University of Tennessee, Knoxville
Trace: Tennessee Research and Creative Exchange

The Harlan D. Mills Collection | Science Alliance

2-1972

Mathematical Foundations for Structured Programming

Harlan D. Mills

Follow this and additional works at: http://trace.tennessee.edu/utk_harlan

Part of the Mathematics Commons, and the Software Engineering Commons

Recommended Citation
Mills, Harlan D., "Mathematical Foundations for Structured Programming" (1972). *The Harlan D. Mills Collection.*
http://trace.tennessee.edu/utk_harlan/56

This Book is brought to you for free and open access by the Science Alliance at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in The Harlan D. Mills Collection by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

Mathematical Foundations for Structured Programming

By Harlan D. Mills

Chief Programmer Team Operations

IBM

FSC 72-6012

MATHEMATICAL FOUNDATIONS

FOR

STRUCTURED PROGRAMMING

by

Harlan D. Mills

February 1972

SEP

Federal Systems Division
INTERNATIONAL BUSINESS MACHINES CORPORATION
Gaithersburg, Maryland 20760

©Copyright 1972 International Business Machines Corporation

The first name in structured programming is Edsger W. Dijkstra (Holland), who has originated a set of ideas and a series of examples for clear thinking in the construction of programs. These ideas are powerful tools in mentally connecting the static text of a program with the dynamic process it invokes in execution. This new correspondence between program and process permits a new level of precision in programming. Indeed, it is contended here that the precision now possible in programming will change its industrial characteristics from a frustrating, trial and error activity to a systematic, quality controlled activity.

However, in order to introduce and enforce such precision programming as an industrial activity, the ideas of structured programming must be formulated as technical standards, not simply as good ideas to be used when convenient, but as basic principles which are always valid. A good example of a technical standard occurs in logic circuit design. There, it is known, from basic theorems in boolean algebra, that any logic circuit, no matter how complex its requirement, can be constructed using only AND, OR, and NOT gates.

Our interest is similar, to provide a mathematical assurance, for management purposes, that a technical standard is sound and practical. This mathematical assurance is due, in large part, to Corrado Böhm and Giuseppe Jacopini (Italy), who showed how to prove that relatively simple (structured) program control logics were capable of expressing any program requirements.

Initial practical experience with structured programming indicates there is more than a technical side to the matter. There is a psychological effect, as well, when programmers learn of their new power to write programs correctly. This new power motivates, in turn, a new level of concentration, which helps avoid errors of carelessness. This new psychology of precision has a mathematical counterpart in the theory of program correctness, which we formulate in a new way.

The mathematical approach we take in formulating structured programming and the correctness problem emphasizes these combinatorial aspects, in order to demonstrate for programmers that correct programming involves only combinatorial selection, and not problems requiring perfect precision, on a continuous scale. Because of this, we are confident that programmers will soon work at a level of productivity and precision which will appear incredible compared to early experience with the programming problem.

CONTENTS

	Page
Complexity and Precision in Programming	1
The Psychology of Precision	2
The Problem of Complexity	3
The Idea of Structured Programming	4
The Correctness of Structured Programs	6
Functions	8
Functions and Rules	8
Function Composition and Completion	10
Special Functions	11
Programs	12
Control Graphs	13
Programs in Flowchart Form	14
Program Execution	16
Proper Programs	17
Program Equivalence	18
Program Expansions	19
Control Graph Labels	20
Program Formulas	22
Program Descriptions	25
Structured Programs	27
The Number of Control Lines in a Proper Program	30
Lemma	30
Proof	30
Structure Theorem	31
Proof	31
Summary	38
Top Down Corollary	39
S-Structured Programs	40
Program Representations	40
Program Trees	41

	Page
Program Correctness	43
Correctness Theorem	43
Proof	44
Correctness Notes	49
Top Down Program Expansions	50
Expansion Theorem (Verbal Version)	51
Expansion Theorem (Set Theoretic Version)	52
Proof	53
Indeterminate Programs	57
References	59

COMPLEXITY AND PRECISION IN PROGRAMMING

The digital computer has introduced a need for highly complex, precisely formulated, logical systems on a scale never before attempted. Systems may be large and highly complex, but if human beings, or even analog devices, are components in them, then various error tolerances are possible, which such components can adjust and compensate for. However, a digital computer, in hardware and software, not only makes the idea of perfect precision possible -- it requires perfect precision for satisfactory operation. This complete intolerance to the slightest logical error gives programming a new character, little known previously, in its requirements for precision on a large scale.

The combination of this new requirement for precision, and the commercial demand for computer programming on a broad scale, has created many false values and distorted relationships in the past decade. They arise from intense pressure to achieve complex and precision results in a practical way without adequate technical foundations. As a result, a great deal of programming uses people and computers highly inefficiently, as the only means presently known to accomplish a practical end.

It is universally accepted today that programming is an error-prone activity. Any major programming system is presumed to have errors in it. Only the very naive would believe otherwise. The process of debugging programs and systems is a mysterious art. Indeed, more programmer time goes into debugging than into program designing and coding in most large systems. But there is practically no systematic literature on this large undertaking.

Yet, even though errors in program logic have always been a source of frustration, even for the most careful and meticulous, this may not be necessarily so in the future. Programming is very young as a human activity -- some twenty years old. It has practically no technical foundations yet. Imagine engineering when it was twenty years old. Whether that was in 1620 or 1770, it was not in very good technical shape at that stage either! As technical foundations are developed for programming, its character will undergo radical changes.

We contend here that such a radical change is possible now -- that in structured programming the techniques and tools are at hand to permit an entirely new level of precision in programming.

This new level of precision will be characterized by programs of large size (from tens of thousands to millions of instructions) which have

mean time between detected errors of a year or so. But to accomplish that level of precision a new attitude toward programming expectations will be required in programmers, as well.

THE PSYCHOLOGY OF PRECISION

A child can learn to play the game of tic tac toe perfectly -- but a man can never learn to saw a board exactly in half. Playing tic tac toe is a combinatorial problem, selecting, at every alternative, one of a finite number of possibilities. Sawing a board exactly in half is a physical problem, for which no discrete level of accuracy is sufficient.

The child who has learned to play tic tac toe need never make a mistake, except through a loss of concentration. In any game he believes important (say played for a candy bar) he is capable of perfect play.

Computer programming is a combinatorial activity, like tic tac toe, not like sawing a board in half. It does not require perfect resolution in measurement and control -- it only requires correct choices out of finite sets of possibilities at every step. The difference between tic tac toe and computer programming is complexity. The purpose of structured programming is to control complexity through theory and discipline. And with complexity under better control, it now appears that men can write substantial computer programs correctly. In fact, just as a child moves from groping and frustration to confidence and competence in tic tac toe, so men can now find solid ground for program development.

A child, in learning to play tic tac toe, soon develops a little theory, dealing with ''center squares'', ''corner squares'', ''side squares'', and the self discipline to block possible defeats before building threats of his own. In programming, theory and discipline are critical, as well, at a man's level of intellectual activity. Structured programming is such a theory, which provides a systematic way of coping with complexity in program design and development. It makes possible a discipline for program design and construction on a level of precision not previously possible.

But for the child, knowing how to play tic tac toe perfectly is not enough. He must know that he knows. This knowing that he knows is a vital ingredient in his self discipline -- knowing that he is capable of analyzing the board, and doesn't need to guess and hope.

It is the same with the programmer. If a programmer knows that what is in his mind is correct, then getting it onto paper precisely is more important, as is checking details of data definitions, and whatever, in his coding process. On the other hand, if a programmer thinks what is in his mind is probably all right, but is subconsciously counting on

debugging and integration runs to iron out logic and interface errors, then the entire process of getting it onto paper and into the computer suffers in small ways to later torment him.

It takes some learning on the part of experienced programmers to discover that structured programs can be written with unprecedented logical and interface precision. As with the child, it is not enough to be able to program with precision. The programmer must know his capability for precision programming in order to supply the concentration to match his capability.

THE PROBLEM OF COMPLEXITY

Five hundred years ago men did not know that the air we breathe, and move through so freely, has weight. Air is hard to put on a scale, or even identify as any specific quantity for weighing at all. But now we know that air has weight -- at sea level, the weight of a column of water 34 feet high.

It is easy to imagine, in hindsight, the frustrations of a well pump manufacturer, whose ''research department'' is operating on the theory that ''nature abhors a vacuum''. Water can be raised up a well pipe 15, 20, then 25 feet, by using a plunger and tightening its seals better and better. All this merely seems to confirm the ''current theory'' about the operation of such pumps. But at 35 feet, total frustration ensues. No matter how tight the seals, the water cannot be raised.

In computer programming today, we do not yet know that ''complexity has weight''. Since it is not easily measured or described, like storage requirements or throughput, we often ignore the complexity of a planned program or subprogram. But when this complexity exceeds certain unknown limits, frustration ensues. Computer programs capsize under their own logical weight, or become so crippled that maintenance is precarious and modification is impossible. Problems of storage and throughput can always be fixed, one way or another. But problems of complexity can seldom be adequately recognized, let alone fixed.

The syndrome of creating unsolvable problems of complexity because of anticipated problems of storage and throughput is well known. It is the work of amateurs. It arises in a misguided arrogance that ''what happened to them won't happen to me!'' But it keeps happening, over and over.

THE IDEA OF STRUCTURED PROGRAMMING

Closely related to many original ideas of E. Dijkstra [10], and using key results of C. Böhm and G. Jacopini [5], P. Naur [32], and R. Floyd [13], structured programming is based on new mathematical foundations for programming (in contrast to the use of programming to implement mathematical processes, or to study foundations of mathematics). It identifies the programming process with a step by step expansion of mathematical functions into structures of logical connectives and subfunctions, carried out until the derived subfunctions can be directly realized in the programming language being used. The documentation of a program is identified with proof of the correctness of these expansions. Aspects of this approach is illustrated as well in work of Ashcroft and Manna [3], Hoare [17], and Wirth [39]. A major application to a programming system of considerable size is described by Baker [4].

Four mathematical results are central to this approach. One result, a ''Structure Theorem'', due in original form to Böhm and Jacopini, guarantees that any flowchartable program logic can be represented by expansions of as few as three types of structures, e.g., (1) f THEN g, (2) IF p THEN f ELSE g, (3) WHILE p DO f, where f, g, are flowcharts with one input and one output, p is a test, THEN, IF, ELSE, WHILE, DO, are logical connectives. This is in sharp contrast to the usual programming practice of flowcharting arbitrary control logic with unrestricted control branching operations.

In block structured programming languages, such as Algol or PL/I, such structured programs can be GOTO-free, and be read sequentially without mentally jumping from point to point. In a deeper sense, the GOTO-free property is superficial. Structured programs should be characterized not simply by the absence of GOTO's, but by the presence of structure. Structured programs can be further organized into trees of program ''segments'', such that each segment is at most some prescribed size, e.g., a page (some 50 lines) in length, and with entry only at the top and exit at the bottom of the segment. Segments refer to other segments at the next level in such trees, each by a single name, to represent a generalized data processing operation at that point, with no side effects in control. In this way, the size and complexity of any programming system can be handled by a tree structure of segments, where each segment -- whether high level or low level in the system hierarchy -- is of precisely limited size and complexity.

The Structure Theorem has a constructive proof, which provides insight, itself, into program design and construction techniques. Although a flowchart may be of any size, the Structure Theorem guarantees that its

control logic can be represented in a finite basis, with a corresponding reduction in the complexity characteristic of arbitrary flowcharts. The Structure Theorem also provides a canonical form for documenting and validating programs, to help define operational procedures in programming.

The second mathematical result is a ''Top Down Corollary'', which guarantees that structured programs can be written or read ''top down'', i.e., in such a way that the correctness of each segment of a program depends only on segments already written or read and on the functional specifications of any additional segments referred to by name. The application of this Corollary requires a radical change in the way most programmers think today, although advocates of ''functional programming'' have proposed such ideas independently, e.g., Randell and Zurcher [40], Landin [22], Strachey [37], Burge [6], and Scott [35]. It is a nearly universal practice, at the present time, to write large programs ''bottom up'' -- coding and unit testing program modules, then subsystems, and finally systems integration and testing. In top down programming, the integration code is written first, at the system, then subsystem levels, and the functional modules are written last. As discussed by Mills [29], top down programming can eliminate the necessity for the simultaneous interface assumptions that frequently result in system errors during integration.

The third mathematical result is a ''Correctness Theorem'', which shows how the problem of the correctness of structured programs can be reduced to function theoretic questions to which standard mathematical practices apply. These questions necessarily go into the context of intentions and operations available for writing programs. Ordinarily, they will require specific mathematical frameworks and procedures for their resolution. Indeed, for complex programs, the mathematical questions may be more comprehensive and detailed than is practical to resolve at some acceptable level of mathematical rigor. But, in any case, the questions can be formulated on a systematic basis, and technical judgements can then be applied to determine the level of validation which is feasible and desirable for a given program.

In this connection, we note that mathematics consists of a set of logical practices, with no inherent claim to absolute rigor or truth, e.g., see Wilder [38, pp 196]. Mathematics is of human invention, and subject to human fallibilities, in spite of the aura of supernatural verities often found in a schoolboy world. But even so, the reduction of the problem of program meanings to such mathematical practices permits the classification and treatment of ideas in terms of processes which have been subjected to considerable analysis and criticism by mankind.

The fourth mathematical result is an ''Expansion Theorem'' which defines the freedom available in expanding any functional specification into a structure at the next level. Perhaps the most surprising aspect of this result is how little freedom a programmer has in correctly expanding programs top down. For example, it will be clear in defining the structure ''IF p THEN f ELSE g'', that the choice of p automatically defines f and g -- that the only freedom in such a structure is in its predicate. Even more surprising, is the result that in the expansion ''WHILE p DO f'', no freedom exists at all in the selection of p -- the looping predicate will be seen to be totally determined by the functional specification itself.

Our motivation in this final result is to exhibit programming as an analysis, rather than a synthesis, activity, that is, to identify the top down programming process as a sequence of decompositions and partitions of functional specifications and subspecifications, each of which produces simpler subspecifications to handle, until finally the level of programming language instructions or statements is reached. This is in contrast to programming as a synthesis of instructions or statements that ''accomplish'' the functional specifications. It is in this distinction that programming emerges as a readily perceived combinatorial activity.

THE CORRECTNESS OF STRUCTURED PROGRAMS

With structured programming, programmers are capable of high precision programming, but, as in tic tac toe, it is important for their concentration to know their own capability for this high precision. The Correctness Theorem provides concepts and procedures for realizing this precision in programming. Correctness proofs are demonstrations of human devising for human consumption. There is no such thing as an absolute proof of logical correctness. There are only degrees of rigor, such as ''technical english'', ''mathematical journal proof'', ''formal logic'', etc., which are each informal descriptions of mechanisms for creating agreement and belief in a process of reasoning.

It is clear that a whole spectrum of rigor will be useful in correctness proofs. A casual program, used in an experimental investigation, may warrant no more than a few lines of explanation. A heavily used program -- say a text editor or a compiler -- may warrant a much more formal proof. London has furnished several realistic examples of proof at a mathematics level [23, 24, 25], including the proof of an optimizing LISP compiler. Jones [20] has given an example of a proof in more formal terms. King [21] and Good [14] have developed more automatic machinery. Dijkstra [9] has illustrated less formal ideas which may be even more convincing in some programs. The persuasion of a proof depends not only on its formality, but on its brevity. Unfortunately, formality and

brevity do not often cooperate, and the programmer has a difficult balancing problem in selecting the best compromise between formality and brevity.

Our approach is functional (or denotational, as used by Ashcroft [2], rather than computational -- instead of proving assertions about computational steps in a program (as introduced by Naur [31], Floyd [12], et. al.), we formulate assertions about functions whose values are computed by programs and subprograms. In this approach, the set theoretic definition of a function as a set of ordered pairs is of critical convenience. For example, an IFTHENELSE subprogram corresponds to a partition of a corresponding function into two subsets of ordered pairs, which, as subfunctions, correspond to the THEN clause and ELSE clause of the original subprogram.

As noted, structured programs admit decompositions into subprograms of very simple types, such as THEN, IFTHENELSE, and DOWHILE subprograms. Our main interest is to show that each type leads to a characteristic logical assertion about the correctness of a subprogram. These assertions are eventually embodied in function theoretic questions, dealing with composition and partition of functions, e.g., for some sets f, g, h, (not necessarily distinct), it is to be proved that

$$f = g * h \qquad \text{or} \qquad f = g \cup h.$$

These relations assert equalities between sets of ordered pairs. There are many acceptable ways in current mathematical practice to prove such assertions, such as an induction over some common structural feature of the sets involved. But such ways are outside our current interest in formulating the assertions themselves.

We recognize, with Floyd [12], that the question of program correctness is simply the question of program meaning, i.e., knowing what a program does. Any program, including pure gibberish, exhibits some behavior, and it is correct with respect to that behavior, independent of what other capabilities may be envisioned for it. In this context, it is crucial to distinguish between correctness and capability. A program under construction top down can be correct at every stage, but not capable of its eventual requirements until completed. An error in a program is an unexpected action. A function theoretic description of the behavior of a program can thus be regarded as a pure description or a normative prescription, but the correctness problem comes down to the agreement between a functional description and a program behavior.

We adopt the common mathematical notion that a <u>function</u> is a set of ordered pairs, c.f., Halmos [15], say

$$f = \{(x_1, y_1), (x_2, y_2), \ldots\}$$

such that if $(x,y) \in f$, $(u,v) \in f$, $x = u$, then $y = v$. The relation $(x,y) \in f$ is often written as

$$y = f(x),$$

and x is called the <u>argument</u>, y is called the <u>value</u> of function f. The sets of first, second members of the ordered pairs of a function are called the <u>domain</u>, <u>range</u> of the function, respectively. In the example above,

$$\text{domain } (f) = \{x_1, x_2, \ldots\}$$

$$\text{range } (f) = \{y_1, y_2, \ldots\}$$

Note these definitions for domain, range include <u>only</u> arguments, values of the function, and no other elements.

Since a function is a set, it makes sense to use the terms ''empty function'', ''subfunction'', ''function partition'', etc., with the word, suffix or prefix ''set'' replaced by ''function'', whenever the conditions further required by a function can be guaranteed to hold. Instances which violate these conditions include the case of the power set -- the set of subsets of a function is not itself a function, but is a set of functions -- and the union of functions -- the uniqueness of a value for a given argument -- may be lost in forming the union of two functions. However, the union of disjoint functions or intersection of two functions is again a function, as is the difference (set) of two functions.

FUNCTIONS AND RULES

In the description of a function f as a set of ordered pairs, it is often convenient to give a <u>rule</u> for calculating the second member from the first, as in

$$f = \{(x,y) \mid y = x^2 + 3x + 2\},$$

or

$$(x, x^2 + 3x + 2) \in f,$$

or even

$$f(x) = x^2 + 3x + 2,$$

where domain (f) is given in some broader context. A rule used in defining a function in this way is not unique. For example, if

$$x^2 + 3x + 2 = (x + 1)\ (x + 2),$$

then the new function and rule

$$g = \{(u,v) \mid v = (u+1)(u+2)\}$$

or

$$g(u) = (u+1)(u+2)$$

defines the same set as before, i.e., f=g (as sets).

If a function is finite, then its enumeration can serve in a rule. The rule is to find any given argument as a first member of an ordered pair, if possible, and to extract the second member, if found, as the value for that argument. Otherwise, if enumeration is impossible or impracticable, a rule must be expressed as an algorithm, possibly very complex, but with unambiguous outcome for every argument.

In programming, there is a direct correspondence to the relationship between functions and rules -- it is between functional specifications and programs. The problem of program correctness then becomes the problem of showing that a given function is defined by a given rule. Perhaps the simplest form of the program correctness problem is defined by function rules of enumeration, or ''table lookup'', If a table lookup program has been proved to be correct previously, then any finite functional specification, entered as a table, can be verified to be correct by verifying the table entries therein.

Since functions are merely sets of ordered pairs, we regard the usual idea of a ''partial function'' to be a relationship between two sets, one of which is the domain of some function under consideration, In our case, we use the term <u>partial rule</u> to mean a rule of computation not always defined over some given set.

FUNCTION COMPOSITION AND COMPLETION

Beyond operations directly inherited from sets, <u>function composition</u> is based on the fact that functions are sets of ordered pairs. A composition of two functions is a new function which represents the successive use of the values of one function as the arguments of the other. That is, we define the new function composition, using an infix notation, i.e.,

$$f * g = \{(x,y) \mid \exists z\ (z=g(x) \wedge y = f(z))\}.$$

If range (g) and domain (f) are disjoint, then f * g is the empty function; otherwise, f * g is just the set of ordered pairs which is defined through the application of g then f to arguments of g to get values of f.

Conversely, we say an ordered pair of functions, (f,g), is a <u>decomposition</u> of a function, h, if h = f * g. Clearly, for any function h, there may be many decompositions.

It is clear that function composition is associative -- i.e., that

$$(f * g) * h = f * (g * h)$$

for all functions f, g, h; hence, the parentheses can be omitted without ambiguity, as in

$$f * g * h$$

Then, the composition of a function with itself can also be denoted simply by an exponent notation, i.e.,

$$f^2 = f * f$$

$$f^3 = f * f^2 = f^2 * f = f * f * f$$

$$f^4 = f * f^3 = f * f * f * f.$$

It will be occasionally convenient to permit a zero exponent, and interpret f^0 as an identity function (see below).

Given a function, we consider its repeated composition with itself, reusing values as new arguments, until, if ever, such values are not members of the domain of the function. The number of compositions then possible depends on the original argument, of course. Thus, we define a <u>function completion</u>, say for function f, to be

$$* f * = \{(x,y) \mid \exists k\ ((x,y) \in f^k) \wedge y \notin \text{domain}\ (f)\}.$$

SPECIAL FUNCTIONS

We identify, for future convenience, several general classes of functions, namely:

a. Identity Functions:

$$I = \{ f \mid (x,y) \in f \supset y = x \}$$

b. Constant Functions:

$$C(a) = \{ f \mid (x,y) \in f \supset y = a \}$$

c. Permutation Functions:

$$P = \{ f \mid \text{domain } (f) = \text{range } (f) \}$$

d. Inverse Function Pairs:

$$R = \{ (f,g) \mid f * g = g * f \in I \}$$

(If $(f,g) \in R$, we say

$g = f^{-1}$ or $f = g^{-1}$.)

We abstract the commonly known idea of a (computer) program as a finite set of functions, called instructions, each with a finite domain contained in a common set, called the data space, and a finite range contained in the cartesian product of the data space and the program, called the state space. Members of the data space, state space are called data values, state values, respectively.

A program execution is a sequence of state values, say

$$s_i = (d_i, f_i),\ i = 0,1,\ldots$$

such that

$$s_{i+1} = f_i(d_i),\ i = 0,1,\ldots$$

which terminates, if ever, when $f_i(d_i)$ fails to exist -- i.e., when $d_i \notin$ domain (f_i). The state value s_0 is called the initial value of the execution. If the execution is finite, say

$$s = s_0, s_1, \ldots, s_n = t$$

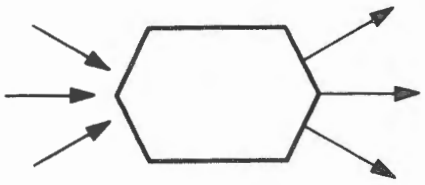
then t is called the final value of the execution.

Since the state space of a program is finite, it is decidable, for every initial value, s, whether that execution terminates, and, if so, what the final value, t, is. Therefore, a program automatically defines a function of ordered pairs (s,t) defined by terminating executions, called the program function. If a program is given by a set P, we denote its program function by [P]. In retrospect, a program is a specific (non-unique) rule for calculating the values of its program function.

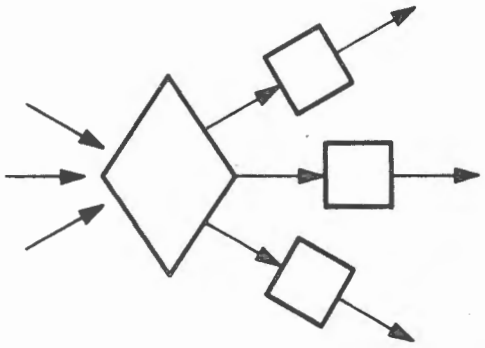
A subprogram is a subset of a program, which inherits its state space. A subprogram execution is a contiguous subsequence of a program execution which terminates, if ever, when an instruction not in the subprogram appears in the state value. To each subprogram corresponds a subprogram function, as well.

CONTROL GRAPHS

The instructions (functions) of a program determine a directed control graph whose nodes are instructions and whose directed lines are the next possible instructions. A node of such a graph may have several input lines and several output lines which denote the direction of control flow, as shown:

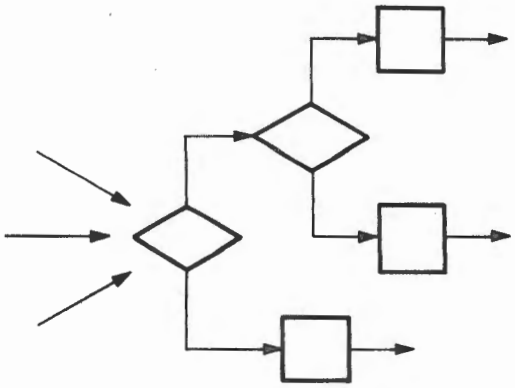


An instruction (node) has a natural decomposition between control and data effects which can be displayed by its partition (of its set of ordered pairs) into subsets, each of whose values contain identical (next) instruction components. The instruction node displayed above then has the form:

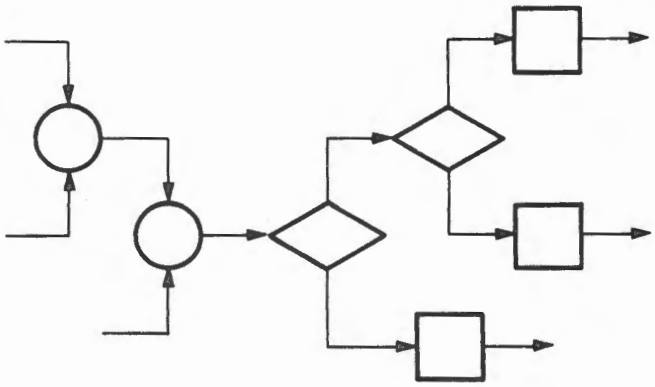


where the diamond (control node) represents an identity function for values in the data space and a square (process node) represents a constant function for values in the program (next instruction). Since the program (set) is finite, this partition can be refined so that control nodes each contain exactly two output lines, called predicate nodes.

From these considerations we are led to directed graphs with predicate and process nodes of the form shown.



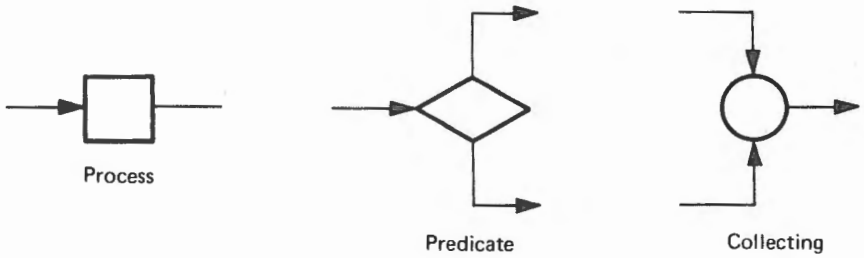
It will be convenient to introduce a symmetry into such directed graphs, by augmenting the original program with ''noop'' instructions (collecting nodes) which collect and transfer control from exactly two input lines each, which we diagram as shown:



Control graphs are also called program schemas [19].

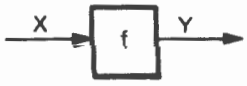
PROGRAMS IN FLOWCHART FORM

We can represent a program in flowchart form. A flowchart is defined by a control graph, and by operations and tests to be carried out on data in a sequence determined by that control graph. As noted, we consider control graphs with only three types of nodes:



The upper and lower lines out of a predicate node are labeled ''True'' and ''False'', respectively, just to be definite, unless otherwise noted.

In a flowchart, each process node is associated with a function, or data transformation, and each predicate node is associated with a predicate function, or a binary valued data test. Each line of a flowchart is associated with a set of possible data states. A set of data states may be the set of all possible machine states, for a program in a machine language, or may be the set of all variables allocated at a point in a program in a programming language. The function associated with a process node maps a set of data states associated with its input line into a set of data states associated with its output line. A function f from X to Y is identified in a flowchart as:

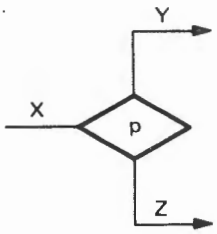


This mapping is a subfunction, say g, of f, namely:

$$g = \{ (x,y) \mid x \in X \wedge (x,y) \in f \wedge y \in Y \}.$$

If $x \notin X$, no such input is possible; if $y \notin Y$, no such output is possible; if $x \in X$ but $(x,y) \notin f$ or $y \notin Y$, the operation is not completed.

The predicate function associated with a predicate node maps the set of data states associated with its input line into the set {True, False} but does not transform data otherwise, that is, the flowchart figure

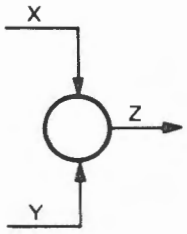


is associated with the identity mappings of data from input to output. But in order to satisfactorily complete the test, the condition

$$x \in X \wedge (((x, \text{True}) \in p \wedge x \in Y) \vee (x, \text{False}) \in p \wedge x \in Z))$$

must be satisfied.

The collecting node is also associated with an identity mapping, from the flowchart figure:



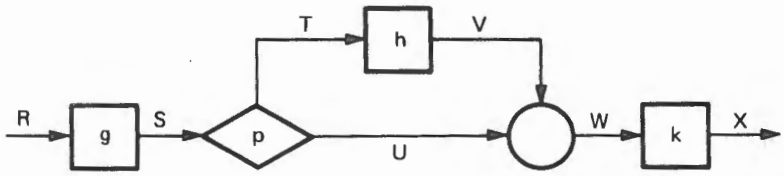
Also, to complete the transfer of control, the condition

$$(x \in X \wedge x \in Z) \vee (y \in Y \wedge y \in Z)$$

must be satisfied. In early practice and in current programming theory, the sets associated with control lines are often taken to be identical -- a ''state vector'' set. However, with data scoping and dynamic storage allocation, as found in contemporary practice, the data space is variable, rather than constant, over a program or flowchart.

PROGRAM EXECUTION

The execution of a program is easily visualized in a flowchart, using the control graph to identify the sequence of operations and tests on data required. For example, consider the program f in flowchart form:



Where possible, <u>initial data</u> $r \in R$ is converted by f into <u>intermediate data</u> $s \in S$, then $t \in T$ and $v \in V$, or $u \in U$, then $w \in W$, and ultimately into <u>final data</u> $x \in X$, by functions g, h, and k, under the control of predicate p. I.e., the program function [f] of program f has values, when they exist, given by:

$$x = k\ (h(g(r))) \qquad \text{if} \qquad p(g(r)) = \text{True}$$

$$x = k\ (g(r)) \qquad \text{if} \qquad p(g(r)) = \text{False}.$$

More precisely, we mean:

$$[f] = \{\ (r,x) \mid r \in R \wedge (\exists s, v\ ((r,s) \in g \wedge (s, \text{True}) \in p \wedge (s,v) \in h \wedge (v,x) \in k)) \vee (\exists s((r,s) \in g \wedge (s, \text{False}) \in p \wedge (s,x) \in k) \wedge x \in x\ \}.$$

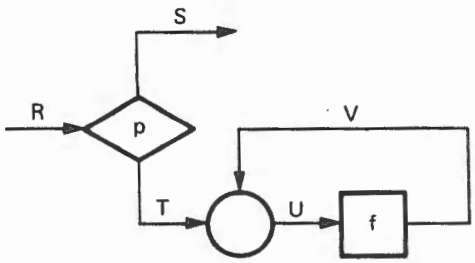
PROPER PROGRAMS

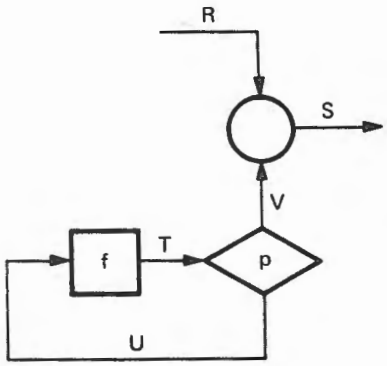
We define a <u>proper program</u> to be a program in which:

a. There is precisely one input line and one output line

b. For every node, there exists a path from the input line through that node to the output line.

Note we admit the possibility of programs with no nodes, a single input/output line. We call such a program λ. Clearly, the program function $[\lambda]$ is an identity function; i.e., $[\lambda] \in I$. In illustration, the following are not proper programs.



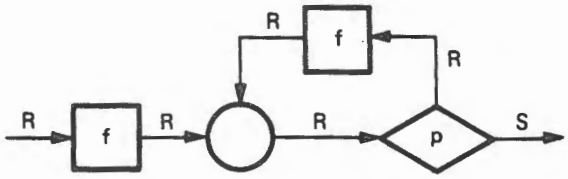


This definition of proper programs is primarily motivated by the interchangeability of proper programs and process nodes in larger programs.

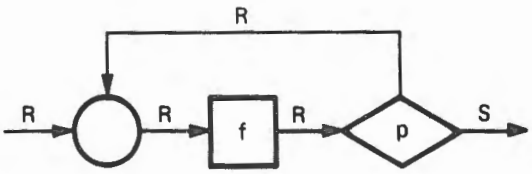
Henceforth, we take the ''proper program'' and ''program'' to be synonymous. If necessary, we will use the term ''improper program'' to refer to a program which is not a proper program.

PROGRAM EQUIVALENCE

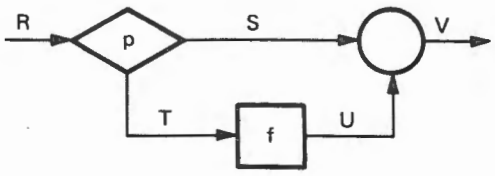
We will say two proper programs are equivalent when they define the same program function, whether or not they have identical control graphs, require the same number of operations, etc. For example, the two programs

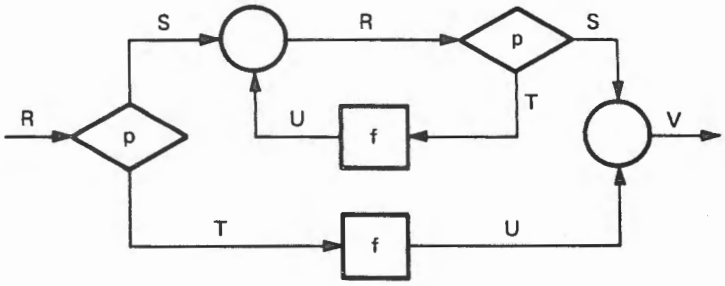


and



have the same program function, as do the two programs:





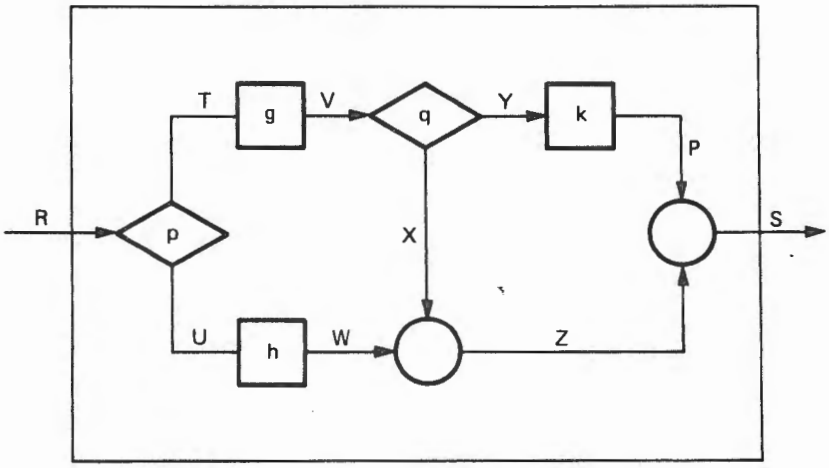
That is, two programs are equivalent if they define the same program function, even though the programs may represent different rules for computing the values of the program function. In particular, given program f and its program function [f], the new program g

is equivalent to f. In this case g is a table lookup version of f.

PROGRAM EXPANSIONS

If a program contains a process node, as

it may happen, that a rule for computing the values of f is defined as another program. We call such a program an expansion of the function f, such as shown next.



In this case, it is asserted that the program function of the latter program is f. That is, any expansion of a function is simply a rule for computing its values, possibly using other functions and predicates to do so.

Programs with loops may or may not terminate. This property of termination partitions an input set R into R_t and $R-R_t$, where R_t is the subset of inputs for which the evaluations terminate. If $R_t \neq R$, then the program defines a partial rule, rather than a rule. Note, in fact, that a program may terminate by reaching an output line (normal termination) or by reaching a node with a data value not in the domain of the corresponding function (abnormal operation termination) or by reaching a line with a data value not in the data space (abnormal storage termination).

CONTROL GRAPH LABELS

The set of all control graphs of proper programs can be enumerated and labeled. The beginnings of such an enumeration is given in Figure 1.

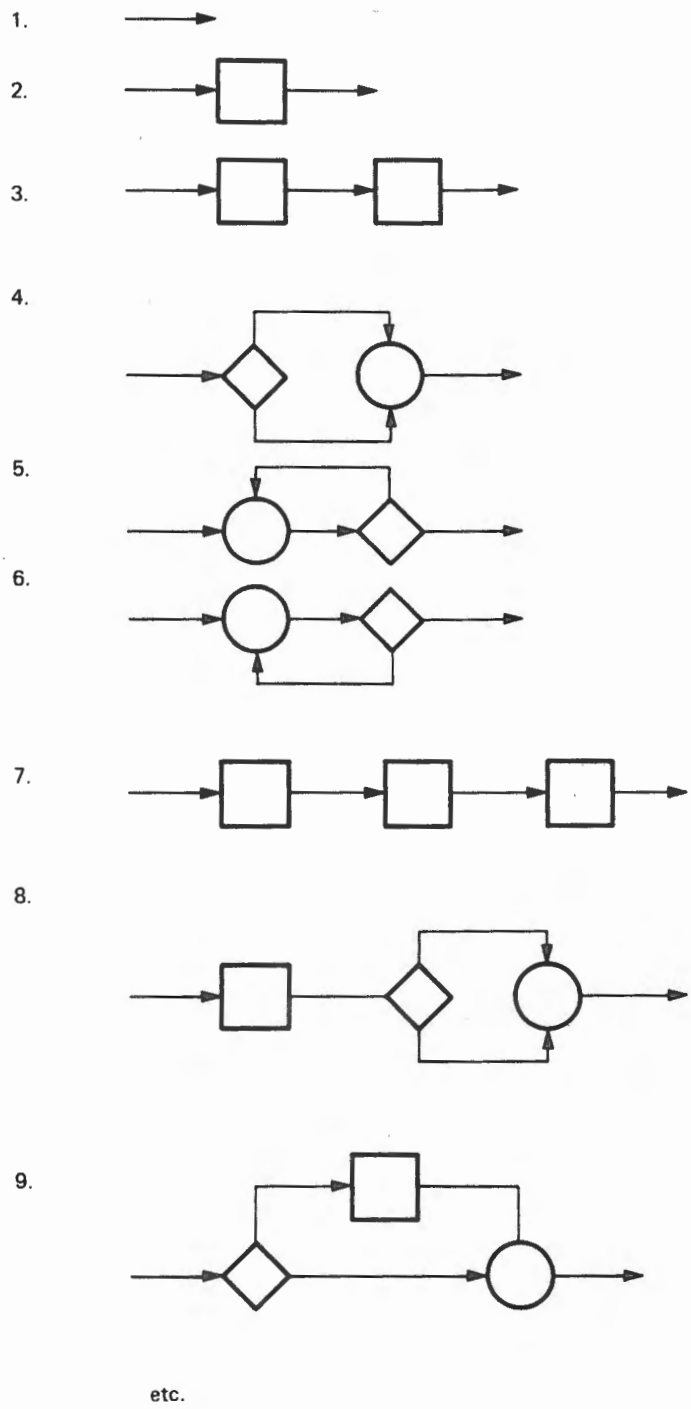
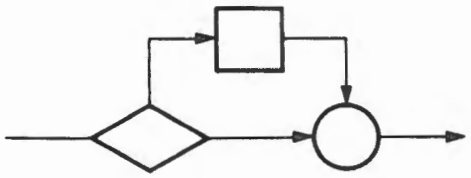
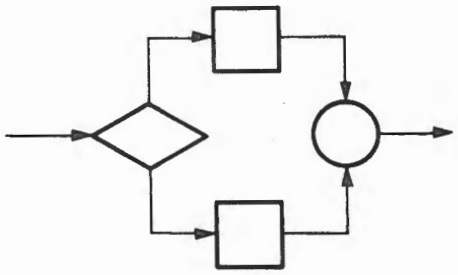


Figure 1. Control Graphs

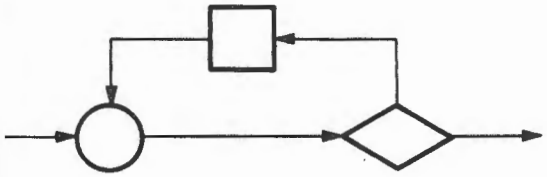
In fact, a few such control graphs are given special mnemonic labels in various programming languages. For example, the following labels are common:



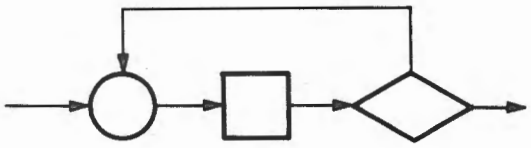
IFTHEN



IFTHENELSE



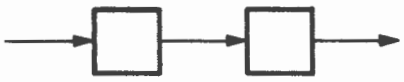
DOWHILE



DOUNTIL

(IFTHEN is 9, in the enumeration started above, IFTHENELSE might be 37, 42, etc.)

However, there is nothing special about these graphs except for their simplicity. Any control graph possibly more complicated than these might be so labeled if it were useful. In particular, we label the sequence of two process nodes



BLOCK

for future reference.

PROGRAM FORMULAS

A program can be given as a <u>formula</u>, by associating an ordering with the set of process nodes, predicate nodes and control lines of its

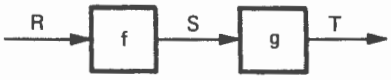
control graph, and by listing the label of its control graph, followed by labels for the functions, predicates and state sets of the program. For notational convenience, we will use parentheses and commas to denote the list structure of a program formula, e.g.,

(A, p, q, f, g, h, R, S, T, U)

means a program given by a control graph labeled A, with predicates p, q, functions f, g, h, and state sets R, S, T, U, associated with the nodes and lines of A. For example

(BLOCK, f, g, R, S, T)

defines a program



whose action on an input $r \in R$ is to produce output $t \in T$ if it exists, such that

$$t = g(f(r)),$$

more precisely,

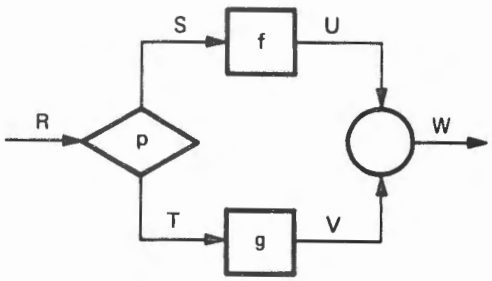
$$[\ (\mathrm{BLOCK},f,g,R,S,T)] = \{\ (r,t)\ |\ \exists\ s\ (r \in R$$

$$\wedge\ s \in S \wedge t \in T \wedge (r,s) \in f \wedge (s,t) \in g)\ \}.$$

The list

IFTHENELSE, p, f, g, R, S, T, U, V, W)

defines a program



which maps any $r \in R$ into some $w \in W$, if it exists, such that

$$w = \begin{cases} f(r) \text{ if } p(r) = \text{True} \\ g(r) \text{ if } p(r) = \text{False} \end{cases}.$$

More precisely,

$$[\ (\text{IFTHENELSE}, p,f,g,R,S,T,U,V,W)]$$

$$= \{\ (r,w) \mid r \in R \wedge w \in W \wedge (((r, \text{True}) \in p \wedge$$

$$r \in S \wedge (r,w) \in f \wedge w \in U) \vee$$

$$((r, \text{False}) \in p \wedge r \in T \wedge (r,w) \in g \wedge w \in V))\ \}.$$

In much of what follows, the list of data sets is not central to the ideas under development. In this case, they will be suppressed. However, such data sets are always implicit to program descriptions and discussions.

Since function composition is associative, i.e.,

$$(f * g) * h = f * (g * h),$$

so is BLOCK formation, i.e.,

$$[\ (\text{BLOCK}, [\ (\text{BLOCK},f,g)], h)\ ] = [\ (\text{BLOCK},f, [\ (\text{BLOCK},g,h))\ ]]$$

and no ambiguity results by extending the meaning of BLOCK to several nodes, e.g.,

$$(\text{BLOCK3},f,g,h) = (\text{BLOCK},(\text{BLOCK},f,g),h),$$

etc. In particular, we permit zero or one nodes in a BLOCK, as in

(BLOCK0) = λ (BLOCK1, f) .

Then, for example, we have the identity

$$f = [\ (\text{BLOCK1},f,\text{domain}(f), \text{range}(f))\ ].$$

It may happen that a function listed in a program formula is, itself, a program function given by another formula, such as

$$(\text{IFTHEN},p, [\ (\text{BLOCK},g,h)]).$$

We extend the idea of program formula to permit the replacement of a program function by its program formula, such as

(IFTHEN,p,(BLOCK,g,h)).

It is clear that, while these are different programs, they have identical program functions, just by the definition of program functions.

PROGRAM DESCRIPTIONS

Flowcharts and formulas are simply two alternative ways of describing (possibly partial) rules, with some internal structure, in terms of other rules (or partial rules). Still another way of description is in programming language text such as

```
IF p THEN
   f
ELSE
  g
ENDIF
```

and

```
WHILE p DO
 f
ENDDO
```

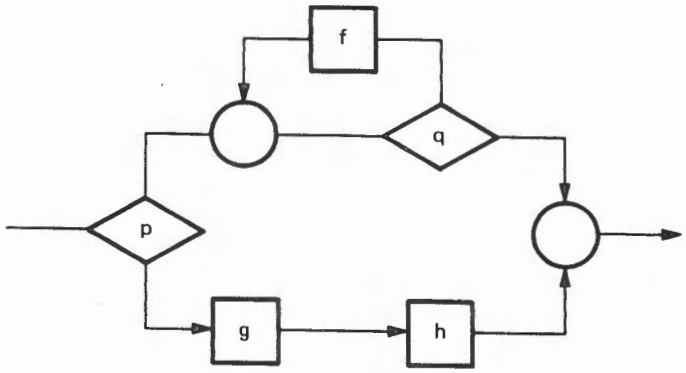
and

```
BLOCK
 f
 g
ENDBLOCK
```

etc. We find all three types of description useful in various circumstances in programming. Typically, flowcharts are useful in general discussions because of their graphics, formulas are useful in stating and proving theoretical properties of such rules, the text is useful in the actual construction of large complex programs. For example, the same program is given in the formula

(IFTHENELSE,p ,(DOWHILE,q,f), (BLOCK,g,h)),

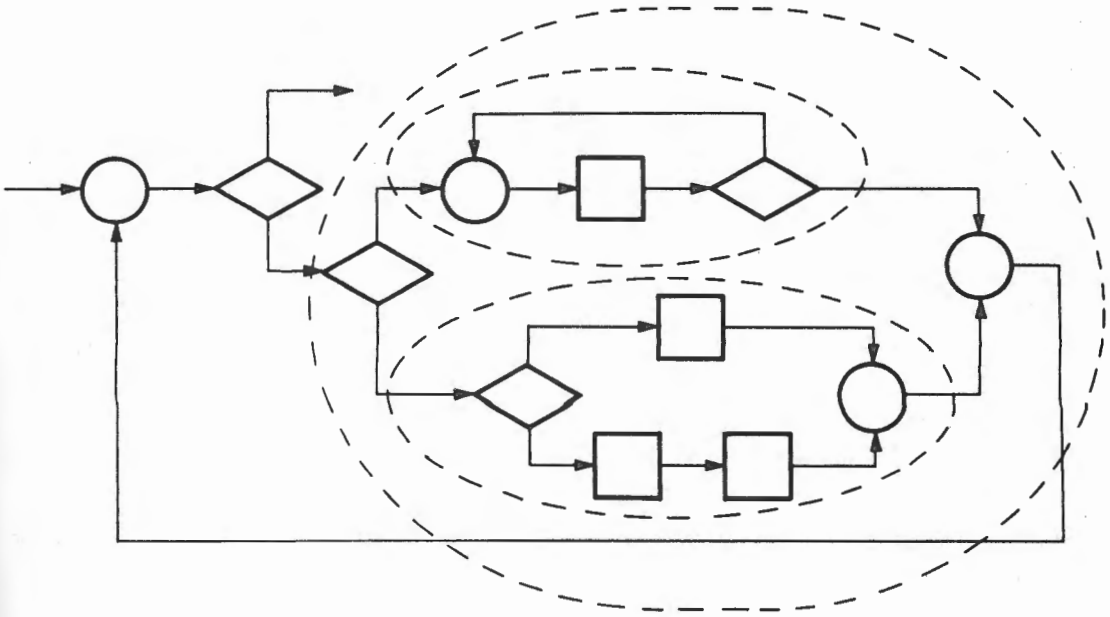
the flowchart,



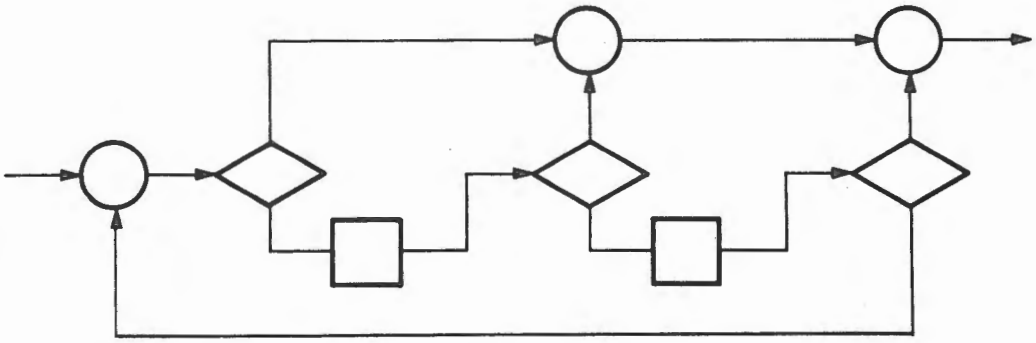
or in program text,

```
IF p THEN
     WHILE q DO
       f
     END DO
ELSE
     BLOCK
       g
       h
     END BLOCK
END IF
```

As flowcharts increase in size, we can often identify patterns which give more coherence and understandability to a whole flowchart. For example, the control graph next



has three definite nested substructures, which are control graphs for proper programs, which make the whole more easily considered. But the following control graph



admits no such structuring. By simply continuing this last pattern indefinitely, it is easy to see that indecomposable control graphs exist of any size.

Having noted that programs of arbitrary size may be indecomposable, we next add the possibility of operations and tests on data outside the original data sets of a program. The additional operations and tests correspond to ''flag'' setting and testing. But we can couch these operations in the concept of a push down stack to show their economy. In addition to the functions and predicates original to a given program, we introduce three new functions and one predicate.

More specifically, we define process nodes with functions named TRUE, FALSE, POP, and a predicate node with function named TOP, which add truth values True, False, remove, and test such truth values in an input data set, respectively. That is, for any data set Y, and $y \in Y$ and $z \in \{ True, False \}$,

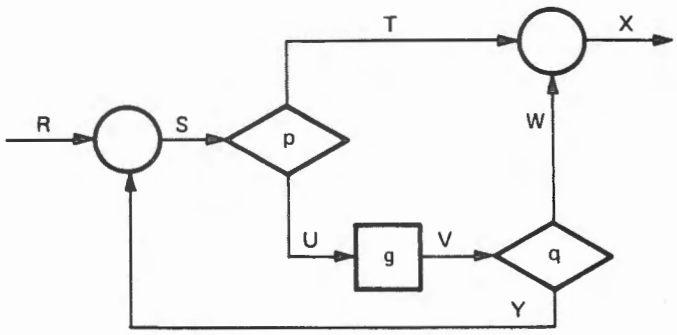
TRUE(y) = (y, True)

FALSE(y) = (y, False)

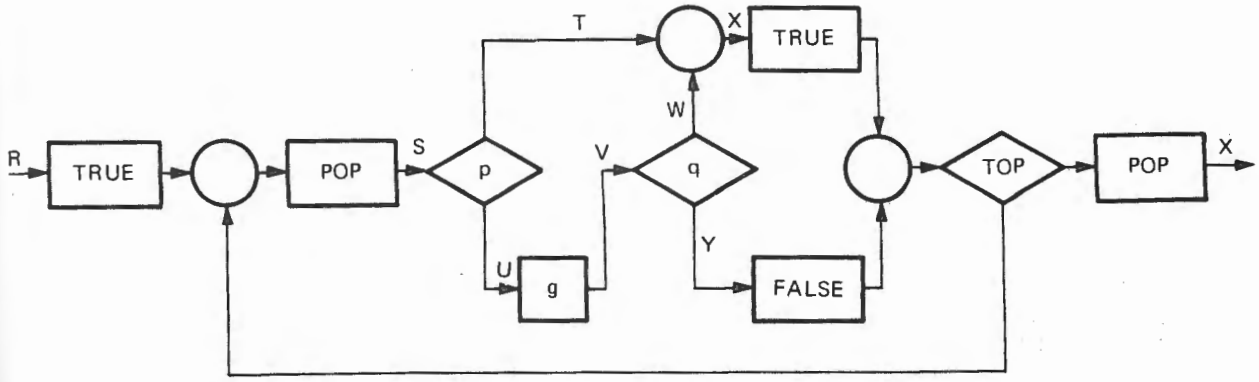
POP(y,z) = y

TOP(y,z) = z

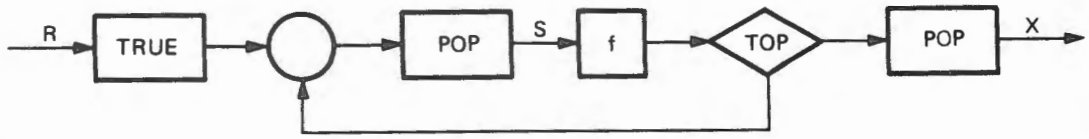
These new functions and predicate allow us to construct explicit control logic in the form of flags. For example, a program whose control structure is in the indecomposable pattern above is shown next.



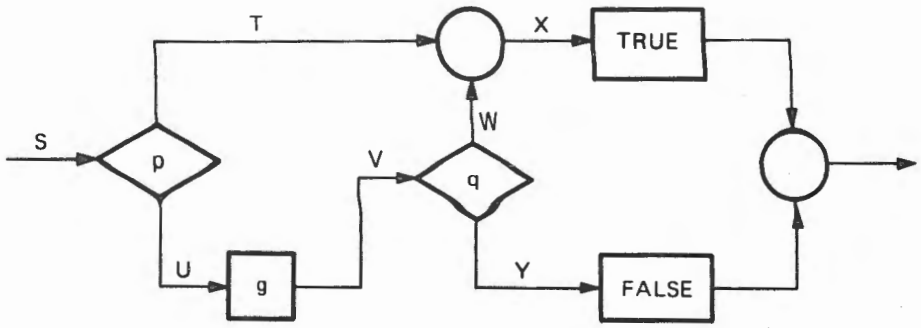
This program is equivalent to the new program, where the output line X and return line Y are tagged and the tag later tested.



Only the original data sets have been shown; the remaining ones can be inferred from the definitions above. Close inspection will reveal that the net effect of TRUE, FALSE, POP, and TOP is to present just the correct original data set to each of the original functions and predicates of the program. It may not be obvious that this equivalent program is of any value in this case. It seems rather more complex -- except that there is now a substructure, a proper program, which contains all the original functions and predicates, and furthermore, has no loop in it. This particular application previews a fundamental construction in the proof of the main Structure Theorem below. As a result, this new program can now be decomposed into two sections, of the forms



where process node f is given by



Before proving this Theorem, we introduce a simple Lemma which counts the control lines of a proper program in terms of its function and predicate nodes.

THE NUMBER OF CONTROL LINES IN A PROPER PROGRAM

Lemma

If the number of function, predicate, and collecting nodes is ϕ, π, γ, respectively, and the number of control lines, i.e., (edges) is e, in a proper program, then

$$\pi = \gamma$$

and

$$e = 1 + \phi + 3\pi \ .$$

Proof

In order to prove this Lemma, count the ''heads and tails'' of the control lines, adjacent to all the nodes, and at the input and output of the program, to get:

Control Line	Input	Function Node	Predicate Node	Collecting Node	Output	Total
Heads		ϕ	π	2γ	1	$\phi+\pi+2\gamma+1$
Tails	1	ϕ	2π	γ		$\phi+2\pi+\gamma+1$

Since the total number of heads must equal the total number of tails, and each equal e,

$$\phi + \pi + 2\gamma + 1 = e = \phi + 2\pi + \gamma + 1$$

and the equations of the Lemma follow.

STRUCTURE THEOREM

Any proper program is equivalent to a program whose formula contains at most the graph labels BLOCK, IFTHENELSE, and DOUNTIL, and additional functions TRUE, FALSE, POP and predicate function TOP.

Proof*

We prove the Theorem by induction on the number of lines of a proper program. The induction step is constructive, and identifies, for any proper program of more than one node, an equivalent proper program which is a formula in at most graph labels BLOCK, IFTHENELSE, and DOUNTIL and new proper programs, each with fewer lines than the initial program.

In order to carry out the induction, we first define a structuring process, S, on any proper program, f, whose result we denote by S(f), as follows. For convenience, we abbreviate the graph labels BLOCK, IFTHENELSE, DOUNTIL to BLK, IF, DO, respectively, in the remainder of the proof.

Since f is a proper program, it has exactly one input and one output. We identify several cases that are possible.

Case 1–No Nodes

If f has no nodes, we define

$S(f) = \lambda$.

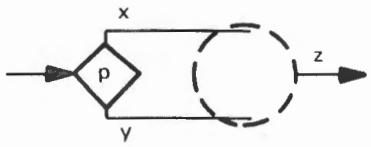
Case 2–One or More Nodes

If f has at least one node, we examine the unique node reached by the input line. There are three possible cases:

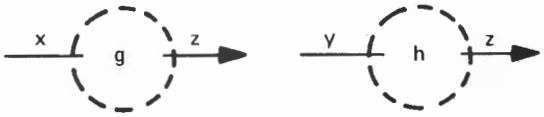
---

*Thanks go to J. Misra for suggestions and assistance in developing the following Proof. Thanks are also due to S. Cole for discussions about the Theorem and methods for its Proof.

Case 2a–Predicate Node. If the first node is a predicate node, then f is of the form



Since f is a proper program, the line z can be reached from both x and y,* and we construct two constituent programs which consist of all nodes accessible in f from x and y, respectively, calling them g and h, respectively.

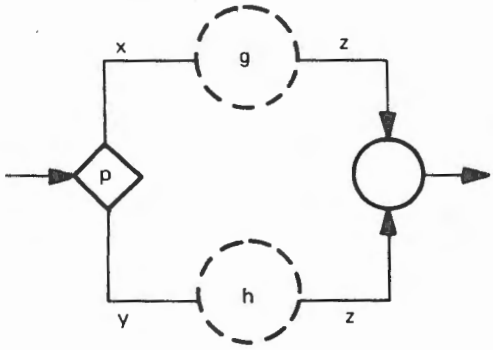


The constituents may contain identical nodes from f, so that g and h represent duplications of parts of f. If a collecting node in g or h is reached by only one input line (the other line in f being in the other constituent), we suppress that collecting node, i.e.,

Note g and h are each proper programs; otherwise f is not a proper program. Note also g and/or h may be λ, a program with no nodes.

*Our definition of proper programs is necessary for this assertion. The proof of Böhm and Jacopini [5] breaks down at this point.

Since each g and h contain at least one less predicate node than does f, at least one collecting node is suppressed in each constituent. Next, we consider the new proper program, (IF,p,g,h),



with the original predicate p and the constituents g and h of f (and a new collecting node, not from g or h). In this case, we define

$$S(f) = (IF,p,g,h).$$

Also, in this case, we observe that

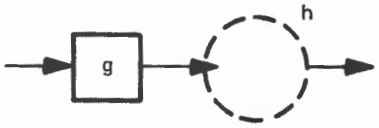
$$e(g) \leq \phi(f)+3(\pi(f)-1)+1 = e(f)-3$$

$$e(h) \leq \phi(f)+3(\pi(f)-1)+1 = e(f)-3$$

since g and h at least do not contain predicate node f. (We use e(f), $\phi(f)$, and $\pi(f)$ to denote the number of lines, function nodes, and predicate nodes, respectively, in f, etc.)

Finally, it is clear by construction that S(f) is equivalent to f .

<u>Case 2b-Function Node.</u> If the first node is a function node, then f is of the form



and h is a proper program, possibly λ. In this case, we define

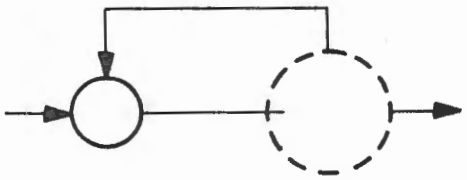
$$S(f) = (BLK,g,h) .$$

Also, in this case, it is easy to count the number of lines in h, given that there are e(f) lines in f. The number is

$$e(h) = (\phi(f)-1)+3\pi(f)+1 = e(f)-1 .$$

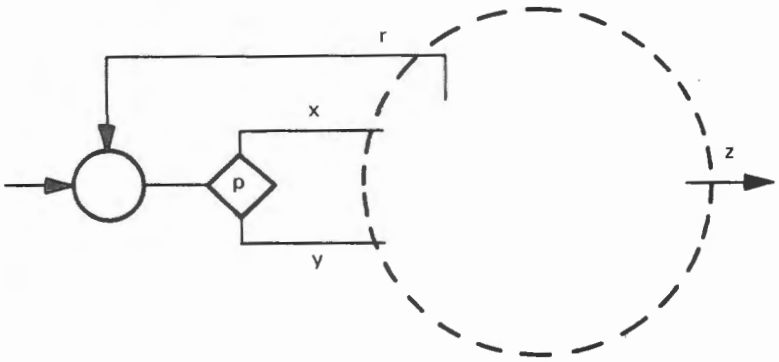
Finally, it is clear by construction that S(f) is equivalent to f.

Case 2c–Collecting Node. If the first node is a collecting node, then f must be of the form

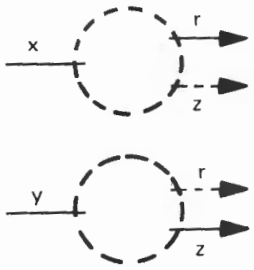


and we examine the next unique node reached from this collecting node. It is clear that such a next node exists, because a predicate node, at least, must be reached in the remaining improper program in order to have two output lines. There are three subcases to be examined.

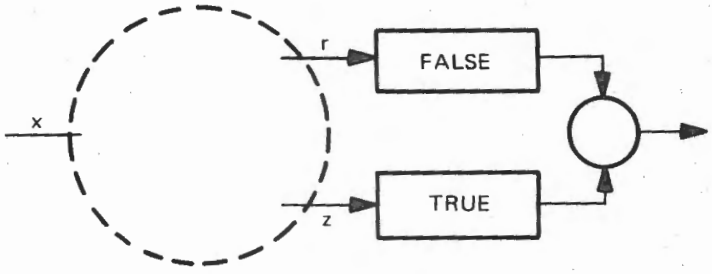
2.c.(1) Predicate Node–If the next node is a predicate node, then f is f is of the form



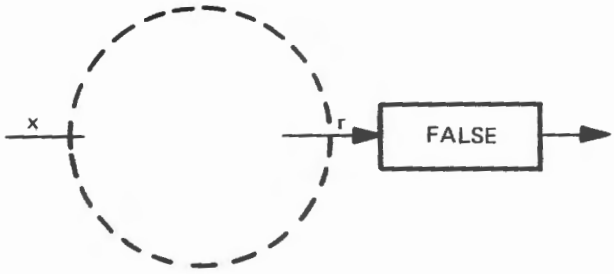
As before, we construct two programs which consist of all nodes which can be reached from x and y, which terminate in z or r. We suppress collecting nodes with only one input, as before. These programs will not be proper programs if both r and z can be reached from x or y. However, since f is a proper program, we know that each constructed program must reach at least z or r, and that each z and r must be reached by at least one constructed program. These constructed programs have the form



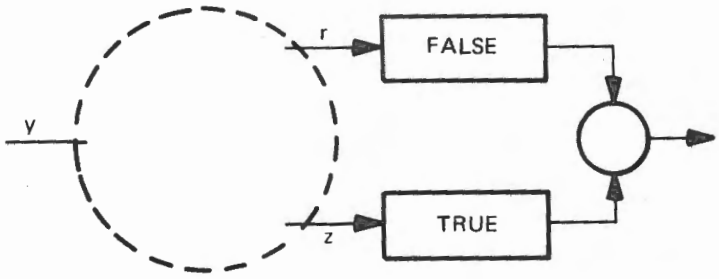
where the solid output line is necessary, and the dotted output line may or may not exist. We use TRUE, FALSE function nodes (to set flags) and possibly collecting nodes to construct new proper programs from these shown, of the form



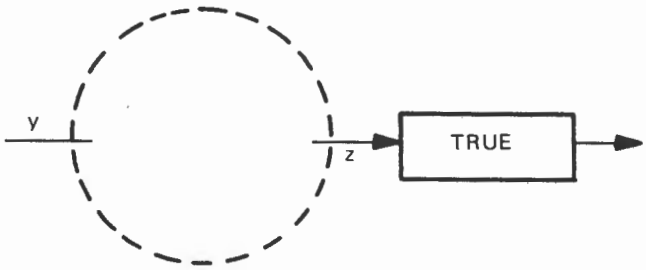
or



and

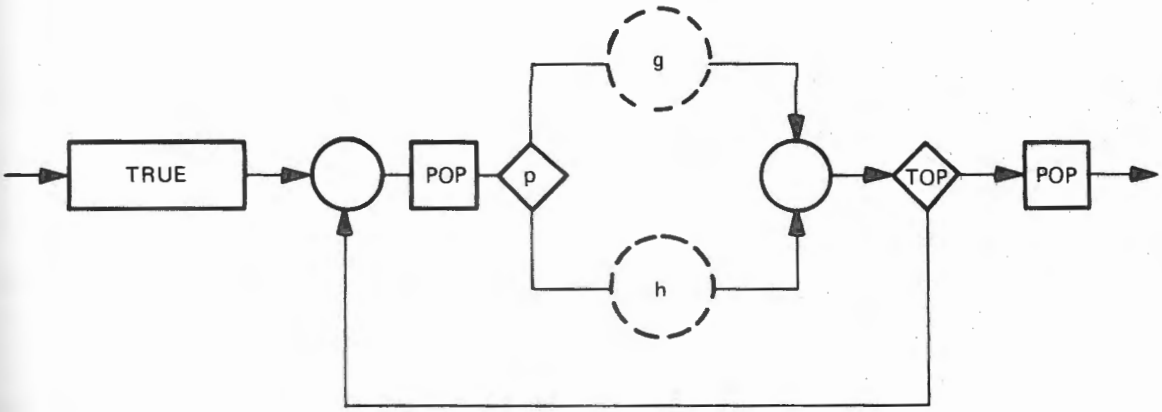


or



depending on whether or not the dotted output lines r and z exist.

We label these proper programs g and h (such that g has at least the r output line and h has at least the z output line). Now, we consider the new program



with g and h as constituent programs. In this case, we define

$$S(f) = (BLK,TRUE,(BLK,(DO,TOP,(BLK,POP,(IF,p,g,h))),POP)) \ .$$

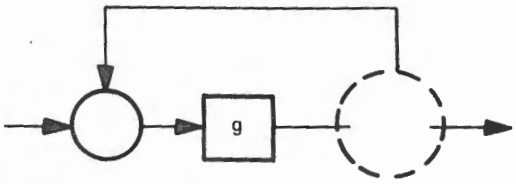
We observe that g and h does not have the predicate node p and each has at most two more function nodes. Hence,

$$e(g) < \phi(f)+2+3(\pi(f)-1)+1 = e(f)-1$$

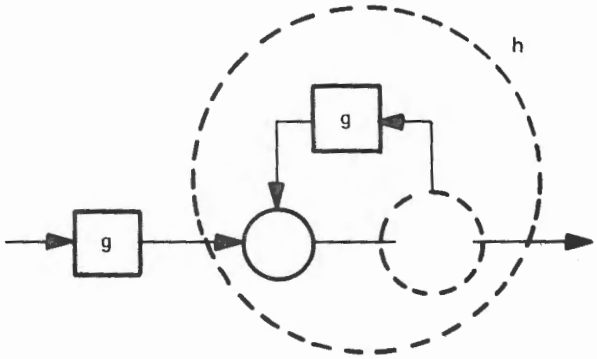
$$e(h) < \phi(f)+2+3(\pi(f)-1)+1 = e(f)-1$$

Finally, it can be verified that S(f) is equivalent to f.

2.c.(2). Function Node—If the next node is a function node, then f is of the form



and we consider the new program



with new program labeled h. In this case, we define

$$S(f) = (BLK,g,h).$$

Also, in this case, we observe directly that

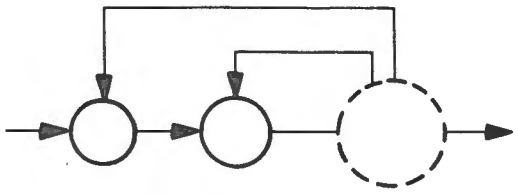
$$e(h) = e(f)$$

but that also, the number of lines, say i(f), required to reach the first predicate of f is reduced by one, i.e., that

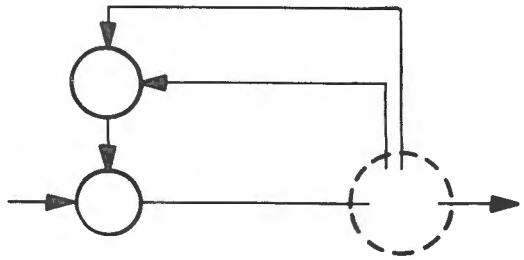
$$i(h) = i(f)-1$$

Finally, it is clear that S(f) is equivalent to f.

2.c.(3). Collecting Node—If the next node is a collecting node, then f is of the form



and we consider the new program



called g. In this case we define

$$S(f) = g$$

Also, in this case, we observe directly that

$$e(g) = e(f)$$

$$i(g) = i(f)-1$$

It is clear that S(f) is equivalent to f.

Summary

This completes the analysis of cases for the input region of f, and the definition of the structuring process S. In summary, in each case, we have defined a new program, S(f), equivalent to f, such that S(f) is a formula in, at most, graph labels BLOCK, IFTHENELSE, DOUNTIL, functions, predicates, and constituent proper programs. In several cases, the number of edges of the constituents of f are seen to be less than the number of edges in f. In two cases, this number of edges was not decreased, but the number of edges from input to the first predicate node, was decreased. It is clear that the number of edges from input to the first predicate node is bounded by the number of edges of a program. When we apply this information to that generated in the case analyses above, we get Table 1.

Table 1. Case Analysis -- Structuring Process

Case	e values		i values
2a	$e(g) \leq e(f)-3$ $e(h) \leq e(f)-3$	, ,	$i(f) \leq e(f)-3$ $i(h) - e(f)-3$
2b	$e(h) = e(f)-1$	,	$i(h) \leq e(f)-1$
2c(1)	$e(g) \leq e(f)-1$ $e(h) \leq e(f)-1$	, ,	$i(g) \leq e(f)-1$ $i(g) \leq e(f)-1$
2c(2)	$e(h) = e(f)$	,	$i(h) = i(g)-1$
2c(3)	$e(g) = e(f)$	,	$i(g) = i(f)-1$

We are now ready to summarize our proof, as follows:

First, it is clear that the Theorem is true for proper programs with one line, for such a program is simply λ.

Next, suppose the Theorem is true for proper programs of n lines or less for $n > 1$. Let f be a proper program with $n + 1$ lines. We apply S to f. If cases 2a, 2b, or 2c(1) apply, we have a new equivalent program, whose constituent programs are proper and have at most n lines; and each such constituent, by our induction hypothesis, satisfies the Theorem. Moreover, the new equivalent program has a formula in, at most, graph labels BLOCK, IFTHENELSE, DOUNTIL, predicates and its constituents. Therefore, the new program satisfies the Theorem. If none of cases 2a, 2b, or 2c(1) apply, then $i(f) < n$, and case 2c(2) or 2c(3) must apply. In each such case, there remains only one constituent, say g, and

$$e(g) = e(f),\ i(g) = i(f)-1$$

Therefore, after, at most, n such applications, case 2c(1) must apply, and the final equivalent program satisfies the Theorem.

This completes the Proof of the Structure Theorem.

TOP DOWN COROLLARY

Any proper program is equivalent to a program of one of the forms

(BLOCK,g,h)

(IFTHENELSE,p,g,h)

(DOUNTIL,p,g)

where p is a predicate of the original program or TOP, and g,h are each proper programs, functions of the original program, TRUE , FALSE , or POP .

S-STRUCTURED PROGRAMS

The Structure Theorem motivates the definition of a structured program as follows:

Let S be any finite set of labels associated with control *graphs of* proper programs. Then any program whose formula contains only graph labels from S is said to be an S-structured program.

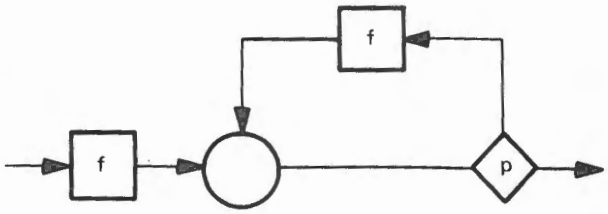
When the prefix ''S'' is not critical, or understood, it will be suppressed.

PROGRAM REPRESENTATIONS

The result of the Structure Theorem is similar to representation theorems in other branches of mathematics, in which it is shown that all elements of a set, or ''space'', can be represented by combinations of a subset of ''basic elements'' of the space. For example, three nonplanar vectors span a three-dimensional Euclidean space, the set { sin nx, cos nx | n=0,1,... } span a set of real functions in the interval [0, 2π] -- i.e., a ''function space''. The foregoing examples refer to linear combination for representation.

In the Structure Theorem, it is shown that three simple types of programs, defined by BLOCK, IF-THEN-ELSE, and DO-UNTIL control graphs, span the set of all proper programs, using substitution of proper programs for process nodes as the only rule of combination. Such a representation theorem permits the resolution of questions of the adequacy of a programming language simply and effectively. For example, all one needs to show a new set of basis programs will span the set of all proper programs is that one can represent BLOCK, IF-THEN-ELSE, and DO-UNTIL programs in this new set.

One simple illustration of a new basis is to represent DOUNTIL in terms of BLOCK and DOWHILE, as follows



or

(DOUNTIL,p,f) = (BLOCK,f,(DOWHILE,p,f)).

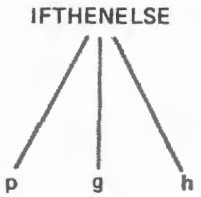
Hence, BLOCK, IFTHENELSE, and DOWHILE, provide a sufficient control structure to represent all proper programs as well as BLOCK, IFTHENELSE and DOUNTIL.

PROGRAM TREES

The formula of a structured program can be displayed in a program tree in a natural way, with the graph labels, functions and predicates assigned to nodes of the tree. For example, the formula

(IFTHENELSE,p,g,h)

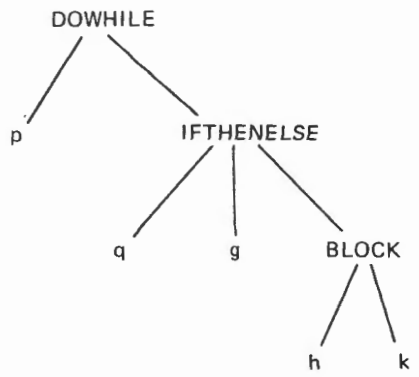
defines the program tree.



and the formula

(DOWHILE,p,(IFTHENELSE,q,g,(BLOCK,h,k)))

defines the program tree



Conversely, given any program tree of graph labels, functions and predicates, the original program can be recovered. In particular, any subtree defined by a node plus all its successors in the tree defines a subprogram of the original program.

The program tree provides a convenient way of visualizing program structure in the form of subprograms. By labeling subprograms, and referring to their program functions at higher levels in the program, an original program of any size can be organized as a set of subprograms, each of a prescribed maximum size.

It is clear that the subprograms so defined are each proper programs. That is, they each map an input data set into an output data set, with no control side effects.

We have already noted that program correctness is a question of predictability. More precisely, given a program, f , and a function, g , we are interested in whether g is the same as the program function [f]. If we know both g and [f], we can resolve the question by comparison. Carrying out such a comparison of two sets is a general mathematics problem whose solution will depend on how the sets are defined. In few cases they will be enumerated. In that case their elements can be ordered and matched, a pair at a time. In most cases such sets will be defined by conditions or rules in some broader (less formal) context than set theory per se. There may be natural numbers involved, in which case inductive definitions and comparisons may be possible. In any case, the techniques for comparison are beyond our present interest, and must be formulated in whatever terms are available.

In the case of structured programs, the program tree permits the decomposition of the correctness problem into a series of nested problems, each of a simple type which can be prescribed in advance.

CORRECTNESS THEOREM

If the formula of a program contains at most graph labels BLOCK, IFTHEN, IFTHENELSE, DOWHILE, and DOUNTIL, and satisfies a loop qualification, then it can be proved correct by a tour of its program tree, in which, at each node, the relevant one of five cases must be proved (data sets suppressed—see below for data set versions):

If f = (BLOCK,g,h), prove

$$[f] = \{(r,t) \mid \exists\, s((r,s) \in [g] \wedge (s,t) \in [h])\}$$

If f = (IFTHEN,p,g), prove

$$[f] = \{(r,s) \mid ((r, \text{True}) \in p \wedge (r,s) \in [g] \vee ((r, \text{False}) \in p \wedge (r,s) \in p \wedge r = s)\}$$

If f = (IFTHENELSE,p,g,h), prove

$$[f] = \{(r,s) \mid ((r, \text{True}) \wedge p \wedge (r,s) \wedge [g]) \vee ((r, \text{False}) \in p \wedge (r,s) \in [h])\}$$

If f = (DOWHILE,p,g), prove

$$[f] = [(\text{IFTHEN},p,(\text{BLOCK},[g], [f]))]$$

If f = (DOUNTIL,p.g), prove

$$[f] = [(\text{BLOCK},[g], (\text{IFTHEN},p,[f]))]$$

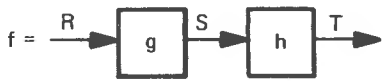
Proof

By hypothesis each node in the program tree is one of the five types listed. Beginning at the root of the tree, the program function [f] of program f is determined by possibly a predicate, and program functions [g], [h] of constituent subprograms g,h, and so on, until functions are reached at the endpoints of the tree. If the program function at each node is known with respect to program functions of its successor nodes, then by finite induction, the program function at the root of the tree is known with respect to the functions in the program.

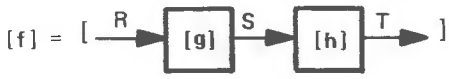
It remains to validate the detailed assertions case by case.

Case f = (BLOCK,g,h)

In flowchart form,



Now



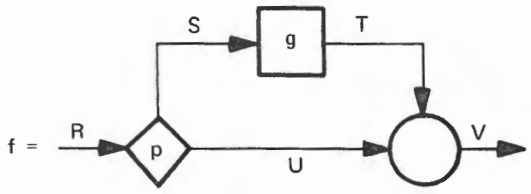
by the definition of program functions [g], [h]. Then, program function [f] can be formulated directly as

$$[f] = \{(r,t) \mid r \in R \;\; \exists \; s((r,s) \in [g] \quad s \in S \quad (s,t) \in [h]) \quad t \in T\}.$$

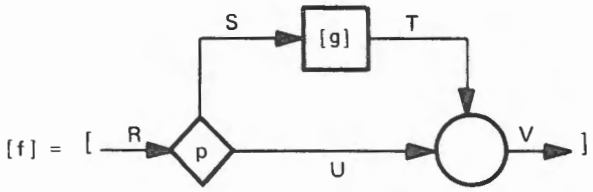
This agrees with the statement of the Theorem with the data sets suppressed.

Case f = (IFTHEN,p,g)

In flowchart form,



Now



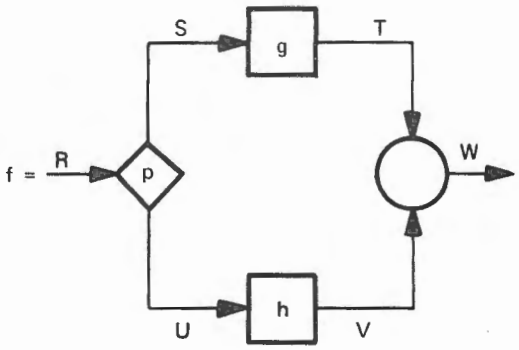
Then

$$[f] = \{(r,v) \mid r \in R \wedge$$
$$(((r,True) \in p \wedge r \in S \wedge (r,v) \in [g] \wedge v \in T)$$
$$\vee((r,False) \in p \wedge r = v \wedge v \in U)) \wedge v \in V\}.$$

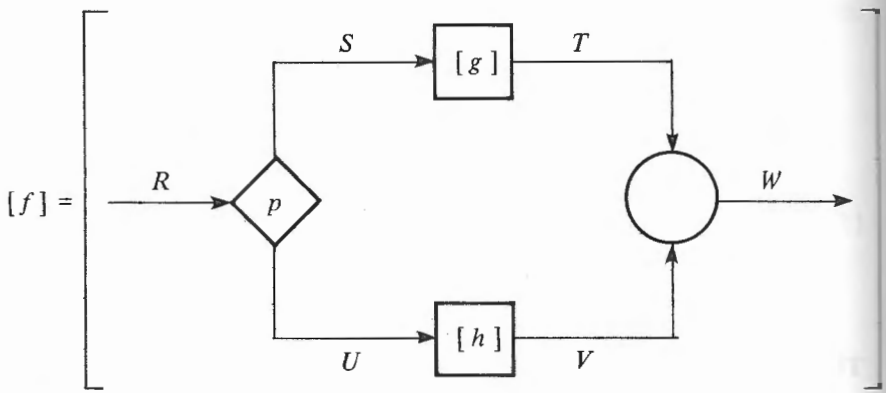
This agrees with the statement of the Theorem with the data sets suppressed.

Case f = (IFTHENELSE,p,g,h)

In flowchart form,



Now



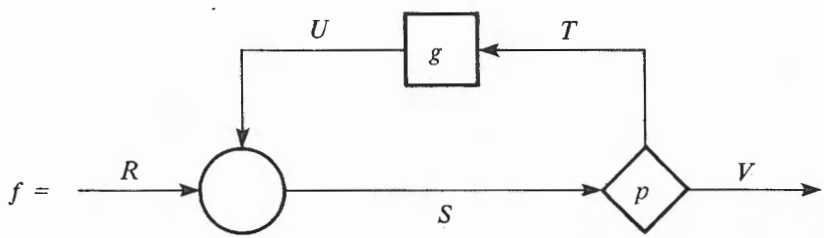
Then

$$[f] = \{(r, w) \mid r \in R \wedge (((r, \text{True}) \in p \wedge r \in S \wedge (r, w) \in [g] \wedge w \in T) \wedge ((r, \text{False}) \in p \wedge r \in U \wedge (r, w) \in [h] \wedge w \in V)) \wedge w \in W\}.$$

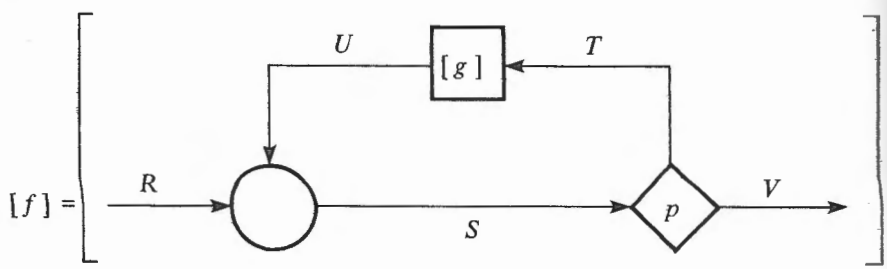
This agrees with the statement of the theorem with the data sets suppressed.

Case $f =$ (DO-WHILE, p, g)

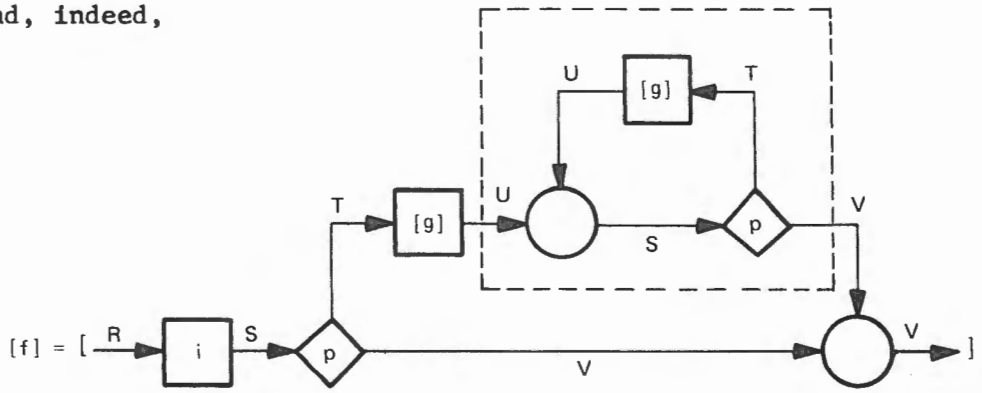
In flowchart form,



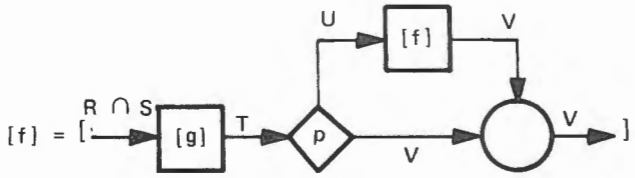
Now



and, indeed,



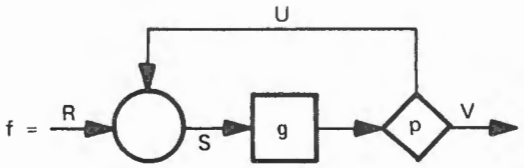
by construction and inspection, where i is an identity function. We note that if R = U then the DOWHILE subprogram in the dotted section has program function [f] i.e.,



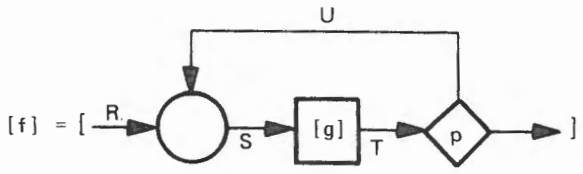
This agrees with the statement of the Theorem with the data sets suppressed. We call the condition R=U the <u>loop qualification</u> on f; i.e., both input lines to the collecting node have identical data spaces.

Case f = (DOUNTIL,p,g)

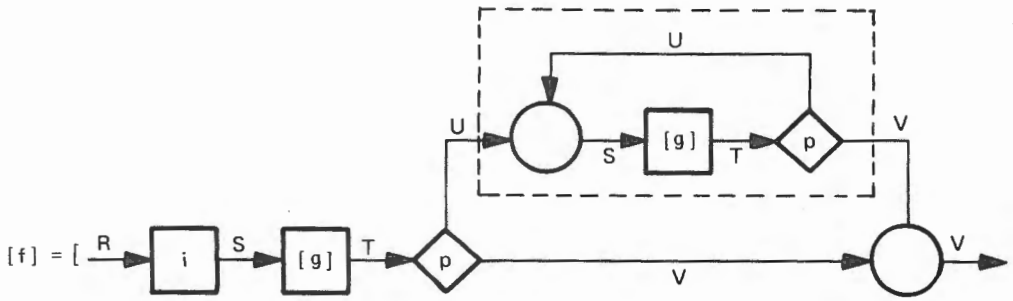
In flowchart form,



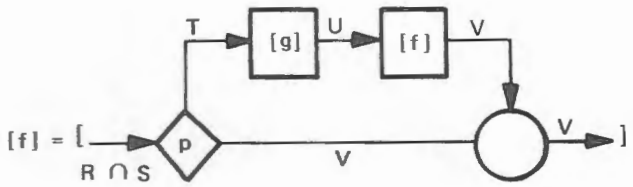
Now,



and, indeed,



by construction and inspection. If R = U, (the loop qualification), then the DOUNTIL subprogram in the dotted section has program function [f], i.e.,



This agrees with the statement of the Theorem with the data sets suppressed.

With this case, the proof of the Theorem is completed.

Correctness Notes

At first glance, the verification conditions for DOWHILE and DOUNTIL seem to involve a recursive relation in program function [f]. But this is not the case; the verification conditions involve [f] as an input, not as an unknown to be solved for.

It is also noteworthy that the top down approach to correctness avoids the problem of incomplete rules (or in other formulations, incomplete functions, for which we have no counterparts), and termination. In a program equation such as

f = WHILE p DO g ,

the functions p and [g] are usually taken to be the ''independent variables'' and the function [f] to be the ''dependent variable'', a ''bottom up'' viewpoint. Of course, even though p and [g] may be given by complete rules, the new rule ''WHILE p DO g'' may turn out to be partial because of nontermination. However, in the top down viewpoint, the function [f] is the ''independent variable'', and the program equation defines ''dependent variables'' p and [g] implicitly, (and meaningfully). Now, since [f] is a function, p and [g] must be defined such that the rule ''WHILE p DO g'' terminates for any input in the domain of [f].

The loop qualification required in the Correctness Theorem is a serious restriction with respect to the allocation and freeing of storage space. If the body of a DO loop allocates or frees space, then the loop qualification is not satisfied, and the reduction of a loop verification to the form of the Theorem is not valid.

Thus far, we have considered programs first, and then their meanings as program functions. In top down programming, we want to reverse that order of conception. That is, given a function (a program specification) we want to find some program (a rule) which has that program function. This reversal of conception allows us to avoid questions of ''partial rules'', ''partial correctness'', and the general termination problem, because they never arise. In the usual way of looking at program equations, such as

f = (DOWHILE,p,g)

the graph label DOWHILE, predicate p, and function or subprogram g, are usually taken to be the ''independent variables'' and program f taken to be the ''dependent variable''. In this case, even though p and g are given by rules defined everywhere on their domains, the new program (DOWHILE,p,g) may not terminate and thus be called a partial rule. One may prove properties relating p and g to f in case of termination to get partial correctness, but one must also establish termination separately to get total correctness.

We observe that if we take f to be the independent variable in the foregoing equation, then these partial rule and partial correctness problems disappear. If f denotes a complete rule, then p and g must denote complete rules, in order to satisfy the equation as dependent variables. That is the essence of top down programming, regarding the constituent subprograms and predicates of an expansion as dependent variables which satisfy a prescribed equation which is inherited top down.

When this approach is taken, perhaps the most surprising result is the amount of freedom available to a programmer in writing a correct program. In the bottom up approach, programming appears to be an activity with almost unlimited freedom to improvise or solve problems in various ways. But in developing a program top down, it is clear that this freedom is highly restricted. At first glance it may seem there is less freedom in programming top down than in bottom up, but a second thought shows that is not the case. They must lead to equivalent results and, in fact, what really is exhibited in the bottom up approach is a false freedom that is subsequently paid for in a painful error elimination process, following an original ''gush of originality.''

In order to exhibit the degree of freedom available in programming, we formulate the Expansion Theorem below in both a verbal and a set theoretic version. The Structure Theorem exhibits characteristics of a completed program, while the Expansion Theorem shows how programs can look at every intermediate stage of their construction. At every such intermediate stage, a program developed in a top down discipline can be guaranteed to

be correct, insofar as it is developed, without the necessity of altering parts of the program already done, in order to accommodate the remaining parts of the program yet to be developed. It is a familiar experience in large program development to get ''90% done'' and to remain at that 90% level for a lengthy period. That phenomena occurs not because the last 10% is difficult to write, but because in order to write the last 10%, critical sections of the first 90% need to be altered. The Expansion Theorem and top down programming can guarantee that the first 90% can remain intact while the last 10% is finished on schedule.

EXPANSION THEOREM (VERBAL VERSION)

In a program function expansion of the form (data sets suppressed -- see below for more detail):

(1) f = [(BLOCK,g,h)]
Any pair (g,h) whose composition is f may be chosen.

(2) f = [(IFTHENELSE,p,g,h)]
Any predicate p with the same domain as f may be chosen, then g and h are fully determined, as the members of the partition of f defined by p.

(3) f = [(DOWHILE,p,g)]
The program function f must be the identity in the intersection of its domain and range, any function g may be chosen whose completion is the varying part of f, and p is fully determined by f and g.

In short, the invention of an IFTHENELSE program is equivalent to a partition of a prescribed program function, while the invention of a DOWHILE program is equivalent to the determination of a function whose completion is a prescribed program function. That is, the only freedom in an IFTHENELSE program is its predicate, and the only freedom in a DOWHILE program is its iterative process -- all other freedoms, in the THEN or ELSE clauses, in the WHILE predicate, are illusions. THEN and ELSE clauses are frequently used for elaborating functional specifications not fully stated; but these are not freedoms of choice, but interpretations of intentions at more detailed levels. The point is that if functional specifications are sufficiently well defined to decide whether a program satisfies them, then there is no freedom beyond the choice of the predicate in an IFTHENELSE program. In the case of the DOWHILE, the question is more subtle, and relates to the character of the termination questions in programming top down, in contrast to bottom up. The WHILE predicate is completely determined on the domain and range of the function (specification). The DOWHILE program must terminate on reaching any element of the range, and must continue otherwise; because,

if not, it cannot possibly satisfy the prestated (top down inherited) function specification.

In order to formulate a more concise, set theoretic version of the Expansion Theorem, we introduce a reinterpretation of the logical constant ''True''. Ordinarily, a predicate is taken to be a function, p, such that

$$\text{range}(p) = \{ \text{True}, \text{False} \}.$$

We reinterpret the constant True by the statement for an associated function

$$\bar{p} = \{(x,y) \mid (x,\text{True}) \in p\} ;$$

i.e., if p(x) is true, then for any element y , $(x,y) \in \bar{p}$.

We also introduce the idea of a refinement of a function, corresponding to the ordinary idea of the refinement of a partition. (A refinement of a partition is simply a new partition, each of whose members is a subset of some member of the original partition.) We form a partition of the domain of a function, called a partition of <u>level sets</u>, or the <u>contour</u> of the function, by grouping arguments which have identical values into subsets of the domain. Then we say one function is a refinement of another if its contour is a refinement of the others.

Finally, we define the <u>fixed points</u> of a function f, denoted as the fixed(f) subset

$$\text{fixed}(f) = \{(x,y) \mid (x,y) \in f \wedge x = y\}.$$

EXPANSION THEOREM (SET THEORETIC VERSION)

In a program expansion of the form (data sets suppressed -- see below for more detail):

(1) f = [(BLOCK,g,h)]

(a) choose function g as any refinement of program function f

(b) then h is uniquely determined by the relation
f = g * h

(2) f = [(IFTHENELSE,p,g,h)]

(a) choose predicate p such that domain(p) = domain(f)

(b) then g and h are uniquely determined by the relations

$$g = \bar{p} \cap f$$
$$h = f - g$$

(3) $f = [(\text{DOWHILE},p,g)]$

(a) verify that
$\text{domain}(\text{fixed}(f)) = \text{domain}(f) \cap \text{range}(f)$

(b) choose function g such that
$* \; g \; * = f - \text{fixed}(f)$.

(c) then p is uniquely determined such that
$p(x) = \text{True if } x \in \text{domain}(g)-\text{range}(f)$
$p(x) = \text{False if } x \in \text{range}(f)$

Proof

Case $f = [(\text{BLOCK},g,h)]$

In flowchart form

Consider the following construction of g, h, R, S, T:

Set $R = \text{domain}(f)$.

Set $T = \text{range}(f)$.

Choose any refinement of f, say g; then for any $x \in R$, $y \in R$,

$$g(x) = g(y) \supset f(x) = f(y)$$

Set $S = \text{range}(g)$

Set $h = \{(s,t) \mid (r,s) \in g \wedge (r,t) \in f\}$

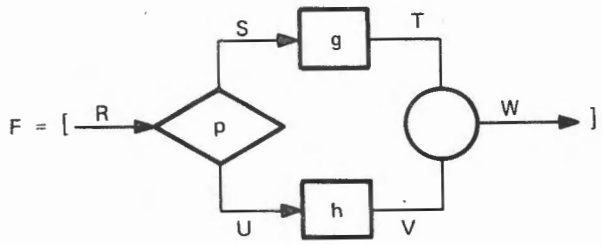
Now, it is easy to verify by this construction that

$$[\ (\text{BLOCK},g,h)] = \{(x,y) \mid (x,y) \in f\}$$

as was to be shown. The function h is uniquely determined in the construction by f and g.

Case f = [(IFTHENELSE,p,g,h)]

In flowchart form



Consider the following construction of p, g, h, R, S, T, U, V, W:

Set R = domain(f)

Choose any predicate p such that domain(p) = domain(f) = R

Set S = $\{ s \mid (s,True) \in p\}$

Set g = $\{(s,t) \mid s \in S \wedge (s,t) \in f\}$

Set T = range(g)

Set U = $\{u \mid (u,False) \in p\}$

Set h = $\{(u,v) \mid u \in U \wedge (u,v) \in f\}$

Set V = range(h)

Set W = $T \cup V$

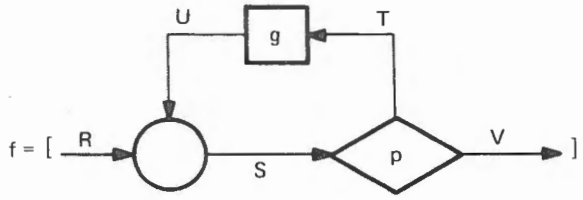
Now, it is easy to verify by this construction that

[(IFTHENELSE,p,g,h)] = $\{(x,y) \mid (x,y) \in f\}$

as was to be shown. Note that g is a subset of f defined by p, i.e., $\bar{p} \cap f$, and h is the complement of g in f, i.e., f - g.

Case f = [(DOWHILE,p,g)]

In flowchart form



Consider (s,v) $\in$ f, i.e., v $\in$ range(f). We note that necessarily p(v) = False. Otherwise the control path to g is taken, and the program cannot terminate with value v, which contradicts the correctness of the expansion.

Next, consider (r,v) $\in$ f such that r $\in$ domain(f) $\cap$ range(f); then p(v) = False by the foregoing remark, and the function g is bypassed, so that necessarily v = r, or r $\in$ domain(fixed(f)). Conversely, if r $\in$ domain(fixed(f)), then r $\in$ range(f) and p(r) = False, hence r $\in$ domain(f) $\cap$ range(f). That is, domain(fixed(f)) = domain(f) $\cap$ range(f) as needed to be shown.

Next, choose function g such that * g * = f - fixed(f). At least one such choice is possible, namely for g = f - fixed(f), since the domain and range of f - fixed(f) is disjoint.

Finally, we have already seen that necessarily p(x) = False when x $\in$ range(f). But clearly, we must have p(x) = True when x $\in$ domain(g), in order that the correct control path be taken to finally reach an output v $\in$ range(f); in addition, since * g * $\supset$ f, then necessarily domain(g) $\supset$ domain(*g*) $\supset$ domain(f), so that x $\in$ domain(g) implies x $\in$ domain(f). Thus, in summary,

p(x) = True if x $\in$ domain(g) - range(f)

p(x) = False if x $\in$ range(f)

as was to be shown.

The data sets required are as follows:

Set R = domain(f)

Set V = range(f)

Set T = domain(g)

Set U = range(g)

Set S = R $\cup$ U

This discussion is concluded with a combinatorial characterization of g, the iterative process of a DOWHILE program:

For function f, consider any superfunction h, such that range (h) = range(f). For each level set, or contour, of h, define any arbitrary set of rooted trees on its elements. If x of domain(h) is a root of such a tree, then we set

$$y(x) = h(x).$$

If $x \in$ domain(h) is not a root of such a tree, let y denote the parent of x in that tree, and define

$$g(x) = y.$$

It is easily verified that any function g so defined, and no other, will satisfy the relation * g * = f.

With this, it is clear that in all three cases, the entire freedom of choice is a combinatorial one. In a BLOCK program, it is the choice of a function; in an IFTHENELSE program, the choice is a partition of a function; in a DOWHILE program, the choice is a tree structure within the level sets of a function.

In certain applications, particularly those of artificial intelligence [33], it is convenient to generalize the idea of a program to a construct which permits ambiguity in execution, rather than uniqueness. For example, an algorithm may specify a selection of a member of some set for processing, without naming a specific member. In this event, intermediate and/or final results may be indeterminate. Such ''indeterminate algorithms'', are often useful in describing the essentials of a process, without getting unduly involved with its specifics. Indeterminate algorithms are also useful for treating a man-machine computing system, in which the actions of men -- say at terminals -- are indeterminate. Then, an entire system can be defined to be governed by an indeterminate algorithm.

Our development of programs, which we call ''determinate programs'', where necessary, can be generalized to include ''indeterminate programs'' by a very simple extension -- namely, by extending the idea of function, throughout, to the idea of relation. A relation is defined to be a set of ordered pairs, without the additional qualification required of a function to provide unique values for given arguments. As with functions, relations inherit set properties. In fact, not only the intersection and difference of two relations are new relations (as in the case of functions), but the union of two relations is also a relation (not generally so for functions). Domains and ranges of relations are defined as for functions.

Next, we define an indeterminate program to be a finite set of relations, called indeterminate instructions, whose domains are each included in a data space, and whose ranges are each included in the cartesian product of the data space and the indeterminate program, again called the state space. An indeterminate program execution is, again, a sequence of state values

$$s = (d_i, r_i),\ i = 0, 1, \ldots$$

such that

$$(d_i, s_{i+1}) \in r_i,\quad i = 0, 1, \ldots$$

which terminates, if ever, when $d_i \notin \text{domain}\ (r_i)$. Precisely as before, all executions which terminate define a set of ordered pairs, now a relation, instead of a function, which we call the indeterminate program

relation; i.e., in retrospect, an indeterminate program is a (nonunique) rule for calculating the members of its relation, using other relations in so doing.

At this point, we leave it to the reader to observe that every construction and theorem goes through for indeterminate programs and their relations, just as for determinate programs and their functions.

REFERENCES

1. Allen, C. D., ''The Application of Formal Logic to Programs and Programming,'' IBM Systems Journal, Vol. 10, No. 1 (1971) pp. 2-38.

2. Ashcroft, E. A., ''Program Correctness Methods and Language Definition,'' Proceeding ACM Conference on Proving Assertions about Programs, New Mexico University, Las Cruces, N. M. (January 6-7, 1972) pp. 51-57.

3. Ashcroft, Edward A., and Manna, Zohar, ''The Translation of 'GO TO' Programs to 'WHILE' Programs,'' Stanford Artifical Intelligence Project, Memo AIM-138, Computer Sciences Department Report No. STAN-CS-71-188 (Jan. 1971).

4. Baker, F. T., ''Chief Programmer Team Management of Production Programming,'' IBM Systems Journal, Vol. 11, No. 1 (1972) pp. 56-73.

5. Böhm, Corrado, and Jacopini, Giuseppe, ''Flow diagrams, Turing machines and Languages With Only Two Formation Rules, Communications of Association for Computing Machinery, Vol. 9 (1966) pp. 366-371.

6. Burge, W. H., ''Some Examples of Programming Using a Functional Notation,'' Second Symposium of Special Interest Group Association for Computing Machinery on Symbolic and Algebraic Manipulation, Los Angeles (March 1971).

7. Burstall, Rod M., ''An Algebraic Description of Programs with Assertions, Verification and Simulation,'' Proceedings Association for Computing Machinery Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, N. M. (1972).

8. Church, A., Introduction to Mathematical Logic, Vol. 1, Princeton University Press, Princeton, N. J. (1956).

9. Dijkstra, E. W., ''A Constructive Approach to the Problem of Program Correctness,'' BIT, Vol. 8, No. 3 (1968) pp. 174-186.

10. Dijkstra, E. W., ''Notes on Structured Programming,'' Technische Hogeschool Eindhoven (THE), (1969).

11. Dijkstra, E. W., ''Structured Programming,'' Software Engineering Techniques, NATO Science Committee (Eds. Burton, J. N. and Randell, B.), (1969) pp. 88-93.

12. Floyd, R. W., ''Assigning Meanings to Programs,'' Proceedings of the Symposium in Applied Mathematics, Vol. 19 (Ed. J. T. Schwartz), American Mathematical Society, Providence, R. I. (1967) pp. 19-32.

13. Floyd, R. W., ''Nondeterministic Algorithms,'' Journal of the Association for Computing Machinery, Vol. 14, No. 4 (Oct. 1967) pp. 636-644.

14. Good, D. I., ''Toward a Man-machine System for Proving Program Correctness,'' Ph. D. Thesis, University of Wisconsin, (1970).

15. Halmos, Paul R., Naive Set Theory, D. Van Nostrand Co., Inc., (Eds. J. L. Kelley, P. R. Halmos), Princeton, N. J., (1960).

16. Hoare, C. A. R., ''An Axiomatic Approach to Computer Programming,'' Communication of the Association for Computing Machinery Vol. 12, No. 10 (Oct. 1969), pp. 576-580, 583.

17. Hoare, C. A. R., Proof of a Program: FIND,'' Communications of the Association for Computing Machinery, Volume 14, No. 1 (Jan. 1971), pp. 39-45.

18. Horning, J. J., and Randell, B., ''Structuring Complex Processes,'' IBM T. J. Watson Research Center, Report RC 2459 (May 1969).

19. Ianov, Iu, ''The Logical Schemas of Algorithms,'' Problems of Cybernetics 1. English translation (Pergamon Press) (1960).

20. Jones, C. B., ''Formal Development of Correct Algorithms: An example based on Earley's recognizer,'' Proceedings of the Association for Computing Machinery Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, N. M. (Jan. 1972) pp. 51-57.

21. King, J. C., ''A Program Verifier,'' Ph.D. Thesis, Carnegie-Mellon University (1969).

22. Landin, P. J., ''A Correspondence between ALGOL 60 and Church's Lambda-Notation,'' Communication of the Association for Computing Machinery 8 (Mar. 1965).

23. London, R. L., ''Certification of Algorithm 245 Treesort 3: Proof of Algorithms—A New Kind of Certification,'' Comm. ACM 13 (1970), pp. 371-373.

24. London, R. L., ''Correctness of a Compiler for a LISP Subset,'' Proceedings of the Association for Computing Machinery Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, N. M. (Jan. 1972), pp. 51-57.

25. London, R. L., ''Proving Programs Correct: Some Techniques and Examples,'' BIT, 10, 2 (1970), pp. 168-182.

26. McCarthy, J., ''Towards a Mathematical Science of Computation,'' Proceedings of the IFIP Congress, North-Holland, Amsterdam (1962).

27. Mendelson, E., Introduction to Mathematical Logic, D. Van Nostrand Co., Inc., Princeton, N.J. (1964).

28. Mills, H. D., ''Syntax-Directed Documentation for PL360,'' Communications of the Association for Computing Machinery, 13, No. 4 (April 1970), pp. 216-222.

29. Mills, H. D., ''Top Down Programming in Large Systems,'' Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, NYU Ed. Randall Rustin, (1971), pp. 41-55.

30. Nahikian, Howard, M., Topics in Modern Mathematics, The MacMillan Co., Collier-MacMillan Ltd., London, Ed. C. B. Allendoerfer, (1966).

31. Naur, P., ''Programming by Action Clusters,'' BIT 9 (1969), pp. 250-258.

32. Naur, P., ''Proof of Algorithms by General Snapshots,'' BIT 6 (1966), pp. 310-316.

33. Nilsson, N. J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, Inc., (1971).

34. Scott, D., ''The Lattice of Flow Diagrams,'' Programming Research Group, Oxford University (1970).

35. Scott, D., ''An Outline of a Mathematical Theory of Computation,'' Programming Research Group, Oxford University (1970).

36. Snowdon, R. A., PEARL: An Interactive System for the Preparation and Validation of Structured Programs, Tech. Report No. 28, University of Newcastle Upon Tyne, Computing Laboratory Ed. Dr. B. Shaw, (1971).

37. Strachey, Christopher, ''Towards a Formal Semantics,'' Formal Language Description Languages, Ed. T. B. Steel, North-Holland (1966), pp. 198-220.

38. Wilder, Raymond L., Evolution of Mathematical Concepts - An Elementary Study, John Wiley & Sons, Inc., New York (1968).

39. Wirth, Niklaus, ''Program development by Stepwise Refinement,'' Communications of the Association for Computing Machinery 14, No. 4 (Apr. 1971), pp. 221-227.

40. Zurcher, F., and Randell, B., ''Iterative Multi-Level Modelling - A Methodology for Computer System Design,'' Proceedings of the IFIP Congress (1968), pp. D138-D142.