JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 404–447 (1973)

Characterizations of Flowchartable Recursions

S. A. WALKER AND H. R. STRONG

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received May 10, 1972

In this paper we give new characterizations for the flowchartability of recursive functionals. Here flowchart means flowchart with counters (or any arbitrary control structure) and nonflowchartability essentially means requiring access to an unbounded number of data locations during computation of the functional. The general question of flowchartability is recursively undecidable. We present here an effective map from the class of all recursions to a subclass of "representatives" for which the question is decidable. The decision provides a good approximation to a characterization for general flowchartability in the following senses: (1) if a representative is flowchartable, then the recursions it represents are, and (2) there is a straightforward method for flowcharting arbitrary recursions, depending only on recursion structure, such that a recursion is flowchartable by this method if, and only if, its representative is flowchartable.

The main results of the paper are (1) that a representative is flowchartable if, and only if, it is simple or linear, and (2) that, when the context is restricted so that only invertible operations are considered, such a representative is flowchartable if, and only if, it is nested. The terms "simple" and "linear" have been defined in previous papers in the area, although they are extended slightly in this one. The term "nested" is introduced here. Simple, linear, and nested recursions are very easy to identify by inspection.

1. INTRODUCTION

In recent work (cf. [1, 2, 5, 7]) the question as to when a recursive function can be flowcharted has been reexpressed in terms of when the recursive function can still be flowcharted if we *systematically ignore certain information* in its recursive definition, namely information about the kind of data involved in the calculations (e.g., integers, finite alphabets of symbols) and information about the constants and operations (e.g., 0, 1, +, −) used in the recursive definition. Thus, the recursively defined Fibonacci function

$$f(x) = \begin{cases} f(x-1) + f(x-2) & \text{if } x > 1 \\ 1 & \text{if } x \leqslant 1 \end{cases}$$

might be investigated in terms of the "recursion schema"

$$f(x) = \begin{cases} a(f(b(x)), f(b(b(x)))) & \text{if } p(x) \\ c & \text{if } p(x) \end{cases}$$

where the latter recursion would be said to be "flowchartable" only if there were a flowchart schema that would serve as a (functionally equivalent) flowchart under *all* interpretations for the constant c, the operations a and b, and the predecate p. Here flowchart means flowchart with counters (or any arbitrary control structure) and nonflowchartability essentially means requiring access to an unbounded number of data locations during computation of the functional. We use the word "recursion" for presentations by means of branched recursion equations of the above kind, and call what is presented a "recursive functional," for it is taken to be a function ranging over arbitrary, unanalyzed subroutines on arbitrary, unanalyzed data types.

This approach has been fruitful in at least four ways: (1) it has provided a way of excluding the obviously inefficient use of encoding (via "Gödel-numberings") for flowcharting a recursive function; (2) it has provided a framework for the discussion of "recursion removal" methods with syntactic characterizations that would be amenable to inclusion in compilers; (3) it has restricted attention to "recursion removal" methods uniformly applicable (without encoding) to all data types which might also have uses in compilers and interpreters; (4) it has provided a measure for comparing the power of different programming languages with respect to which many models of programming languages and even standard languages actually do differ.

Unfortunately, one can apply a technique used by Luckham, Park, and Paterson (to show the undecidability of functional equivalence for program schemata) to show that the flowchartability of recursions is recursively unsolvable. Thus [3, Theorem 4.1] contains a description of an effective procedure for converting a Turing machine and input into a program schema which diverges under all interpretations if, and only if, the Turing machine fails to halt on the input. We can easily extend this procedure to produce an equivalent recursive schema. Let the principal function letter of this schema be g (with one argument). Let f, h, a, b, c, and p not appear in this schema. Then the schema formed by adding

$$f(x) = a(g(x), h(x))$$

$$h(x) = \begin{cases} a(h(b(x)), h(c(x))) & \text{if } p(x) \\ x & \text{if } -p(x) \end{cases}$$

and taking f as the new principal function letter is flowchartable if, and only if, the Turing machine fails to halt on its input. Since the halting problem is recursively unsolvable, so is the flowchartability problem.

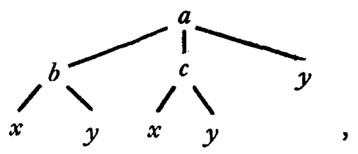
The undecidability of flowchartability for recursions merely solved a (difficult) problem for those interested in comparing powers of programming languages, for it implies the existence of nonflowchartable recursions (actual examples were presented in [7 and 5]). But it presents serious difficulties to those who would like to have general procedures for translating recursions into flowcharts when possible (recursive subroutines tend to require more time and space in existing programming languages than equivalent nonrecursive ones).

We look at the work in this paper as simultaneously exemplifying two different approaches that can be taken in this situation. First, it is an example of the fact that the problem of flowchartability is sensitive to just how much information is allowed to be taken into consideration. Recall that to go from a definition of a recursive function to a recursion, we "forget" certain information. In this paper, we forget a little more information, going from an arbitrary recursion to what we call its "canonical representative." It turns out that flowchartability *is* decidable for canonical representatives, and has a very simple syntactic characterization, moreover. A recursion is flowchartable if its representative is; however, it can be flowchartable while its representative is not.

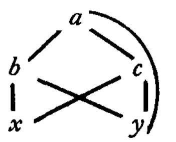
We also applied this point of view in a special case (for which we had encountered a number of examples). If we restrict attention to recursions in which all the operations are assumed to be invertible, and, with this addition, use only the information contained in a canonical representative, once more flowchartability turns out to be decidable (we have more information now, so it's a different problem), and to have an extremely simple characterization. The invertibility result can also be expressed as follows: if we only allow information about which operations are invertible (in addition to the information contained in the canonical representative) the only recursions that are nonflowchartable without invertibility information that could possibly become flowchartable are the ones that satisfy the characterization. Note that, for recursions with invertible operations we lose the relationship between canonical representative and recursion: flowchartability for the former no longer implies flowchartability for the latter (see Section 7). This is because operations can increase their numbers of arguments in passing from recursion to representative so that invertibility assumptions are stronger for the representative.

The second way we view this work is complementary to the first, because it shows that in going to canonical representatives we have not forgotten so much information that our results are about irrelevant problems. At the time this work began, certain "recursion removal" techniques were known to be applicable to special classes of recursions (thus iterative, linear, and simple recursions were translatable to flowcharts by uniform methods). It turns out that the canonical representatives can be used to characterize the recursions flowchartable by the combination of all these techniques; i.e., a recursion turns out to be flowchartable by one of these methods if, and only if, its representative is flowchartable. Returning briefly to the first point of view: if we restrict attention to the amount of information contained in a canonical representative, the known methods are the best possible.

We will now give a brief outline of the contents of this paper. In Section 2, we review such concepts as freeness and flowchartability as applied to branched recursion equations. We also introduce the concept, basic to our current work, of the *collapsed tree* or *execution graph* of an expression. This is the graph that results when common subexpressions are "collapsed," e.g., the expression $a(b(x, y), c(x, y), y)$ has the tree



while it has the collapsed tree, or execution graph



Some old concepts are redefined in terms of this new one, e.g., now a mutual recursion is said to be *simple* if each of its collapsed trees satisfies the requirement that all paths from argument symbols (not constants) to root pass through all occurrences of nonterminals.

In Section 3 we discuss flowchartability for unrestricted (mutual) recursions and provide a broad characterization of flowchartability for a slightly restricted class of recursions. We use this characterization to obtain our main results in Sections 5 and 6.

In Section 4 we introduce the second new concept that is crucial to this work, that of the *canonical representative* of a recursion.

In Section 5 we present our first main result, namely, that *a canonical representative is flowchartable if and only if it is either simple or linear.* The resulting characterization of flowchartability for nonlinear recursions is much stronger than the corresponding result in [7], where the significantly more restrictive concept of operational translatability was required. In the linear case, the result first occurred in [5]. The apparent extension of the linear case in [2] corresponds to applying [5] to a decomposition into mutual recursions as described in [7].

In Section 6 we present our second main result. By assuming operations *invertible*, we mean that we consider only interpretations such that, for every operation b of n arguments occurring in its definition, and for every $i = 1, 2, \ldots, n$, there is an operation d_i such that

$$d_i(b(x_1, \ldots, x_n)) = x_i .$$

Further, let the nonterminals in a collapsed tree of a recursion equation be *nested*, if, for every argument, there is a path from the argument to the root that contains all the nonterminals of the tree. The nonterminals of a recursion are then said to be *nested* if they are nested in every collapsed tree. Our result can then be stated as follows: *Assuming invertibility, a canonical representative is flowchartable if and only if its nonterminals are nested.* By considering the proof we note that for the sufficiency we can weaken our invertibility requirement and that we need not restrict ourselves to canonical representatives.

In Section 7 we note certain straightforward extensions of our results and discuss some of the limitations inherent in our characterizations.

Section 8 is an appendix which provides a categorization by examples of the various types of behavior of recursions with respect to flowchartability, and can be read as a summary of the results of this paper and many of those of [5, 7, and 8]. Flowcharts are exhibited (more than one where the different results apply to the same examples).

2. Basic Concepts: Expression Graphs, Flowcharts, Recursive Functionals

In this section we will introduce the concept of the *expression graph* or *collapsed tree* of an expression, and some other new concepts related to it. We will also review, usually by examples, certain basic concepts whose precise definition may be found in [7].

First, consider an expression built up from arbitrary operation symbols

$$a, b, c, \ldots,$$

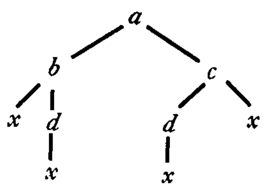
and arguments

$$x, x_1, x_2, \ldots, y, \ldots,$$

for instance

$$a(b(x, d(x)), c(d(x), x))$$

or, in tree form:



(1)

The *expression graph*, or *collapsed tree* of an expression is the graph that results when the tree form of an expression is "collapsed" by eliminating all repetitions of common subexpressions (and transferring the edges from eliminated copies of subexpressions to the single copies that are allowed to remain). Thus

$$\begin{tikzcd}
 & a \arrow[dl, dash] \arrow[dr, dash] & \\
b \arrow[dr, dash] \arrow[ddr, dash] & & c \arrow[dl, dash] \arrow[ddl, dash] \\
 & d \arrow[d, dash] & \\
 & x &
\end{tikzcd} \tag{2}$$

is an expression graph for expression (1). We assume an upward direction on the edges of such a graph. It is clear that an expression graph is a finite, connected, ordered, labeled, acyclic, directed graph, with a single *root*: node with no *successors*, and arbitrarily many *leaves*: nodes with no *predecessors*. Now let

$$L, L_1, L_2, \ldots,$$

be "location variables." A *method of calculating an expression* will be a sequence of instructions (assignment statements) which can best be understood by considering an example, say a method of calculating (1) above:

$$\begin{aligned}
&L_2 \leftarrow d(L_1) \\
&L_3 \leftarrow b(L_1, L_2) \\
&L_1 \leftarrow c(L_2, L_1) \\
&L_3 \leftarrow a(L_3, L_1)
\end{aligned} \tag{3}$$

This method of calculating (1) must also tell us that the argument x occurs in location L_1 in the beginning of an execution of the sequence of instructions (3), and that the value of the expression will be found in L_3 at the end. (The first instruction is, of course, to be read: apply the operation d to the data in location L_1 and put the result in L_2, etc.)

Researchers in this area (cf. [5, 7, and 9]) think of the calculation of an expression in terms of the following graph game.

Let a particular expression be given and think of its expression graph as a "game board." Assume you have an infinite supply of stones with labels on them, say

$$L_1, L_2, \ldots .$$

Now consider the (one-person) game $\mathcal{G}$ in which the following are the only permissible moves.

1. At any move of the game you can place a stone on a leaf.

2. At any move in the game you can put any stone (but only one per move) on a node that has all the nodes directly below it covered with stones. The game ends whenever a stone is placed on the root of the graph.

Here, putting a stone labeled L_i on a node n corresponds to calculating the expression corresponding to the subgraph whose root is n and storing the result in location L_i. The goal of the game is to minimize the number of stones (locations) used. It is not difficult to see that any way of playing out the game on the expression graph of an expression E corresponds to a method of calculating E (and any method of calculating E that does not correspond to such a game must contain a subset of instructions that does). Thus we will speak of a way of playing the game on a graph alternatively as a way of *completing* the game or *executing* the graph. When thinking in terms of playing the game on an expression graph, we say a node, or the subexpression whose root is the node, is *covered* when there is a stone on the node. We say a path is *closed* if at least one of its nodes is covered, *open* otherwise.

The concept of *flowchart* (schema) we use in this paper is a simple one involving assignments, branching on predicates assumed to be incompatible, and the use of

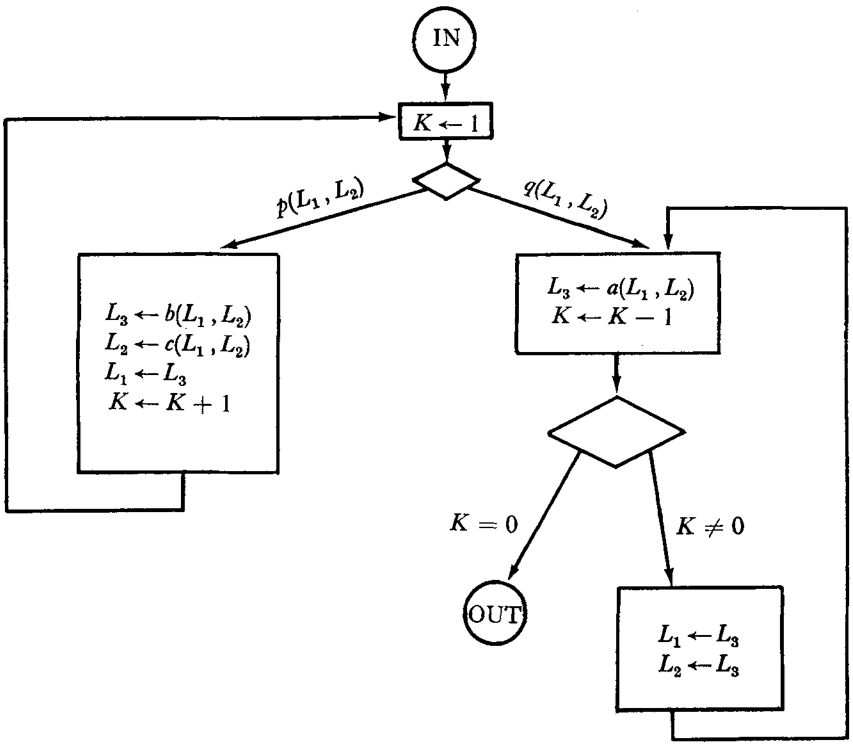


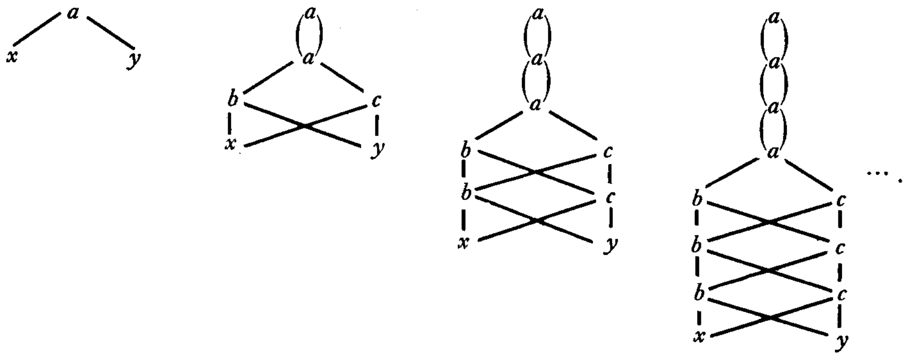
FIG. 1. Example of a flowchart.

counters to which the number one may be added or subtracted, and on which equality to zero may be tested. A straightforward connection between this concept and the game $\mathscr{G}$ is implicit in the following.

LEMMA 1. *If a flowchart mentioning n data (i.e., noncounter) locations computes an expression E, then one can extract from the flowchart computation a way to play the game $\mathscr{G}$ on the expression graph for E that uses at most n stones.*

We will illustrate the meaning of this lemma by considering the preceding flowchart (Fig. 1) as a (completely general) example.

This flowchart has one counter K and three data locations L_1, L_2, L_3, where L_1 and L_2 are assumed to hold "input data" at the beginning, and L_3 to hold the value of the function computed at the end. It is not difficult to see that the expression graphs of the expressions calculated by this flowchart are:



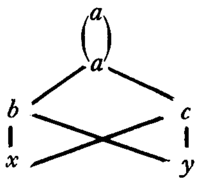
To see how the flowchart indicates ways of playing the game $\mathscr{G}$ on the expression graphs, let us consider the path through the flowchart that computes the second expression. The sequence of assignments made to data locations along this path is as follows:

1. $L_3 \leftarrow b(L_1, L_2)$
2. $L_2 \leftarrow c(L_1, L_2)$
3. $L_1 \leftarrow L_3$
4. $L_3 \leftarrow a(L_1, L_2)$
5. $L_1 \leftarrow L_3$
6. $L_2 \leftarrow L_3$
7. $L_3 \leftarrow a(L_1, L_2)$.

Corresponding to this sequence, for each of the seven steps we have the following values in the location to which the assignment was made (input x and y being in L_1 and L_2 originally):

1. L_3: $b(x, y)$
2. L_2: $c(x, y)$
3. L_1: $b(x, y)$
4. L_3: $a(b(x, y), c(x, y))$
5. L_1: $a(b(x, y), c(x, y))$
6. L_2: $a(b(x, y), c(x, y))$
7. L_3: $a(a(b(x, y), c(x, y)), a(b(x, y), c(x, y)))$.

We can read this last list as a set of instructions as to how to play the game on



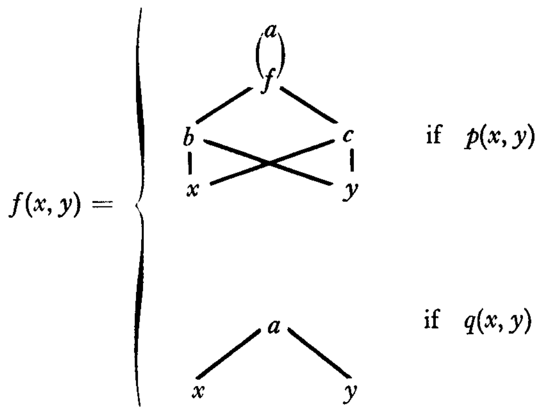
as follows. We take three stones L_1, L_2, L_3 and start by placing L_1 and L_2 on x and y, respectively. Line 1 tells us to put the stone labeled L_3 on the node labeled b (i.e., the node corresponding to the subexpression $b(x, y)$). Line 2 tells us to move the stone L_2 to the node labeled c, etc.

We make much use of the contrapositive of Lemma 1.

In this paper, we consider *recursive functionals* presented by (branched) recursion equations, which we call *recursive schemata or recursions*, e.g., corresponding to the same functional presented by the flowchart example above, we have the branched recursion equation

$$f(x, y) = \begin{cases} a(f(b(x, y), c(x, y)), f(b(x, y), c(x, y))) & \text{if } p(x, y) \\ a(x, y) & \text{if } q(x, y), \end{cases}$$

or, writing the right-hand sides of the rules of the equation as collapsed trees or expression graphs (as we usually think of them),



The function letters in the recursions will be called *nonterminals* and be denoted by f, g, h, while the operation and argument symbols will be called *terminals* and be denoted by a, b, c, d, e, and x, y, z, respectively. With each recursion is associated a principal function letter ("f" in the above example). The *calling graph* of a recursion is the directed graph with nodes representing function letters and edges representing the relation "defined in terms of." If the calling graph of a recursion is s strongly connected, we say the recursion is mutual. Thus the system

$$f(x) = \begin{cases} a(g(b(x))) & \text{if } p(x) \\ c(x) & \text{if } q(x) \end{cases}$$

$$g(y) = \begin{cases} f(e(y)) & \text{if } r(y) \\ d(y) & \text{if } s(y) \end{cases}$$

is mutual while the system

$$f(x) = \begin{cases} a(f(b(x))) & \text{if } p(x) \\ c(x) & \text{if } q(x) \end{cases}$$

$$g(y) = \begin{cases} f(e(y)) & \text{if } r(y) \\ d(y) & \text{if } s(y) \end{cases}$$

is not. The partial reduction of the problem of flowchartability to the case of mutual recursions (by means of a decomposition into mutual components) has been discussed elsewhere ([7 and 9]). Unless otherwise indicated by context, all recursions will be assumed to be mutual in the rest of this paper.

Although our results can be extended to more complicated cases, we also assume here that the arguments of the predicates in the branched recursion equations are precisely the same as the arguments of the function being defined and that there are no 0-ary function letters, although we do allow 0-ary operation symbols (*constants*).

A *production sequence* for a recursive schema with a principal function letter f of rank n can now be defined to be a sequence of collapsed trees $t_0, \ldots, t_{2m}$ such that $t_0 = f(x_1, \ldots, x_n)$, and for each $0 \leqslant i \leqslant m - 1$ there is a rule (for a k-ary g) in the recursion equations

$$g(y_1 \cdots y_k) \to t \qquad \text{if} \quad p(y_1 \cdots y_k)$$

such that

(1) t_{2i+2} is the result of replacing a node g of t_{2i} by t, removing the leaves of t and replacing them by connections to the appropriate arguments of g in t_{2i}, and finally eliminating any repetitions of subexpressions that may have been introduced;

(2) t_{2i+1} is the result of replacing the leaves of $p(y_1, \ldots, y_k)$ by those same subgraphs that were the arguments of the replaced g.

Further, a *computation sequence* is a production sequence in which each substitution is applied to the root of a subgraph that contains no nonterminals. (The notions, *production sequence* and *computation sequence* are defined as in [7] except that their elements are collapsed trees.)

We wish to make use of the natural concept of the *image* of a node in later graphs in a production sequence. Since we go from step to step in a production sequence by substituting expression graphs for nodes and then collapsing, each node in a (nonpredicate) graph will go into a node or a graph in the next nonpredicate graph (in the latter case we will call the root of the graph substituted for the node the *image* of the node, in the first case the node itself). We say a node n in an expression graph E of a production sequence is the *direct image* of a node n^* in an earlier graph E^* if it is the image of that node and no substitutions have been made in the graphs between E^* and E for any of the nodes in the subgraph of E^* that has its root at n^* nor for any of their images in the graphs between E^* and E. For example, suppose

$$\begin{tikzcd} p \arrow[d, dash] \\ \ldots, t, b, u, \ldots \arrow[d, dash] \\ x \end{tikzcd}$$

were part of a production sequence, with

$$t = \begin{tikzcd} & g \arrow[dl, dash] \arrow[dr, dash] & \\ f \arrow[dr, dash] & & a \arrow[dl, dash] \\ & b \arrow[d, dash] & \\ & x & \end{tikzcd} ,$$

$$u = \begin{tikzcd} \begin{pmatrix} g \\ a \end{pmatrix} \arrow[d, dash] \\ b \arrow[d, dash] \\ x \end{tikzcd}$$

Then (the node labeled) g in u is the image, but not the direct image of (the node labeled) g in t; a in u is the image, but not the direct image of f in t; a in u is the direct image of a in t; and b in u is the direct image of b in t. Also, we may say the node labeled b in

$$\begin{tikzcd} p \arrow[d, dash] \\ b \arrow[d, dash] \\ x \end{tikzcd}$$

is the direct image of the node labeled b in t. The notion of direct image is extended from nodes to paths or subgraphs in the obvious way.

The ideas of a consistent computation or production sequence, of an interpretation of a schema, of the value of a schema at an interpretation (and the corresponding concepts for flowchart schemata) are the natural ones, and will not be reviewed here.

We recall that a *terminating production sequence* is a finite production sequence in which the last expression graph contains no nonterminals; that an expression (graph) is called *terminal* if it contains no nonterminals; that the rule of an equation

$$f(x_1, ..., x_n) = t \qquad \text{if} \quad p(x_1, ..., x_n)$$

is *linear* if t has at most one nonterminal (viewed as a collapsed tree); that a recursion schema is *linear* if each of its rules is; that a recursive schema for an n-ary f is *free* if every computation sequence beginning with $f(x_1, ..., x_n)$ can be extended to a consistent terminating computation sequence.

We can now define a mutual recursion to be *simple* if the collapsed trees in its defining equations have the property that all paths from arguments to root pass through all occurrences of nonterminals. See the Appendix (Section 8) for examples of simple and nonsimple recursions.

Here we will use a more manageable notion of conservativeness than that of [7], namely, a recursion will be said to be *conservative* if each argument occurring on the left-hand side of a recursion equation rule also occurs in the collapsed tree on the right-hand side. Thus

$$f(x, y) = \begin{cases} f(a(x), y) & \text{if } p(x, y) \\ b & \text{if } q(x, y) \end{cases}$$

is conservative while

$$f(x, y) = \begin{cases} f(a(x), x) & \text{if } p(x, y) \\ b & \text{if } q(x, y) \end{cases}$$

is not.

3. A Broad Characterization of Flowchartability

Let us first consider some general properties of flowcharts. The counter facility we allow them provides us with a *control* feature capable of computing any partial recursive function. The only limitation of a flowchart (as a program for deterministic computation) is the fixed, finite number of data storage locations. Each of these locations is capable of holding any arbitrary datum; but there are no pairing and unpairing functions available. Consequently, two data items cannot be coded and stored in one location and then separately retrieved. We say that a functional is *flowchartable* if it can be presented by such a flowchart. We say that a recursion is *flowchartable* if the functional it presents is.

Since a flowchart has arbitrary control it can simulate the computation sequence of a recursion symbolically. In order to compute the functional presented by the recursion, it is thus sufficient to execute each symbolic expression which appears as an argument to a predicate in the computation sequence and to execute the terminal symbolic expression, if any. If there is a finite uniform bound on the number of locations required to execute these expressions (over all consistent computation sequences of the recursion), then the recursion is flowchartable. For example, this implies that any recursion with only 1-ary and 0-ary operation symbols is flowchartable. As a partial converse, consider a flowchart which presents the same functional as a recursion. Consider an argument to a predicate in a consistent computation sequence of the recursion such that, depending on the value of the predicate, the computation sequence can be consistently terminated in two different ways, producing two distinct terminal trees. The flowchart must execute this argument. Thus the number of locations mentioned in the flowchart bounds the number of locations required to execute all of these expressions as well as all terminal expressions of consistent computation sequences of the recursion. Under restrictions on the recursions considered, we thus obtain necessary and sufficient conditions for flowchartability in terms of the terminal expressions alone.

THEOREM 1. *A conservative recursion is flowchartable if, and only if, there is a uniform bound on the number of locations required to execute terminal expressions of consistent computation sequences of the recursion.*

We will use this characterization in later sections.

4. CANONICAL REPRESENTATIVES OF RECURSIONS

The following algorithm for obtaining a *canonical representative* for a mutual recursion will also serve to define equivalence of recursion structure in a slightly narrower way than was chosen in [7], i.e., two recursions have equivalent structure if they have the same canonical representatives modulo a one-to-one correspondence between argument, operation, predicate, and recursive function variables. (To understand the essence of the algorithm, the reader is advised to read through it first, ignoring what happens to constants, i.e., to read parts 1c, 1d, 2, and 3.)

Recall that we assume paths of a collapsed tree directed upward, from leaf to root, so that the root has no successors and the leaves have no predecessors. If two distinct nodes of a collapsed tree have some common immediate successor, we say they are *siblings*. An occurrence of an operation symbol is said to be a *quasi-constant* if there are no arguments or nonterminals below it. A *maximal quasi-constant* is a quasi-constant which either has no successors or which has an immediate successor which is not a quasi-constant. The first part of the algorithm below is designed to achieve the effect of absorbing constants into their successors when feasible.

The algorithm will be given in three steps.

(1) For each (nonpredicate) collapsed tree on the right side of a rule of the recursion,

(a) Treating occurrences of maximal quasi-constants one-at-a-time, delete the predecessor edges of the maximal quasi-constant, delete any quasi-constants (except the root) which have no successors, and duplicate occurrences of the maximal quasi-constant (now constant) so that each constant has at most one successor;

(b) Add edges so that each constant immediately covers its siblings, if any, and add edges and argument symbols so that each constant without siblings immediately covers all argument symbols from the left side of the rule;

(c) Add operation symbols so that there is an operation symbol immediately above an argument symbol on each path from a leaf to the root;

(d) Add argument symbols and edges so that each operation symbol immediately above an argument symbol immediately covers all argument symbols from the left side of the rule.

(2) Rename all operation and predicate symbols so that there are no repetitions.

(3) Determine which function letters from the left sides of the rules cannot be terminated. Delete all defining rules for these symbols and all rules in which they occur.

The following example (Fig. 2) illustrates the application of the algorithm to a recursion presented as a set of rules.
Note that, in the absence of constants, the statement of the algorithm would be much simpler. The complex method of eliminating constants is needed to guarantee that the algorithm transforms simple recursions into simple recursions. Thus, we have a transformation algorithm such that, if the standard techniques for flowcharting simple or linear recursions apply to a recursion, then they apply to its transform, while any flowchart for the transform of a recursion can easily be modified to produce a flowchart for the original. (By the undecidability of flowchartability there are pathological cases in which a recursion is flowchartable (by nonstandard methods) while its transform is not. This situation will be discussed further in Section 7.)

Lemmas 2 through 6 below apply to any recursion resulting from the application of steps (1) and (2) with or without step (3).

LEMMA 2. *Every canonical representative is conservative.*

Since there are no repetitions of operation symbols in the rules of a canonical representative, each occurrence of an operation symbol indicates a unique rule by which it was introduced. By induction on the length of the production sequence, we have

LEMMA 3. *Each labeled path from a leaf to some node of an expression graph in a production sequence of a canonical representative is unique up to direct image, i.e., there is an earliest occurrence of the path of which all others are direct images.*

As an immediate corollary, we have the following.

LEMMA 4. *Every computation sequence of a canonical representative is consistent.*

Thus, by Theorem 1, and Lemmas 2 and 4, showing the flowchartability of a canonical representative reduces to showing that there is a uniform bound on the number of locations required to execute terminal expressions of computation sequences of the recursion.

In the induction used to prove Lemma 3, we could consider the expression graph results of substitutions in a production sequence just before, instead of just after, collapsing. Thus, we see that, if t is the right-hand expression of a rule for f in a

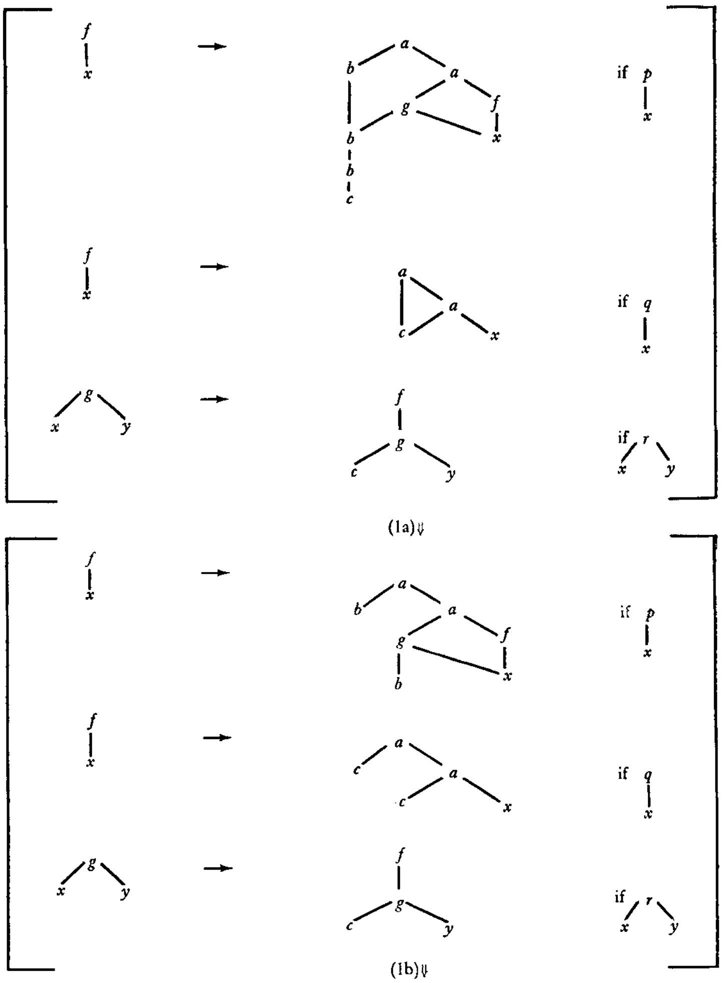
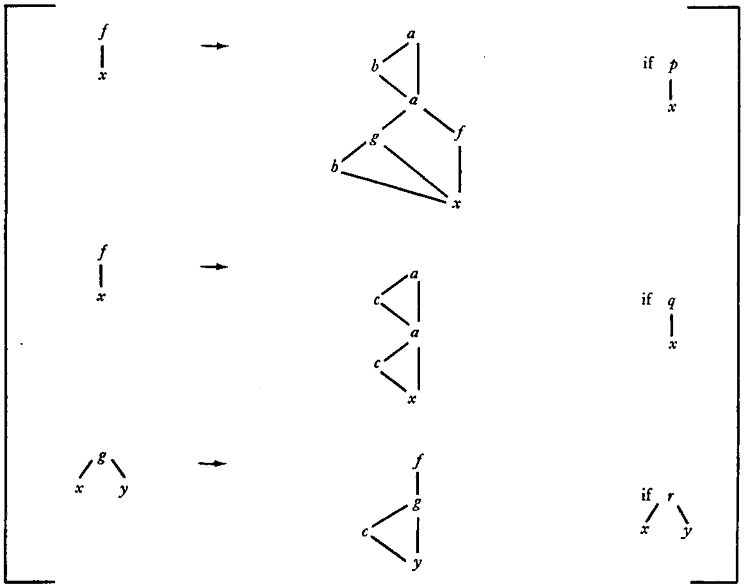


FIG. 2 (*continued*)



(1c)⇓

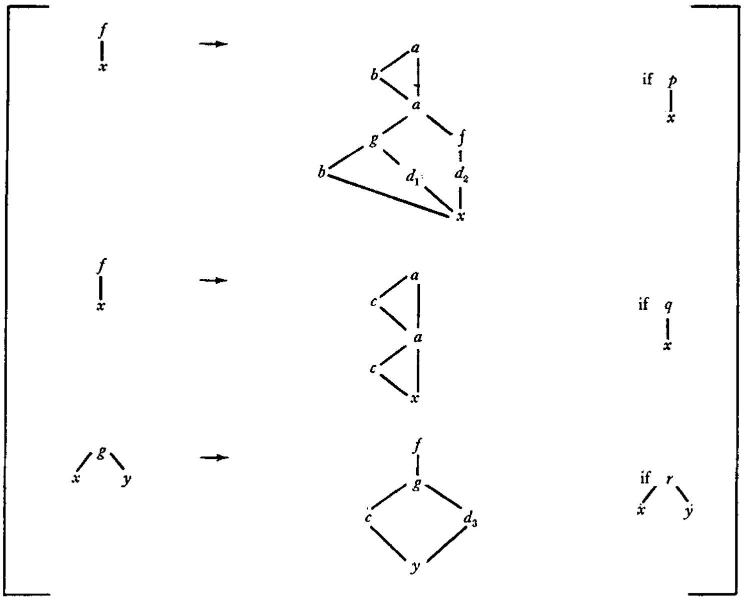
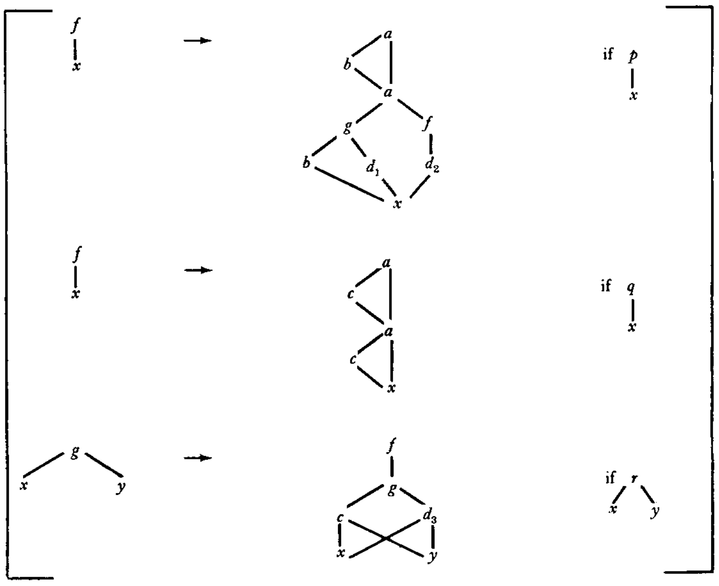
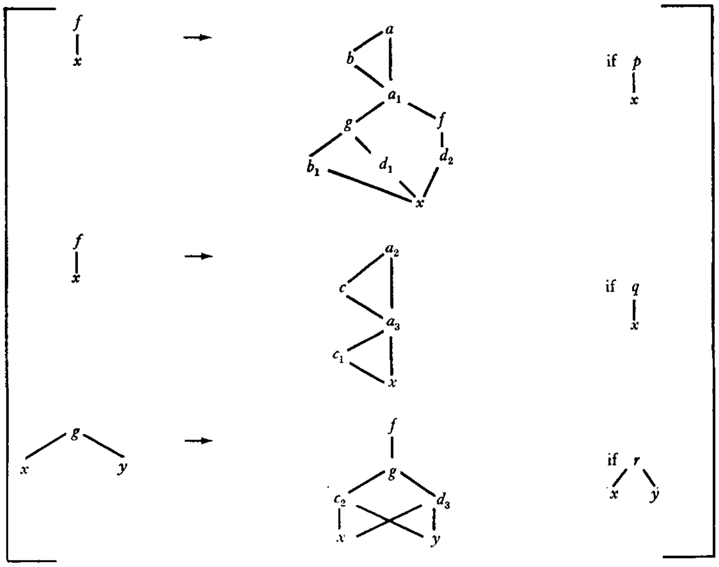


FIG. 2 (*continued*)



(2)⇓



(3) ⇓

FIG. 2 (*continued*)

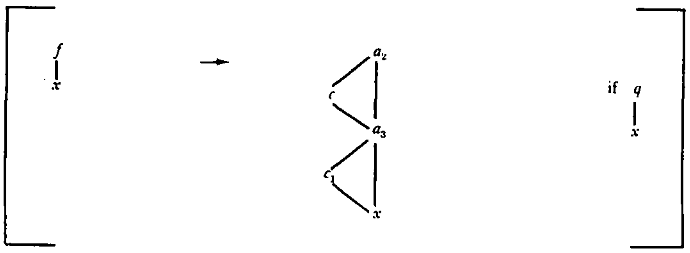


FIG. 2. Application of the algorithm for obtaining the canonical representative.

canonical representative, and if t is substituted for f in E, no collapsing is necessary to produce the subsequent expression E^* of the production sequence; and, consequently, there are no edges from a node n of t in E^* to nodes outside t unless n is a root or leaf of t. We summarize this property of canonical representatives by

LEMMA 5. *There is no collapsing in a production sequence of a canonical representative.*

A straightforward induction will give the following.

LEMMA 6. *Each terminal expression of a production sequence of a canonical representative $\mathscr{C}$ is also the terminal expression of a computation sequence of $\mathscr{C}$.*

In order to carry out step 3 of the algorithm, we need the following lemma.

LEMMA 7. *If $\mathscr{R}$ is a recursion produced by carrying out steps* (1), *and* (2) *of the algorithm, then one can effectively determine which function symbols of $\mathscr{R}$ can be terminated.*

Proof. Let F_1 be the set of function symbols which can be terminated by one application of a rule. Given F_n, let F_{n+1} be the union of F_n with the function symbols which produce, by one application of a rule, expressions which are terminal except for occurrences of symbols from F_n. Since R has only finitely many symbols, there must be a first n such that $F_n = F_{n+1}$. This F_n is the set of function symbols which can be terminated.

After step (3) we have the following.

LEMMA 8. *Every canonical representative is free.*

Since we have defined production and computation sequences in terms of collapsed trees, the notion of free is broader here than in [S], including, e.g.,

$$f(x) = \begin{cases} a(f(b(x)), f(b(x))) & \text{if } p(x) \\ c(x) & \text{if } -p(x) \end{cases}$$

as a free (and linear!) recursion.

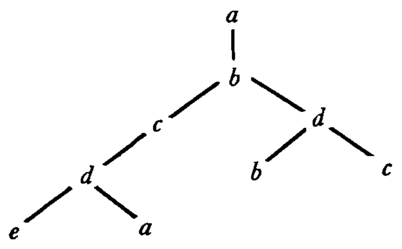
5. Characterization of Flowchartability for Canonical Representatives

This section is devoted to the proof of the first of our two main theorems.

Theorem 2. *A canonical representative is flowchartable if and only if it is either linear or simple.*

That a linear or simple canonical representative (actually a linear or simple recursion not necessarily possessing all the properties of a canonical representative) is flowchartable was implicit in [7 and 8] (simple) and in [5] (linear), and we will not repeat those proofs here. (The appendix reviews some of the translations involved in the proofs, however.) We must thus show that if a canonical representative is neither simple nor linear, it is not flowchartable.

We first define an expression graph to be a *binary tree of depth n* if it is a rootward directed (true) tree such that (1) every node has one or two predecessors; (2) every path from leaf to root passes through exactly n two-predecessor, or *branch*, nodes. Thus



is a binary tree of depth 2.

We will use the following lemma in the proof of Theorem 2.

Lemma 9. *If a binary tree T of depth n is a subgraph of an expression graph E,* *then any flowchart that can calculate E must mention more than n data (noncounter) locations.*

Appealing to Lemma 1, we will use the graph game terminology and show that if E is an expression graph with a binary tree T of depth n as a subgraph, it requires more than n stones to complete the game on the graph E. (This proof is similar to one found in [5 and 7] and uses the simpler concepts of [5].)

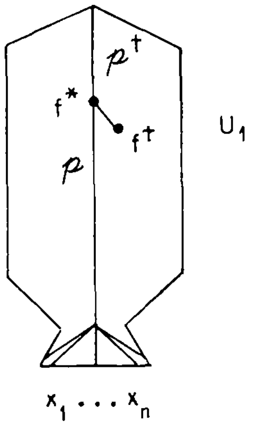
To see this, we first note that any way of completing the game on E with n stones would provide us with a way of completing the game on T with n stones. *We emphasize here that adding edges to an expression graph cannot reduce the number of stones required to play the game.* It thus suffices to show that one cannot play the game on a binary tree T of depth n with only n stones. For this, consider the last time (in any playing of the game) that there is an open path $\mathscr{P}$ from some leaf ℓ to the root of T. At this point every other path from a leaf to the root of T must be closed, and at the next step, a stone must be placed on the leaf ℓ. Thus, at that next step all the nodes on $\mathscr{P}$ above ℓ are still uncovered. But for each of the n branch nodes above ℓ on $\mathscr{P}$ we can consider a path from the root to a leaf of T that follows $\mathscr{P}$ to the given branch point and then takes the opposite edge to that taken by $\mathscr{P}$. Since all these paths are closed by assumption, there must be some node covered on each of them, and, of course, on the part of them that differs from $\mathscr{P}$. Since we are dealing with a tree, these subpaths are clearly disjoint, providing us with n distinct covered nodes in addition to ℓ, thus at least $n + 1$ altogether.

We will now prove Theorem 2 by showing that if a canonical representative $\mathscr{R}$ with principal function letter f is nonlinear and nonsimple, we can find a sequence $\{V_i\}_{i=1}^{\infty}$ of terminal execution graphs for $\mathscr{R}$ such that for each i, V_i contains a binary tree T_i of depth i as a subgraph.

So assume $\mathscr{R}$ is a nonlinear nonsimple canonical representative with (an n-ary) principal function letter f. Since $\mathscr{R}$ is mutual, nonlinear, and free, there is a production sequence from $f(x_1, ..., x_n)$ to a collapsed tree with two (possibly nested) occurrences of f, one of which has no nonterminals beneath it. Since it is mutual and not simple, there is a production sequence from $f(x_1, ..., x_n)$ to a collapsed tree with an occurrence of f which is not on every path from argument to root. Since every production sequence is consistent, these two production sequences can be combined to produce a production sequence from $f(x_1, ..., x_n)$ to an execution graph U_1 with two occurrences of f such that one, which we denote by f^*, is on a path $\mathscr{P}$ from an argument to the root that does not contain the other f, which we denote by $f^{\dagger}$.

For simplicity we now divide our proof into the considerations of two different cases (but we note that we could combine our two procedures for generating the sequences $\{V_i\}_{i=1}^{\infty}$ mentioned above to form a more complicated procedure that would work in both cases).

In the first case, every path from $f^{\dagger}$ to the root of U_1 misses f^*. Proof of nonflowchartability generalizable to this case is given in [7 and 5], and one can be given that is very similar to the proof of Theorem 3 of this paper (see Section 6). Let us thus turn to the second case, and assume that in U_1, there is a path $\mathscr{P}^\dagger$ from $f^\dagger$ to the root of U_1 that passes through f^*:

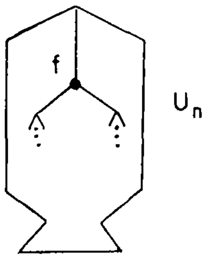


We will now show how to form a sequence $\{U_i\}_{i=1}^{\infty}$ of expression graphs of $\mathscr{R}$ (whose first member is our U_1 described above and from which $\{V_i\}_{i=1}^{\infty}$ will be formed by terminating nonterminals) such that

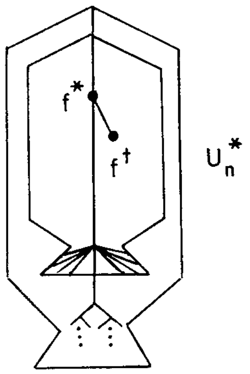
(1) for each i, U_i contains a binary tree T_i of depth i as a subgraph;

(2) an f occurs below the root of T_i, closer to the root than any branch point (other than itself, if it is one).

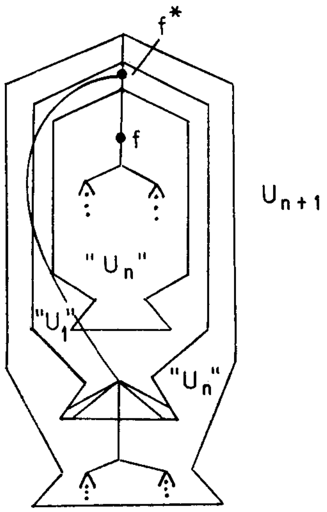
It is clear that U_1 satisfies (1) and (2). Having constructed a U_n satisfying (1) and (2), we construct U_{n+1} as follows: We first form a $U_n{}^*$ by expanding the f of U_n given by condition (2) according to the production sequence given by U_1. Thus, if we have a U_n with its binary subgraph of depth n:



then $U_n{}^*$ is



To obtain U_{n+1}, we then expand the $f^\dagger$ (just introduced by the U_1 expansion) according to the production sequence used to obtain U_n:



Since no collapsings can occur (Lemma 5) and since there is an operation symbol immediately covering all the arguments in U_1, conditions (1) and (2) hold for U_{n+1}: our U_1-type expansion has provided for another branching at the top of the binary subgraph and also the f to satisfy (2).

To finish the proof we note that we can terminate all the nonterminals in each U_i to form a V_i and can choose appropriate subpaths within the graphs we substitute so as to again have binary trees of depth i in each V_i. This completes the proof of Theorem 2.

6. The Flowchartability of Canonical Representatives with Invertible Operations

In this section we present our second main result, namely, a simple and decidable characterization of flowchartability for canonical representatives under interpretations in which all the operations are assumed to be invertible. The proof of the result actually shows us how to flowchart recursions that are not necessarily canonical representatives and in which, moreover, not all the operations are assumed to be invertible.

Thus, let us say that a recursion $\mathscr{R}$ has *invertible operations* if we consider only interpretations such that, for every operation b of n arguments occurring in the definition of $\mathscr{R}$, and for every $i = 1, 2,..., n$, there is an operation d_i such that

$$d_i(b(x_1, ..., x_n)) = x_i .$$

(In terms of playing the graph game $\mathscr{G}$, this means we can put a stone on a node if *any* node directly above it has a stone on it.) Further, let the nonterminals in a collapsed tree of a recursion equation be *nested*, if, for every argument, there is a path from the argument to the root that contains all the nonterminals of the tree.

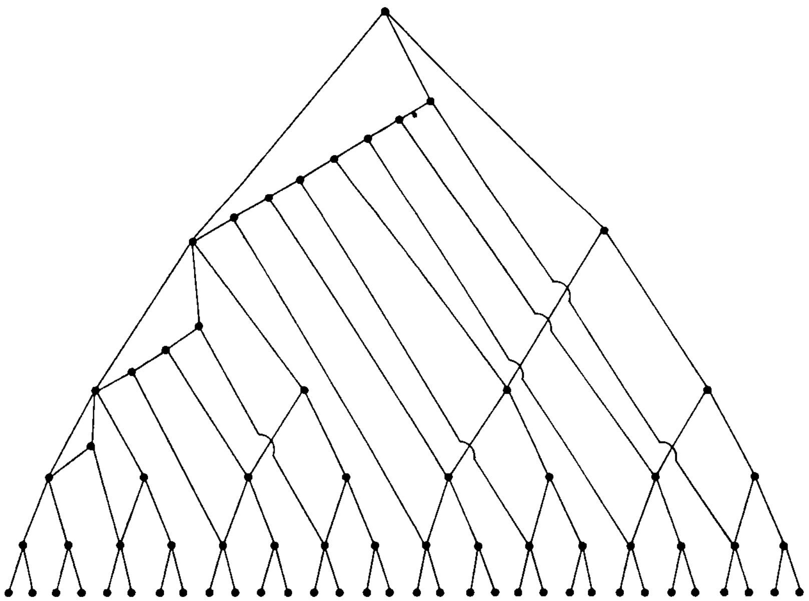


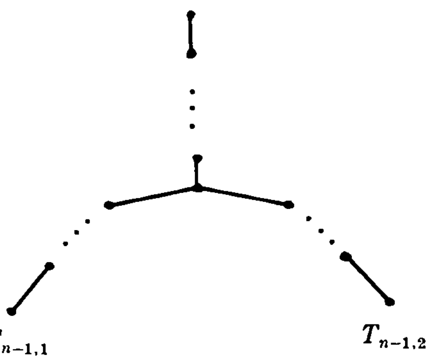
FIG. 3. Graph containing a binary subgraph of depth 5 but requiring only three registers for execution assuming invertibility.

The nonterminals of a recursion are then said to be *nested* if they are nested in every collapsed tree of the rules of the defining equations. Because of the similarity of its proof to that of the last section, we first present the "negative" half of our result.

THEOREM 3. *If the nonterminals of $\mathscr{R}$, a canonical representative with invertible operations, $\mathscr{R}$ are not nested, $\mathscr{R}$ is not flowchartable.*

When we restrict ourselves to interpretations in which the very strong invertibility we have defined holds, in order to prove nonflowchartability, it does not suffice to show that we can find terminal expression graphs with binary subgraphs of arbitrary depth. In fact, the preceding graph (Fig. 3) contains a binary subgraph of depth 5, but the graph game (modified to allow downward movements) can be played on it with only 3 (not 6) stones (the example generalizes to an expression graph with a binary subgraph of *arbitrary* depth that can be computed with three locations). By considering binary tree subgraphs that are "independent" enough, however, we can obtain our desired result.

Let us first consider the following way of looking at a binary tree T_n of depth n. T_n is clearly of the form



where $T_{n-1,1}$ and $T_{n-1,2}$ are binary trees of depth $n - 1$. Each of these subtrees can then also be viewed similarly (as binary trees of depth one with leaves certain binary trees of depth $n - 2$), etc. We say that this way of looking at T_n provides us with a *hierarchical decomposition* of T_n (clearly there may be more than one such decomposition for a given T_n).

Further, if S is a subset of nodes of an expression graph E, we define the *closure* S^* of S in E to be the subgraph formed by all the paths in E connecting points of S. We note that if the nodes of S form a tree, S^* consists of the paths in E between the leaves and root of S. For convenience we will use the notation S^{**} to denote the closure in E of the set consisting of the nodes of S and the root of E.

We can now state the lemma we need for the proof of Theorem 3 as follows.

LEMMA 10. *Let E be an expression graph with a binary tree T of depth n as a subgraph. Even assuming full invertibility for all the operations in E, any flowchart computing E will require at least n locations if there is a hierarchical decomposition of T satisfying the following conditions*:

(1) *if S is a subtree in the decomposition of T, every path in E from a leaf of S to the root of E (i.e., a maximal path in S^{**}) must pass through the root of S*;

(2) *if S and S' are subtrees in the hierarchical decomposition of T that are of the same depth, but which are distinct with respect to the decomposition, then S^{**} and S' are disjoint.*

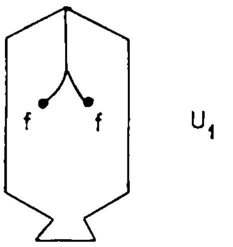
Proof. Assume given expressions E and T satisfying the hypothesis of the lemma. We will show that it requires at least n stones to compute the graph game on E.

Thus, consider the last time in a playing of the game *on* E in which there is a leaf l of T for which *every* path from l to the root of E is open. At the next step some path from l to the root of E must be closed, and in fact all of them, for the only possibility is the placing of a stone on l. (This is because all other nodes on paths from l to the root of E have at least one of their successors uncovered and thus cannot be covered in the usual upward moving way. Moreover, since *none* of the paths have *any* stone on them, invertibility assumptions cannot be used to move a stone back down an edge.)

Now let T_1 be the binary tree of depth 1 (in the assumed hierarchical decomposition) that contains l, let T_2 be the tree of depth 2 that contains T_1, etc. We thus know that at the step mentioned above in which a stone is placed on l, there is some stone in $T_1{}^*$. To prove the lemma, we now show that for $i = 2,..., n$, there is a stone on a node in $T_i{}^*$ that is not a node of T_{i-1}^*. Since by (1) all of the paths from l to the root of E pass through the root r_i of T_i, none of the paths from r_i to the root of E can be closed. Now consider any leaf l' of T_i not in T_{i-1}. By assumption, some path from l' to the root of E must be closed. Each such path must pass through r_i, however, and all paths from r_i to the root of E are open. Thus, there must be a path from l' to r_i that is closed. It remains to see that the node closing this path, which is clearly in $T_i{}^*$, is not in T_{i-1}^*. But this is clear from condition (2) above (since l' was assumed to be in a subtree of depth $i - 1$ different from T_{i-1}).

We can now prove Theorem 3 by showing that if we have a canonical representative $\mathscr{R}$ whose nonterminals are not nested and assume invertibility, then we can find a sequence of terminal expressions $\{V_i\}_{i=1}^{\infty}$ for $\mathscr{R}$ such that for every i, V_i has a binary subgraph of depth i for which there is a hierarchical decomposition satisfying the conditions (1) and (2) of Lemma 10.

Thus let $\mathscr{R}$ be a canonical representative with invertible operations and with principal function letter f whose nonterminals are not nested. By an argument similar to the one at the beginning of the proof of Theorem 2, we can find a production sequence that leads to a collapsed tree of the form



where there may be nonterminals other than the distinguished f's, but where there is no path that nests the two f's. We call these two f's *leaves* of U_1. For each n, U_n will have 2^n f's which are distinguished as leaves. To form U_{n+1}, we expand each of the leaves of U_n according to the production sequence used to obtain U_1 and call the f's in the result that came from the U_1 leaves in the copies of U_1, the *leaves* of U_{n+1}. In addition, we also distinguish the roots of the copies of U_1, and call them *root nodes* of U_{n+1} (along with the images of the root nodes of U_n and the root of the whole graph). Thus, for example, we have Fig. 4

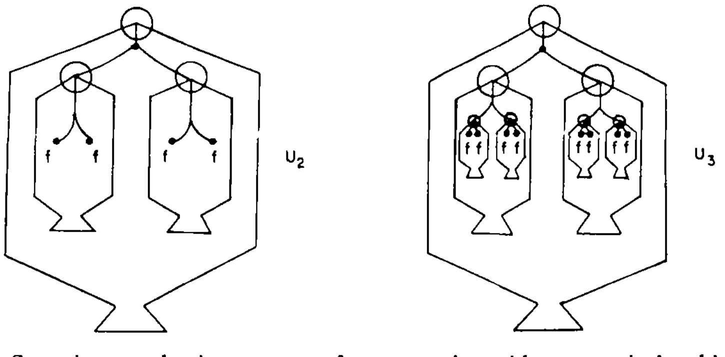


FIG. 4. Steps in a production sequence for a recursion with nonterminals which are not nested.

where the root nodes are circled and the leaves labeled with f's. The terminal trees we wish to consider are the expression graphs $\{V_i\}$ that result when we systematically terminate all the nonterminals in the graphs of the sequence $\{U_i\}$. We call the images of the root nodes and leaves of U_i root nodes and leaves of V_i. By Lemma 5, there is a binary tree T_i of depth i imbedded as a subgraph in V_i in such a way that the leaves of V_i are the leaves of the T_i and the root nodes of V_i form the roots of the trees which can be used for a hierarchical decomposition of T_i satisfying the conditions of Lemma 10.

This completes the proof of Theorem 3.

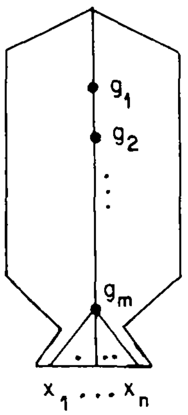
We now consider the converse of Theorem 3. We will actually prove a stronger theorem, this being perhaps our most interesting new result. First, a definition: a *path* $\mathscr{P}$ in a collapsed tree of a recursion is said to be *invertible* if for every operation b on $\mathscr{P}$, if the root of the i-th argument of b is on $\mathscr{P}$, there is an operation c_i such that $c_i(b(x_1, \ldots, x_n)) = x_i$. (This means we have just the operations that allow us to move down edges from operations on $\mathscr{P}$.) We now show the following.

THEOREM 4. *If $\mathscr{R}$ is a conservative recursion such that for every expression graph E in a rule of $\mathscr{R}$ and for every argument x of E, there is an invertible path from x to the root of E that nests all the nonterminals of E, then $\mathscr{R}$ is flowchartable.*

Proof. We will show that if $\mathscr{R}$ is a recursion satisfying the conditions of the theorem, there is a uniform bound on the number of locations required to execute terminal expressions of its computation sequences.

For simplicity, we first note that for any rule in $\mathscr{R}$, we can choose the invertible paths from root to arguments nesting the nonterminals so that they coincide above the lowest nonterminal. Thus, every rule is of the form

$f_i(x_1, \ldots, x_n) =$



if $p_i(x_1, \ldots, x_n)$

where the g's are all the nonterminals and the invertible paths are denoted by straight lines. For each rule r_i, let N_i be a number of locations that would suffice to compute the expression graph of the rule if the nonterminals were assumed to be operations and no invertibility assumptions were made (for instance, N_i can trivially be taken to be the number of nodes in the graph). Let $N = \max\{N_i\} + 1$. We will now sketch a proof showing that it requires at most N locations to calculate any terminal expression of a computation sequence of the recursion.

We first remark that in this proof when we speak of a node being (an occurrence of) a nonterminal, or an argument of a nonterminal, we will mean it is such a nonterminal or argument, or the *image* of one. Further, if g is a nonterminal introduced into a graph by a substitution for a particular nonterminal f, we say f is the *parent* of g, g the *child* of f.

Now let us consider for a moment the form a computation sequence for $\mathscr{R}$ must take. The properties one must prove by induction about each graph in the computation sequence are the following:

(1) there is an invertible path containing all the nonterminals (both actual nonterminals and images of nonterminals);

(2) there is an invertible path from any nonterminal child to any argument of its parent.

The reader might find it easier to verify these properties after considering an example. For instance, a computation sequence for f might take the form of Fig. 5

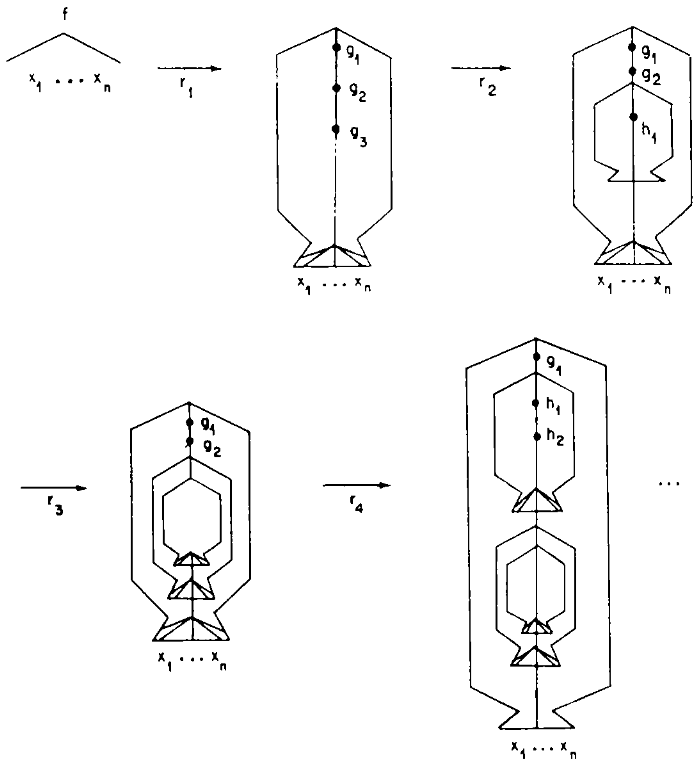


FIG. 5. A computation sequence for a recursion with nested nonterminals.

and might have a terminal expression of the form of Fig. 6 (here we have "labeled" the nonterminals with the names of the rules they were expanded by):

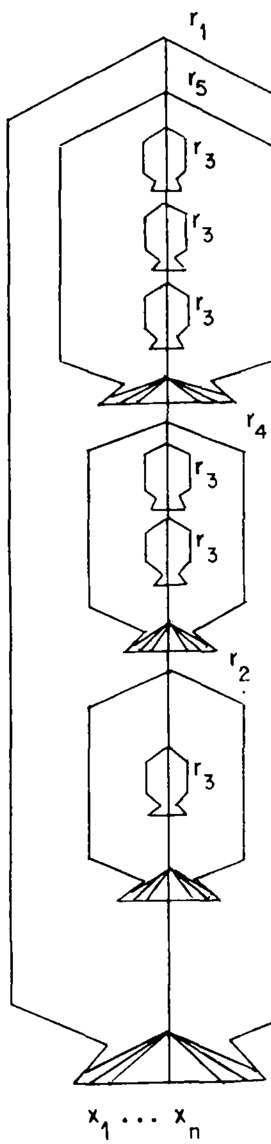


FIG. 6. A typical terminal expression for the sequence of Fig. 5.

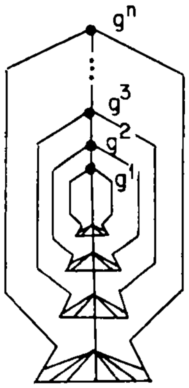
To show that we can calculate an arbitrary terminal expression graph E for $\mathscr{R}$ with N locations, we will show:

(a) (the subgraph whose root is) the lowest nonterminal can be calculated with N locations;

(b) if a given nonterminal can be calculated with N locations, the next nonterminal above it can be calculated with N locations.

(Here "above" and "below" are defined with respect to the path given by property 1 above.)

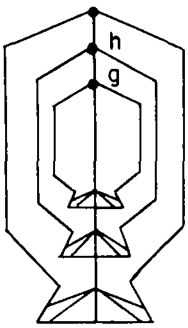
First let us see (a), i.e., that we can calculate the lowest nonterminal g^1 (an image, here, of course) in E with N locations. Clearly g^1 will be the image of the lowest nonterminal in its parent g^2, which will be the image of the lowest nonterminal in its parent g^3, etc. Thus, the part of the terminal tree we are concerned with has the following appearance:



(where g^n is the image of the original f). Now for each g^i in question, let r^i be the rule by which g^i was expanded and let $\mathscr{P}^i$ be a procedure which calculates the expression graph E^i of the rule (see the beginning of the proof) with N_i locations. Now we note that the subgraph with root g^i is the same as E^i except that the nonterminals and arguments have been replaced by terminal graphs.

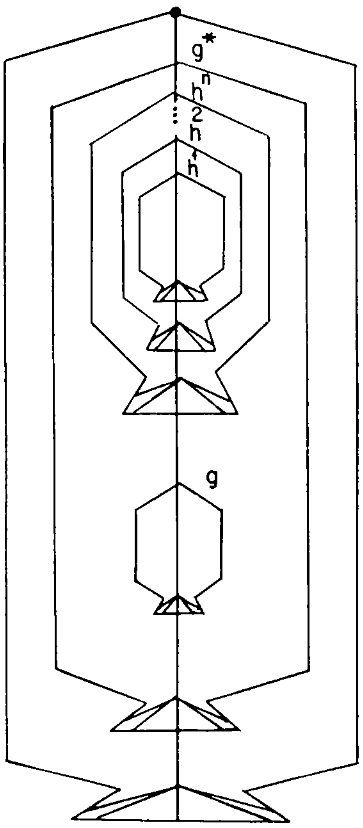
To calculate the subgraph whose root is g^1, we now proceed as follows: We start calculating E as if it were E_n (which requires $N_n < N$ locations) until the point at which we would calculate the value of its lowest nonterminal g^{n-1} if it were an operation. At this point, we have all the arguments of g^{n-1} stored. We now start calculating the subgraph with these arguments rooted at g^{n-1} as if it were E^{n-1} (using N_{n-1} locations) until the point at which we would calculate the value of g^{n-2}, etc. This process stops when we calculate the subgraph with root at g^1 as desired (here we use at most $N - 1$ locations).

Now let us prove (b), i.e., that if a given nonterminal g can be calculated with N locations, the first nonterminal h above it can be calculated with N locations. There are two cases to consider here. The first is one in which h is the parent of g. Here the part of the expression graph we are interested in is of the form



To calculate the subgraph with root at h, we first calculate the one with root at g with N locations as we can do by assumption. Leaving the value of g in a location, we then use the invertibility property provided by condition (2) and calculate all the arguments of h (with respect to the graph game, this means taking stones and running down the invertible paths to the required arguments; thus this requires only as many stones as h has arguments, i.e., $<N$). We then start calculating the subexpression with root at h according to the procedure $\mathscr{P}_h$ corresponding to the rule r_h by which h was expanded until we reach a point at which we want to "compute" the value of a nonterminal from the expression graph for r_h. If that nonterminal was below g we again use invertibility (this time through property (1)) to obtain the value of its subgraph. If the nonterminal is g, we just skip the step.

In the second case, then, the upper nonterminal, h^1, is not the parent of g, and the part of the expression graph we are interested in has the form (where we denote the parent of g by g^*, the parent of h^1 by h^2, etc.):



We calculate h^1 with N locations in the following manner. First, we calculate g with N locations, using our assumption. Leaving g in a location, we get the arguments of g^* by the invertibility provided by (2). We then start calculating g^* using the procedure corresponding to the rule used to obtain it and obtaining nonterminals below g by the invertibility provided by (1). At the point at which the nonterminal h^n would be calculated if it were an operation (i.e., we have all its arguments stored), we proceed as described in the proof of (a).

This completes the proof of Theorem 4.

We can finally combine Theorems 3 and 4 to obtain a complete, simple and decidable characterization of flowchartability for canonical representatives with invertible operations.

THEOREM 5. *A canonical representative with invertible operations is flowchartable if, and only if, its nonterminals are nested.*

7. Conclusion

In discussing the notion of a recursion presented in Section 2, we restricted attention to mutual recursions with no complicated predicate expressions. Our results can be extended in a straightforward way to much more complex recursions following, for example, the decomposition techniques presented in [8]. We have not mentioned any restriction to "total" interpretations since none is needed for our results: the arguments of recursive functionals (corresponding to operation and predicate symbols) are partial functions and predicates. However, totality assumptions are made in many of the papers to which we refer.

Since the assumption that operations and predicates are total has been traditional in this field, some care is needed in comparing our results with those of others. In particular, the new result of Plaisted [6], that a flowchart augmented by only one counter can be simulated by a "pure" flowchart with no counters, may depend on totality assumptions. In the context of this warning, we note that pure flowcharts may be obtained via [6] for all the linear recursions and all the nonlinear, simple, single consequent recursions, e.g., example V from the appendix. A decision for recursions represented by example VI would complete a characterization for "pure" flowchartability.

Our first main theorem, Theorem 2, characterizes flowchartability for canonical representatives. For general recursions, we noted earlier that simple recursions and linear recursions have been shown to be flowchartable. Also, a recursion is flowchartable if its canonical representative is. However, even for free, conservative recursions, it is possible for the recursion to be flowchartable while its canonical representative is not. Consider, for example,

$$f(x,y) = \begin{cases} f(x, f(x, a(y))) & \text{if} \quad p(x,y) \\ b(x,y) & \text{if} \quad \neg p(x,y). \end{cases}$$

In practice, however, we can usually use the characterization for canonical representatives (and its method of proof) to decide flowchartability for general recursions.

Invertibility assumptions change this situation. It can happen that a recursion is not flowchartable, even with invertibility assumptions, but its canonical representative is. A counterexample to Theorem 4 without the conservative hypothesis is

$$f(x) = \begin{cases} f(a(x, f(b(x)))) & \text{if} \;\; p(x) \\ f(d(f(b(x)))) & \text{if} \;\; q(x) \\ f(e(f(b(x))) & \text{if} \;\; r(x) \\ c & \text{if} \;\; s(x) \\ x & \text{if} \;\; t(x). \end{cases}$$

We omit the long, but straightforward, proof that this example is not flowchartable, even with invertibility, in favor of a suggestion to apply Lemma 10. Since the example is nested, Theorem 5 states that its canonical representative is flowchartable, given appropriate invertibilities.

We summarize our characterizations (Theorems 2 and 5) by examples in the appendix. In this paper the notions of consequent and subconsequent are only defined ostensively. For precise definitions, see [7].

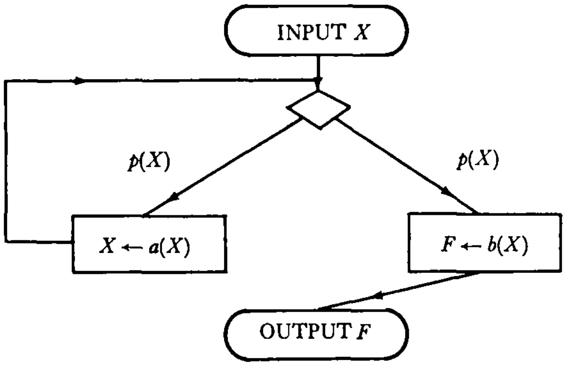
8. Appendix

I. Iterative Form.

Recursion:

$$f(X) = \begin{cases} f(a(X)) & \text{if} \quad p(X \\ b(X) & \text{if} \quad \neg p(X). \end{cases}$$

Flowchart:

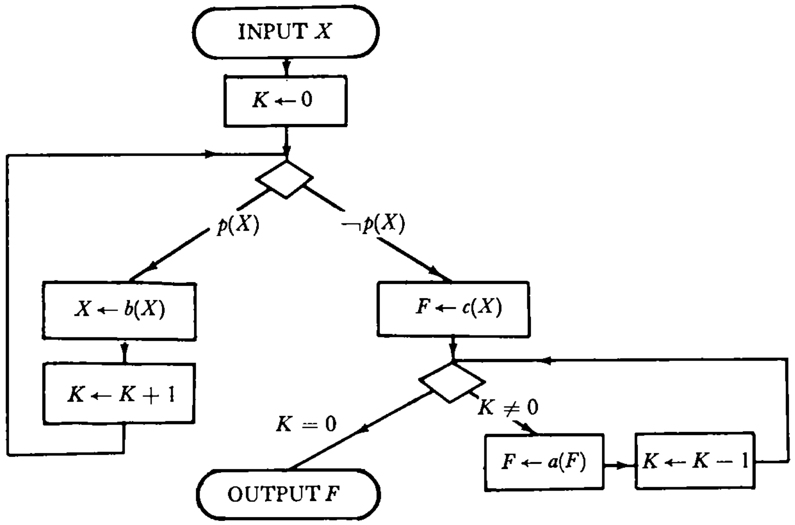


II. Linear—Simple—No Multiple Subconsequents (see [7] for definition).

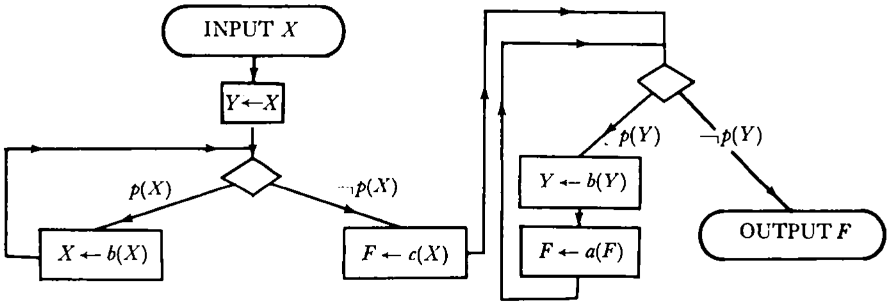
Recursion:

$$f(X) = \begin{cases} a(f(b(X))) & \text{if} \quad p(X) \\ c(X) & \text{if} \quad \neg p(X). \end{cases}$$

Flowchart (with one counter):



Flowchart (no counters):

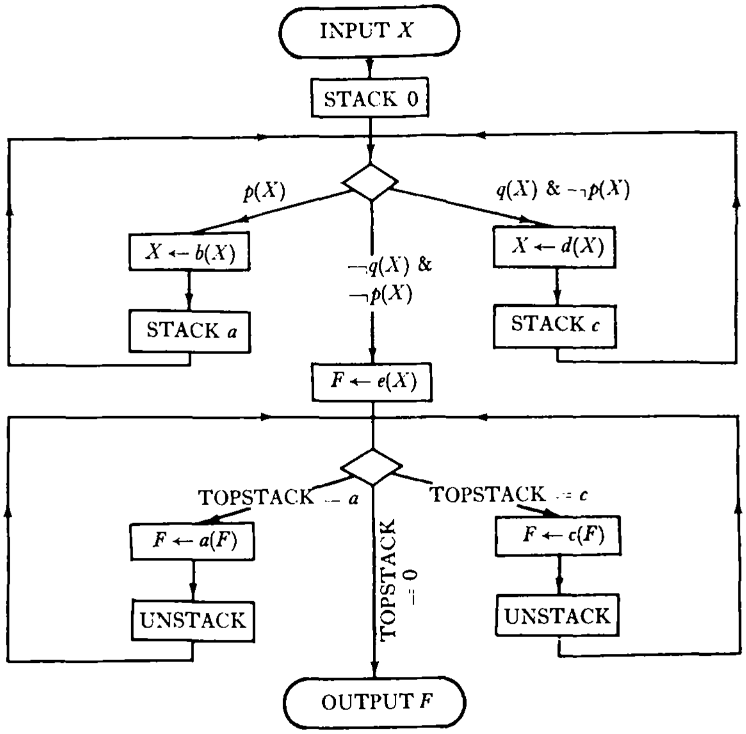


III. Linear—Simple—Multiple Subconsequents.

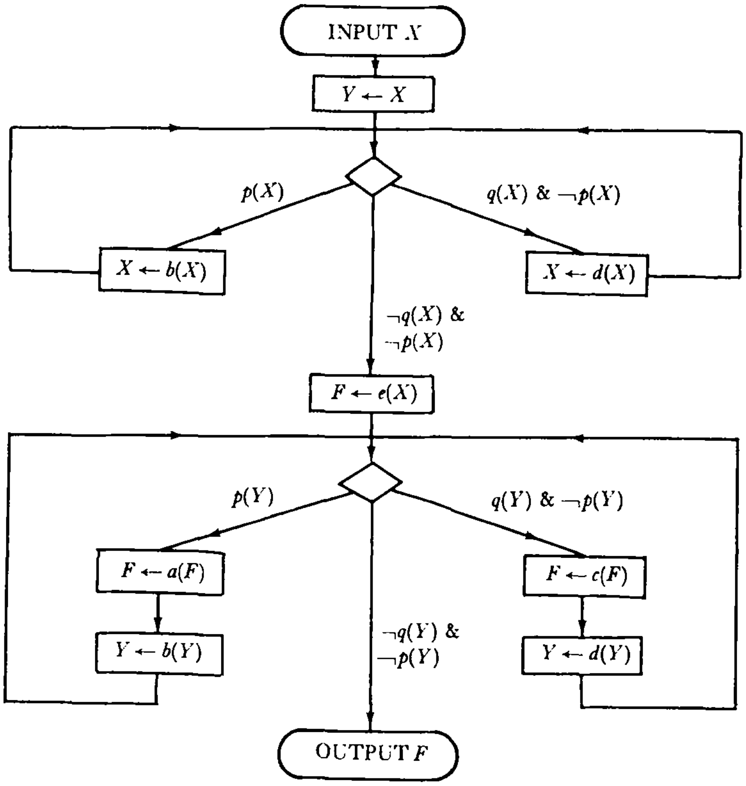
Recursion:

$$f(X) = \begin{cases} a(f(b(X))) & \text{if } p(X) \\ c(f(d(X))) & \text{if } q(X) \,\&\, \neg p(X) \\ e(X) & \text{if } \neg q(X) \,\&\, \neg p(X). \end{cases}$$

Flowchart (with control stack):



Flowchart (no counters):

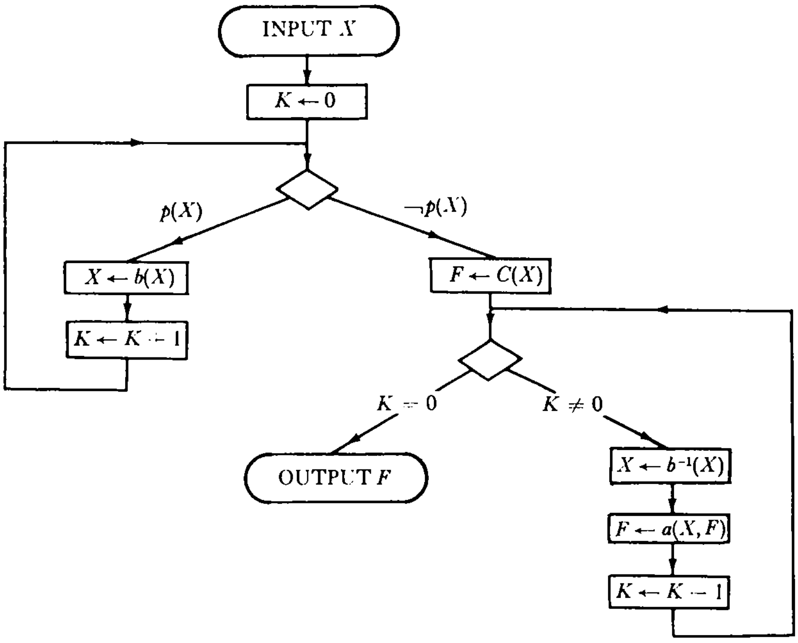


IV. Linear—Not Simple

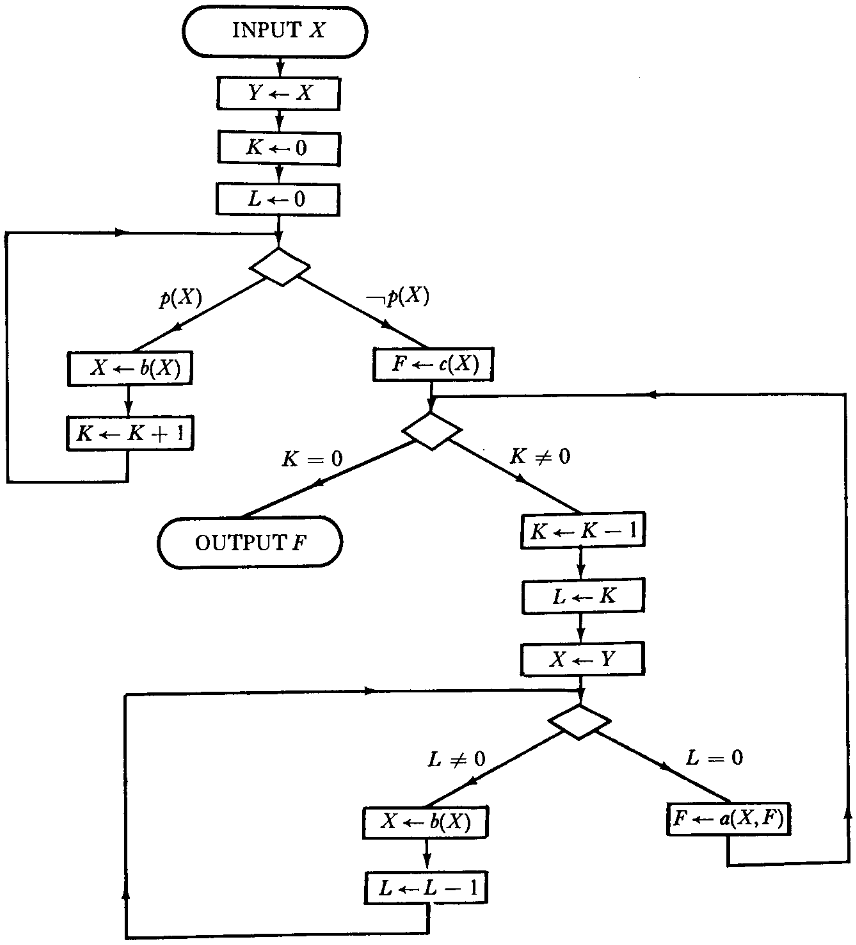
Recursion:

$$f(X) = \begin{cases} a(X, f(b(X)) & \text{if} \quad p(X) \\ c(X) & \text{if} \quad \neg p(X). \end{cases}$$

Flowchart (with one counter):
Assuming invertibility of b



Flowchart (with two counters):



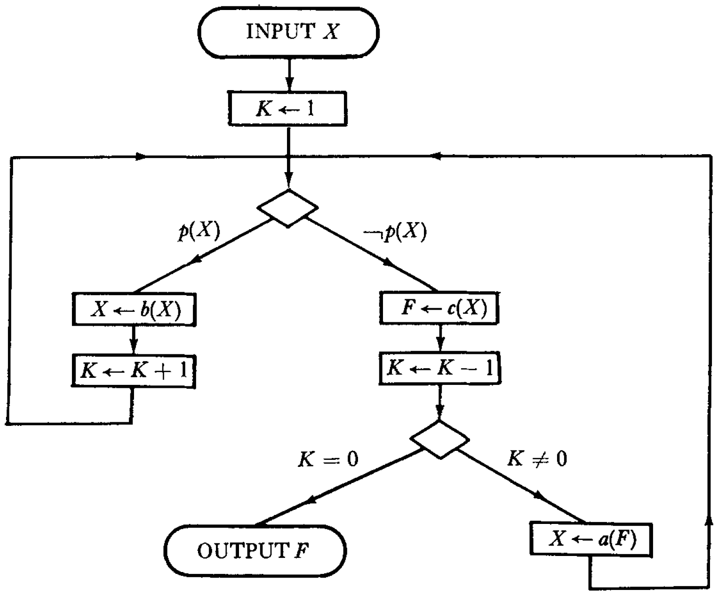
Flowchart (with no counters)
(See [5])

V. Nonlinear—Simple—No Multiple Consequents (see [7] for definition)

Recursion:

$$f(X) = \begin{cases} f(a(f(b(X)))) & \text{if} \quad p(X) \\ c(X) & \text{if} \quad \neg p(X). \end{cases}$$

Flowchart (with one counter):

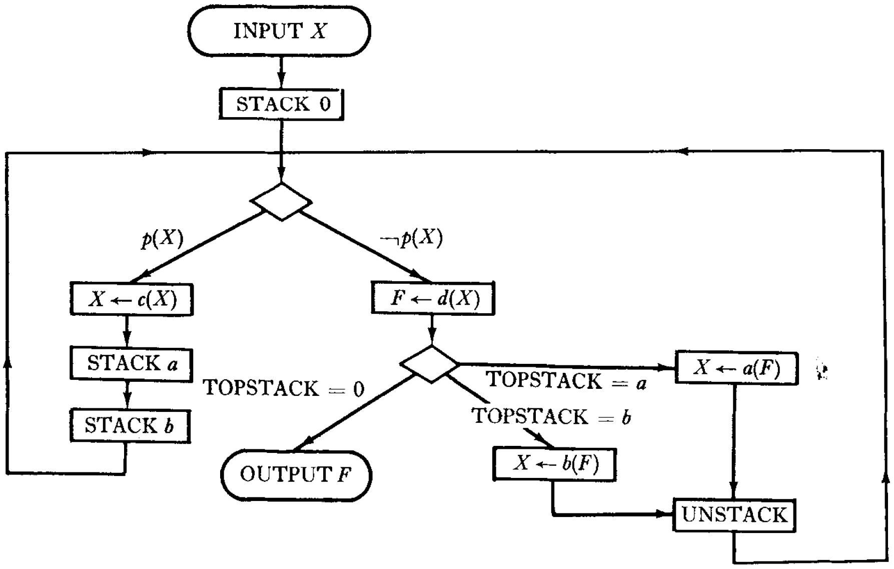


VI. Nonlinear—Simple—Multiple Consequents.

Recursion:

$$f(X) = \begin{cases} f(a(f(b(f(c(X)))))) & \text{if} \quad p(X) \\ d(X) & \text{if} \quad \neg p(X). \end{cases}$$

Flowchart (with control stack):



VII. Nonlinear—Not simple—Nested.

Recursion:

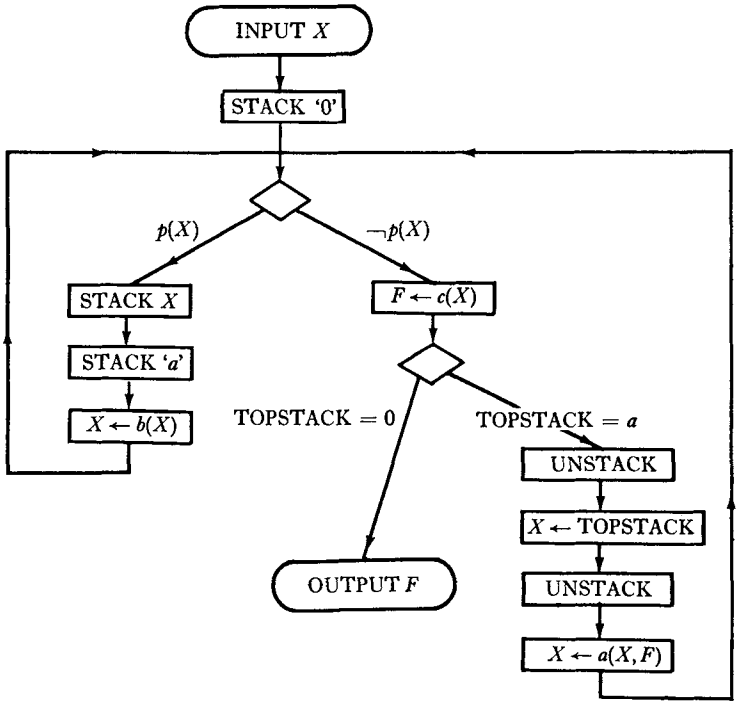
$$f(X) = \begin{cases} f(a(X, f(b(X)))) & \text{if} \quad p(X) \\ c(X) & \text{if} \quad \neg p(X). \end{cases}$$

Not flowchartable with arbitrary control.
If we denote by t^*, t without its root and by $t^\dagger$, t without its argument, the following sequence of expressions is not uniformly executable with a finite number of locations:

$$t_1(X) = c(a(X, c(b(X))))$$

$$t_{n+1}(X) = c(a(t_n{}^\dagger(X), t_n{}^*(b(t_n{}^\dagger(X))))).$$

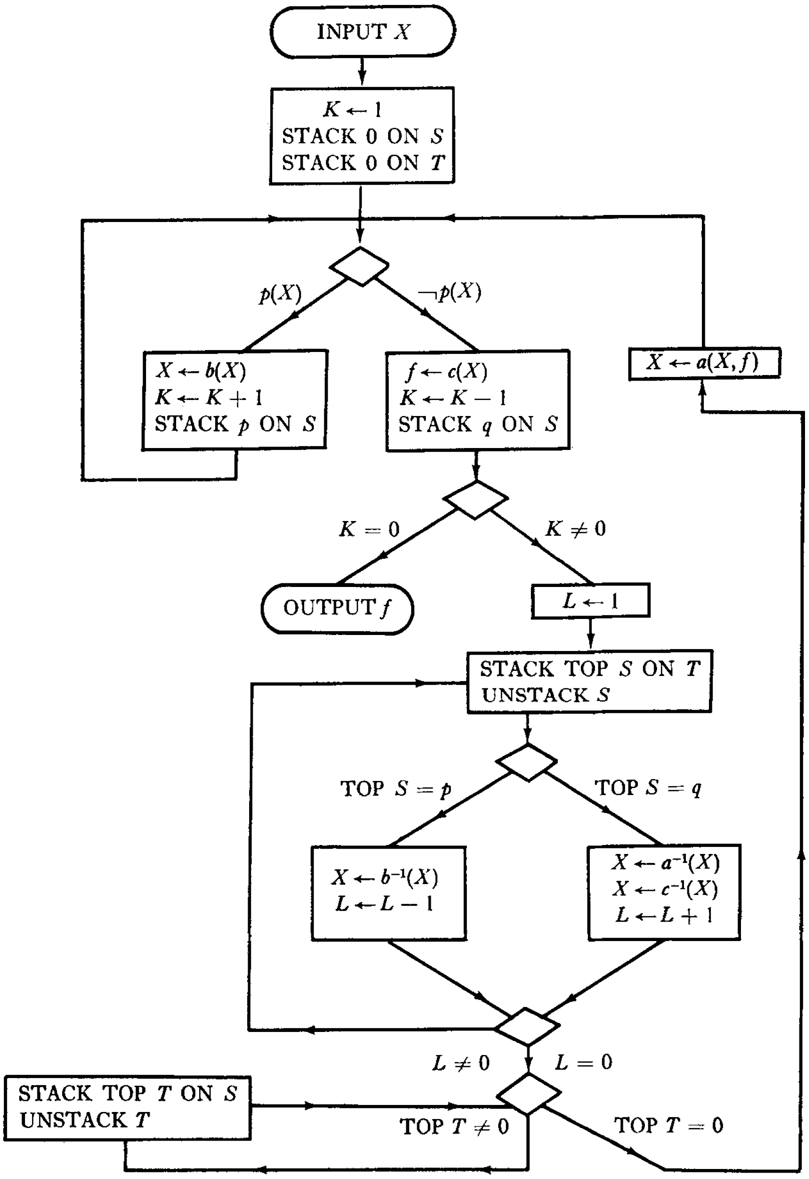
"Flowchart" (with *data storage* stack):



Flowchart (with counter K and control stacks S and T).

Assuming invertibility of a, b, and c:

$$a^{-1}(a(X, Y)) = Y$$
$$b^{-1}(b(X)) = X$$
$$c^{-1}(c(X)) = X$$

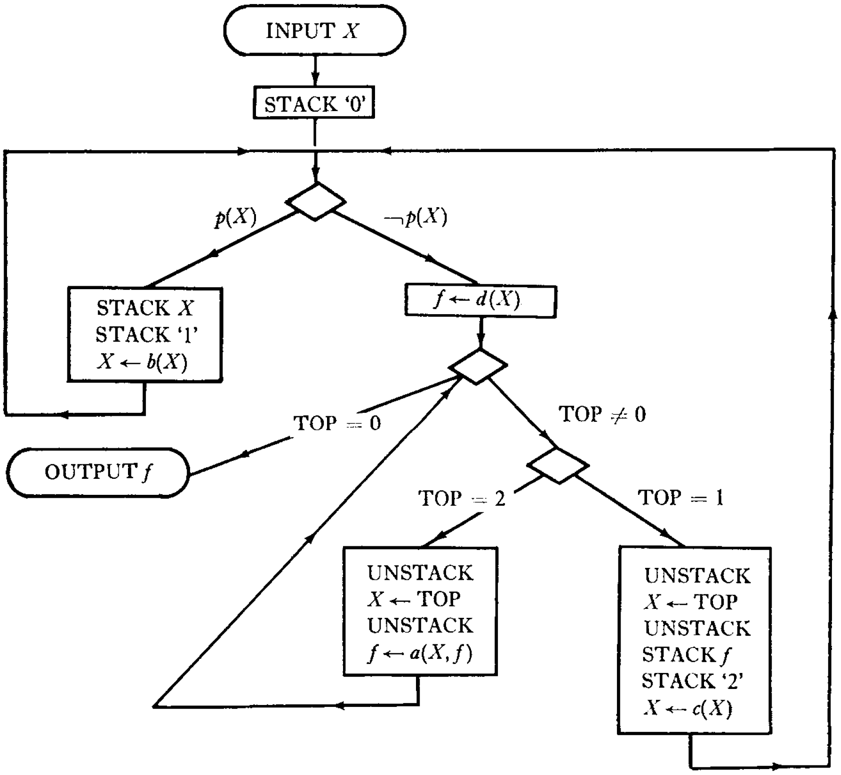


VIII. Nonlinear—Nonsimple—Nonnested.

Recursion:

$$f(X) = \begin{cases} a(f(b(X)), f(c(X))) & \text{if} \quad p(X) \\ d(X) & \text{if} \quad \neg p(X) \end{cases}$$

Not flowchartable with arbitrary control.
"Flowchart" with *data storage* stack:



Acknowledgments

The authors wish to thank B. K. Rosen for his interest and helpful suggestions. They also wish to thank Mrs. J. Bennett for her patient and efficient work in preparing the manuscript, and Mrs. K. Fanti for preparation of the revision.

REFERENCES

1. J. W. DEBAKKER AND D. SCOTT, "A Theory of Programs," unpublished (as described in [2]), August, 1969.
2. S. J. GARLAND AND D. C. LUCKHAM, "Program Schemes, Recursion Schemes, and Formal Languages," Technical Report UCLA-ENG-7154, June, 1971.
3. D. C. LUCKHAM, D. M. R. PARK, AND M. S. PATERSON, On formalized computer programs, *J. Comput. System Sci.* 4 (1970), 220–249.
4. M. S. PATERSON, "Equivalence Problems in a Model of Computation," Ph.D. Dissertation, Univ. of Cambridge, Cambridge, England, 1967.
5. M. S. PATERSON AND C. E. HEWITT, Comparative Schematology, Record of Project MAC Conference on Concurrent Systems and Parallel Computation, June, 1970, 119–128, ACM, New Jersey, December, 1970.
6. D. A. PLAISTED, Flowchart Schemes with Counters, Conference Record of Fourth Annual ACM Symposium on Theory of Computing, 1972.
7. H. R. STRONG, Translating recursion equations into flow charts, *J. Comput. System Sci.* 5 (1971), 254–285.
8. H. R. STRONG AND S. A. WALKER, "Properties Preserved Under Recursion Removal," IBM Research Report RC-3633, November, 1971.
9. S. A. WALKER, "Some Graph Games Related to the Efficient Calculation of Expressions," IBM Research Report RC-3628, November, 1971.