IMPL EMENTING RECURSION IN FORTRAN

by

Irving N. Rabinowitz*

Technical Report #95

April 1977

* Permanent Address: Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903, U.S.A.

This work was performed while the author was on leave under the Rutgers
Faculty Academic Study Program. This support is gratefully acknowledged.
ABSTRACT

Methods of removing explicit recursion in a systematic way are described, with emphasis on their expression in Fortran, though the techniques are generally applicable. We show how certain local optimization criteria can be applied during the transformation of a program from recursive to stacking form.
We shall describe techniques for converting programs written in recursive form into ones in which no recursive calls appear. The resulting program will perform exactly the same sequence of operations as the original one (except for minor local optimizations that may be made during the conversion process), and so this process should not be thought of as producing an efficient iterative program from a possibly inefficient recursive one. (Indeed, the finding of an efficient iterative algorithm from a recursive definition calls for a good deal of ingenuity on the part of the programmer, and in many cases is the most creative part of the problem-solving process.) It has been shown experimentally [2,4] that both memory space and running time may be reduced by conversions of this sort in Algol and PL/I (both of which allow recursive procedures). Although this is useful, especially when performed mechanically by a compiler or preprocessor, the savings tend to be small compared with the saving possible by an algorithm fundamentally different from the recursive one. A dramatic illustration of this may be seen in Rice's [15] iterative algorithm for the calculation of the Ackermann function. The time and space required for the program are orders of magnitude smaller than even the most optimized version of the recursive program.

A more cogent reason for conversion of recursive programs to ones in which no recursive calls appear (which we may call "stacking programs") is that three of the most widely used programming languages (FORTRAN, COBOL and BASIC) do not permit explicit recursion. As a result, most programmers using these languages either make no use whatever of recursion, thereby giving up one of the sharpest tools in their kit, or rediscover the stack afresh each time an unavoidable recursion shows up. The second approach carries with it all the difficulties of incorporating the book-keeping problems of stacks into the first program to solve the problem, difficulties of assessing the correctness of the program, difficulties of changing it. Examples of such stacking programs may be seen in [8], for example.

A mechanically applicable method of producing such a program from its recursive form is preferable from a number of standpoints. One of the most important is that the recursive program that embodies a recursive definition generally parallels the definition very closely, so that the
correctness of the program is not hard to demonstrate, at least informally. Indeed, in the case of many functions defined recursively, the program is simply a transcription of the mathematical definitions into the locutions of the programming language. Our old friend the Ackermann function is defined \[^{[14]}\] for \(M \geq 0\) and \(N \geq 0\) by

\[
A(M,N) = \begin{cases} 
N + 1 & \text{if } M = 0 \\
A(M-1,1) & \text{if } N = 0 \\
A(M-1, A(M,N-1)) & \text{otherwise}
\end{cases}
\]

A "FORTRAN" procedure to evaluate this is

```Fortran
integer function A(M,N)
integer m,n
if (m.ne.0) go to 10
    A = n+1
    return
10 if (n.ne.0) go to 20
    A = A(m-1,1)
    return
  otherwise
20 A = A(m-1, A(m,n-1))
    return
end
```

One can hardly doubt that this program carries out its intended function. Even a formal proof would be a relatively easy task (if the semantics of Fortran were a little better defined!). In contrast, a stacking version of this algorithm \[^{[8]}\], in Fortran takes 25 statements, including 6 GO TO statements. Its correctness is less easy to see. But if the stacking program were derived from the recursive one by an "automatic program-transformer", its obscurity would not be a handicap. The program documentation need not even include it, but can be restricted to the recursive version. In a sense, the stacking program can be thought of as the high-level analogue of the machine code produced by a compiler. Under ordinary circumstances it would not be incorporated as part of the documentation, since only the computer would want to read it. If our
compiler can make the transformation, we don't need the output version except in machine-readable form. Any changes to the program that have to be made, either because of detected errors or because of changes in problem specification, can be made in the recursive source version. Recompilation (or pre-processing, or manual conversion) then provides a new "object program", with more assurance of its correctness than we would have by making changes to the stacking version. Knuth [11] has pointed out that many of the problems of programming can be overcome with the help of such program-transformers.

The use of a stack for implementing recursive programs (including programs that have block structure as well as recursive calls) is well known in principle, and has of course been applied in compilers since the earliest days. The best description of how this is to be done is still the early classic paper by Dijkstra [7].

In a nutshell, the idea is to maintain a stack, the top level of which contains the values of the parameters and local variables, and a return address for the current activation of the recursive program. All work of this activation is done by accessing this top level. Any recursive calls made by the current activation will use higher levels of the stack (in exactly the same way), discarding them when they are finished. Thus after completion of such a recursive call, the stack will be returned to its state before the call, and the current activation can proceed as if nothing unusual has happened. When it exits, it deletes its information from the top of the stack and exits to the place which had been placed there by the previous call, which activated it. This previous caller is then the current activation, and all proceeds safely until an exit is attempted with the stack empty, at which point the whole procedure stops with a return to the external calling program. The process which we describe is nothing more than a systematic application of these ideas to a program written in explicitly recursive form. The language vehicle is Fortran, though the same ideas hold for recursive programs written in any language.

Knuth [11] has described the process of conversion from recursive to stacking programs in much the same way as we do here. In [13] Knuth
and Schwarcfiter present an extended example of the technique for an Algol program, which is, however, closely intertwined with the actual problem being solved. A similarly extended example for a PL/I program is in Auslander and Strong [2]. Day [6] describes this process for Fortran quite systematically, but without introducing any of the possible optimizations. Darlington and Burstall [5] apply similar and more extended methods in their system.
1. THE BASIC TRANSFORMATION PROCESS

To illustrate the process of transformation we will write a program to evaluate the number of partitions of an integer. The partitions of an integer $N$ are the different ways (disregarding the order of terms) in which $N$ can be expressed as a sum of integers. For example, the integer 4 can be expressed as

$$4$$
$$3 + 1$$
$$2 + 2$$
$$2 + 1 + 1$$
$$1 + 1 + 1 + 1$$

so that the number of partitions is 5. One way to determine the number of partitions is to define a function $Q(M,N)$, whose value is the number of partitions of $M$ in which no term is larger than $N$. The desired number for $N$ is then $Q(N,N)$. This function can be defined as follows [3]:

$$Q(M,N) = 1 \quad \text{for all } N$$
$$Q(M,1) = 1 \quad \text{for all } M$$

For the cases in which neither $M$ nor $N$ is 1,

$$Q(M,N) = \begin{cases} 
Q(M,M) & \text{for } M < N \\
1 + Q(M,M-1) & \text{for } M = N \\
Q(M,N-1) + Q(M-N,N) & \text{for } M > N 
\end{cases}$$

A pseudo-Fortran procedure for this function may be written immediately from the definition (Figure 1):

* The origin of this definition may be seen from the following considerations:
  (a) There is only one partition of the number 1, regardless of the limit on the size of the terms. Thus
      $Q(1,N) = 1 \quad \text{for all } N$.
  (b) Similarly, if the largest term is 1, there is only one partition of any $M$ (namely $1+1+\ldots+1$), so that
      $Q(M,1) = 1$, for all $M$. 
100 INTEGER FUNCTION Q(M,N)
200 IF(.NOT. (M .EQ. 1) .OR. (N .EQ. 1)) TO TO 500
300 Q = 1
400 RETURN
500 IF(.NOT. (M .LT. N) ) GO TO 800
600 Q = Q(M,M)
700 RETURN
800 IF(.NOT. (M .EQ. N) ) GO TO 1100
900 Q = 1 + Q(M,M-1)
1000 RETURN
1100 Q = Q(M,N-1) + Q(M-N,N)
1200 RETURN
1300 END

(c) In the case $M < N$, it is apparent that there are no partitions whatever in which terms larger than $M$ appear. Thus we have
\[ Q(M,N) = Q(M,M) \quad \text{for} \quad M < N. \]

(d) In the case $M = N$, note that if we can find $Q(M,N-1)$, i.e., the number of partitions with terms $\leq M-1$, we know $Q(M,H)$, since there is only one partition unaccounted for, namely $M$ itself. Then
\[ Q(M,M) = 1 + Q(M,M-1) \quad \text{for} \quad M = N. \]

(e) Finally, when $M > N$, we argue so: Enumerate the partitions of $M$ in which no term is larger than $N-1$, to get $Q(M,N-1)$. Which partitions are left? Those which include one or more terms of value $N$. If we now take away an $N$ from each of these, we are left with all the partitions of $M-N$ (in which we can still go up to $N$--- there are one or more $N$'s in the sums). Hence we have
\[ Q(M,N) = Q(M,N-1) + Q(M-N,N) \quad \text{for} \quad M > N. \]

It can be seen that the above definition allows us to evaluate $Q(M,N)$ for all integers $M$ and $N \geq 1$. (a) and (b) are terminal conditions in which no recursion is needed, while (c)-(e) cover the three possibilities for general $M$ and $N$.

This definition can be simplified by combining cases (c) and (d) to get
\[ Q(M,M-1) = 1 + Q(M,M-1) \quad \text{for} \quad M < N. \]

We will not make this simplification, in order to illustrate the effect of an iterative form of recursion.
Note that the conditions in the IF's are negated only to make both the conditions and the textual order of the program the same as those of the definition. Since Fortran has no if-then-else (as yet), and since the only useful statement in a logical IF is a GO TO, the above slightly clumsy program results. The statements are all numbered for reference purposes though we only need statement numbers 500, 800, and 1100.

Let us now proceed to transform this program into a stacking program, step-by-step. Some preliminary changes should be made, partly for cosmetic purposes so that we can avoid a few little details later, and partly to ease the job of seeing where the more substantive changes have to be made. The first of these is trivial:

**Step 1.** Put a label on the first executable statement of the program; let's call it `start`. This will be needed later, since whenever we make a recursive call, we will in effect be saying "go to start of the program and execute it afresh." Of course, if the statement already has a label, don't bother. In the example, `start` is statement number 200.

**Step 2.** Bring the parameters of the routine into its body, so that their values are available for future use. One way to do this is as follows:

Suppose the parameter of the routine is `x` (where just for simplicity we assume one scalar parameter). Change every reference to `x` in the body to `xx`, say, and add the statement "`xx = x`" just after the routine heading. This has the effect of making the parameter `x` a so-called "value" parameter, and in a language like Algol is accomplished by specifying the parameter as "value x". The renaming of `x` is then done by the compiler. In Fortran it must be done by the programmer.

A much easier way of accomplishing the same thing is to leave unchanged all references to `x` in the body, and change the name of the dummy variable in the heading. It usually saves a lot of writing to replace the heading statement by

```
FUNCTION f(xx)
  x = xx
```

In our example we write

\begin{align*}
100 & \text{ INTEGER FUNCTION } Q(MM,NN) \\
M & = MM \\
N & = NN .
\end{align*}

The reason for bringing the parameters into the routine is that during the course of the recursion we need not only to have access to the current values of the parameters, but also to be able to change these values. That is, we are going to sprinkle assignment statements of the form "\( x = \ldots \)" throughout the body. If the parameter is treated, as it is in Fortran, as a "reference", each such assignment will result in changing the value of the actual parameter in the calling program.*

This is ordinarily an undesirable state of affairs, as it leads to non-obvious side-effects in a program, and can be disastrous, particularly when the actual parameter is a constant. In Fortran this has long been recognized as a problem that could crop up because of a programmer's error.

Thus, consider the following statements in a program.

\begin{verbatim}
CALL SUB(1) \\
I = 1
\end{verbatim}

with the subroutine \texttt{SUB} defined as

\begin{verbatim}
SUBROUTINE SUB(N) \\
N = 2 \\
RETURN \\
END .
\end{verbatim}

Clearly an error of one kind or another has occurred. The effect of the error is that the storage location (in the calling program) which held the number called "\texttt{I}" , now holds a different number, the one that ought to be called "\texttt{2}". Because of this, when the "\texttt{I = 1}" statement is

* In some Fortran implementations the assignment of the new value to the actual parameter does not take place until exit from the routine, but regardless of what compiler is used the actual parameter is going to get changed.
executed, the value that \( l \) receives is 2, not 1, and the programmer receives a rude shock when the program bombs in a most inexplicable way.

This odd behavior is not restricted to the case of a constant argument, but shows up in other, more reasonable-appearing cases. It is not uncommon for an argument to be used as a counter for a loop in the subroutine, so that a statement of the form \( X = X = 1 \) may be repeatedly executed. By the time the routine has finished, the value of the actual argument (the variable in the calling program) is nowhere near its original value, and if the author of the calling program is unaware of the fact that the routine changes its argument, he too will be surprised.

In some languages, this problem is easily overcome. Algol, for example, allows the author of the subroutine to isolate a parameter from the calling program by specifying it as a "call-by-value" parameter. Some more modern languages, like Algol 68 and Pascal, require specification of all parameters, and inconsistent uses can be spotted during compilation. Indeed, in such languages the standard method of parameter passing is by value, and it is the type or mode of the parameter that determines whether assignment may be made to it. Thus an integer parameter will have a value passed to the called program while a parameter declared as reference-to-integer will have the value of the address passed, thus allowing it to be used as a changeable quantity.

**Step 3.** If recursive references occur within expression, "parse" the expressions so that each such reference sits alone on the right side of an assignment. (In SUBROUTINES this is not a problem, as all references occur alone in CALL statements.) For each occurrence of \( f(a) \) in an expression, introduce a statement \( f = f(a) \) where as usual, the name on the left side is that of the FUNCTION. In case several calls occur in the same expression, we have to introduce other local variables to save values of \( f \). For example, if we have

\[
a = f(b) + f(c) \times f(d),
\]

we might replace it by the sequence of statements

\[
f = f(b); \quad f1 = f; \quad f = f(c); \quad f2 = f; \quad f = f(d); \quad a = f1 + f2 \times f.
\]
Two points should be noted here: First, it is not advisable to compress the last two statements into the single one

\[ a = f_1 + f_2 * f(d). \]

Although there is no real harm in doing so, it forces us to keep some irrelevant detail in mind, particularly the location of the result of the call \( f(d) \), and the location to which to return after making the call. It is better if \( \text{all} \) calls of \( f \) are replaced by the \( "f = f(a)" \) form, just to avoid confusion. Second, why did we write two statements for each call, where one would do? That is, why not have written

\[ f_1 = f(b); \quad f_2 = f(c)? \]

The reason for this is a little more subtle, reflecting the fact that the routine we are playing with is both a calling program and a called program. As a called program, its value, by language convention, is to be stored in the variable named \( f \). As a calling program, its value is in an otherwise anonymous variable which we denote by \( "f(a)" \). The first we take care of during the writing of the program, by making sure that the value of the function is stored into \( f \) at some point. The second can easily be taken care of by a compiler, since it has access to the anonymous location. We are not so lucky, because the whole point of this exercise is to be able to get rid of the denotation \( "f(a)" \) and replace it by statements that have the same effect, but are acceptable to the language in which we are writing. In effect, we must know, when we write \( "f(a)" \), where this value is. But we see that it can only be in the variable called \( f \), and nowhere else. Thus we are forced to explicitly move the value from \( f \) to \( f_1 \), while a compiler that accepts recursive calls knows where the result is and can arrange the move behind the scenes.

Another place where expressions should be transformed is when the recursive call occurs as part of the test value in an \( \text{IF} \) statement. If the recursive function occurs in the expression \( e \) in the statement

\[ \text{IF (e) } \ldots, \]

it should be parsed into
\[ v = e \]
\[ IF (v) \]

where \( v \) is a local logical variable. Then the above expression transformation may be applied to \( e \) outside the IF-statement.

In our example, we rewrite statements 900 and 1100 (not 600, as that is already in the appropriate form):

\[
900 \quad Q = Q(M, M-1) \\
\quad Q = 1 + Q
\]

and

\[
1100 \quad Q = Q(M, M-1) \\
\quad Q1 = Q \\
1110 \quad Q = Q(M-N, N) \\
\quad Q = Q1 + Q.
\]

Of course, having introduced a new name, \( Q1 \), we have to declare it by

\[
\text{INTEGER} \quad Q1
\]

at the beginning of the program.

The above three steps have not changed the program in any serious way, and in particular, it can be seen that they have not changed the value computed by the program. Step 1 has no effect whatever. Step 2 may make a difference to whatever program uses our routine for \( f \), but has no effect on the value of \( f \). The third step is simply doing the

\[ \text{INTEGER} \quad Q1 \]

* It may be argued that this very change vitiates our entire process. The whole idea is to make transformations in such a way as to ensure that all values computed by the program are unchanged, and we have already made a change, at least to the calling program. This really is not strictly true, in the sense that a *properly constructed* recursive program will not change the values of its parameters [9], so that Step 2 has no effect on such a program. If a recursive routine does change its parameters in the course of its computation, then we must fall back on the argument that it should not. For a number of reasons, some practical, some theoretical, functions ought not change their parameters. Among the practical reasons is that a calling program should be able to have confidence that what a function sub-program does is to compute a value, and not have the programmer worrying about what
same job that the compiler does, just more explicitly. One more such inexhaustible, though useful, change will be given as Step 5 below. Step 4 is the first of the substantive changes.

Step 4. Replace any calls to $f$ which immediately preced a RETURN as follows:

For $f = f(a)$ write $x = a$
RETURN
GO TO start,

where start is the label put on the first executable statement of the program in Step 1. For our example, only statements 600 and 700 have this configuration, and they may be replaced by

600 $N = M$
700 GO TO 200.

(To be perfectly precise, we should write $M = M$ for the setting of the first argument; but of course we can take short cuts when we see them.)

It is not obvious that this transformation preserves the effect of the program. A theoretical justification may be found in [17], where it is shown that a recursive scheme of this sort (the so-called "iterative" form) is translatable into a looped flowchart in just this way. We can see the propriety of the change by appealing to the stacking implementation of recursion.

As mentioned previously, the current activation of $f$ has access to the top of a stack in which are stored the values of its parameters (say $\pi$), the values of its local variables (call them $v$), and its return address, $r$. The state of affairs just before executing the call to $f(a)$ is shown in Figure 2(a). We are currently at a depth $n$ in the recursion, i.e., there have been $n$ calls that haven't yet been completed. The place to which the current activation is to return is $r_n = Sr'$, and the value of the

\[ y = \pi + f(\pi) \]

it does to his other variables. If $\pi$ is changed by $f$, a statement of the form "$y = \pi + f(\pi)$" can become a loser in a big way. A theoretical reason for avoiding side-effects of this sort is that it becomes extremely difficult to prove that a program is correct when they are present.
(local) variable \( f \), which is not kept on the stack in this scheme, is whatever its current value is, say \( f_n \). To execute the call, we add an \((n+1)\)th level to the stack, containing \( x_{n+1}^* = a \), space for the \( x_{n+1} \), and \( r_{n+1} = S_r \), to get Figure 2(b), and then go to start. It is now crucial to realize that the effect of the call to \( f(a) \) must be only to put the value of \( f(a) \) into location \( f \), and to get rid of anything on the stack above level \( n \). If this is properly done, then Figure 2(c) shows the state after completion of the call, when control has been sent to \( S_r \). Finally, since \( S_r \) is just the return statement, we pop level \( n \) off, and go to \( S_r' \), with \( f \) still holding the value of \( f(a) \) (Figure 2(d)).

Now consider what happens when the transformation of Step 4 is made; i.e., start with things as shown in Figure 2(a). Then instead of adding to the stack, we replace \( x_n \) by \( a \), to get Figure 2(e), and go to start. As before, what happens at levels of the stack above \( n \) won't affect level \( n \) and below, and at the completion of the calculation of \( f(a) \), its value will be in \( f \). At exit, level \( n \) will get popped, and control will go to \( r_n = S_r' \) (Figure 2(f)). We see now that both paths lead to exactly the same state of affairs, namely a stack with \( n-1 \) levels in it, and control at \( S_r' \).

In short, we are saying that since we won't need any local variables after the call to \( f(a) \), simply overwrite the current level of the stack and instead of then doing pop \( n+1 \), return to \( n \), pop \( n \), return to \( n-1 \), just do pop \( n \), return to \( n-1 \).

There is a little sleight-of-hand being pulled, as the stack won't look exactly as we have shown it. Actually, the top level of the stack is kept in the variables local to the routine, except for the return address \( r \), which really is on the stack. Furthermore, \( r_n \) (the place to which the depth-\( n \) activation is to return) is not on level \( n \), but on level \( n-1 \). This is not absolutely necessary, but is done to make the handling of the code more convenient (as will be seen in Step 6b). The effect is exactly the same as if we put \( r_n \) onto the \( n \)th level, and accessed it in a slightly messier way. The only question to worry about is that of level zero, which does not exist. "Level zero" holds the return address to which the procedure is to go after completing all its work, i.e., the address in the program which
originally called \( f \). We access it by executing a real \( \text{RETURN} \) statement when the stack becomes empty.

**Step 5.** This is another cosmetic change, to save writing at the next stages, and consists of getting rid of all but one \( \text{RETURN} \) statement by changing all but one (call it \( \text{finish} \)) to "\( \text{GO TO finish} \)". The only reason to make it the fifth step rather than the fourth is that Step 4 gets rid of some \( \text{RETURNs} \) itself. Which \( \text{RETURN} \) is chosen as the \( \text{finish} \) statement is a matter of taste. I prefer to make it the \( \text{RETURN} \) from the first non-recursive alternative. In the example, we will use 400 as the \( \text{finish} \) statement, and change 1000 and 1200 to:

\[
\begin{align*}
1000 & \quad \text{GO TO} \quad 400 \\
1200 & \quad \text{GO TO} \quad 400
\end{align*}
\]

At this point the program has been transformed to the form shown in Figure 3, which is still in pseudo-Fortran. We now proceed to make the changes to the program to turn it into executable Fortran. These steps involve deciding what needs saving on and restoring from the stack, which is the basic requirement, and choosing a reasonable representation of the stack within the confines of Fortran's very restricted data representation capabilities.

**Step 6a.** Replace each statement of the form

\[
\begin{align*}
\text{Sc} & \quad f = f(a) \\
\text{Sr} & \quad \text{return}
\end{align*}
\]

by the sequence of statements

\[
\begin{align*}
\text{Sc} & \quad \text{save } x, v, r \\
& \quad x = a \\
& \quad \text{GO TO start} \\
\text{Sr} & \quad \text{restore } x, v.
\end{align*}
\]

In this, \( x \) represents the arguments of the function, \( v \) the local variables; \( r \) is an integer which will be used to select statement number \( \text{Sr} \) out of a computed "GO TO", and should therefore be between 1 and the number of statements of the form \( f = f(a) \). Note that while we save \( r \),
on the stack prior to making the next recursive call into a loop, we do not bother to restore it after we return. We will, when returning, use \( r \), which is on the same stack level as \( x \) and \( v \). Since we will have no further use for it, it can be deleted from the stack in the return statement. This, incidentally, is why it is convenient to make \( r \) the last element of the `save` list, as we will see in Step 6b.

In a compiler's translation of a recursive call, all the parameters of the call will be put on the stack, as will all the local variables (though that costs nothing, as it is merely a compile-time allocation of local variable space). We don't have to move everything for our transformation, but only those parameters and local variables which will still be used between the return to \( Sr \) and the first assignment to them after that. These are the so-called "live" variables. The detection of live variables in our example is easy enough to do by a casual inspection. In more complicated routines it is not so easy to see what needs to be saved. We will discuss this question in Section 5:

In the example, we have three calls to transform. The call at 900 doesn't need either \( M, N \), or \( Ql \) to be saved, as none of them will be used between returning and exiting. Thus we write

\[
900 \text{ save } 1 \\
\text{ } \\
N = M-1 \\
\text{GO TO 200.} \\
\]

\[901 \text{ restore} \]

The call at 1100 needs to save \( M \) and \( N \), since they are both used after the return and before the exit. However, we don't have to save \( Ql \), as that is defined later. So,

\[
1100 \text{ save } M, N, \text{ and } 2 \\
N = N-1 \\
\text{GO TO 200} \\
\]

\[1101 \text{ restore } M \text{ and } N. \]

The call at 1110 requires the saving of \( Ql \), since that is
active before and after the call, but neither \( M \) nor \( N \) needs to be saved:

\[
\begin{align*}
&M = M - N \\
&\text{GO TO } 200 \\
&\text{restore } Q1.
\end{align*}
\]

Step 8b. Now that we have taken care of the stacking and unstacking of values, the part of the stack manipulation process that remains is what to do at exit from the procedure. The only surviving RETURN statement (the one we have called \textit{finish}) is to be transformed to

\[
\begin{align*}
&\text{finish} \quad \text{IF (stack is empty) RETURN} \\
&\text{IRETRN} = \text{top element of stack} \\
&\text{GO TO} \quad (51, 52, \ldots, 5k), \ I\text{RETRN}.
\end{align*}
\]

If there is nothing saved on the stack, and it is time to exit from the procedure, then it must be the case that there are no incomplete calls, so we simply execute the RETURN statement, which finds the (otherwise inaccessible) address to which to return in the calling routine. If the stack is not yet emptied, then there are recursive calls still undone, and the number that indicated the place to which to return has been put at the top of the stack by the most recently executed \textit{save}. So we go there. There are \( k \) possible return points within the procedure, selected by the integers 1 through \( k \), selecting the statement numbers \( 51 \) through \( 5k \), respectively. Each of these corresponds to one of the statements \( f = f(a) \).

A convenient local optimization can be used in the special case where there is only a single recursive call in the entire procedure, a circumstance which occurs often enough to be paid attention to. In this case, we don't have to select a return point, as there will only be one, say \textit{Sreturn}. In this case, the \textit{finish} coding is just

\[
\begin{align*}
&\text{finish} \quad \text{IF (stack is empty) RETURN} \\
&\text{GO TO } \text{Sreturn}.
\end{align*}
\]
Now that the stack has been introduced, it remains to represent it. Although there are many ways to do this, the simplest method for a language like Fortran is to use an array to hold the saved values, together with a pointer to the last-used element of this array. If the pointer is called $K$, then adding an element to the stack means storing into stack $(K+1)$ and increasing $K$ by 1; taking an element off means moving it from stack$(K)$ to its destination and decreasing $K$ by 1; testing for emptiness is just asking whether $K$ is zero; and the element at the top of the stack is in stack$(K)$.

**Step 7a.** Assume that STACK is a one-dimensional array large enough to hold everything that we may ever wish to put into it. Then the four stack operations are done by:

1. Replace
   "save $x_1, x_2, \ldots, x_n$"
   by the statements
   \[
   \begin{align*}
   &\text{STACK}(K+1) = x_1 \\
   &\text{STACK}(K+2) = x_2 \\
   &\vdots \\
   &\text{STACK}(K+n) = x_n \\
   &K = K+n
   \end{align*}
   \]

2. Replace
   "restore $x_1, x_2, \ldots, x_n$"
   by the statements
   \[
   \begin{align*}
   &K = K-n \\
   &x_1 = \text{STACK}(K+1) \\
   &x_2 = \text{STACK}(K+2) \\
   &\vdots \\
   &x_n = \text{STACK}(K+n)
   \end{align*}
   \]

3. Replace "stack is empty" by "$K = 0$".

4. Replace "top element of stack" by
   "STACK$(K)$ ; $K = K-1$".
In our example we will have

900  STACK(K+1) = 1  "save 1"  (10)
     K = K+1

901  CONTINUE  "restore"  (11)

1100  STACK(K+1) = M  "save M, II, and 2i"
      STACK(K+2) = N
      STACK(K+3) = 2
      K = K+3

1101  K = K-2  "restore M and N"  (13)
      M = STACK(K+1)
      N = STACK(K+2)

1110  STACK(K+1) = Q1  "save Q1 and 3"
      STACK(K+2) = 3
      K = K+2

1111  K = K-1  "restore Q1"  (15)
      Q1 = STACK(K+1)

400  IF (K .EQ. 0) RETURN  (16)
     IRETRN = STACK(K)
     K = K-1
     GO TO (901, 1101, 1111), IRETRN

One exception to this rule may occur. The recursion may be such that no stacking at all has to be done. This may occur when (a) there is only one internal recursive call, so that we don't have to stack a return address; and (b) none of the arguments or local variables have to be used after the call. (This corresponds to the linear, simple, no multiple subconsequent case of [18].) Although there is now no stack to maintain, it is still necessary to know the depth of recursion, so that we can terminate properly. The test for "empty stack" is still "K .EQ. 0", but the "save" and "restore" coding degenerates, not to "K = K+0", and "K = K-0", but to "K = K+1" and "K = K-1". The finish coding is still as described in Step 6b.
Step 7b. Declare the stack as an array, and initialize it to "empty" by

\[
\text{INTEGER STACK(large enough)} \\
K = 0. 
\]

The question of how large is "large enough" will be discussed subsequently. It cannot be determined on purely syntactical grounds (as all the other transformations could be), but depends on knowing something about what the program does, not simply what it looks like.

The final version of the program, which we get from Figure 3 by applying (7) - (17), is shown in Figure 4. (Some slight liberties have been taken to put more than one statement on a line.)
2. DEALING WITH THE STACK

(a) Size  The question of how large is "large enough" is one which cannot be answered by a purely syntactic analysis of the program. In all but the most special cases of recursion, the depth of recursion (which is essentially the same as the size of the stack) depends on the arguments of the procedure. In principle, we would like to have an open-ended stack, limited only by the amount of storage available. In languages with recursion, this is generally available, as the compiler can arrange for interaction with the operating system at run time. In a language like Fortran, with purely static, compile-time allocation of data space, the problem becomes considerably harder. We have to know how much stack space we will need, and are in the usual dilemma: Too much space allotted causes the cost to go up, too little causes the program to fail.

In the present example, the amount of storage needed for a call to \( Q(n,n) \) is \( 4n-5 \) integers (for \( n \geq 2 \)). For the Ackermann function the space needed for \( A(m,n) \) is just \( A(m,n)-2 \) (for \( m \geq 1 \)), an exceptionally large number for \( m \geq 4 \) (and pretty big even for \( m = 3 \)).

While there is no real substitute for a detailed analysis, in practice it is sometimes not available. A safeguard against the grisly consequences of a size estimate that is too small is an overflow test at each insertion [12]. If the recursion demands more space than has been dimensioned, such a test can at least allow an orderly retreat rather than a complete (and often inexplicable) rout. Notice that no test for underflow is needed in our particular use of a stack.

(b) Data Types  In the example, the only data in the stack were integers. Frequently enough, especially in non-numerical work, we find it necessary to save quantities of different types, e.g., integers and reals. If this were all, there would be little problem (in Fortran), as the usual dodge of equivalencing arrays, say STACK and ISTACK, can be used. In the more general situation, where we may have to stack objects which don't take the same amount of space, we have to resort to different stack representations than the simple array. In some other languages where dynamic storage allocation is possible, a one-way linked list is a convenient way to stack objects of different types, since we can usually attach an object of almost
any shape to a list. In Fortran, the restriction to arrays can be overcome by a similar trick, though at the rather heavy cost of simulating a storage management scheme for a large array space.

For the conversion of a recursive to a stacking program, the problem is somewhat simpler, since we don't need quite the generality afforded by a dynamic allocation scheme. For a particular program we have only a limited number of data types, say $T_1, T_2, \ldots, T_n$. Suppose, for example, that in the $Q(M,N)$ program, we had declared $Q$ as a REAL FUNCTION, and the local variable $Q_1$ as REAL. Then we would have to stack integers and reals, at different times. Leaving out the possibility of equivalencing a real and an integer array, we can deal with the stack in two ways:

1. Keep two stacks in parallel, with a single stack pointer, putting integers into $\text{ISTACK}(K)$ and reals into $\text{RSTACK}(K)$; or
2. Keep two independent stacks, with a pointer for each, storing integers into $\text{ISTACK}(IK)$ and reals into $\text{RSTACK}(RK)$.

The first representation has the drawback of leaving a lot of unused space in one or more of the stacks, while the second requires the maintenance of two counters, and either a closer analysis of the needed stack size or an overflow test at each insertion. It is not especially obvious that the $4N$ places needed for the $Q$ program are split between $2N$ integers, $N/2$ reals, and $3N/2$ addresses.

(c) Locating the Stack. We have described a situation where we could easily declare the stack as a local quantity, but in more general cases, it may be desirable to make it global, i.e., in COMMON. The question comes up when we have more than one recursive subroutine to deal with, since a single subroutine has no need of COMMON. As described in [7], only a single stack is necessary for the realization of recursion (as well as block structuring and expression evaluation), so that putting one (or several, as described in (b) above) into COMMON is adequate for even the most complex recursions. However, whether a variable goes into COMMON is more a matter of overall program design than one of implementing a specific technique. In the case of several non-interacting recursive routines it may be preferable from the standpoint of modularity to give each routine its own stack.
If a single stack in COMMON is used for the separately recursive parts of a large program, the treatment of the stack must be modified from the description in Section 1. Consider the situation where, among the subroutines of a large program, we have a recursive routine \( f(x) \), and a recursive routine \( g(x) \) such that \( f \) is self-contained (as the program is), while \( g \) calls upon \( f \) as a "closed" subroutine. That is, although both are recursive, neither interacts with the other except as calling and called routines. In this case, suppose \( g \) has recursed to level \( n \) in the common stack, and then executes a call to \( f \). If \( f \) has its own local stack, all proceeds as before, and no special action has to be taken. But if \( f \) is to use the same stack as \( g \) (and by implication, the same stack pointer, which is also in COMMON), the initialization of the stack within \( f \), and the testing for empty must be treated a little more carefully. In effect, \( f \) must note where the current top of stack is when it is entered, and must leave when the top returns to this point. With the counter \( K \) in COMMON, initialization within \( f \) should not be "\( K = 0 \)", but

\[ K_{SAVE} = K, \]

...to record the top of stack position at entry. The test for completion ("stack is empty") is then

\[ K \text{ .EQ. } K_{SAVE} \]

rather than "\( K \text{ .EQ. } 0 \)".

An alternative to this is to give \( f \) its own counter, say \( K_F \), to count levels of recursion. The counter \( K \) is then used as before for the stacking and unstacking, but doesn't need initializing or testing. Initialization is to "\( K_F = 0 \)" and the ending test is "\( K_F \text{ .EQ. } 0 \)".

Still another alternative strategy is to condense \( f \) and \( g \) into a single routine. These questions are pursued further in Section 4.
3. **SUBROUTINES**

(a) **Result and update parameters.** In illustrating the transformations on FUNCTION subprograms, much was made of the fact that the function should not change its arguments, as part of the general principle of avoiding unwanted side-effects. This is not the whole story, as we very often want, indeed need, to modify the values of parameters. In SUBROUTINE subprograms, or non-typed procedures in other languages, the most important way to communicate results back to the caller is through the list of parameters. (Another very important — and a lot more dangerous — way is to alter the values of global or COMMON variables.) Fortunately, the treatment of recursive SUBROUTINES can be reduced to much the same kind of transformations as we described for FUNCTIONs. The basic idea is to consider each output value as the result of a function evaluation, so that we can treat them as the outputs of FUNCTION subprograms [10].

A result, or output, parameter is one which is not used in the routine until it has been given a value within the routine. That is, in any call to the subprogram, the parameter is the location into which a value is to be put, not where it already exists. An update parameter is one which has a value at the call (as usual), but which is changed during execution of the subprogram. Needless to say, in any call, these parameter positions can hold only variable names, not expressions or constant values. Result parameters are a special case of update parameters, and allow a tiny bit of optimizing during the conversion to a stacking program.

Consider a subroutine $s$, with value parameter $x$ (input only), and two variable parameters, the update parameter $y$ and the output parameter $z$. In the body of $s$ there is a recursive call:

```
SUBROUTINE s(x,y,z)
  ...
  CALL s(a,u,v)
  ...
END.
```

At the internal call the variable $u$ has been assigned some value, but $y$ has not, and $u$ and $v$ are different variables. $a$ is an expression,
After the call, \( a \) will be unchanged, but \( u \) and \( v \) will have been assigned new values. The treatment of the argument \( x \) is precisely the same in all respects as in the case of a FUNCTION. To handle the arguments \( y \) and \( z \) we can think of the subroutine as effecting the calculation of a couple of function values which are assigned to these outputs. In the calling program, where \( p \) and \( q \) are different variables,

\[ \text{CALL } s(e,p,q) \]

can be considered as assigning some values to \( p \) and \( q \) by the simultaneous evaluation of the conceptual or fictitious function calls

\[ <p = Y(e,p), \ q = Z(e,p)> . \]

(The angular brackets don't have any formal meaning, but are simply a way of reminding us that these are not sequential function calls of two separate functions.) Taking this point of view lets us deal with the arguments of \( Y \) and \( Z \) as value parameters so that subroutines which return values through parameters can be treated almost like functions that don't.

The internal call to \( s \) should then have the same effect as

\[ <u = Y(a,u), \ v = Z(a,u)> . \]

The only difference between this and a recursive call of a FUNCTION \( f(x) \) (aside from the conceptual simultaneity) is the lack of a name for the place where the result appears. In the FUNCTION case, the result is in a location named \( f \), which was used exactly like any other local variable. Here we don't have locations named \( Y \) and \( Z \) automatically manufactured for us, but we do have \( y \) and \( z \) that can be used for this purpose. Using the convention that \( Y \) and \( Z \) leave their results in \( y \) and \( z \) respectively, the internal call should behave like

\[ <y = Y(a,u), \ z = Z(a,u)> \]

immediately followed by:

\[ <u = y, \ v = z> . \]
The replacement of CALL s(a,u,v) by this sequence is the analogue of Step 3 of the basic transformation process, and lets us handle the rest of the conversion almost exactly as before. The major difference will be that y and z have to be treated a little more carefully than f in the FUNCTION, since we don't have access to addresses of objects in the transformed code. Because of this, we will have to consider y and z as being 'used' at each RETURN statement, which will have as its primary consequence that they may be found to be live at a call even if the original text of the subroutine does not mention them between the CALL and any ensuing RETURN. The reason for the sudden appearance of a use of y and z at the return is that we have to use a "value-result" type of parameter passing. Rather than being able to directly use the address of u and v in the internal call, we have to go through the assignment to y and z first. Thus, to protect y and z from unwanted changes, we will move the original value of y in, and at the end, move final values of y and z out.

The transformation process is then as follows. We parallel the steps previously described.

Step 1 is the same.

Step 2 is the same, except for output arguments, which only need to have their names changed. Thus the heading of s can be changed to

\[
\text{SUBROUTINE } \bar{s}(\bar{x}, \bar{y}, \bar{z}) \\
\bar{x} = \bar{x} \\
\bar{y} = \bar{y} \\
\bar{z} = \bar{z}
\]

Step 3 is the consideration of recursive CALL statements as the conceptual function calls and assignments.

Step 4 doesn't apply except for SUBROUTINES which have no update or result arguments. In effect, the creation of the assignments following the "function" calls eliminates such iterative calls. The exceptions are those subprograms that only change COMMON variables or have only side-effects, such as printing. For these, Step 4 is the same.

Step 5 is the same.
Step 6a is slightly different, in that the "call"

\[ \text{Sc: } \langle y = Y(a,u), z = Z(a,u) \rangle \]

can't be separated from the assignments

\[ \text{Sr: } \langle u = y, v = z \rangle. \]

In the FUNCTION case it could be since \( f \) never had to be stacked. Here we must take care to restore values to \( y \) and \( z \) after we have made the assignments. Thus we save and restore by

\[ \text{Sc: } \]
\[ \text{save live variables and } r \]
\[ x = a \]
\[ y = u \]
\[ \text{GO TO start} \]
\[ \text{Sr: } \]
\[ u = y \]
\[ v = z \]
\[ \text{restore live variables.} \]

Step 6b is modified also, to take into account the movement of update and output values to the calling program. Instead of writing

\[ \text{finish: } \text{IF (stack is empty) RETURN,} \]

we assign the variables before RETURNing, by

\[ \text{finish: } \]
\[ \text{IF (stack is not empty) GO TO finish 1} \]
\[ y = y \]
\[ z = z \]
\[ \text{RETURN} \]
\[ \text{finish1: } \text{RETURN = ...} \]

This is the reason that \( y \) and \( z \) must be considered to be used at all RETURN statements (all RETURNs lead to finish).

Step 7 is the same,

(b) Arrays as Stackable Items. In constructing an algorithm, it may be very natural to introduce arrays which have to maintain their values across
recursive calls. Although this offers no problems in principle - they get stacked like any other live local variables - there is the obviously undesirable cost of moving a large object back and forth between stack and local space. We can derive a bad stacking program from a bad recursive one.* An array may be just another variable from the point of view of recursion removal, but it pays to consider it as a special kind from the stand point of efficiency.

We have been considering the process of conversion as a relatively mechanical one, not considering the questions of efficiency except locally. This is partly due to the principle that an inefficient correct program is almost always more useful than an efficient wrong one. That is, if we have what appears to be a correct algorithm (however we have decided that), our conversion method lets us derive an equally correct program, so that we don't have to worry so much about debugging the resulting program as if we had written it from scratch. On the other hand, if we can work at the algorithm level, where things are not so cluttered with language dependencies, it is often possible to make major changes without complicating things to the point where we have doubts about the correctness.

In the case of a recursion that uses arrays (or sets or trees or any other big things) whose values have to be saved from time to time, a useful thing to investigate is whether the arrays can be dealt with in a way that makes their handling more efficient. One feature that recurs over and over in such algorithms is the fact that large objects tend to be handled by small updates at each level of recursion, rather than being completely changed. Often what we're doing in a recursion is dealing with a data object whose structure is naturally defined recursively, in such a way that the transition from one state of the object to the next involves only a small addition to or deletion from or change to the object. In such circumstances it can be very profitable to separate the object from the process that deals with it. In effect, the object being computed is global to the routine computing it, and those parts of the object that don't change between levels of recursion don't get saved, but just sit quietly out of the way.

* Just as we did in Section 1.
Let us look at a standard example of a recursive routine, the inorder traversal of a nonempty binary tree. This is similar to Knuth's "sturdy toddler" (Example 6 of [11]), but with a major difference: We want a subroutine that does not print the nodes, but produces an array of these nodes as its output. A naive, but very natural statement of this algorithm is

"Recursively call the routine to get the list of nodes in the left subtree; add to it the name of the root; call again to get the right subtree's list; add this to the list being constructed as output, and exit."

Suppose the tree is represented by three integer arrays in COMMON, giving the node number in NAME, and the node numbers of the left and right children in LEFT and RIGHT respectively. We can assume a declaration like

\[
\text{INTEGER NAME(100), LEFT(100), RIGHT(100)}
\]

\[
\text{COMMON NAME, LEFT, RIGHT .}
\]

If there is no subtree on that side of node, \( I \), then \( \text{LEFT}(I) \) or \( \text{RIGHT}(I) \) is zero, where \( I \) is the subscript of ("pointer to") the tree. A pseudofortran program to realize the above algorithm is that of Figure 5. (The IF-THEN, ELSE, and END-IF statements have obvious meanings, and may even get into the new Fortran definition. The array statements are abbreviations of DO-loops. We use this notation just to avoid avoidable obscurity.)

Applying our conversion rules in a straightforward way will cause us to save only \( I \) (and a return address) at the first call, but \( \text{ASIZE} \) and all of \( A \ (1: \text{ASIZE}+1) \) (as well as the return) at the second, since they are both live at the call. On top of this, since the result of a call is always in \( A \) and \( \text{ASIZE} \), by our conventions on fictitious functions, we also have to move them into \( B \) and \( \text{BSIZE} \) after the return from the second call. There is altogether too much hustle and bustle in a program as simple as this.

The key to avoiding all the furniture-moving is to recognize that the essential work of the routine \( IN \) is adding the name of the root to what has been gathered from the left subtree. That part of the array is already
complete and unchangeable. Using the whole array as a parameter is just forcing us to save things that aren't changed, except by our construction. The appropriate solution to our problem is to think of the output array \( A \) as being a global array, and the effect of \( \text{IN} \) being to add a node name to it. (This view is already implicit in Knuth's presentation, where our array corresponds to the output file being lengthened by a \( \text{print} \) statement at each recursive call.) Doing things this way, we can recast our algorithm to

"Assuming \( A \) and \( \text{ASIZE} \) as global, recursively call \( \text{IN} \) to fill up the first part of \( A \) with the nodes of the left subtree; add the name of the root; call again to fill in the rest of \( A \) with the right subtree's nodes; exit."

By declaring something like

```plaintext
INTEGER A(100), \text{ASIZE}
COMM\(\text{ON A, ASIZE,}
```

the whole program of Figure 5 collapses to

```plaintext
SUBROUTINE \text{IN}(T)
INTEGER T

1 IF (LEFT(T) .EQ. 0) GO TO 2
   CALL \text{IN} (LEFT(T))
2 A (\text{ASIZE+1}) = \text{Name}(T); \text{ASIZE} = \text{ASIZE+1}
3 IF (RIGHT(T) .EQ. 0) GO TO 4
   CALL \text{IN} (RIGHT(T))
4 RETURN
END
```

An external initialization of \( \text{ASIZE} \) to zero is needed, after which the recursive subroutine fills out the array, one node at a time. By the nature of the data structure and the order in which it is traversed, we fill \( A \) in the proper order, from low subscripts to high.
(c) Statement Labels as Parameters; RETURN n Statements. Don't use them. In Fortran, they can't be manipulated as data objects, and in other languages they tend to institutionalize the practices so abhorrent to "go-to-less" programming. The only use for an alternate return from a subroutine to some remote place is to indicate an irreparable failure of the whole program. If that happens, take a dump and go home.
4. MUTUAL RECURSION

We have, up to now, concentrated on how to deal with a single self-contained (as far as control is concerned) recursive program. A more general situation that can occur is where we have a group of mutually recursive routines. A group of routines $f_1, f_2, \ldots, f_n$ is mutually recursive if, for any pair $f_i$ and $f_j$, $f_i$ can call $f_j$ (possibly through a chain of calls), and $f_j$ can call $f_i$ (ditto). More precisely, let us symbolize "$f$ calls $g$ explicitly" by

$$f \rightarrow g.$$

Then "$f$ calls $g$", symbolized by

$$f \rightarrow * g,$$

if there is a chain $h_1, \ldots, h_m$ such that

$$f \rightarrow h_1 \rightarrow \ldots \rightarrow h_m \rightarrow g.$$

The group $f_1, \ldots, f_n$ is mutually recursive if for any pair $f_i$ and $f_j$ ($1 \leq i, j \leq n$),

$$f_i \rightarrow * f_j \quad \text{and} \quad f_j \rightarrow * f_i.$$

Handling such systems of routines is not essentially different from what has been described already. In short, we will convert such a group into a single subprogram to which the basic transformations can be applied. Each such subprogram can be handled independently, as ordinary non-recursive routines, or all the subgroups of a larger group can be lumped together into one grand program. This latter route is not recommended.

(a) An Example. As an example of such programs we will use a "recursive descent" parser for a very small fragment of a programming language. To avoid cluttering up the example we write the syntax in the form of diagrams. [19], in Figure 6. A valid "program" in this language can be generated by following the arrows starting at "program". When a box is met, switch to that diagram, returning when the exit arrow is followed. Words in boxes
with round ends and symbols in circles are terminal symbols (Three boxes, namely "identifier", simple type", and "constant", can be considered as being defined by other such diagrams whose details are unimportant.) When the exit arrow from "program" is followed, a valid program has been generated.

These diagrams can also be used to parse a given string. Each diagram corresponds to a function which tries to match its sequence of boxes against a part of the input string, by calling the functions corresponding to those boxes. If the function succeeds in the match it returns the value .FALSE. (no error has been found), otherwise .TRUE. Each box for a defined type corresponds to a statement like

\[ \text{IF}(\text{IDENT}(x)) \ldots \]

which is a call to the function IDENT(x) which tries to match the portion of the input string, starting at the point being scanned currently, against its definition of an identifier. Terminal symbols correspond to calling the non-recursive function \( \text{RECOG}(x) \) which tests if its argument is exactly matched by the input string at the current point. If so it advances the input pointer past the part that matched, and returns .FALSE. Forks in paths correspond to forks in the programs, and loops in paths correspond to loops in the programs.

The programs for "type" and "field" are shown in Figure 7, and illustrate all of these features. We have shown an input argument \( x \) for each function, and indicated the argument of each call by the expressions \( a(x), b(x), \) etc. This is only a bit of pedagogical license, as no arguments are needed for this simple parser's functions. However, since we do have to deal with arguments in more general cases, it is worthwhile having them here, so we can see what must be done to handle them.

We have nine functions corresponding to these diagrams, some of which are recursive. In addition, we have four non-recursive functions for "identifier", "simple type", "constant" and "recognition" of terminals. Nothing need be done with these. The nine functions defined as in Figure 7 must be analyzed and converted.
(b) Detection of Mutual Recursion. To decide which of our nine functions need to be converted, we first have to find out who depends on whom. Of course we could take all of them as a single group, assuming them all to be mutually recursive (as a compiler could do without half trying), but this would be more work than necessary for our manual conversion technique, and would also introduce unnecessary stacking operations at run time. It is much more efficient to treat the program in pieces.

A straightforward method of finding the groups of mutual recursions is by analysis of the dependency graph of the functions. The nodes of this graph are the names of the functions, and an arrow proceeds from node \( x \) to node \( y \) if function \( x \) calls function \( y \) explicitly. Our nine functions have the dependency graph of Figure 8, which does not include any of the functions known \textit{a priori} to be non-recursive. This kind of diagram can be constructed quite easily from even a large group of functions, especially when they have some rational underlying structure that we know about. Figure 8 does not look like Figure 9, which happens to be precisely the same, because Figure 8 was constructed from the syntax diagrams in an orderly way, while Figure 9 started with an alphabetical list of the nine nodes which were then connected.

The definition of mutual recursion can be interpreted in terms of the dependency graph. The relation \( f \rightarrow g \) can be interpreted as "an arrow goes from node \( f \) to \( g \)" and \( f \rightarrow \ast g \) as "there is a path from \( f \) to \( g \)". Then a set of routines is mutually recursive if their nodes form a subgraph, each of whose nodes is reachable from any other by some path. These sets of nodes are the "strong components" of the dependency graph, and there are algorithms which can determine them quite efficiently. For small graphs it can be done pretty easily by inspection, especially when the graph has a reasonable structure. To find strong components in Figure 9 is messy. In Figure 8 we can find a couple immediately. "Program" and "block" are obviously non-recursive components. On the left we see that "type" and "field" are strongly connected. The functions for them are mutually recursive. On the right, "statement", while self-recursive, is not called by any other routines which it calls, so can be dealt with as described in Section 1. However, below it, a small amount of path tracing shows that (a) "expression", "term", "factor" and "variable" form a
mutually recursive group, and (b) this group is called from "statement" at two different places.

(c) Conversion Rules for TYPE and FIELD. Let us consider first the pair of functions TYPE and FIELD, which are called by BLOCK, and whose pseudo-Fortran forms are in Figure 7. These are an instance of a recursive system with one distinguished function, in this case TYPE. That is, the function FIELD is completely subsidiary to TYPE, as far as any external calls are concerned, and it is therefore invisible to the outside world. We can therefore incorporate FIELD into TYPE, and have only one FUNCTION. Once this is done, we can deal with this single routine by our previously described methods. We have

Step 1. Merge the group of mutually recursive routines into a single one, whose first statement is the header for the distinguished function. As part of this merger, we must delete the END and heading statements of each function, leaving only one of each.

This merging has to be done with some care. Since we're making a single program out of separate ones, there will be clashes of names and statement numbers. These must be avoided. In Figure 7, we have cunningly avoided it by:

(a) Making the argument names different, and

(b) Using different sets of statement numbers. In addition, if we had any variables local to each subprogram, we would have to

(c) Use a different set of names for the local variables.

One further change that has to be made, not because it's logically necessary, but because most Fortran compilers want it, is to

(d) Move all declarations to the beginning.

In the present case, this is the "INTEGER Y" and "LOGICAL IDENT". Note that there is still a declaration of "LOGICAL FIELD". This used to refer to the external function of that name, which we have just caused to vanish. However, we will be using a variable with that name, so the declaration of FIELD as LOGICAL has to stay, though with a different meaning now.
Having merged the two FUNCTIONs, we are in a position to apply the basic transformation process to the result. About the only point to be careful of is in the assignment of argument values and going to the start of the function. The call

\[
\text{FIELD} = \text{FIELD}(d(X))
\]

(which resulted from the basic transformation of

\[
\text{IF(FIELD}(d(X))) \text{ GO TO 99}
\]

into

\[
\text{FIELD} = \text{FIELD}(d(X)) \text{ IF(FIELD) GO TO 99)
\]

is to be converted to

\[
Y = d(X) \text{ GO TO 4.}
\]

The call

\[
\text{TYPE} = \text{TYPE}(c(X))
\]

should become

\[
X = c(X) \text{ GO TO 1.}
\]

That is, a recursive call \( f_i = f_i(a) \) to function \( f_i \), whose formal parameter is \( x_i \), and whose first statement is \( \text{start}_i \) is to be replaced by

\[
x_i = a \text{ GO TO start}_i .
\]

This is the reason for making the argument names of the different functions distinct.

All other aspects of the conversion are as described earlier. If we are dealing with SUBROUTINEs as well as FUNCTIONs, the fictitious functions of Section 3 are used.

(d) Conversion Rules for VARIABLE and EXPRESSION. VARIABLE and EXPRESSION (together with TERM and FACTOR) form a mutually recursive group. This
differs from the TYPE-FIELD one in that only TYPE was called from outside, and FIELD could be incorporated into it. Here neither VARIABLE nor EXPRESSION can have its identity submerged, since both are used by STATEMENT. In effect, we want a single program (the one that must be the outcome of our changes) to provide one or another output, depending on how it is called.

The problem is not primarily one of transforming a recursive to a stacking program, but of finding a representation in the language that allows the conversion to be done by techniques that we already know about. The constraints we suffer from are:

1. STATEMENT should not be changed. Its calls are to be logical functions VARIABLE and EXPRESSION, and they ought to stay that way. It is an unhappy period in the development of a program when one must change a program just because a lower-level program has difficulties.

2. VARIABLE and EXPRESSION therefore have to be FUNCTIONS merged into one program, but they have to have two results.

The first constraint is one of programming style, which ought not to be treated lightly. The second asks for an unreachable goal, given that we are dealing with old-fashioned algorithmic languages in which a function has only a single unstructured output.

One solution to our problem is to change the unreachable goal by some fancy Fortran footwork. The obvious solution to the two output problem is to merge the functions into a SUBROUTINE with two outputs. But there must also be a way to select one of two entry points.* We can handle this by making the heading

```
SUBROUTINE VAREXP(I,VX,EX,VR,ER),
```

in which \( I \) selects which entry point is wanted (and which input and output, as well), \( VX \) is the input value if VARIABLE is entered, \( EX \) if EXPRESSION, and \( VR \) and \( ER \) hold the result at exit. The extra coding to handle it this way is not prohibitive, though messy.

---

* The ENTRY statement solves most of our problems, but isn't standard Fortran.
Given this merged program (which is now convertible as described), the function VARIABLE can be simply

```
LOGICAL FUNCTION VARIABLE(X)
CALL VAREXP(1,X,0,V,E)
VARIABLE = V
RETURN
END.
```

$X$ is an input parameter to the subroutine when we are trying to parse a variable, while the third argument isn't used. $V$ is the result of the search for a variable, and $E$ will be ignored. The first parameter is the signal for VAREXP to find the proper entry. EXPRESSION is similar.
5. LOOKING FOR LIVE VARIABLES

In Step 6a of the transformation we arrange to save on the stack all parameters and all local variables, as well as a return address. It was pointed out that it isn't necessary to save everything, but only those values which will be needed after return. Unfortunately, we cannot always be sure which values will be needed, so we usually have to settle for a less than optimal saving of those values which may be needed. More exactly, we can arrange not to save those values which will definitely not be needed.

The theory of such "live-dead" analysis has been studied extensively, and general techniques developed for compiler optimization are well described in many places, e.g., [1], [16]. We will not describe the general algorithms, which determine the state of health of all variables at all points of the program, but will concentrate on the simpler problem of making this determination at a single point, namely the point of a recursive call. For programs which are not too large, and especially for programs which are well structured, the analysis, though tedious, can pretty easily be done by the programmer. An important point to realize is that the analysis can be done on the original recursive version of the program, whose paths of control are simpler than those of the transformed one.

The essence of the search is to examine every path from the point of call to all RETURN statements. If an already-defined variable is never used along any of these paths, it is dead and need not be saved. If, on every path, it is assigned a value before it is used, then it is dead. Otherwise, it is live. The difficulty in the analysis is that we have to look at every such path. Fortunately, the very expression of an algorithm in recursive form tends to cut down the amount of path tracing that's needed.

The basic tool for the analysis is a skeleton version of the flowchart of the program, the "data-flow" graph. This is the detailed flowchart with all details eliminated, but with use and definition information added at each box. A very simple example of this is the data flow graph.
for our program of Section 1, which is shown in Figure 10. The appearance of a variable name on the right indicates a use, on the left a definition. The numbers key the graph to the text of Figure 1, and have no other significance. The recursive calls are marked with an "X". Note that statements 900 and 1100 have been expanded to their parsed forms, i.e., the program is shown after Step 3 has been done.

The recursive call at 600 immediately precedes a RETURN, so we don't have to consider it further. Nothing is used between the call and the end. The call at 900 starts a path in which the only value used is that of the Q defined there, so nothing needs saving. The first call at 1100 starts a path on which only M and N are used before they are assigned to (the Q1 that's also used isn't "already defined" at this point, so it doesn't need saving). Finally, the path following the second call at 1100 has a use of only Q1 before definition. These are the same conclusions we arrived at earlier, in less systematic way.

Branches of control after a recursive call add paths which must be traced, but also allow the programmer to be smarter than an optimizing compiler if he knows something that it can't know. Consider the flow graph of Figure 11(a). Looking at both paths from the call, we see that x and y are live on the left path, x and z on the right, so that we have to save all three before the call. However, we may have certain information about the value of x at the call. For example, we may know that its value is such as to determine the subsequent path. Then we can make the test before the call to decide to save x and y if the left path will be taken, or x and z if the right path. It is as if we transform the program to that shown in Figure 11(b), but without duplicating the calls. Of course, 11(b) shows even more: x doesn't have to be saved at all.

Loops containing a recursive call always give rise to two paths from the call, the one that stays in the loop and the one which exits. These must be considered as separate paths in the search. The path to the exit is an ordinary one, but the path that lies in the loop need be followed only back to the same call. Figure 12(a) shows a DO-loop, with the
increment and test appearing explicitly at the bottom of the loop. The two paths are shown in (b). On the first path, which exits, we see that \( i \) is live, but \( x \) is not used. \( x \) is not dead, however, since on the second path we see a use before definition. Thus both \( i \) and \( x \) are live at the call.
REFERENCES


Figure 2
INTEGER FUNCTION Q(MM,NN)
INTEGER Q1
M = MM
N = NN
200  IF(.NOT.((M .EQ. 1) .OR. (N .EQ. 1))) GO TO 500
300  Q = 1
400   RETURN
500  IF(.NOT. (M .LT. N)) GO TO 800
600  N = M
700   GO TO 200
800  IF(.NOT. (M .EQ. N)) GO TO 1100
900  Q = Q(M,M-1)
      Q = 1 + Q
1000 GO TO 400

THE 'OTHERWISE' CASE
1100 Q = Q(M,N-1)
      Q1 = Q
1110 Q = Q(M-N,N)
      Q = Q1 + Q
1200 GO TO 400
1300 END

Figure 3
100 INTEGER FUNCTION Q(MM,NN)  
   INTEGER Q1, STACK (large enough); K = 0  (17)  
   M = MM; N = NN  
200 IF (.NOT.((M .EQ. 1) .OR. (N .EQ. 1))) GO TO 500  
300 Q = 1  
400 IF (K .EQ. 0) RETURN  
   IRETNR = STACK(K); K = K-1  
   GO TO (901, 1101, 1111), IRETNR  
500 IF (.NOT.(M .LT. N)) GO TO 800  
600 N = M  
700 GO TO 200  
800 IF (.NOT.(M .EQ. N)) GO TO 1100  
900 STACK(K + 1) = 1; K = K+1  
   N = M - 1; GO TO 200  
901 CONTINUE  
   Q = 1 + Q; GO TO 400  
1100 STACK(K + 1) = M; STACK(K + 2) = N  
   STACK(K + 3) = 2; K = K + 3  
   N = N - 1; GO TO 200  
1101 K = K - 2; M = STACK(K+1); N = STACK(K+2)  
   Q1 = Q  
1110 STACK(K+1) = Q1; STACK(K+2) = 3; K = K+2  
   M = M - N; GO TO 200  
1111 K = K-1; Q1 = STACK(K+1)  
   Q = Q1 + Q  
1200 GO TO 400  
1300 END

Figure 4
Figure 5

SUBROUTINE IN(T,A,ASIZE)
INTEGER T, A(1), ASIZE
INTEGER B(100), BSIZE
1 IF (LEFT(T) .EQ. 0) THEN
   ASIZE = 0
ELSE
   CALL IN(LEFT(T),A,ASIZE)
END IF
2 A(ASIZE + 1) = NAME(T)
3 IF (RIGHT(T) .EQ. 0) THEN
   BSIZE = 0
ELSE
   CALL IN(RIGHT(T),J,BSIZE)
END IF
4 A(ASIZE + 2: ASIZE + BSIZE + 1) = B(1: BSIZE)
ASIZE = ASIZE + BSIZE + 1
RETURN
END
Figure 6
Figure 6 (cont'd)
LOGICAL FUNCTION TYPE(X)
INTEGER X.
LOGICAL STYPE, FIELD, RECOG

1 IF (STYPE(a(X))) GO TO 2
   TYPE = .FALSE.; RETURN
2 IF (RECOG('ARRAY')) GO TO 3
   IF (RECOG('()')) GO TO 99
   IF (STYPE(b(X))) GO TO 99
   IF (RECOG('')) GO TO 99
   IF (RECOG('OF')) GO TO 99
10 IF (TYPE(c(X))) GO TO 99
    TYPE = .FALSE.; RETURN
3 IF (RECOG('STRUCTURE')) GO TO 99
20 IF (FIELD(d(X))) GO TO 99
   IF (NOT. RECOG(';')) GO TO 20
   IF (RECOG('END')) GO TO 99
    TYPE = .FALSE.; RETURN
99 TYPE = .TRUE.; RETURN
END

LOGICAL FUNCTION FIELD(Y)
INTEGER Y
LOGICAL IDENT, TYPE, RECOG

4 IF (IDENT(e(Y))) GO TO 98
   IF (NOT. RECOG(',')) GO TO 4
   IF (RECOG(':')) GO TO 98
30 IF (TYPE(f(Y))) GO TO 98
    FIELD = .FALSE.; RETURN
98 FIELD = .TRUE.; RETURN
END

Figure 7
Figure 3

Diagram showing relationships between concepts such as statement, variable, expression, block, type, field, and term.
Figure 10
(a) 

(b) 

Figure 11