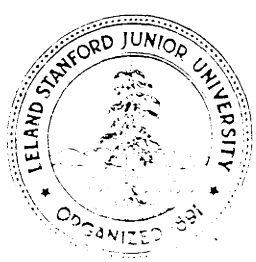
REMOVING TRIVIAL ASSIGNMENTS FROM PROGRAMS

by

Bernard Mont-Reynaud

STAN-CS-76-544
MARCH 1976

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



Removing Trivial Assignments from Programs

Bernard Mont-Reynaud

Abstract

An assignment X ← Y in a program is "trivial" when both X and Y are simple program variables. The paper describes a transformation which removes all such assignments from a program P , producing a program P' which executes faster than P but usually has a larger size. The number of variables used by P' is also minimized. Worst-case analysis of the transformation algorithm leads to nonpolynomial bounds. Such inefficiency, however, does not arise in typical situations, and the technique appears to be of interest for practical compiler optimization.

Keywords and phrases: optimizing compilers, program optimization, program transformation, program schemas, register allocation, renamings of variables.

CR categories: 4.12

This work was supported by a graduate fellowship from the IBM Corporation; the National Science Foundation, grant MCS 72-03752 A03, at Stanford University, and the Office of Naval Research, contract N00014-75-C-0816, at Stanford Research Institute.

1. Introduction.

An assignment X ← Y in a program is "trivial" when both X and Y are simple program variables. An empirical study of FORTRAN programs conducted by D. Knuth [1] suggests that trivial assignments occur quite frequently in practical programs. Such assignments are also introduced when rewriting recursive definitions as iterative ones. In this paper we consider a transformation which removes all trivial assignments from programs. The method can be impractical in pathological cases, but behaves quite efficiently in most typical situations. It is thus of interest for practical compiler optimization.

Let us consider Euclid's algorithm for computing the greatest common divisor of two nonnegative integers:

gcd(A,B) = (if B = 0 then A else gcd(B, A mod B)) .

Here x mod y denotes the remainder of the integer division of x by y . This concise recursive definition is easily implemented in iterative form (see Figure 1).

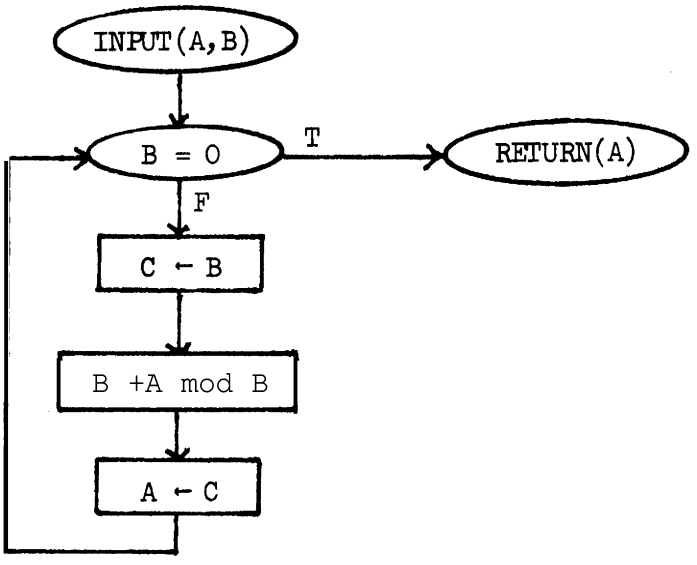


Figure 1. Flowchart GCD1.

The program GCD1 performs one test and three assignments for each iteration. Two of the assignments, $C \leftarrow B$ and $A \leftarrow C$, are trivial, that is, both the left- and right-hand sides are simple program variables. We will show how to transform any flowchart involving trivial assignments into an equivalent flowchart which has no such assignments. For example, there is a flowchart for Euclid's algorithm in which only one test and one assignment are needed for each iteration. Consider Flowchart GCD2, shown in Figure 2. Note that every execution of the loop in GCD2 corresponds to two iterations in GCD1. This will be called 2-fold loop unrolling.

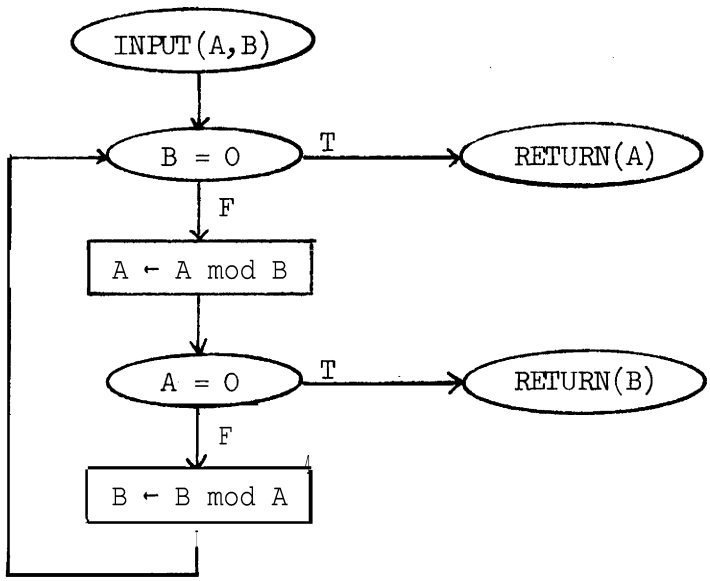


Figure 2. Flowchart GCD2.

The reader should convince himself that the two flowcharts are indeed equivalent, in a rather strong sense: The computation performed by GCD2 is step-wise identical to that performed by GCD1, except for renamings of variables and the omission of trivial assignments. We observe that GCD2 runs faster and uses fewer variables than GCD1.

Such optimizations can be carried out systematically, using the technique described below. Since the transformation is independent of the interpretations of the program variables (e.g., as integers) or of the primitive operations (e.g., the mod operation), it is best viewed as a transformation of program schemas. We introduce basic concepts and notations for discussing flowchart schemas and describe an algorithm to transform them into schemas without trivial assignments. The algorithm is then strengthened to minimize the number of variables used. The inclusion of the technique in a practical system (e.g., an optimizing compiler) raises some difficulties which are discussed briefly. The examples given in Appendices A-E illustrate further aspects of this technique.

2. Basic Concepts and Notations.

2.1 Flowchart Schemas.

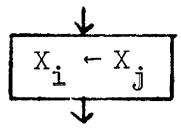
In this section we briefly define a certain class of flowchart schemas. This class is essentially equivalent to those used in classic papers on the subject [2, 3]; minor differences in notation are introduced for convenience in stating and illustrating our transformations.

A flowchart schema (or simply a schema) is a directed graph whose nodes represent computational instructions or boolean tests. It uses a setofvariables, $X = \{X_1, X_2, \ldots, X_N\}$; a set of function symbols, $F = \{f_1, f_2, \ldots\}$ (including constants); and a set of predicate symbols, $P = \{p_1, p_2, \ldots\}$. In the following we let $\bar{x}$ stand for a finite sequence of elements of X ; for example, $\bar{x}$ may be (X_3, X_1, X_4, X_1) . Then we let $f_k(\bar{x})$ and $p_k(\bar{x})$ stand for $f_k(X_3, X_1, X_4, X_1)$

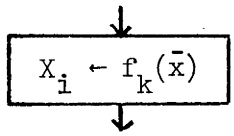
and $p_k(X_3,X_1,X_4,X_1)$, with $f_k \epsilon F$ and $p_k \epsilon P$. Let X_i and X_j be arbitrary variables in X .

We consider the following kinds of nodes.

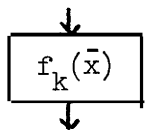
trivial assignment node:



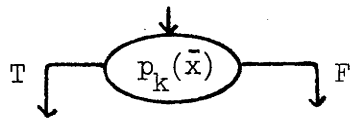
(proper) assignment node:



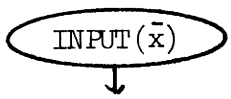
effect node:



test node:



start node:



stop-node:

Flowchart schemas are constructed by combining one start node with one or more nodes of the other kinds.

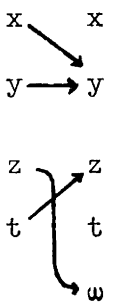
These definitions are largely self-explanatory except for the role of effect nodes. Effect nodes may represent operations such as altering data structures or printing intermediate results, which do not affect the values of the schema's variables. More generally the interpretations of function and predicate symbols are allowed to have side-effects as long as all changes to the values of the variables X_i are explicitly made by assignments.

The flowcharts GCD1 and GCD2 are examples of schemas in our class. (Note that A mod B should really be written mod(A,B) to fit our definitions; similarly we need to write equalzero for the test $B = 0$.) The example given in Appendix A illustrates the use of effect nodes in representing destructive operations on data structures.

2.2 Renamings of Variables.

The use of renamings of variables, that is, mappings from the set $X = \{X_1, X_2, \ldots, X_N\}$ of variables into itself, is central to our technique. The relevance of such renamings to problems of register allocation has already been noted elsewhere [4]. In our case we allow many-to-one mappings. In terms of register allocation, this may be called register sharing: at some point in the execution of a program, a single register holds the values of several variables.

We also make use of partial mappings, or total mappings from X into $X \cup \{\omega\}$, where ω stands for "undefined". A typical renaming of $X = \{x,y,z,t\}$ is the mapping S defined by $S(x) = S(y) = y$, $S(z) = \omega$ and $S(t) = z$:



We can write $S = \begin{pmatrix} x & y & z & t \\ y & y & \omega & z \end{pmatrix}$, extending the notation for permutations, or simply S = (y y ω z) since the upper line is held constant for a given schema. There are $(N+1)^N$ distinct renamings of X when $|X| = N$.

It is convenient to borrow from the vocabulary of register allocation when describing properties of renamings. For the current example, we say that the variables x and y are found in the "register" y . Register y is "shared" since it holds more than one variable. Registers x and t , which hold no variable, are said to be "available" or "free". This suggestive terminology, however, does not limit the technique to register machines, or to cases where there are enough registers to hold all program variables. The actual register allocation and generation of appropriate load and store operations, on a register machine, are not considered here.

Given a renaming S of the set X of variables, and a functional term f(2) , where $\bar{x}$ is the argument list (a sequence of elements of X), we use $f(S(\bar{x}))$ to denote the expression obtained by simultaneously substituting $S(X_1)$ for X_1 , . . . , $S(X_N)$ for X_N in $f(\bar{x})$. This notational convenience also applies to predicates and to the special functions INPUT and RETURN used in the start and stop nodes. It is defined only when ω does not appear after the substitution.

3. Basic Algorithm for Removing Trivial Assignments.

3.1 A Simple Example.

Before considering the algorithm in its full generality, let us follow its operation on a simple example of a straight-line flowchart. (See Figure 3.) The input flowchart has three variables x , y , z . We begin the construction of the output flowchart by copying the start node, and we use (x y ω) for initial mapping since z has no value so far; x and y might as well be mapped to themselves. Processing the trivial assignment node z ← x results in changing the mapping to (x y x) ,

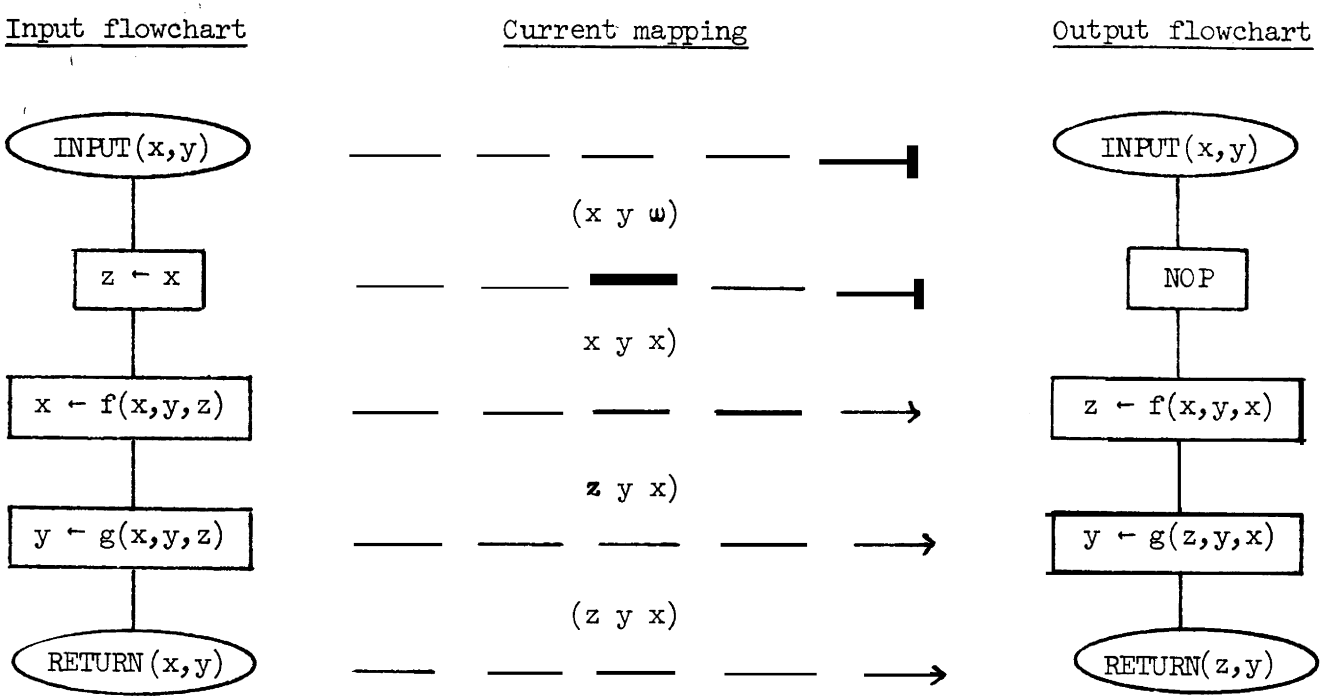


Figure 3. Transformation of a straight-line schema

reflecting the fact that z is now found in register x . This node, like all trivial assignments, needs no counterpart in the output flowchart; it is convenient to use a no-operation (NOP) node for this case. Next we process the assignment $x \leftarrow f(x,y,z)$. Its counterpart has the right hand side $f(x,y,x)$, obtained by the obvious substitution of registers for the corresponding variables. For the left hand side, we can't use register x since it is currently shared by variables x and z . But register z is free, so we can use it to hold the value $f(x,y,x)$. The mapping changes to (z,y,x) , reflecting the fact that x is now found in register z . The right hand side of the transform of the assignment $y \leftarrow g(x,y,z)$ is now clearly $g(z,y,x)$. For the left hand side, we can use y since it is not shared. The mapping is unchanged. Copying the stop node with variables renamed completes the transformation.

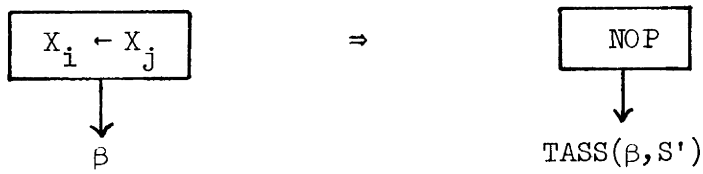
3.2 Case of Loop-free Schemas.

The algorithm uses the auxiliary recursive function TASS (for "remove Trivial ASSignments"). Thisfunction takes two arguments: a reference α to a node in the input flowchart, and a mapping S . The call $TASS(\alpha,S)$ creates a new node in the output flowchart and returns a reference $\alpha.S$ to that node. The algorithm is defined by six transformation rules, one for each kind of node. The first rule initializes the computation, and the remaining five rules define the recursive function TASS. Given the current mapping S , the node α (on the left) is transformed to the node $\alpha.S$ (on the right). The letters β and y stand for references to nodes in the input flowchart.

3.2.0 Start node:

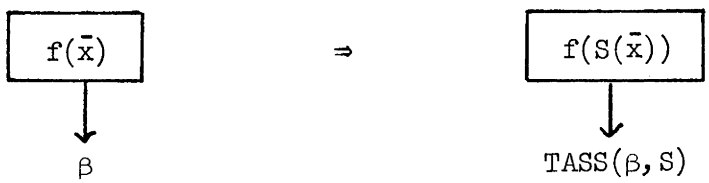
where S_0 is the initial mapping $\lambda u.$(if $u \epsilon \bar{x}$ then u else ω) .

3.2.1 Trivial assignment node:

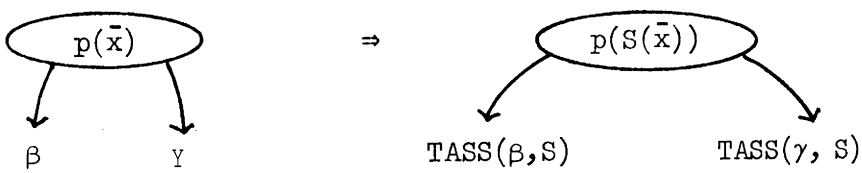


where $S' = \lambda u.$(if $u = X_i$ then $S(X_j)$ else $S(u)$) .

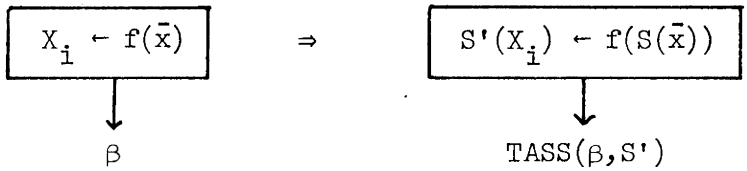
3.2.2 Effect node:



3.2.3 Test node:

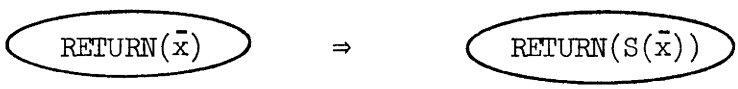


3.2.4 Assignment node:



where S' is determined as follows. Let S'' be the mapping $\lambda u.(\text{if } u = X_i \text{ then } \omega \text{ else } S(u))$. Choose R arbitrarily among the free registers of S'' (note that there is at least one such register). Let S' be the mapping $\lambda u.(\text{if } u = X_i \text{ then R else } S''(u))$. (The reason we define S' using S'' and not S will become clear in Section 4.2.)

3.2.5 Stop node:



The rules 3.2.0 - 3.2.5 completely define the transformation for loop-free flowcharts. The algorithm amounts to a forward propagation of a mapping through the input flowchart. For each node encountered, a copy is created in the output flowchart, with variables renamed as dictated by the current mapping.

For simplicity in stating the algorithm, we have transformed trivial assignment nodes to NOP nodes. It is easy to imagine how such nodes can be eliminated from the output flowchart (or better, how the algorithm could be adapted to avoid their generation in the first place).

3.3 Treatment of Loops.

The algorithm described so far does not terminate when the input flowchart has a loop. This case will be handled in the following way. We strengthen the definition of TASS so that the reference (or node name) $\alpha.S$ returned by the call TASS(α,S) is canonically associated with the pair (α,S). The nodes α of the input flowchart are initially given unique names. The distinct mappings S which arise during the computation also have unique names; for example, they may be encoded as integers between 0 and $(N+1)^N$. One can construct a unique name $\alpha.S$ by pairing the names of α and S in any reasonable way. A critical property here is that we can compute the name $\alpha.S$ which will be returned by the call TASS(α,S) before we determine the attributes associated with that name, and in particular before any recursive call is made. We will use a global variable, CREATEDNODES, to keep track of the set of names $\alpha.S$ corresponding to all the calls TASS(α,S) performed so far. Initially CREATEDNODES is a set of one element, the name of the start node in the output flowchart. (At the end of the process, CREATEDNODES is the set of nodes of the output flowchart.)

The recursive function TASS becomes:

```
TASS(α,S):
begin let α.S be the unique name canonically
          associated with the pair (α,S) ;
      if α.S ∉ CREATEDNODES
          then include α.S in CREATEDNODES, and compute the attributes
               (contents and successors) of α.S using the appropriate
               rule among 3.2.1 - 3.2.5 (the successors are determined
               by recursive calls of TASS)
          else do nothing;
   return α.S as the value of the function
end
```

Termination is now insured, since every edge of the input graph is followed at most once for each of a finite number of mappings.

3.4 Correctness and Worst-case Analysis.

Proving (or even stating precisely) the correctness of the algorithm falls outside the scope of this informal paper. The idea behind the proof is fairly simple, however:

(a) For loop-free flowcharts it is sufficient to prove (by considering rules 3.2.0 - 3.2.5 individually) that the input and output flowcharts are logically equivalent.

(b) For flowcharts with loops we consider the infinite tree schema associated with the output schema (see Figure 4). Imitating part (a) above, we show that the nonterminating algorithm defined by rules 3.2.0 - 3.2.5 constructs an infinite tree schema which is equivalent to the input flowchart. Then we show that the introduction of the global variable CREATEDNODES described in Section 3.3 results in a schema withloops, such that the associated infinite tree schema is precisely the schema just shown to be equivalent to the input flowchart.

A rough analysis of the algorithm shows that, if n and e denote the number of nodes and edges of the input flowchart, n' and e' the same quantities in the output flowchart, and N the number of variables:

- we have $n' \leq n(N+1)^N$ and $e' \leq e(N+1)^N$;
- the number of calls of TASS during the execution of the algorithm is exactly e' ;

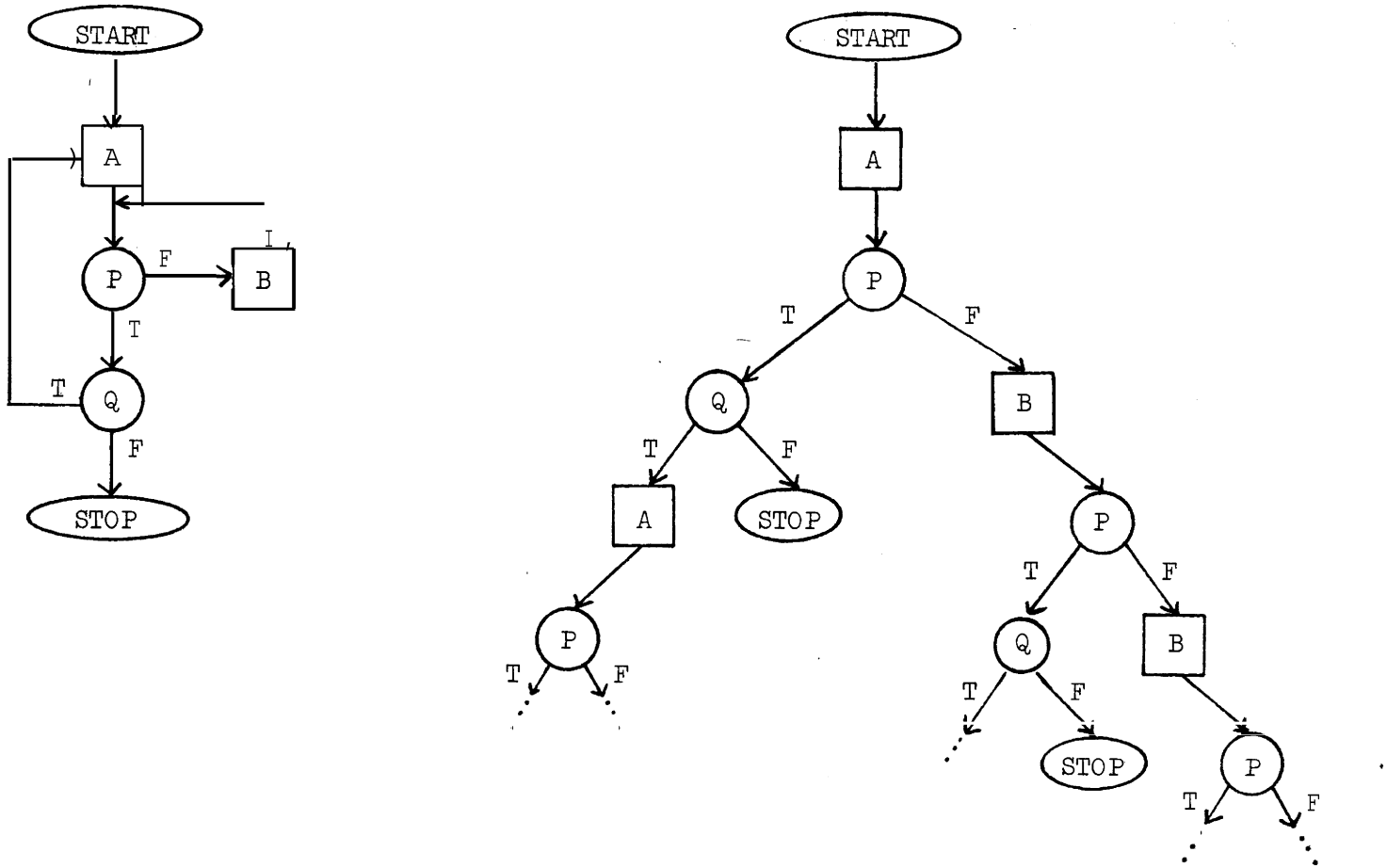


Figure 4. A schema with loops and the associated infinite tree schema.

- if suitable representations are chosen for the set CREATEDNODES and for the mappings, the total running time is $O(e'(N + \log e'))$.

Thus the algorithm is "efficient" in terms of the size of its output. However the size of the output can grow more than polynomially with the size of the input.

4. An Improved Algorithm.

4.1 Back to the gcd Example.

The algorithm described in Section 3, when applied to GCD1, does not produce GCD2, as might be expected, but GCD3 (Figure 5). One may wonder why there are two occurrences of the test (B = 0), together with their associated stop nodes; it seems that we could jump directly from the node [B ← C mod A] initial test (B = 0) . The reason is that the two tests in question are generated under different mappings: $\begin{pmatrix} A & B & C \\ A & B & \omega \end{pmatrix}$ for the first, $\begin{pmatrix} A & B & C \\ A & B & A \end{pmatrix}$ for the second. The basic algorithm overlooks the crucial fact that the variable C is "dead", that is, its value is no longer needed, when we reach the test B = 0 . The mapping should have been $\begin{pmatrix} A & B & C \\ A & B & \omega \end{pmatrix}$ in both cases.

There is another difference between GCD3 and GCD2: the former uses three variables and corresponds to a 3-fold loop unrolling of GCD1 (cf. exercise 1.1.3 in [5], first edition, p. 465), while the latter uses only two variables and its loop covers two iterations of GCD1. This difference will also be removed by the inclusion of dead variable analysis.

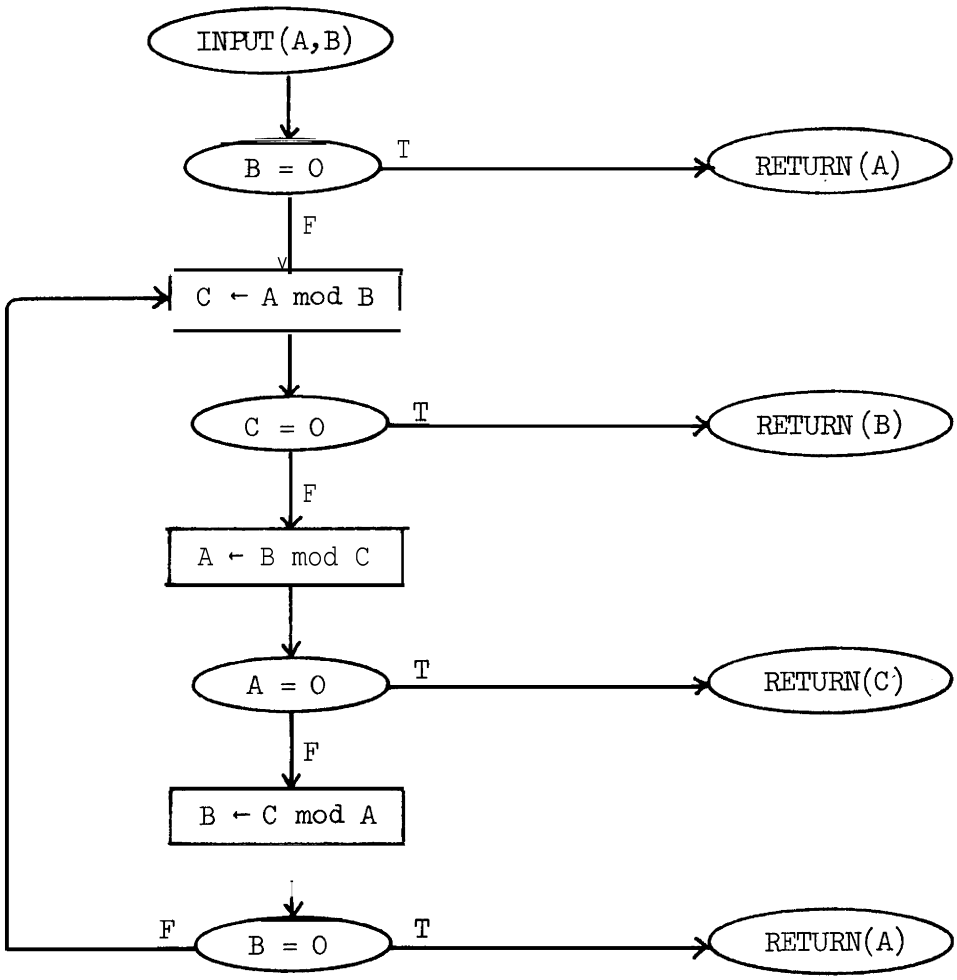


Figure 5. Flowchart GCD3.

4.2 The Improved Algorithm.

The reader is referred to [6] for a general treatment of dead variable analysis.

For our purposes it is sufficient to know that the "last uses" of each variable can be identified in the program text. More precisely, we assume that dead variable analysis has been performed prior to our algorithm and that each node α in a flowchart schema now has an additional attribute: last_used_at(α) , which is the set of variables (possibly empty) which become dead at a . We need to be even more specific. If α is an assignment of the form $x \leftarrow f(x,y)$, we consider the following steps:

(1) f(x,y) is evaluated: x and y 'are live.

(2) x is now dead, since it will receive a new definition before it is used again (possibly y also dies here).

(3) x gets a new definition, and is live again.

In such a case, we would include x in the set last used at(a) : the new value of x might well be placed in a register different from the register which held x when evaluating f(x,y) .

Rule 3.2.1 is modified by replacing

' $S' = \lambda u.(\text{if } u = X_i \text{ then } S(X_j) \text{ else } S(u))$ '

with

' $S' = \lambda u.(\text{if } u = X_i \text{ then } S(X_j) \text{ else if } u \in \text{last\_used\_at}(\alpha)$
then ω else $S(u))$ '.

Rule 3.2.4 is modified by replacing

t $S'' = \lambda u.(\text{if } u = X_i \text{ then } \omega \text{ else } S(u))$ '

with

' $S'' = \lambda u.(\text{if } u \in \text{last\_used\_at}(\alpha) \text{ then } \omega \text{ else } S(u))$ ' .

It can be shown easily that $S''(X_i)$ is ω , whether or not X_i belongs to last_used_at(α) , so that S'' always has at least one free register R , as before.

Rules 3.2.2 and 3.2.3 are modified by replacing the recursive calls TASS(*,S) by TASS(*,S") , where S" is again defined as $\lambda u.$(if $u \in$ last_used_at(α) then ω else S(u)) . Rule 3.2.5 is unchanged.

The resulting algorithm has a source of nondeterminism, due to the arbitrary choice of a free register among the available registers, in the case of a proper assignment. This nondeterminism can be removed by ordering the set X of variables and choosing the free register of lowest possible rank in that set. The ordering is such that the input variables (those appearing in the arguments to INPUT in the start node) precede other variables in the ordering of X . With these conventions, the modified algorithm minimizes the number of variables used; that is, if k is the largest number of variables simultaneously live at any point in the input flowchart, then the output flowchart has at most k variables. Other ways of taking advantage of the nondeterminism (for example, to minimize the size of the output flowchart) will not be considered here.

5. Discussion.

5.1 Interest of the Technique.

The removal of trivial assignments, as performed by the basic algorithm, does not dramatically change the time complexity of a program. Instead, the transformation usually reduces the constants involved in the analysis of the program; in the case of the gcd algorithm, the work done by the inner loop is reduced from one division, one test, and three assignments

to one division, one test, and one assignment. On the other hand, trivial assignments are quite frequent in practical programs. In an empirical study of a representative sample of FORTRAN programs [1, p 112], D. Knuth reports that 35 percent of all assignments, or 23 percent of all statements executed, have no arithmetic operation on the right-hand side. These percentages represent dynamic counts as the programs were being executed, not merely static counts on the program text. Unfortunately [1] does not tell how many of these assignments are indeed trivial; no distinction is made there between simple variables and array elements. The examples given in Appendices A-E, particularly in Appendix C, should help convey the potential of the technique.

On a register machine, additional savings may result from the reduction in the number of variables used. Also, independently of the removal of trivial assignments, the use of renamings and node copying solves the problem of "optimizing register allocation around a loop" [7], by unrolling loops as many times as necessary to achieve the optimization. Surprisingly, this is done without any explicit consideration of the loops in the input flowchart.

5.2 Practical Difficulties.

One major drawback of the technique is that the size of the output flowchart can exceed any fixed polynomial in the size of the input. A remarkable example of this behavior, due to R. S. Boyer [8], is presented in Appendix E. There are also cases where only minor gains in efficiency are obtained at the expense of major increases in program size (see Appendix D). These difficulties can be remedied in several ways, including (a) the use of effort bounds and cost functions to

decide whether the transformation should be applied or not; (b) working from the inside out, that is, beginning with inner loops; and especially (c) the combination of (a) and (b).

Another practical difficulty, which is familiar in object code optimization, arises from the idiosyncrasies of the primitive machine operations. For example, when the operation $C \leftarrow A \bmod B$ used by the gcd algorithm is implemented using a single hardware division instruction, one will not usually be free to choose the register C independently of A and B . Tuning the method to a particular machine architecture is a problem in itself.

5.3 Extensions.

The practical difficulties discussed in the previous paragraph point to various improvements and extensions of the method. Some other extensions under investigation are:

- Including the actual register allocation within the technique; in the virtual register allocation currently performed, there is no limit on the number of registers.
- Performing the transformation on an Algol-like text (source language) rather than on flowchart schemas (intermediate language).
- Defining a metaalgorithm which generalizes the technique described in this paper.
- Adding transformations to the class covered by the metaalgorithm, such as boolean variable elimination and various optimizations associated with loop unrolling.

Acknowledgments.

While teaching a data structure course at Stanford University in 1973, Edward McCreight showed how a certain machine-language program for destructive list reversal could be improved by a rather tricky use of loop unrolling (see Appendix A). The puzzlement created by this isolated example motivated an investigation which eventually led to the ideas expressed in this paper.

These results would never have been obtained, however, without the illuminating comments and continued encouragement provided by the author's thesis advisor, Donald E. Knuth. Thanks are also due to Bob Boyer, Rob Shostak and Jay Spitzen of Stanford Research Institute for enjoyable discussions and helpful suggestions.

References

[1] D. E. Knuth, "An empirical study of FORTRAN programs," Software - Practice and Experience 1 (1971), 105-133.

[2] D. C. Luckham, D. M. R. Park and M. S. Paterson, "On formalized computer programs," Journal of Computer and System Sciences 4 (1970), 220-249.

[3] Z. Manna, Mathematical Theory of Computation, (McGraw-Hill, 1974), 448 p.

[4] L. Logrippo, "On some equivalence-preserving transformations in program schemas," Proving and Improving Programs, I.R.I.A. Symposium held at Arc et Senans, France, July 1975.

[5] D. E. Knuth, Fundamental Algorithms, The Art of Computer Programming 1 (Reading, Mass.: Addison-Wesley, 1968, 2nd edition 1973), 634 pp.

[6] J. Cocke and J. T. Schwartz, "Programming languages and their compilers," Courant Institute of Mathematical Science, New York University, 1970.

[7] K. Kennedy, "Index register allocation in straight line code and simple loops," in Design and Optimization of Compilers (R. Rustin, ed.) Prentice-Hall, Englewood Cliffs, N. J., 1972, 51-64.

[8] R. S. Boyer, personal communication, December 1975.

-Appendix A. Destructive list reversal.

The example in Figure A-1 is due to Edward McCreight of Xerox Palo Alto Research Center. The program takes as input a pointer to a linked list, and returns a pointer to the reversed list, obtained by destructive updating of the original list. The meaning of **resetlink(P,Q)** is LINK(P) ← Q . The first form is used to make it clear that the corresponding node is an effect node, not an assignment node. The constant null is written **null()** , i.e., as a function without arguments, to distinguish it from a variable. The test is null(P) checks whether P = null .

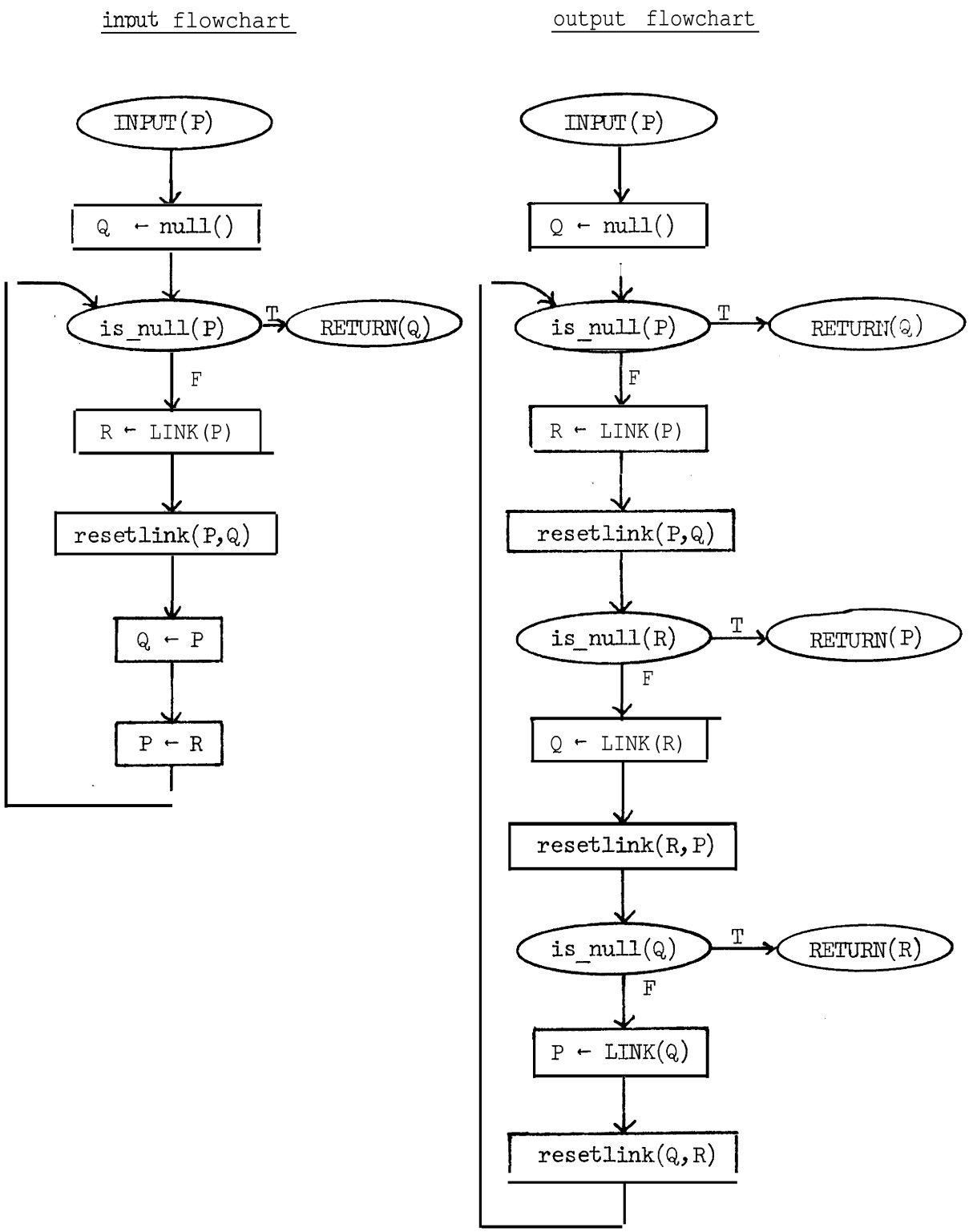


Figure A-1. Destructive list reversal.

Appendix B. Finding the maximum in an array.

The example in Figure B-1 computes the index of the maximum element in an array X(1),...,X(n . At first it may seem that the trivial assignment j ← i in the input flowchart cannot be removed; the reader is invited to try removing it himself before looking at the output flowchart.

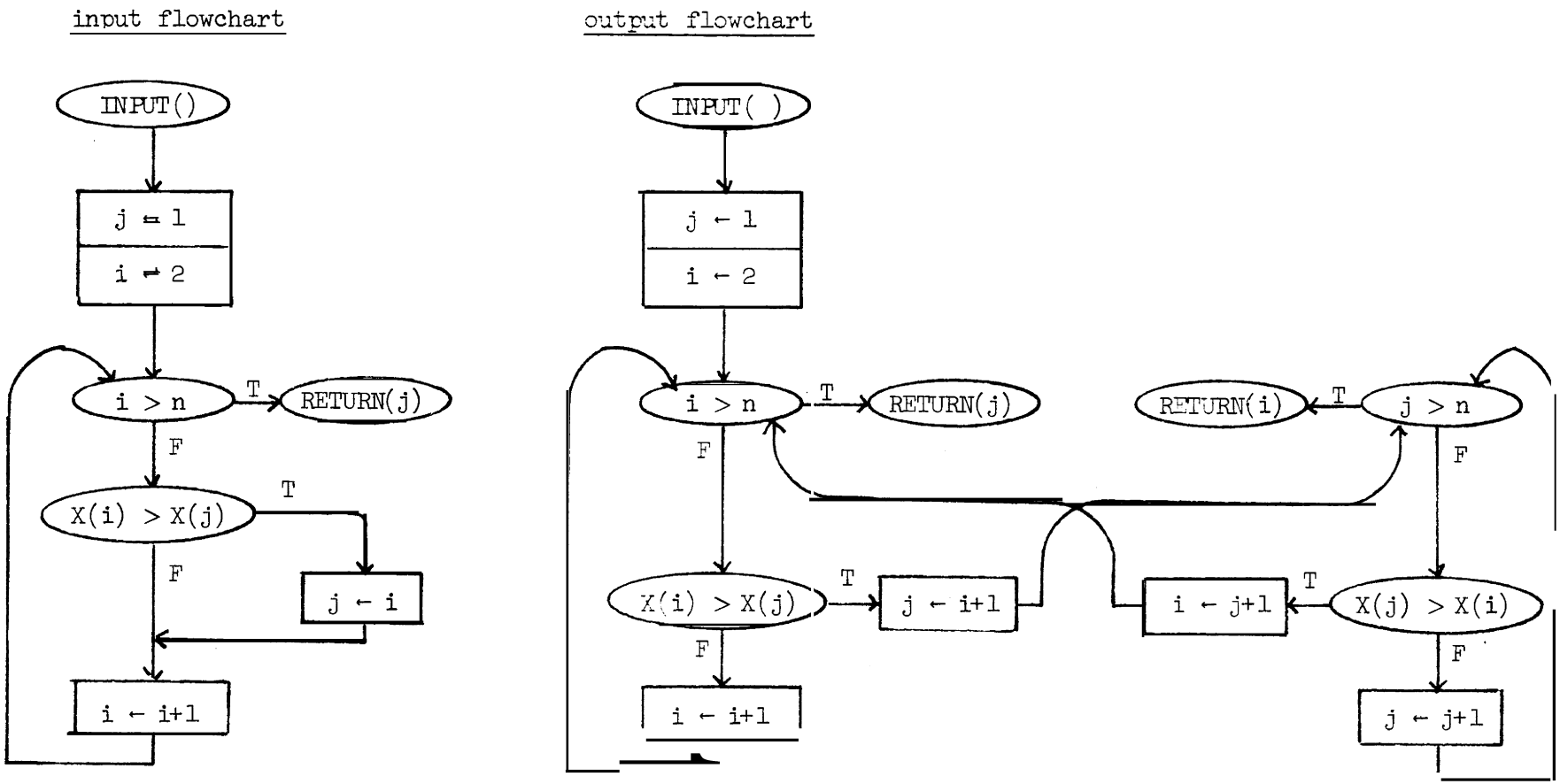


Figure B-1 Finding the maximum in an array

Appendix C. Recurrence relations.

The example in Figure C-1 is typical of all recurrence relations of the form $a_n = f(a_{n-1}, a_{n-2}, \ldots, a_{n-k})$. The program computes a_n , given the value of n . The transformation results in k-fold loop unrolling and saves k assignments per iteration. The figure illustrates the case $k = 3$.

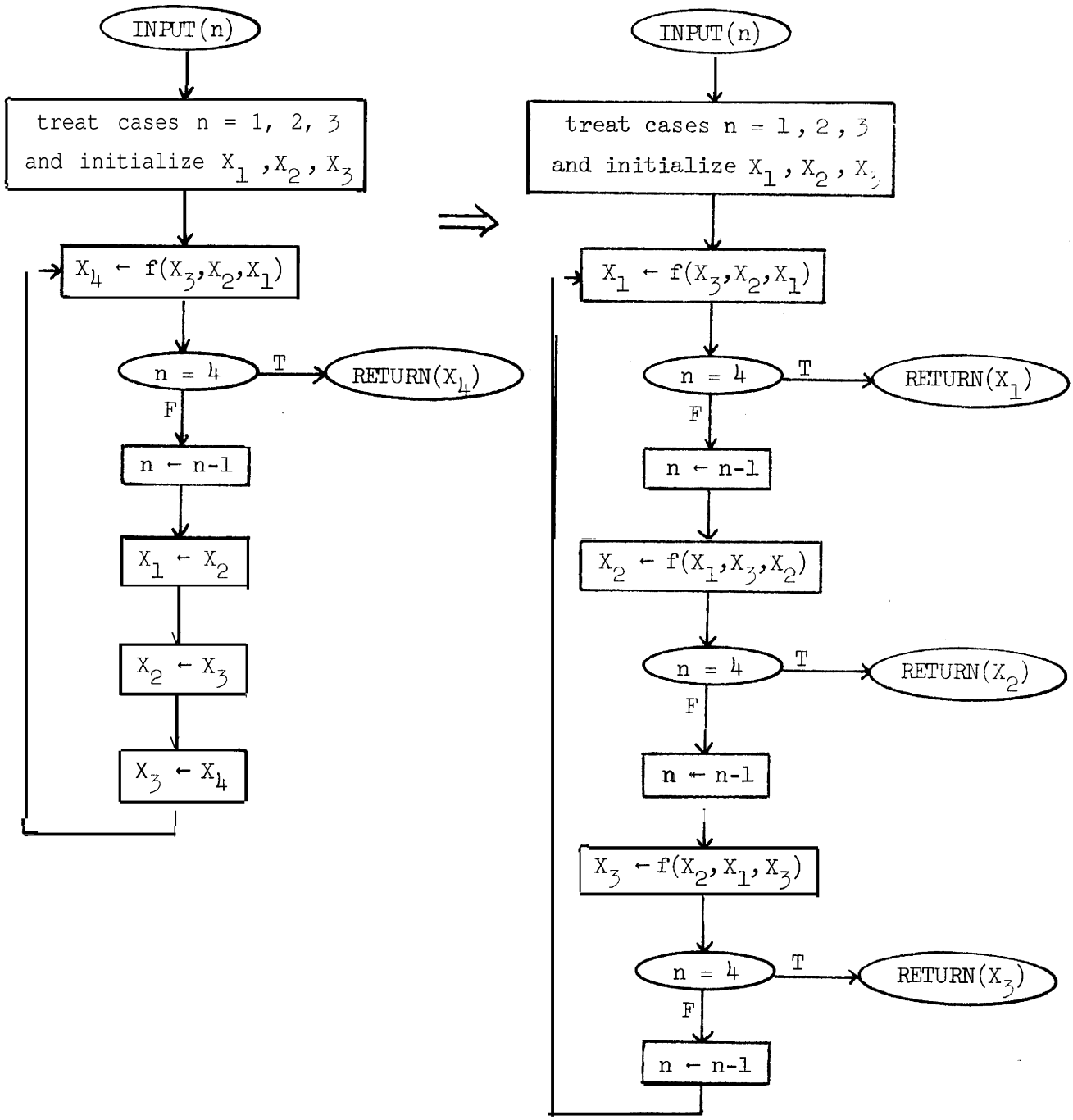


Figure C-1. Recurrence $a_n = f(a_{n-1}, a_{n-2}, a_{n-3})$.

Appendix D. A costly optimization.

input

output

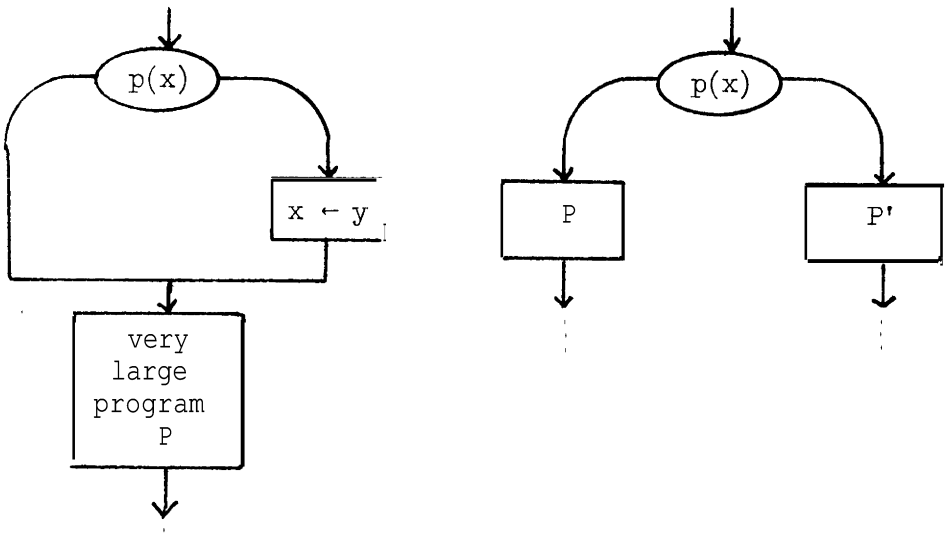


Figure D-1. A costly optimization.

Figure D-1 illustrates the need for evaluating the gain in execution time versus the gain in program size (and/or for focusing on inner loops).

Appendix E. A pathological case.

The program shown in Figure E-1 takes n inputs $X_1, X_2, \ldots, X_n$ and returns them in ascending order of the values. The auxiliary variable X' helps perform exchanges.

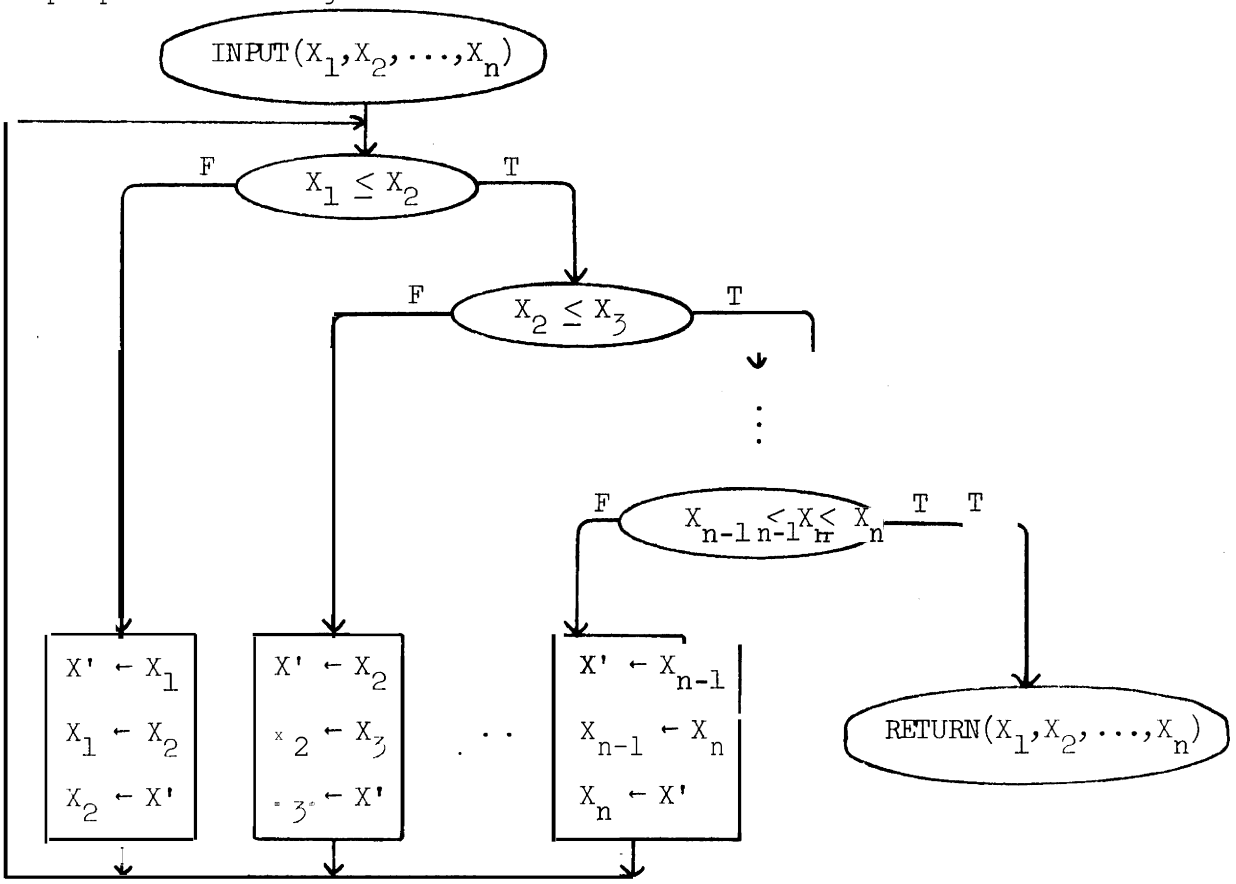


Figure E-1. Sorting n variables.

This flowchart has n+1 variables and 4n-2 nodes. The transformed flowchart has n.n!+1 nodes: the stop node, and each of the (n-1) test nodes are copied exactly once for every permutation of the variables $X_1, \ldots, X_n$.