals and return address (which is the address to which we will go after performing the RETURN) onto the stack. Call these locals (L) and return (L) for convenience, since they are the values needed at level L of the recursion.
2. We call the routine with the parameters needed, and also pass to it the place to return to when the next level of recursion finishes. This return address is the address of the RETURN statement immediately following this call.
3. When this next level finishes, it restores locals (L) and goes to return (L+1), i.e., to the RETURN statement.
4. Since level L is now done, it no longer needs any of the locals (L), but simply restores locals (L-1), and goes to return (L).

We can short-circuit this process of pushing and popping by just replacing locals (L) by their new values (or rather by providing the new values of the parameters and letting the routine deal with its locals), and not even bothering to come back to the RETURN, but allowing the next level to return to return (L).

The net result of this is that we can replace the two recursive calls on lines 40 and 51 by

```
40          M = M - 1 ; N = 1 ; GO TO 10
```

and

```
51          M = M - 1 ; N = I ; GO TO 10
```

A couple of points should be noted: The starting point of the routine is line 10, not line 01, as line 01 is to be done only at the original call, not on subsequent calls; and the assignment of the value of the function to the variable A need not be done at these places, since we have a local variable A which holds the result of the function call, and we have access to it at any point of the subroutine. In effect, the variable A is a syntactic artifact of the language meaning something like "A is to hold the value of the function A(...)", not "the value of the function is to be stored in a local variable named A". In our case, this allows us to avoid worrying about having to assign it to the local variable A, since that will be taken care of elsewhere (See step (5)).

Step (3) There is only one remaining recursive call, the one on line 50. Quite mechanically, we could write

```
50          CALL PUSH(M,N,$51)      This saves things
            M = M ; N = N - 1       This sets up the arguments
            GO TO 10                This now calls A(M,N-1)
```

```
51          CALL POP(M,N) ; I = A       The original line 50 has been
                                        completed now
            M = M - 1 ;     N = I;      GO TO 10
```

Although this would be adequate in general, it is obvious that we can write this program a little more neatly. First, since there are no other recursive calls, we don't have to pass the return address to be stacked. We will always return to line 51. Second, the assignment M = M is obviously redundant. Third, the local variable I, which occurs only at the line 51 group of statements, can be eliminated. Finally, since N is immediately assigned a new value after each call, it does not have to be stacked. As a result of these minor optimizations, we can write, instead of the above,

```
50          CALL PUSH(M) ; N = N - 1 ; GO TO 10
51          CALL POP(M)  ; M = M - 1 ; N = A ; GO TO 10
```

For reference, let us put our program as so far transformed into readable form:

```
00          INTEGER FUNCTION A(MN,NN)
01          M = MM ; N = NN
10          IF ( M .NE. 0) GO TO 30
20          A = N + 1 ; RETURN
30          IF (N .NE. 0) GO TO 50
40          M = M - 1 ; N = 1 ; GO TO 10
50          CALL PUSH(M) ; N = N - 1 ; GO TO 10
51          CALL POP(M) ; M = M - 1 ; N = A ; GO TO 10
60          END
```

Step (4) Since there is only one surviving RETURN (the one at line 20), we can do part (a) by replacing it by

```
IF ( "stack is empty" )  RETURN

    "else pop the stack and go to the place which was
     saved in the stack at the last call"
```

The else case has to be expanded in general, when there are several internal recursive calls. In the present program we had no need to save the return address, since there is only one, and furthermore, we have no need to pop anything, since we anticipated this by doing the popping at line 51. Thus part (b) is just the GO TO. All we have to say is

```
IF ( "stack is empty" )     RETURN

   GO TO 51
```

in place of the RETURN statement of line 20.

Step (5) Finally we have to deal with the stack operations. This has been left for last in much the same way that declarations of variables are left for last. We don't know exactly what the stack should look like until we have written the program to deal with it. At this point we are able to say that the stack can be represented as an array of integers, with the top of the stack pointed to by a variable which is an integer. The emptiness of the stack is decided by asking if this pointer is zero, and the stack is initialized to the empty state by setting it to zero. Pushing means advancing the pointer and putting the number to be saved into the stack at that place, and popping is the inverse of this.

Choosing a reasonably large number for the stack size, to allow for a fairly deep recursion, we can summarize this step by the following changes to the program:

```
05          DIMENSION  ISTK(200) ; K = 0
```

This both declares the stack and initializes it to be empty. Then the RETURN statement of line 20 becomes

```
                    IF ( K .EQ. 0)  RETURN
                        GO TO 51
```

CALL PUSH(M) and CALL POP(M) can be replaced by

```
                    K = K + 1 ; ISTK(K) = M
```

and

```
                    M = ISTK(K) ; K = K - 1
```

respectively.

With these changes, the final version of the program is

```
00      INTEGER FUNCTION A(MM,NN)
01      M = MM ; N = NN
10      IF (M .NE. 0)  GO TO 30
20      A = N + 1 ; IF (K .EQ. 0)  RETURN
                        GO TO 51
30      IF (N .NE. 0) GO TO 50
40      M = M - 1 ; N = 1 ; GO TO 10
50      K = K + 1 ; ISTK(K) = M ; N = N - 1 ; GO TO 10
51      M = ISTK(K) ; K = K - 1 ; M = M - 1 ; N = A ; GO TO 10
60      END
```

REFERENCES

1. Schwartz, J.T. (1973) On Programming; Installment II: The SETL Language and Examples of Its Use, New York University Computer Science Department; (Available from Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, N.Y. 10012).

2. Meissner, Loren P. (1975) On Extending Fortran Control Structures to Facilitate Structured Programming, Sigplan Notices, 10, 9, September 1975, 19-29.

3. Bohm, C., and Jacopini, G. (1966) Flow Diagrams, Turing Machines, and Languages With Only Two Formation Rules, CACM 9, May 1966, 366-371.