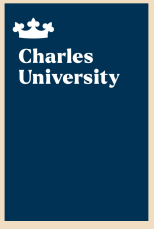
Mathematical science of computation and structured programming

Cultures of Programming — Part 4

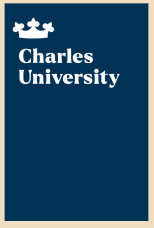
Tomas Petricek, Charles University, Prague

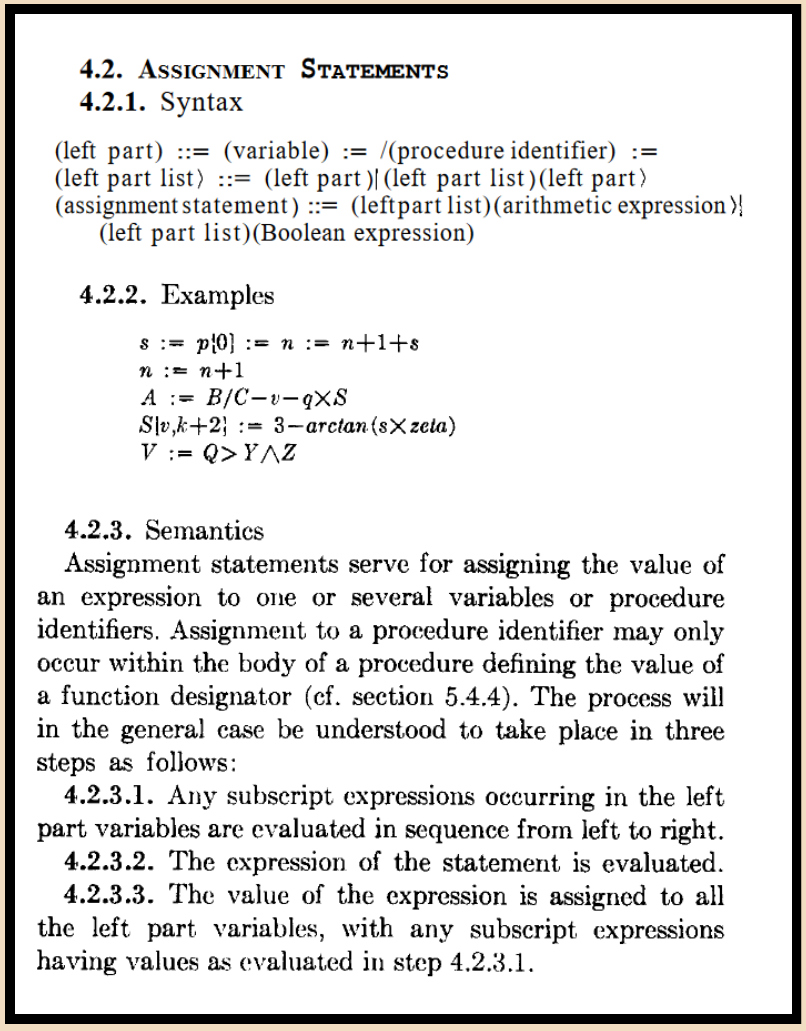
tomas@tomasp.net
https://tomasp.net
@tomasp.net



Mathematical culture

Proving program properties





4.2. Assignment Statements
4.2.1. Syntax

⟨left part⟩ ::= ⟨variable⟩ := |⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩|⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩|
⟨left part list⟩⟨Boolean expression⟩

4.2.2. Examples

$s := p[0] := n := n+1+s$
$n := n+1$
$A := B/C-v-q\times S$
$S[v,k+2] := 3-arctan(s\times zeta)$
$V := Q>Y\wedge Z$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

Mathematical science of computation

Programs as formal entities
Algorithms as objects to study
Formal mathematical reasoning

Algol paradigm

Charles University

Algol paradigm

Mathematical Science (McCarthy, 1962)

- Two procedures (functions) are equivalent
- Procedure terminates for all input data
- Translation correctly translates procedures
- Translation increases efficiency

Charles University

McCarthy's factorial proof

$$g(n, s) = \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ g(n-1, n \times s)$$

$$\begin{aligned}
n! \times s &= n! \times s && (1)\\
n! \times s &= \texttt{if}\ n = 0\ \texttt{then}\ n! \times s\ \texttt{else}\ n! \times s && (2)\\
n! \times s &= \texttt{if}\ n = 0\ \texttt{then}\ 1 \times s\ \texttt{else}\ ((n-1)! \times n) \times s && (3)\\
n! \times s &= \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ (n-1)! \times (n \times s) && (4)\\
g(n, s) &= \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ g(n-1, n \times s) && (5)
\end{aligned}$$

Charles University

Algol paradigm

Proving programs correct



Peter Naur (1966)

Proofs of algorithms by general snapshots



Robert Floyd (1967)

Assigning meaning to programs



Tony Hoare (1969)

An axiomatic basis for computer programming

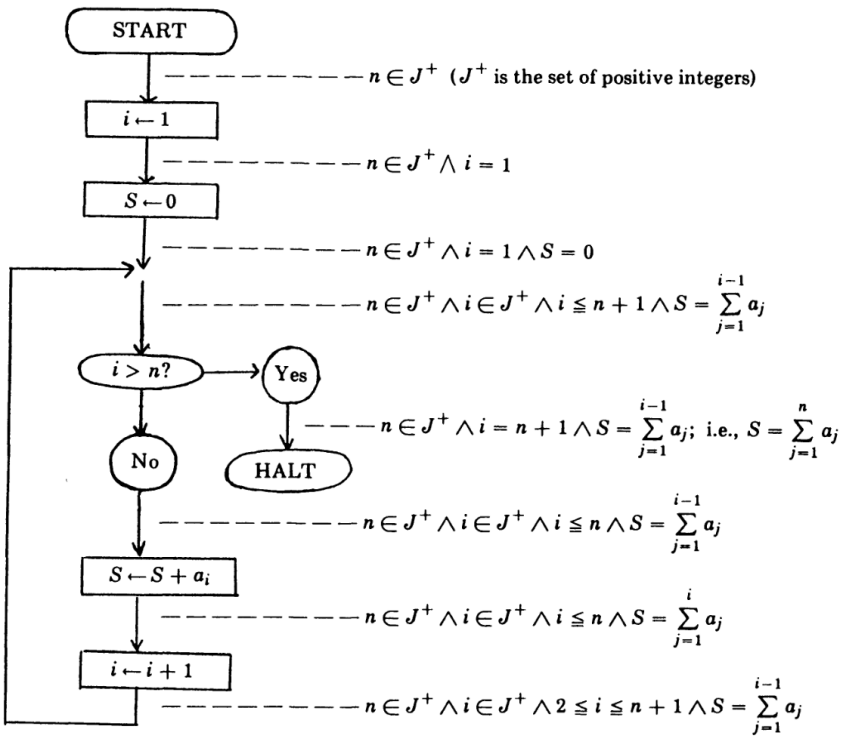


Fig. 1. Flowchart of program to compute $S = \Sigma_{j=1}^{n} a_j$ ($n \geqq 0$).

Floyd (1967)

Cites McCarthy and uses a subset of Algol

But uses flowcharts!

Common visual representation of programs at the time

Charles University

```
s = 1; i = 1;

while i < n do

  i = i + 1;
  s = s * i;

end
```

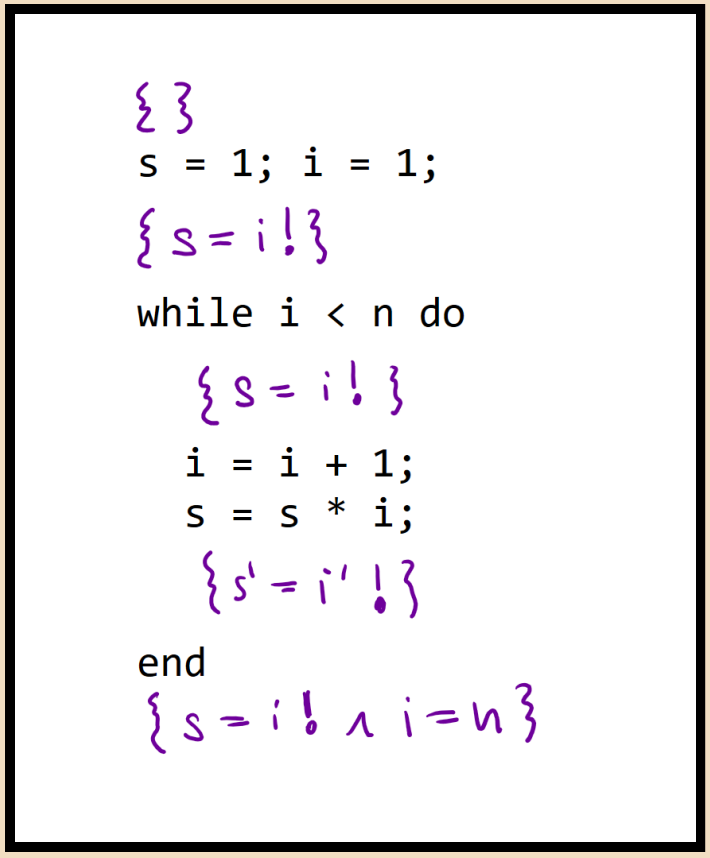
Hoare (1969)

Annotate programs using tripes $\{P\}\, c\, \{Q\}$ where:

- P pre-condition holds before
- c command to run
- Q post-condition holds afterwards

Using modern notation for (now called) Hoare-triples

Charles University



Hoare (1969)

Reasoning about programs using Hoare triples

Composition rule!

If $\{P\}\, c_1\, \{Q\}$
and $\{Q\}\, c_2\, \{R\}$
then $\{P\}\, c_1; c_2\, \{R\}$

Charles University

Proofs of programs in practice

Proof of a Program: FIND

C. A. R. HOARE
Queen's University, Belfast, Ireland*

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constucting the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

2. The Program "Find"

The purpose of the program Find [4] is to find that element of an array $A[1:N]$ whose value is fth in order of magnitude; and to rearrange the array in such a way that this element is placed in $A[f]$; and furthermore, all elements with subscripts lower than f have lesser values, and all elements with subscripts greater than f have greater values. Thus on completion of the program, the following relationship will hold:

$$A[1], A[2], \cdots, A[f-1] \leq A[f] \leq A[f+1], \cdots, A[N]$$

This relation is abbreviated as Found.

One method of achieving the desired effect would be to sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N+1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N+1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that the array is *split at* the given point into two parts (see Table I(b)).

The reasoning of the previous paragraph assumes that there is some means of distinguishing small elements from large ones. Since we are interested only in their comparative values, it is sufficient to select the value of some arbitrary element before either of the scans starts; any element with lower value than the selected element is counted as small, and any element with higher value is counted as large. The fact that the discriminating value is arbitrary means that the place where the two scans will meet is also arbitrary; but it does not affect the fact that the array will be split at the meeting point, wherever that may be.

Now consider the question on which side of the split the fth element in order of value is to be found. If the split is to the right of $A[f]$, then the desired element must of necessity be to the left of the split, and all elements to the right of the split will be greater than it. In this case, all elements to the right of the split can be ignored in any future processing, since they are already in their proper

* Department of Computer Science

Prevention of System Deadlocks

A. N. HABERMANN
Carnegie-Mellon University, Pittsburgh, Pennsylvania*

A well-known problem in the design of operating systems is the selection of a resource allocation policy that will prevent deadlock. Deadlock is the situation in which resources have been allocated to various tasks in such a way that none of the tasks can continue. The various published solutions have been somewhat restrictive: either they do not handle the problem in sufficient generality or they suggest policies which will on occasion refuse a request which could have been safely granted. Algorithms are presented which examine a request in the light of the current allocation of resources and determine whether or not the granting of the request will introduce the possibility of a deadlock. Proofs given in the appendixes show that the conditions imposed by the algorithms are both necessary and sufficient to prevent deadlock. The algorithms have been successfully used in the THE system.

KEY WORDS AND PHRASES: multiprogramming, time-sharing, scheduling, resource allocation
CR CATEGORIES: 3.72, 4.32, 6.20

Introduction

Consider a set of sequential processes which share system resources in such a way that (1) while a resource A is allocated to a process P_i no other process P_j can seize A, and (2) an allocated resource is not released until it has fulfilled its task.

Dijkstra shows in [2] and [6] how the use of system resources can be made exclusive by means of the primitives P-operation and V-operation. Dennis and Van Horn [3] use the macro instructions *LOCK* and *UNLK*, specially designed for this purpose. These techniques have in common that a process P_i performing a P-operation (e.g. *LOCK* instruction) will get the resource requested if it is free and, if not, process P_i is held up until a matching V-operation (e.g. *UNLK* instruction) is performed. The techniques also have in common the fact that they do not imply any rules for resource allocation. This is still a matter of system design.

If the total request for resources of a certain type can exceed the amount available, some strategy is needed to order the assignment of resources in time. Care must be taken that the strategy applied cannot cause a deadlock: i.e. the situation that cooperating processes prevent their mutual progress even though no single one requests more resources than are available.

Suppose, for instance, each of two processes P_1 and P_2 needs the resources A and B at the same time, i.e. when either A or B has been allocated to it. P_1 will certainly not release the allocated resource before the other one has also been allocated to it (likewise P_2). Suppose the strategy is such that allocation will follow on request if the resource is free. Then the situation may arise that P_1 has seized resource A and P_2 resource B. From then on, P_1 and P_2 will wait for each other indefinitely, since P_1 certainly does not release A until it has seized B, i.e. not until P_2 has released B, while P_2 will not release B until it has seized A.

One way out of the difficulty is to permit a set of processes to run concurrently only if together they do not claim more resources than are available [4]. However, we will see that such a strategy is unnecessarily restrictive.

In this paper algorithms are described by means of which a deadlock can be prevented, and yet as many resource requests are granted as can be allowed, while neither are resources reserved on account of a claim, nor is resource request restricted to a few processes. The algorithms decide only whether or not granting a request can produce a deadlock, so that assignment rules (according to priority rules, for instance) can be implemented freely.

In the author's thesis [5] different sorts of resource requests are discussed, in particular, the case of temporary lease, i.e. when a process asks for a resource and can promise its release unconditionally, as well as the case of a single directed request, i.e. when a process can continue independently of resource activity. Though [5] also contains the theorems and proofs, albeit in a slightly different form, the aim of the present paper is to show the usefulness and applicability of the algorithms which prevent a deadlock.

Another Form of Deadlock

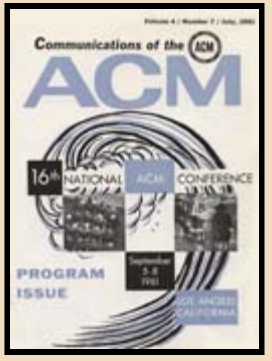
When resource activity itself may imply resource request, another form of deadlock can occur. Let, for instance, a disk device be controlled by a sequential process D that runs synchronously with the device interrupts. D is to be considered a resource, to which tasks can be

* Computation Center

Charles University

Proofs of programs in practice

Prevention of deadlocks (1969)



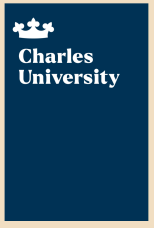
- Abstract algorithm written as pseudocode
- Model using states and transitions
- Not actual source code!

Proof of FIND (1971)

- Part of Hoare's QuickSort algorithm
- Find f^{th} element in an array and rearrange
- $A[1], \ldots, A[f-1] \leq A[f] \leq A[f+1], \ldots, A[N]$
- … it may be possible to enlist the aid of a computer

Mathematical culture

Structuring programs



Loc	Order	Meaning	Comment
0:	T0S	$m[0]$=A; ABC=0	
1:	H2S	R=$m[2]$	Put 10<<11 in R
2:	T0S	$m[0]$=A; ABC=0	
3:	E6S	goto 6	Jump to main loop
4:	P1S	data 2	The constant 2
5:	P5S	data 10	The constant 10
6:	T0S	$m[0]$=A; ABC=0	Start of the main loop
7:	I0S	$m[0]$=rdch()	Get operation code
8:	A0S	A+=$m[0]$	Put it in A
9:	R16S	ABC>>=6	Shift and store it
10:	T0L	$w[0]$=AB; ABC=0	so that it becomes the senior 5 bits of $m[0]$ $m[1]$ is now zero
11:	I2S	$m[2]$=rdch()	Put next ch in $m[2]$
12:	A2S	A+=$m[2]$	Put ch in A
13:	S5S	A-=$m[5]$	A=ch-10
14:	E21S	if A>=0 goto 21	Jump to 21, if ch>=10

EDSAC instructions

EDSAC initial orders
(first few instructions)

Low-level assembly-like

Write to register

Transfer control (jump)

Basic arithmetic

See the EDSAC poster

Charles University

Structured programming

Block-based languages

- Control structures like `while` and `if` instead of `goto`
- Popularized by Algol

```
s = 1; i = 1;                    s = 1; i = 1;
while i < n do                L1 if i = n then goto L2
  i = i + 1;                      i = i + 1;
  s = s × i;                      s = s × i;
end                               goto L1;
print(s);                     L2 : print(s);
```

For structured programming

- Supports reasoning and readability

Against structured programming

- Can we trust compiler generated code?
- Can we really express everything well?

Charles University



Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** *B* **then** *A*), alternative clauses (**if** *B* **then** *A*1 **else** *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2, \cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

Dijkstra (1968)

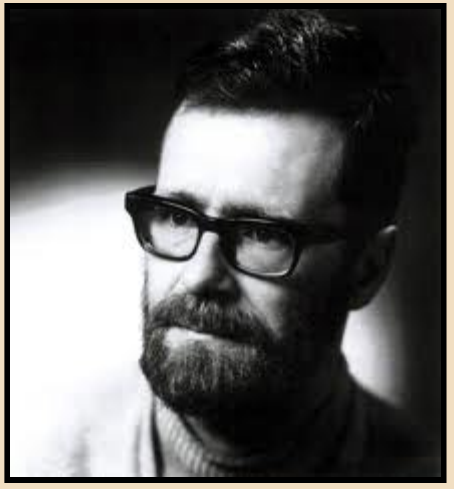
"...the quality of programmers is a decreasing function of the density of **`go to`** statements in the programs they produce."

"...the **`go to`** statement should be abolished from all 'higher level' programming languages"

Charles University

Dijkstra on the go to statement

Making programs understandable



- Programs are read more than written
- Shorten conceptual gap between static program and dynamic process

You do not need go to

- Structured program theorem (Böhm–Jacopini)
- Any flowchart can be encoded using just three control structures (sequence, selection, iteration)

Charles University

Formal reasoning about programs

```
s = 1; i = 1;                    s = 1; i = 1;
while i < n do            L1     if i = n then goto L2
   i = i + 1;                    i = i + 1;
   s = s × i;                    s = s × i;
end                              goto L1;
print(s);                 L2 :   print(s);
```

Why structured programming helps

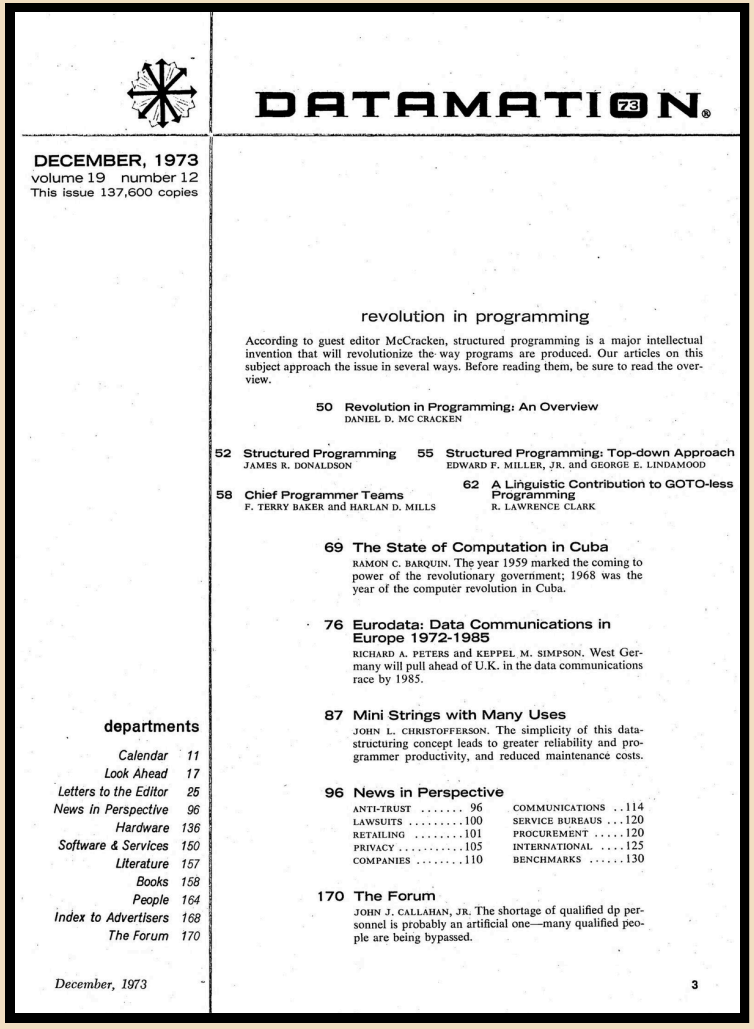
- Recall that if $\{P\}c_1\{Q\}$ and $\{Q\}c_2\{R\}$ then $\{P\}c_1; c_2\{Q\}$
- Works better with block-based languages!
- No need to model register and memory states

Charles University

Managerial culture

Structuring teams

Charles University



DATAMATION 73

DECEMBER, 1973
volume 19 number 12
This issue 137,600 copies

revolution in programming

According to guest editor McCracken, structured programming is a major intellectual invention that will revolutionize the way programs are produced. Our articles on this subject approach the issue in several ways. Before reading them, be sure to read the overview.

50 Revolution in Programming: An Overview
DANIEL D. MC CRACKEN

52 Structured Programming
JAMES R. DONALDSON

55 Structured Programming: Top-down Approach
EDWARD F. MILLER, JR. and GEORGE E. LINDAMOOD

58 Chief Programmer Teams
F. TERRY BAKER and HARLAN D. MILLS

62 A Linguistic Contribution to GOTO-less Programming
R. LAWRENCE CLARK

69 The State of Computation in Cuba
RAMON C. BARQUIN. The year 1959 marked the coming to power of the revolutionary government; 1968 was the year of the computer revolution in Cuba.

76 Eurodata: Data Communications in Europe 1972-1985
RICHARD A. PETERS and KEPPEL M. SIMPSON. West Germany will pull ahead of U.K. in the data communications race by 1985.

87 Mini Strings with Many Uses
JOHN L. CHRISTOFFERSON. The simplicity of this data-structuring concept leads to greater reliability and programmer productivity, and reduced maintenance costs.

96 News in Perspective
ANTI-TRUST 96
LAWSUITS 100
RETAILING 101
PRIVACY 105
COMPANIES 110
COMMUNICATIONS .. 114
SERVICE BUREAUS ... 120
PROCUREMENT 120
INTERNATIONAL 125
BENCHMARKS 130

170 The Forum
JOHN J. CALLAHAN, JR. The shortage of qualified dp personnel is probably an artificial one—many qualified people are being bypassed.

departments

Calendar 11
Look Ahead 17
Letters to the Editor 25
News in Perspective 96
Hardware 136
Software & Services 150
Literature 157
Books 158
People 164
Index to Advertisers 168
The Forum 170

Revolution in programming

(Datamation magazine, 1973)

According to guest editor McCracken, structured programming is a major intellectual invention that will revolutionize the way programs are produced. Our articles on this subject approach the issue in several ways. Before reading them, be sure to read the overview.

Charles University

Chief Programmer Teams

Project management at IBM

- New York Times data bank project
- Overall structure by one person
- Chief programmer + support

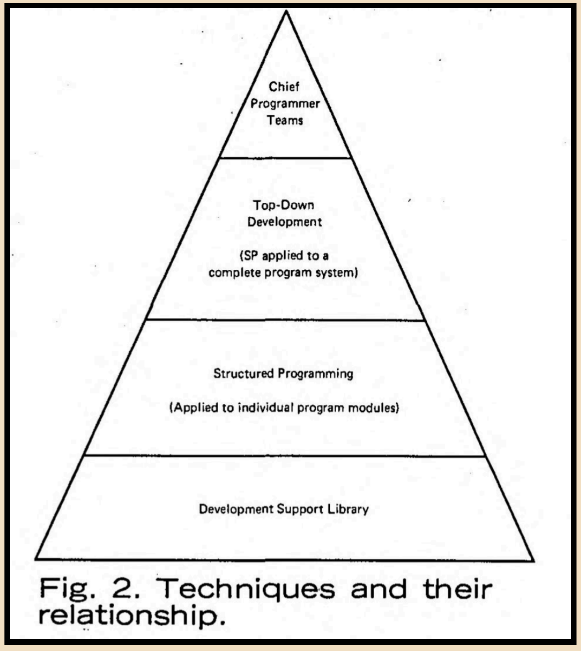
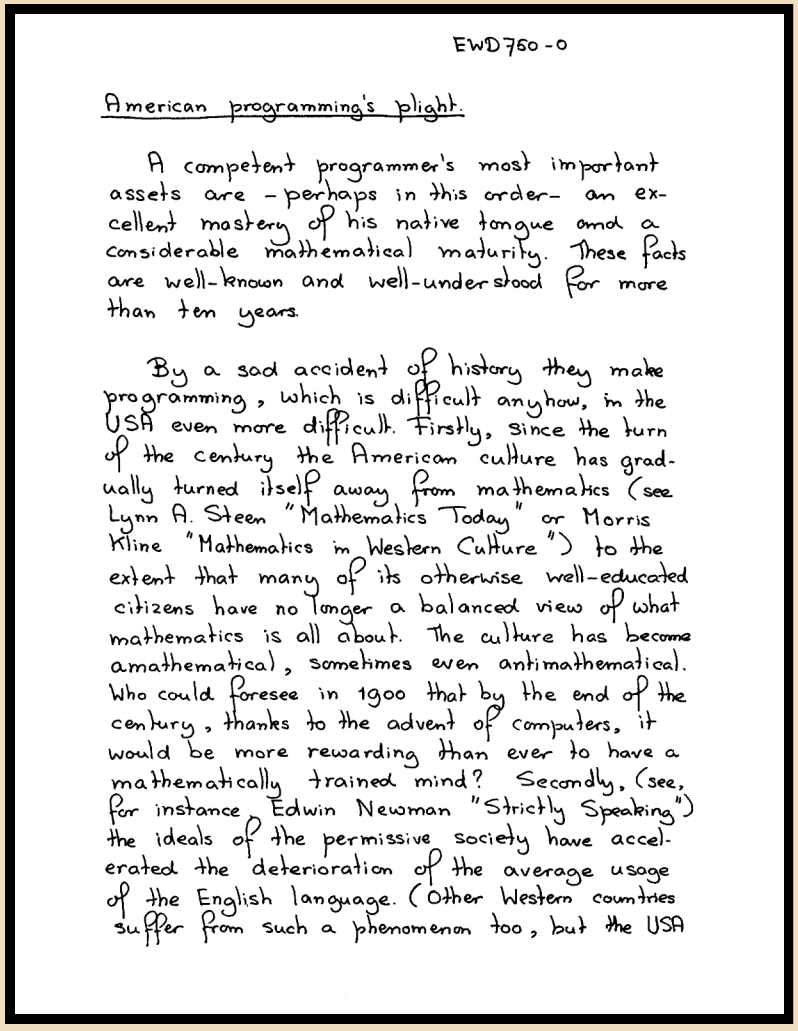


Fig. 2. Techniques and their relationship.

Structured programming

- Top-down design of programs vs. top-down management of teams

Charles University



EWD750-0

American programming's plight.

A competent programmer's most important assets are - perhaps in this order - an excellent mastery of his native tongue and a considerable mathematical maturity. These facts are well-known and well-understood for more than ten years.

By a sad accident of history they make programming, which is difficult anyhow, in the USA even more difficult. Firstly, since the turn of the century the American culture has gradually turned itself away from mathematics (see Lynn A. Steen "Mathematics Today" or Morris Kline "Mathematics in Western Culture") to the extent that many of its otherwise well-educated citizens have no longer a balanced view of what mathematics is all about. The culture has become amathematical, sometimes even antimathematical. Who could foresee in 1900 that by the end of the century, thanks to the advent of computers, it would be more rewarding than ever to have a mathematically trained mind? Secondly, (see, for instance, Edwin Newman "Strictly Speaking") the ideals of the permissive society have accelerated the deterioration of the average usage of the English language. (Other Western countries suffer from such a phenomenon too, but the USA

Dijkstra's response

How to program if you cannot

"the prevailing management philosophy [makes] companies as independent as possible of the competence of their employees"

Charles University

Structured programming

Mathematical culture

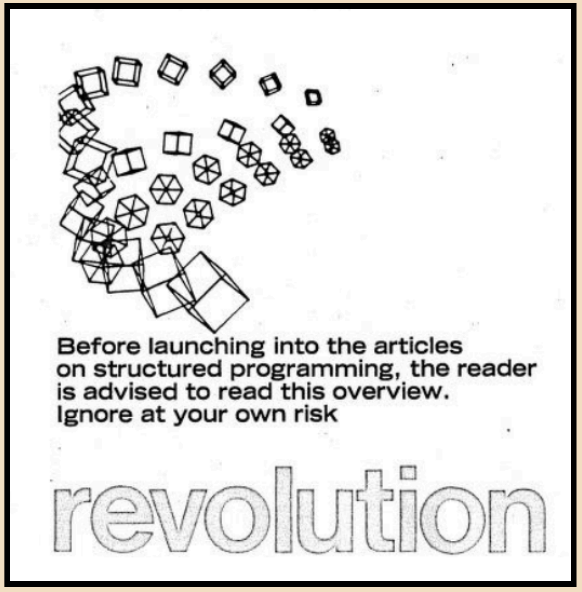
- Mathematical reasoning
- Compositional semantics

Engineering culture

- Understanding code
- Individual responsibility

Managerial culture

- Top-down software development
- Structuring teams



Charles University

Cultures of Programming



Home → Continue to Part 5

Charles University