

REFERENCE



# NBS TECHNICAL NOTE 842

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

## Concepts in Quality Software Design

— QC —  
100  
.U5753  
no. 842  
1974

## NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards<sup>1</sup> was established by an act of Congress March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau consists of the Institute for Basic Standards, the Institute for Materials Research, the Institute for Applied Technology, the Institute for Computer Sciences and Technology, and the Office for Information Programs.

**THE INSTITUTE FOR BASIC STANDARDS** provides the central basis within the United States of a complete and consistent system of physical measurement; coordinates that system with measurement systems of other nations; and furnishes essential services leading to accurate and uniform physical measurements throughout the Nation's scientific community, industry, and commerce. The Institute consists of a Center for Radiation Research, an Office of Measurement Services and the following divisions:

Applied Mathematics — Electricity — Mechanics — Heat — Optical Physics — Nuclear Sciences<sup>2</sup> — Applied Radiation<sup>2</sup> — Quantum Electronics<sup>3</sup> — Electromagnetics<sup>3</sup> — Time and Frequency<sup>3</sup> — Laboratory Astrophysics<sup>3</sup> — Cryogenics<sup>3</sup>.

**THE INSTITUTE FOR MATERIALS RESEARCH** conducts materials research leading to improved methods of measurement, standards, and data on the properties of well-characterized materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; and develops, produces, and distributes standard reference materials. The Institute consists of the Office of Standard Reference Materials and the following divisions:

Analytical Chemistry — Polymers — Metallurgy — Inorganic Materials — Reactor Radiation — Physical Chemistry.

**THE INSTITUTE FOR APPLIED TECHNOLOGY** provides technical services to promote the use of available technology and to facilitate technological innovation in industry and Government; cooperates with public and private organizations leading to the development of technological standards (including mandatory safety standards), codes and methods of test; and provides technical advice and services to Government agencies upon request. The Institute consists of a Center for Building Technology and the following divisions and offices:

Engineering and Product Standards — Weights and Measures — Invention and Innovation — Product Evaluation Technology — Electronic Technology — Technical Analysis — Measurement Engineering — Structures, Materials, and Life Safety<sup>4</sup> — Building Environment<sup>4</sup> — Technical Evaluation and Application<sup>4</sup> — Fire Technology.

**THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY** conducts research and provides technical services designed to aid Government agencies in improving cost effectiveness in the conduct of their programs through the selection, acquisition, and effective utilization of automatic data processing equipment; and serves as the principal focus within the executive branch for the development of Federal standards for automatic data processing equipment, techniques, and computer languages. The Institute consists of the following divisions:

Computer Services — Systems and Software — Computer Systems Engineering — Information Technology.

**THE OFFICE FOR INFORMATION PROGRAMS** promotes optimum dissemination and accessibility of scientific information generated within NBS and other agencies of the Federal Government; promotes the development of the National Standard Reference Data System and a system of information analysis centers dealing with the broader aspects of the National Measurement System; provides appropriate services to ensure that the NBS staff has optimum accessibility to the scientific information of the world. The Office consists of the following organizational units:

Office of Standard Reference Data — Office of Information Activities — Office of Technical Publications — Library — Office of International Relations.

<sup>1</sup> Headquarters and Laboratories at Gaithersburg, Maryland, unless otherwise noted; mailing address Washington, D.C. 20234.

<sup>2</sup> Part of the Center for Radiation Research.

<sup>3</sup> Located at Boulder, Colorado 80302.

<sup>4</sup> Part of the Center for Building Technology.

acc. -  
100  
5753  
842  
74

Ref.

# Concepts in Quality Software Design

---

S. L. Stewart, Editor

Systems and Software Division  
Institute for Computer Sciences and Technology  
U.S. National Bureau of Standards  
Washington, D.C. 20234

From Seminars at the  
National Bureau of Standards in 1972

t. Technical note no. 842

Conducted by

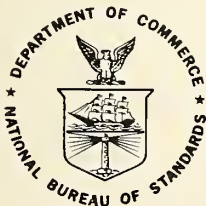
S. Rao Kosaraju

Johns Hopkins University  
Baltimore, Md. 21218

and

Henry F. Ledgard

University of Massachusetts  
Amherst, Mass. 01810



---

U.S. DEPARTMENT OF COMMERCE, Frederick B. Dent, Secretary  
NATIONAL BUREAU OF STANDARDS, Richard W. Roberts, Director

Issued August 1974

**Library of Congress Catalog Card Number: 74-600130**

**National Bureau of Standards Technical Note 842**

Nat. Bur. Stand. (U.S.), Tech. Note 842, 89 pages (Aug. 1974)

CODEN: NBTNAE

U.S. GOVERNMENT PRINTING OFFICE  
WASHINGTON: 1974

---

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402  
(Order by SD Catalog No. C13.46:842). Price \$1.25

## FOREWORD

In the summer of 1972, three activities came together in a fortuitous combination. The Institute for Computer Sciences and Technology at the National Bureau of Standards, which was already active in the broad field of computer utilization, organized a specific project on the subject of software quality. Several staff members had long-standing interests in this topic which they had been pursuing as part of the exploratory development program or in support of other projects. Summertime also made available two fine computer scientists from the academic world: Prof. Rao Kosaraju, from Johns Hopkins, and Prof. Henry Ledgard, now at the University of Massachusetts.

To start the new project in software quality, a series of seminars was organized with a designated speaker and topic for each meeting, but the structure was quite informal with free discussion. Five of the seminars are presented here, consisting of those given by Prof. Kosaraju and Prof. Ledgard. These are not transcripts, but rather they are re-worked versions based on recordings and notes. The authors and editors have attempted to preserve some of the informality of the oral presentation without its redundancy. I hope these written versions will serve the reader as a source of ideas as the original seminars served the participants.

These seminars discussed three broad questions: (1) how to design the solution of a programming problem? (2) how to program that design? and (3) how to organize programmers to do the above when the problem is too big for one designer/programmer? Definitive answers to all these questions are still in the future of quality software research but several good answers have been proposed in recent years.

Chapter 1 gives background and overview material and puts the problem of software quality in perspective. Chapter 2 looks at the top-down approach to problem solving in the specific context of developing algorithms. The idea is the familiar one of breaking down a large problem into a combination of smaller and simpler problems. This process is repeated until the problems are solvable by known methods. It sounds easy, but effective criteria for decomposing the larger problems are not easily stated, and the technique remains more of a craft than a part of software engineering.

Top-down design can lead naturally to structured programs, and this relationship has led many people to use the terms interchangeably. Chapter 3 covers structured programs in detail. Together with Chapter 2 both subjects are discussed in detail, but an informal definition at this point may help. Structured programming is programming with certain restrictions on the flow of control from one portion of the code to another. Exact definition of the restrictions varies from one programmer to another, but in general, execution of each block of code must be started at the beginning and terminated at the end, i.e. there should be no branching out of (or into) the middle of a block of code except to execute a block of closed code (subroutine). The common programming constructs of decision



making and looping can be handled by special blocks designed for these cases. The general constraint on branching out of (or into) the middle of blocks leads to the nickname "GOTO-less" programming.

With top-down design, it is natural for each subproblem to result in a block of code in the structured programming sense. This is possible, if the branching and looping block types are sufficiently sophisticated to relate the solution of subproblems to the decomposition of the original problem. The experiences of Dijkstra [1972], Wulf [1972], and Mills [1972] show that structured programming is practical. Now the questions are: what are the limitations in application, and how can structured programming be employed in the software production cycle? Except for BLISS [Wulf, 1971] most work in structured programming has been done by having the programmer voluntarily restrict himself to looping and branching control blocks when they are available (ALGOL60, COBOL, PL/1, etc.) or carefully using "good" GOTO's. It is important to note that programs may be well structured without top-down design. Regardless of design methodology, if the programmer adheres to structure rules, the result will be a structured program.

In the interval since Dijkstra's famous letter [1968b] started the GOTO discussion, there has also been more thought on other sources of program interaction besides control structures. In particular, study is now turning toward the relation between scope of variables and block structure and the data structures at the interface between program modules. Development of formal notation has gone hand in hand with increased understanding of the programming process, from syntax and control structures to data structures and modularity. Chapter 4 ranges over some of the ideas in the latter areas that will be influential.

Chapter 5 deals with an important practical distinction between the analytic problem of proving programs correct, in a mathematical sense, and the synthetic problem of constructing a correct program. In the latter may be found a method for significantly improving the quality of production software.

All references are to a combined bibliography at the end. In the text, citation is by first author and year.

The two speakers not only made the seminar series a success but also labored long and hard as authors preparing this material for publication. I would also like to thank Dennis Fife, Elizabeth Fong, Richard Freemire, Tony Horowitz, Belkis Leong-Hong, Gordon Lyon, Kirk Rankin, Susan Reed, Rona Stillman, and Justin Walker for their work as speakers, editors, and reviewers. My special thanks go to the division secretarial staff for their continuing support.

S. L. Stewart

## ABSTRACT

A seminar series on quality software, sponsored by the Systems and Software Division, was held at the National Bureau of Standards during the summer of 1972. This Note includes five of these seminars in edited form. (I) A brief background provides motivation for studies in software quality. The authors mention some factors which influence software manufacture, and propose measures which might quantify concepts of "software quality". Several approaches to establishing program correctness receive attention. (II) Elements of top-down programming are sketched out and then examined in detail. An extended critique of another top-down experiment provides example material. (III) Powers of various structured control constructs are compared within a framework of weak and strong program equivalence. Results include a demonstration that Dijkstra's D-programs are strongly equivalent to programs built from functions and one-input/two-output predicates. (IV) After a review of Quine's notion of referential transparency, the author examines elements of good and bad programming practice. In addition, a table of programming proverbs provides guidance to a programmer, and should be especially useful to a novice. (V) Discussions on problem and program specification provide an introduction to a review of proof-of-correctness techniques. Then, noting some practical limitations on proving correctness, the author goes on to examine selected facets of program synthesis.

KEYWORDS: Control structures; GOTO-less programming; program validation; programming; proofs of correctness; referential transparency; software quality; structured programming; top-down programming.

## CONTENTS

	Page
Foreword . . . . .	iii
S. L. Stewart	
Chapter 1. Perspectives on Quality Software . . . . .	1
S. Rao Kosaraju and Henry F. Ledgard	
Chapter 2. The Case for Top-Down Programming . . . . .	18
Henry F. Ledgard	
Chapter 3. Structured Programs . . . . .	33
S. Rao Kosaraju	
Chapter 4. Towards a Formalization for Quality Software . . . .	46
Henry F. Ledgard	
Chapter 5. Correctness of Programs - Writing Correct Programs .	66
S. Rao Kosaraju	
Bibliography . . . . .	80



# PERSPECTIVES ON QUALITY SOFTWARE

S. Rao Kosaraju and Henry F. Ledgard

*A brief background provides motivation for studies in software quality. The authors mention some factors which influence software manufacture, and propose measures which might quantify concepts of "software quality". Several approaches to establishing program correctness receive attention. Discussion ends with some observations on software production management.*

## 1. INTRODUCTION

In the past five years there has emerged a growing concern over the quality of computer programs (software). While difficulties associated with software have accompanied digital computers since their advent, only recently have software design and manufacture begun to be studied intensively.

There have been very few documented cases of software development costs. One interesting historical note goes back to Charles Babbage. Over 100 years ago the Royal Society underestimated costs in building the difference engine by a factor of 50. While the difference engine was never completed, it seems that ever since computing began, we have been plagued by embarrassments of development costs of computing systems.

Some have claimed that manufacturers might pay a million dollars or more to prove that some programs are correct. This statement is easily accepted if we consider something like a large operating system. A certain, well known operating system has been estimated to have cost \$40 M and to have consumed about 5,000 man-years. Average cost per man seems low, but it must include secretaries and the help of people who are not included in the total. In any case, its testing consumed \$20 M of the \$40 M.

Critical operating system features such as ease of maintenance, extendability, human engineering, structure, and documentation have not been so well funded. In spite of the documented need for awareness and action in software engineering, see for example the NATO report [Buxton, 1970], major projects are still undertaken with little attention to these factors.

Other indications of project investments are available. Paul Peach [1965] at System Development Corp. conducted a study of sample programming projects. After a survey of computer installations, he concluded that the average commitment to a production programming project was 77 man-months. This underscores the fact that production projects are major activities requiring a good deal of manpower.

## 1.1 Problems

As noted by Smith [1972], there are several severe problems that arise in the production and use of many software systems.

- (1) There are too many bugs.
- (2) Many systems are difficult to use. Furthermore, many system features are rejected or ignored by users as unwieldy.
- (3) Many systems are too complex to design and maintain. Lack of visible logical structure has caused systems to wither away.
- (4) There are too many system crashes. Catastrophic failures arise almost daily in some systems.
- (5) There is a failure to meet costs and schedules.

In short, these are some of the rather distressing issues that lead us to be concerned with the area of quality software. Hopefully, the next ten years will see drastic improvements in the area.

What are the reasons for the above mentioned problems? For one, there is a tendency by many programmers to produce elegant, sophisticated, or clever code. Few would dare modify the hand crafted code of most programming systems. Second, there is a lack of quality control. Imagine the reception accorded to a quality control analyst among a group of programmers. Could he, as in an electronics plant where modules are periodically inspected, sample programs, read them, and submit a quality review? Programmers would be unlikely to welcome such an audit. Yet any neutral observer would conclude that some form of quality control is sorely needed. Third, we tend to staff projects too rapidly, emphasizing quantity of code rather than quality of the result. Fourth, formal guidelines or standards are unfortunately scarce. We could give many more reasons for our dilemma.

## 1.2 Factors in Quality Software

Figure 1 outlines roughly what we consider constitutes quality software. There seem to be two issues here -- measurement of what is quality and control over software production to insure that quality is maintained. Under measurement we would include:

- (1) Definition. How do we define a program? This includes specification of the problem and documentation of the finished product.
- (2) Clarity. There should be some measure of how clear a program is, i.e., how easy it is to read, understand and use.
- (3) Reliability. Under reliability we include correctness, testing for errors, and error tolerance -- the kind of errors one can tolerate in a large system.
- (4) Flexibility. Here we include:
  - (a) the changeability of systems, for example, ease of correcting bugs and an ability to replace an algorithm with a more efficient one.
  - (b) maintenance of systems to meet alternations in specifica-

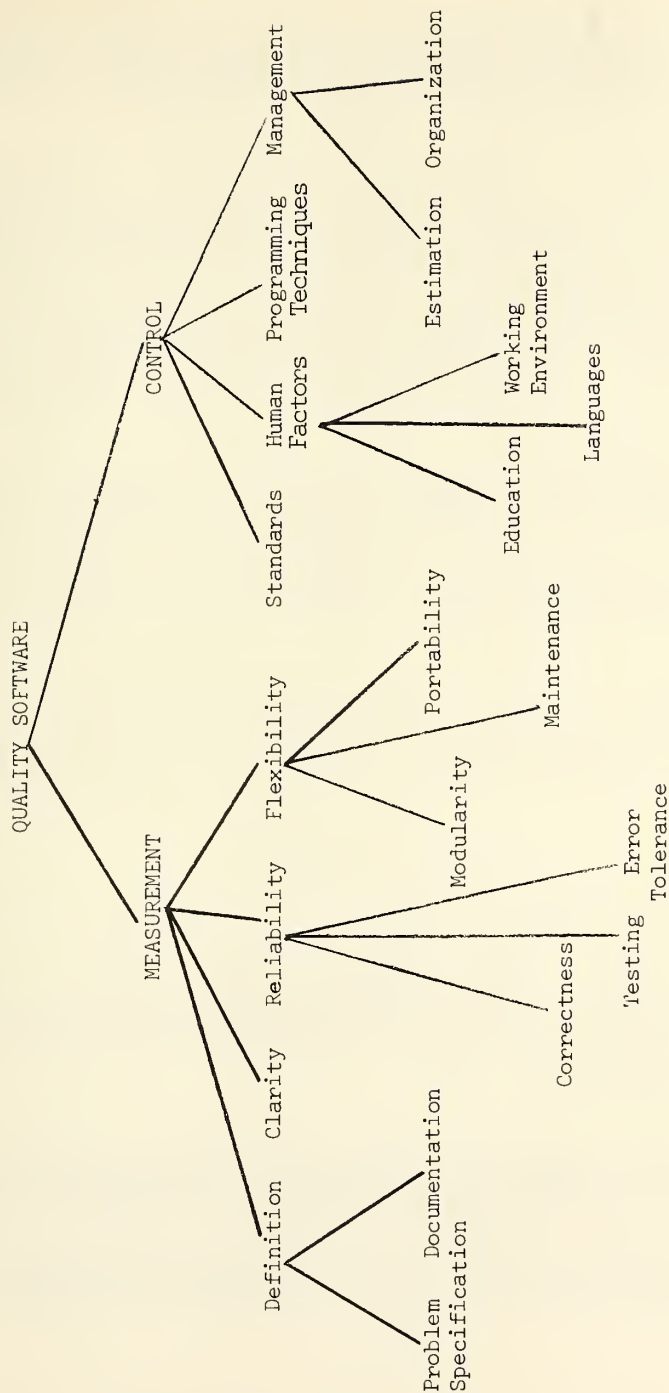


Figure 1: The Quality Software Tree

- tions, and
- (c) portability of systems required to move a program from one installation to another installation.

Under control we have:

- (1) Standards. Standards have had a difficult time in the computing field. Even efforts to make common the standard definition of FORTRAN have not been very successful. Most installations have neither a proper subset nor superset of standard FORTRAN.
- (2) Human Factors. How do we educate programmers so that they produce good software? What kind of languages do we give them so that it is easier to produce quality programs? What is the influence of the day-to-day things like coke machines or elevators on programmers? Weinberg is a leading advocate of this area.
- (3) Programming Techniques. What kind of programming techniques lead to quality programs? There has been some good work recently. We are thinking specifically of stepwise refinement programming by Wirth and excellent ideas proposed by Dijkstra.
- (4) Management. How do we estimate the difficulty, cost, and time of programming? How do we organize management for effective quality programming?

Broadly speaking, the above discussion indicates what we mean by "Quality Software". We will elaborate what has been done in each of the areas represented by nodes of the tree. For some of the nodes, our explanations will overlap; rather sadly, some nodes of the tree will be entirely neglected.

## 2. MEASUREMENT OF QUALITY

### 2.1 Definition

Parnas [1972] has been doing substantive work on how we define programming problems. Three areas of his work are relevant to our objectives today. First of all, he considers when we make certain design decisions. The biggest problem here is that we often make decisions without realizing that we have made them. This is a pervasive issue in day-to-day programming.

For example, consider designing a program to play checkers. A "board" often sneaks into our design without our realizing that there are alternative methods of programming checkers that have no direct analog to a board. For example, rather than a 64-element array denoting the board, one could consider two 12-element lists containing the locations of pieces owned by each player. Implicit decisions on the need for certain data structures often rule out good alternative algorithms.

Parnas is also a strong advocate of functional program specifications. That is, one should specify a program by listing its input-output characteristics and net effect, rather than by algorithmically describing its operation. It is not only confusing to describe programs algorithmically: it also obscures their functional properties.

## 2.2 Reliability

Recently there has been a great deal of interest in assessing the reliability of programs. Suppose we are given a function  $f$  and a program  $P$ . The reliability of  $P$  is a measure of how far it is from a program that realizes  $f$ , i.e., by how much we would have to change  $P$  to make it a program that exactly computes  $f$ . Reliability is thus closely related to the concept of "distance" between programs, i.e.

$$d(P_1, P_2) = \text{"closeness" of } P_1 \text{ and } P_2$$

Normalizing the distance function we should have  $0 \leq d(P_1, P_2) \leq 1$ .

Intuitively,  $d$  should at least satisfy the following conditions:

- (1)  $d(P_1, P_2) = 0$  iff  $P_1 = P_2$  or both  $P_1$  and  $P_2$  are syntactically correct and compute the same function.
- (2)  $d(P_1, P_2) \geq 0$
- (3)  $d(P_1, P_2) = d(P_2, P_1)$

Given any program  $P$  and any function  $f$ , we will denote the reliability of  $P$  in realizing  $f$  as:

$$\underline{\text{Rel}}(P, f) = 1 - \min_Q d(P, Q)$$

where  $Q$  exactly realizes  $f$ . We could obviously define an ordering relation based on the reliability of programs as follows:

$$\begin{aligned} P &\geq_f Q \quad (P \text{ is more reliable than } Q) \\ \text{iff } \underline{\text{Rel}}(P, f) &\geq \underline{\text{Rel}}(Q, f). \end{aligned}$$

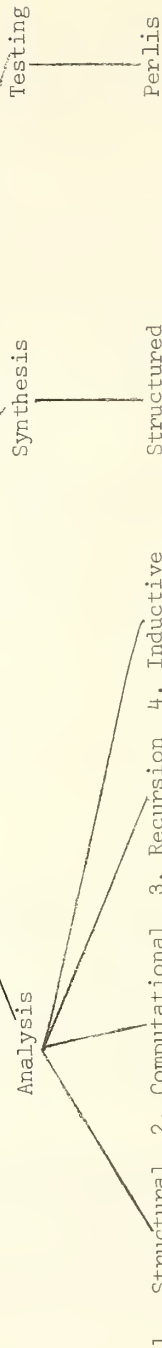
Hence  $P$  correctly realizes  $f$  if and only if  $P$  exactly computes the function  $f$ . We note that this is so iff  $\underline{\text{Rel}}(P, f) = 1$ .

## 2.3 Correctness of Programs

Most definitive results in quality software are in methods for establishing program correctness, depicted in Figure 2. There are three sometimes broadly disagreeing schools of thought.

- (1) Analysis Methods. Given the program  $P$  and a problem  $f$ , prove that  $P$  computes  $f$ .

# METHODS FOR CORRECTNESS OF PROGRAMS



- 1. Structural
  - Induction
    - McCarthy
    - Burstall
- 2. Computational
  - Induction
    - Gorn
- 3. Recursion
  - Induction
    - McCarthy
- 4. Inductive
  - Assertions
    - (Interpretation of Flow-charts)
    - Floyd
    - Naur
    - Manna
    - King
    - London
    - Hoare
    - (Goldstine & Von Neumann)
- Structured
  - Approach
    - (Dijkstra, Mills)
- Testing
  - Perlis

Figure 2: Methods for Establishing Program Correctness



- (2) Synthesis Methods. Given only the problem  $f$ , construct a program  $P$  to compute  $f$ .
- (3) Testing Methods. Given  $P$  and  $f$ , find a suitable number of inputs to test that  $P$  computes  $f$ .

Basically, analysis advocates settle for nothing less than formally proving the correctness of a given program. Testing advocates feel that a judicious selection of test cases gives the necessary amount of confidence in a program. We believe that the most promising approach lies in synthesis methods. This school says that we should revise our method of programming in order to guarantee, by the structure of a program, that the program is correct. Presumably, we could amplify this with some analysis or testing.

## 2.4 Analysis Methods

In analysis methods, we assume that the problem is defined formally. This is generally done in either of two ways:

1. If  $f$  is given as a program  $P_1$ , then the correctness proof of a program  $P_2$  reduces to the equivalence problem for programs. That is,  $P_2$  is a correct representation of  $P_1$  iff  $P_2$  is equivalent to  $P_1$ .
2. If  $f$  is defined by its input/output response, then  $P$  can be considered correct if it defines the same mathematical function from the set of inputs to the set of outputs.

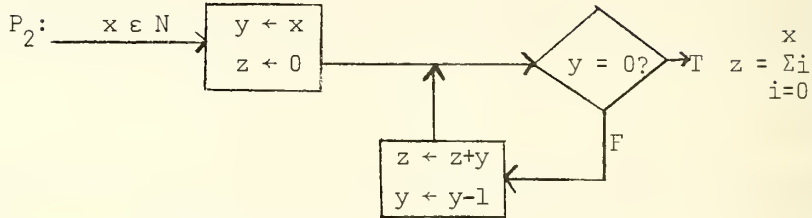
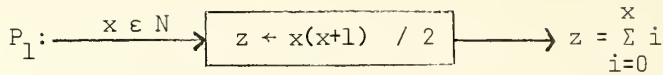
It is not our intention to present all the technical details of analysis methods. Instead we will give a general idea of the essential features of each.

### 2.4.1 Structural Induction

This type of proof of correctness is a generalization of the normal inductive proofs encountered in number theory. For example, let  $P_1$  and  $P_2$  be any two programs mapping  $x$  into  $\sum_{i=0}^x i$  where  $i, x \in \mathbb{N}$ . In general the input domain may not be as structured as the set of natural numbers, but if the structure of the input domain is specified inductively, then we can establish equivalence of  $P_1$  and  $P_2$  by proving their equivalence for:

- (1) all the basic elements and
- (2) all the inductive constructs.

As an exercise consider the following programs:



Here we must prove that:

$$P_1(0) = P_2(0)$$

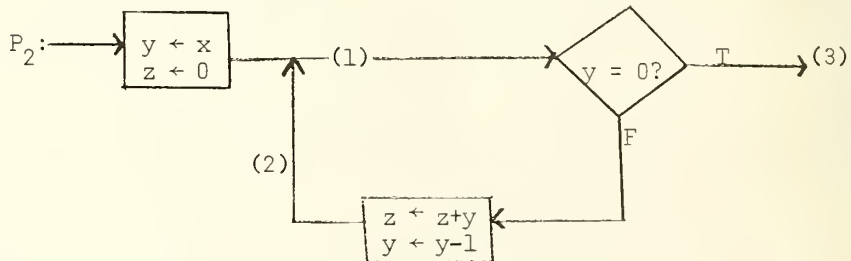
$$P_1(k) = P_2(k) \rightarrow P_1(k+1) = P_2(k+1)$$

The equality of  $P_1$  and  $P_2$  on the input domain  $\mathbb{N}$  then follows.

#### 2.4.2 Computational Induction

Computational induction differs from structural induction in that the induction is made over time (number of steps) rather than input structure. The theorem exploited by this technique is the following: If a property initially holds and if this property is left invariant by each step of the computation, then the property holds at every step of the computation. Thus if the program terminates, then the property holds at the end of the program.

Let us consider the same  $P_2$  as in the previous example



We claim that the property  $q$  given by

$$z = \sum_{i=y+1}^x i \quad (\text{where} \quad \sum_{i=x+1}^x i = 0)$$

is left invariant at each of the points 1, 2, and 3. At the program exit,  $y = 0$ , so  $z = \sum_{i=1}^x i$ . This idea was first applied to programs by Gorn [1968].

#### 2.4.3 Recursion Induction

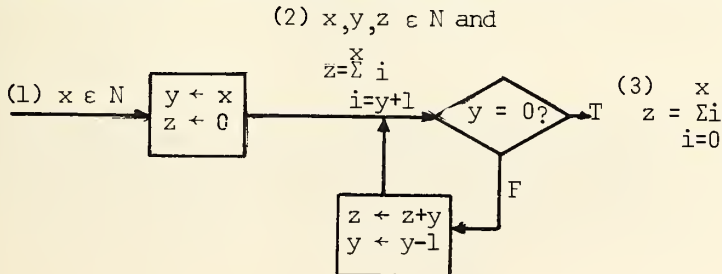
In this method we are interested in showing that programs  $P_1$  and  $P_2$  compute identical functions. We look for a super problem  $F$  that has a set of solutions  $(Q_1, Q_2, \dots)$ , where any two solutions have the property that if both of them terminate for a given input then they have the same output value. I.e., if  $Q_i(x)$  and  $Q_j(x)$  terminate then  $Q_i(x) = Q_j(x)$ , and if  $P_1$  and  $P_2$  are both solutions of  $F$ , then  $P_1$  and  $P_2$  are equivalent on domain  $X$ .

The skill is in choosing a suitable  $F$ . Usually the fixed points of  $F$  are taken as the solutions of  $F$ ; in this method you generalize the problem and attack the general problem.

#### 2.4.4 Inductive Assertions

In this method we assert several conditions that should hold at various points in the program. We must show that if any such point is reached during the computation, then the corresponding condition holds. In particular if the input and the output points are two such points, then we claim that for any input satisfying the input assertion, the output assertion holds if the program terminates.

Using our previous example, we can apply the method to prove the correctness of the following. Here we have three assertions tagged to points (1), (2), and (3) of the flow chart.



This method was introduced by Floyd and Naur independently, and almost at the same time. Manna formalized their ideas and further obtained a condition (which can be justified on intuitive grounds, independent of his formalism) for program termination.

There are various disadvantages to analysis methods:

1. Formally, we are trying to solve an undecidable problem. So the techniques cannot be completely mechanized.
2. The methods are too involved for large programs.
3. These methods are almost impossible to extend to parallel processing schemes: e.g., using inductive assertions the number of assertions to be verified can grow exponentially compared to sequential schemes.

But there is one important psychological advantage. Being aware of the above techniques may help us in writing correct programs, even though we might not want to bother with formal proofs.

## 2.5 Synthesis Methods

The next important class is the synthesis methods. In these methods, we do not write just any program. Instead, we try to develop a correct program using only well understood and simple structures at each stage. Thus the cardinal principle in these methods is to structure the program as simply as possible, going to a complex structure only if it is absolutely necessary.

This is not an entirely new idea, but the first one to stress and to make use of this concept was Dijkstra [1968]. He claims that he used this philosophy in the design of THE system. Three major projects have had designs motivated by synthesis methods.

1. Harlan Mills - N. Y. Times Information retrieval system.
2. Niklaus Wirth - PASCAL
3. William Wulf - BLISS

In the N.Y. Times problem the designer was restricted to simple control structures. PASCAL and BLISS are based on another school of thought, a restriction within the language itself such that the programmer is constrained to pick simple structures.

The following views, expressed at the NATO conference [Buxton, 1970], are illuminating:

Wirth: "In fact, the idea of program verification has influenced the design of PASCAL considerably. Several commonly used features of other languages have been deliberately omitted from PASCAL, because they appear to be too sophisticated for presently known methods of proof."

Lang: "You describe some intellectual tools which you say have increased your own programming efficiency about five times.  
 (to Dijkstra) Have you been able to transmit these to your colleagues with a similar effect?"  
 Dijkstra: "Yes. In my experience, the intellectually degrading influence of some educational processes is a serious bar to clear thinking. It is a requirement for my applicants that they have no knowledge of FORTRAN."  
 Bauer: "...one can't help mentioning PL/1 here."

From the above comments, it is obvious that Wirth and Dijkstra strongly believe in the synthesis approach.

## 2.6 Testing Methods

Some people feel that the analysis methods are too involved and instead we should resort to testing. The following remark by Perlis [Buxton, 1970] illustrates this philosophy:

"The problem is to isolate the right test cases, not to prove the algorithm, for that follows after the choice of the proper test cases."

- Q. Does Perlis support his assumption that for any program he can indeed isolate the proper test cases or does he just assert this?
- A. At least in the conference records he does not give any examples. But one of the examples we give later on in a different context might illustrate how test cases could be selected.
- Q. In the theory of computing, doesn't the principle of structural induction go back to the middle 30's?
- A. It might be even earlier than that. But within the context of programs, it originated with McCarthy. The general idea is as old as mathematics.
- Q. Do Mills, Dijkstra, and Wirth actually prove that by eliminating complex structures the proof of program correctness becomes easier?
- A. They are more interested in convincing arguments for the correctness of a program rather than a formal proof. The control structures do not automatically generate a correct program. After all, it is well known that any program can be converted into an equivalent program which uses only the simple constructs. But when program development is geared to the synthetic approach, it appears easier to write correct programs (or at least more reliable ones).

## 2.7 Program Testing and Error Tolerance

There are three broad classes of program errors, Syntactic, Semantic and Simple Semantic (border-line) bugs.

Syntactic bugs are the bugs detected when the program is considered a syntactic string of the language. Some examples are:



1. Syntax errors within single statements, for example typing errors, and
2. Syntax errors across several statements, for example missing labels or END brackets for blocks.

Mathematically, correction of syntactic bugs consists of finding the "nearest" syntactically good program. It amounts to: Given any program  $P$ , find a  $P_1$  such that  $P_1$  is syntactically correct and

$$d(P, P_1) = \min_Q d(P, Q), \text{ where } Q \text{ is syntactically correct [Lyon, 1972].}$$

Semantic bugs are errors in the execution phase. They are the hardest to correct. Mathematically, semantic bugs arise in two ways. Assume that a program  $P$  is intended to compute a function  $f$ .

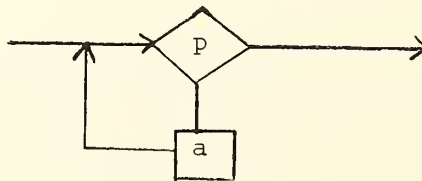
1.  $P$  does not halt on a valid input  $x$
2.  $P$  halts on  $x$  and  $P(x) \neq f(x)$ , i.e.,  $P$  gives incorrect output.

Suppose we want to test a program which is claimed to compute  $\sqrt{x}$ ,  $x \in \mathbb{N}$ . We do not have any idea about the program; that is, we are not allowed to look at the source code. What types of errors could occur? If there are errors, how could we find them? Since we cannot look at the program, we have to consider it as a system where only inputs and outputs are accessible.

- (a) Initial value checks: Say check for 0, 1, 2, 3
- (b) Select a few large  $x$ 's of the form  $n^2$
- (c) Ceiling operation error - Select a few large  $x$ 's of the form  $n^2 - 1$ , or  $n^2 + 1$
- (d) Others: Select a large  $n$  and a few  $x$ 's in between  $n^2$  and  $(n + 1)^2$

By making all these tests we will have a high degree of confidence about the program. But by no means are the tests a conclusive proof of correctness. After all, for any selected set of tests, we could write a program which is incorrect, but correct on the selected set.

In the case where we know the program structure we can select the test cases dependent on the program. In some instances it is effective for tracing nontermination. For example, in





we can select a few  $x$ 's satisfying  $p$  and a few  $x$ 's not satisfying  $p$ .

In the more general case, we make use of both problem and program in selecting test samples. While there is no known theory to handle bugs, inductive assertions methods described previously might prove to be helpful in selecting critical test samples.

Simple semantic bugs are errors occurring during execution that could be classed as domain problems. For example, an array  $A$  may have been defined from 1 to  $n$ , and during computation we might access  $A(I)$  where  $I$  has the value 0 or  $n+1$ . Examples in this class might include the following:

1. Usage of un-initialized variables
2. Illegal branches into DO groups
3. Array bounds
4. Misuse of pointer references
  - (a) to I/O buffers no longer available
  - (b) to data with attributes that differ from those of the name specified.
5. Misuse of storage allocations
  - (a) attempts to retrieve a value from storage that has previously been freed
  - (b) attempts to free storage that has already been freed.

## 2.8 Flexibility

The next concept that seems relevant to quality software is flexibility, under which we include modularity, maintenance, and portability. We could conceivably measure the modularity  $M(P)$  of a program  $P$  by the number of subprograms in it. We could measure modularity of a programming language  $L$  by a function  $M_L$  which can be defined. For example, consider two possible definitions:

- (1)  $M_L(n) = \text{Min } [M(P) \mid P \text{ is any } n\text{-instruction program in } L]$
- (2)  $M_L(n) = \text{Expectation of } M(P), \text{ where } P \text{ is any } n\text{-instruction program in } L$

To find the value of the modularity of a language at  $n$ , find the modularities of all  $n$  instruction programs in the language and take either the minimum or some weighted average. If we accept the second definition, then Dijkstra's class of structured programs turns out to be more modular than the general class of programs.

Maintenance is a very large issue in today's systems, since we must frequently alter a working system to fit the needs of its users. Different users often require changes in the specifications to adapt the system to their needs. Almost no work exists in this area.

Portability measures the ease of transferring a program from one environment to another. Perlis [Buxton, 1970] defines portability as the property of a system which permits it to be mapped from one environment to a different environment.

A cardinal principle in preserving portability is "never use implementation dependent features." Suppose something is undefined in a programming language. In any implementation it has to be defined in some way. But there is no guarantee that it will be defined consistently in all implementations.

There are some interesting comments in the NATO conference [Buxton, 1970] on this topic:

Feldman: "If you are using a system very hard and pushing it to its limits, you cannot expect that system to be portable."

Buxton: "The business of a manufacturer is to get his clients and then to keep them...Portable software would be a disaster because they might be able to take useful application programs elsewhere."

### 3. CONTROL OVER QUALITY SOFTWARE PRODUCTION

We now discuss the other side of quality software, the control over software projects to insure quality. Currently, programming is very much an individual activity. The personality of a programmer may become injected into his code. This personalization is highly undesirable when the code is part of a publicly used product. Moreover, programming has become somewhat of a "cult" in the sense of an esoteric occupation rampant with local conventions and an over-concern with detail. Sometimes it is difficult to see the broader picture. To correct this problem Mills contends that programming should become a higher intellectual activity than at present, and for example, we should segregate "coding" from "programming". In any case, control over the quality of software will be difficult.

#### 3.1 Human Factors

One of the most ignored aspects in programming is the importance of human factors, an issue amplified in Weinberg's book, *The Psychology of Computer Programming* [1971]. This major thrust is to study computer programming as a human activity. Basically, Weinberg's book is an expose of entirely under-estimated human factors in programming. There is a very well-documented set of examples. Some of the features he talks about are:

- (a) The value of reading programs
- (b) Ironical situations that develop in day-to-day working environments at particular installations
- (c) The questionable value of efficiency in programs, namely that it is over-played in developing programs.

- (d) The appeal for practical experiments to study human factors in programming.
- (e) "Egoless programming," i.e., the value of divorcing one's personality from programs.

Even factors like the size of a chair, the Hawthorne effect, or distance to the nearest coke machine are factors in day-to-day programming. Weinberg's book contains numerous examples of the influence of these factors over the success or failure of programming prospects. We cannot help but mention that this book is a superb motivation for the need for formal measures of quality software.

### 3.2 Language Developments

The book by Sammet [1969] contains almost no mention of quality software as being an important design factor in any language. The two recent major language developments that are notable for their concern over quality software are BLISS [Wulf, 1971] and PASCAL [Wirth, 1971b]. BLISS is a language designed for systems programming. It is designed around a machine, which is catastrophic from our point of view. It is highly expression oriented. Expressions seem to be things that programmers can handle well, whereas command features of a language give problems. BLISS has a restricted control structure. The most notable feature of BLISS is the facility for data structures. We will not go into it here, except to note that BLISS has few built-in structures and it is easy to change the representation of a structure without altering operations over the structure.

PASCAL is an attempt to combine:

- (1) top-down programming - hence it allows certain extension mechanisms allowing a user to define new structures and new forms of expressions that he is designing top-down.
- (2) restricted control structures - almost like Dijkstra's constructs IF-THEN-ELSE and DO-WHILE.

There has been no report on the success of this language.

### 3.3 Methods of Programming

We tried to look at methods of programming and programming techniques. While there have been many casual attempts to define programming in simple terms, for example,

- 1. Analysis
- 2. Flow charting
- 3. Coding

4. Debugging
5. Production
6. Documentation

truly viable methodologies for programming are just beginning to emerge.

Wirth [1971a] considers programming "a methodology of constructive reasoning applicable to any problem capable of algorithmic solution." Wirth has introduced the notion of "stepwise refinement." He has elaborated by examples and documentation (critical in this case) a notion of programming as a construction of a branching tree, going from an informal description in successive levels of refinement until the desired code is produced. He emphasizes use of "natural" notation in designing programs and advocates clear isolation of when certain decisions about programming are made.

Many consider Dijkstra as the father of the field of quality software. His efforts are devoted mainly to programming techniques, as are Wirth's. Among other things, he made a strong case for designing large programs in some systematic manner. Dijkstra brought the idea of top-down programming into prominence. He is a proponent of correctness via structure rather than exhaustive testing, and is noted for the quote: "Program testing can be used to show the presence of bugs, but never to show their absence."

Dijkstra was the first to exhibit a simultaneous demonstration of top-down programming, structure, and correctness. Along with suggestions for some higher-order theory for program correctness, he has elaborate discussions with examples and insights on everything he writes about. His work is highly technical and profound in comparison with the work of many others.

#### 3.4 Management Techniques

One of the documented successful software management efforts is the chief programmer team experiment by Baker and Mills at IBM [Baker, 1972]. The experiment was to construct a large (83,000 lines) information retrieval system for the N. Y. Times. The notable management personnel are:

- (1) Chief Programmer - a senior person (sometimes mentioned as a super-programmer) who is the nucleus of the programming project. He specifies and integrates everything about the system.
- (2) Back-up Programmer and Other Team Members - the chief programmer will have reports from a back-up person, a program production librarian (who keeps track of everything written and everything that exists in the library) and consultants.

In a chief programmer team, there is a clear separation between clerical work and programming. The novel organization factors are:

- (a) it is functional in nature
- (b) it has an extensive library and a librarian
- (c) it uses exclusively top-down programming, including Dijkstra's notion of structured programming.

As reported by Baker [1972], the result of this experiment was a well-documented, nearly error-free product, receiving wide acceptance by its users. An independent analysis of this project has not been recorded, but we believe it to be significant.

#### 4. CONCLUSION

The notion of "programming" as evolved in the last five years is a new, emerging methodology. A large number of important computer scientists have devoted their energies to the area we call "quality software". We believe that the next ten years will bring significant changes to an area that has long been considered of secondary importance.



Henry F. Ledgard

*Elements of top-down programming are sketched out and then examined in detail. An extended critique of another top-down experiment provides example material. Summary remarks and a discussion transcript end the paper.*

## 1. INTRODUCTION

I would like to discuss in detail a technique now known as top-down programming. Let me first state what I believe is the pure notion of the top-down approach. From the start, one writes code in the source language at hand. The first written piece of code is the top-most level, e.g. the calling procedure. The sub-procedures are then written, these are later split up, and so on until the entire program is coded. One may assume dummy procedures for those sections that are left unspecified; later these will be spelled out.

This definition of top-down is the mirror image of the "bottom-up" method, where you usually write the lower procedures first and later write the upper levels. The two definitions then, are symmetrical. Now, when I use the word top-down, I am going to mean something different from that, something that is more like successive refinement.

What is called successive refinement is a method of programming where again one starts at the top level, but one is not constrained to work in a particular programming language. One generally starts with an improvised language, which in some sense is mechanical to the user. One might use statements like "compute the n-th prime number", "find the roots of the equation" or "process the payroll". This is the method advocated mainly by Dijkstra and Wirth. With successive refinement we don't arrive at our target language until we are near the bottom level, and the program is almost in front of our eyes.

As an aside, here is a statement from a paper that appeared in 1968. The statement is originally from "Computer-Aided Design: "A Statement of Objectives", by Douglas T. Ross, et. al, 1960 [quoted from Tou, 1970].

"The manner of stating problems is also of primary importance. You must be able to state problems outside in, not inside out. Under the new philosophy, successive stages of problem statements are greater and greater refinements of the original statement of the problem."

I am not sure whether the author fully understood the impact of this statement for programming. He meant it about design processes. However, I do not think the jump from design processes to programming is quite natural.



## 2. TOP-DOWN PROGRAMMING

For the purposes here, I shall give my own definition of the top-down approach. I use the words "top-down" only because the words "stepwise refinement", though wisely chosen, are too long. Long words just never sail. "Top-down programming" is a process for writing computer programs that has at least the following properties:

1. The programmer initially defines the overall structure of his program in the English language. From statements in English that are machine and language independent, the programmer moves towards a final machine implementation in the target programming language.
2. The programmer initially uses expressions that are relevant to the problem, even though the expressions cannot be directly transliterated into the target language.
3. The programmer works in levels. He concentrates on critical, broad issues at the initial levels, and postpones details (e.g., input/output, choice of identifiers, or data representation) until lower levels.
4. At each level, the programmer considers alternative ways to refine some parts of the previous level. He may look a level or two ahead to determine the best way to proceed. After making his decision, he writes down the next level in the general form of a "program".
5. After completing step 4, the programmer must rewrite the program as a correct formal statement. This step is critically important. He must insure that all arguments to unwritten procedures or sections of code are explicit and correct, so that further sections of the program can be written independently without later changing the specifications or the interfaces between modules.
6. Steps 4 and 5 are repeated as often as necessary to move from the general statement of the problem in the English language to the completed program in the target language.

Initial statement of the problem

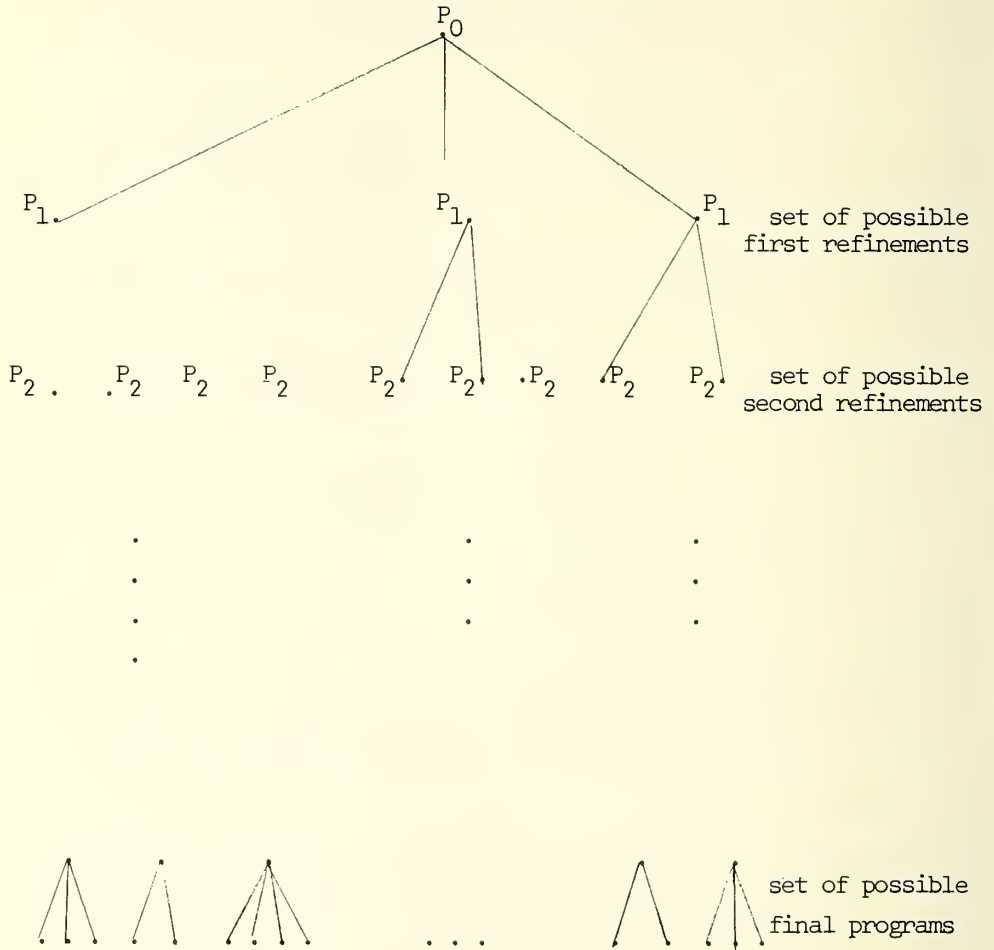


Figure 1: Branching structure of the top-down programming process.

In top-down programming we usually start off by writing what I am going to call  $P_0$ , an outermost description of the program, see Figure 1.

$P_0$  will be something like:

- Play checkers
- Compute the first N prime numbers
- Compute the Chebychev approximation of a function

As we proceed down the tree from  $P_0$ , we have many design choices. The  $P_1$  level will be the set of possible refinements of  $P_0$ , and it is important that we elect a wise choice based on the broad characteristics of the problem. The  $P_2$  level gives us another set of choices, and so forth for each successive level. At the bottom level we have a set of programs which all perform the same task. From among this set we will have selected some particular program by our route down the tree. That will be the final program.

We must make one important point here. Suppose we have written some level of our program,  $P_n$ , and we are about to choose a suitable  $P_{n+1}$ . One of my major points here is that we usually cannot go directly from  $P_n$  to  $P_{n+1}$ . This is a bit like expecting a burst of divine intervention. Trying to examine my own approach to this process, I usually select some tentative  $P_{n+1}$  that at the time is very vague, too vague to be called a refinement. I then find myself experimenting at lower levels, trying to get a feel for the consequences. Furthermore, I find myself reaching down the tree much further than I really want to go, just to see how the tentative choice will come out. Finally, when I have satisfied myself that it all makes sense, I return to  $P_n$ , and select a specific refinement  $P_{n+1}$ . This lookahead helps insure a wiser choice among the particular alternatives at  $P_{n+1}$ . So I claim, there is much more looking ahead in the top-down approach than is apparent in the literature. With the exception of Dijkstra, few authors recognize this point.

One final point. After you have selected a level, you should stop and explicitly formalize the interfaces. This is very different from strict stepwise refinement, where the formalization does not really occur until you hit the target language. My claim is that you should explicitly define each sub-section of code up to input-output, and debug the "program" at this level. More on this will be said later.

### 3. AN EXTENDED EXAMPLE

I wish next to discuss a paper that appeared in B. I. T. [Henderson, 1972]. While I think it is a good paper, I take serious issue with its basic point. The paper is called "An Experiment in Structured Programming" and was written by Henderson and Snowdon. Briefly, a program was written to process a stream of telegrams. The number of words in each telegram was counted and telegrams were printed with appropriate spacing on an output medium. An experiment was then conducted in which the program was debugged and a detailed analysis of its structure was made. Henderson was the programmer and Snowdon conducted the experiment.

If I read the paper correctly, the program was used in many lectures before it was tested. Lo and behold, a bug was later found. No one had caught the bug before. The claims made by the authors are the following: (1) there is an over simplification with the stepwise refinement method, (2) the method leads to a false sense of security, and (3) data structure decisions sneak in that are not really consciously made by the programmer. While I think these criticisms are good, I don't think they had anything to do with the error. I think the error was just a careless mistake. Had the programmer been careful, especially if he had formalized and debugged each of the levels, the mistake would not have arisen.

When I first read the paper, I was quite surprised that there was a mistake, but I didn't read the program carefully enough. When one looks at the proposed modules on first reading, one agrees that they are correct. Later on, the authors state that there is an error. This is really unfair, because in programming the problem, I don't think most of us would make the error. However, an error was made. And the programmer was using the method of stepwise refinement. That much is true. Therefore, we can't just ignore the error. We have to either explain it away or criticize the method. My claim is that while the error did occur it should have been avoided.

Let us look at the problem in some detail. We have an input stream that consists of telegrams each of which is followed by the sequence "ZZZZ". The stream is terminated by a blank telegram followed by "ZZZZ". There exists a buffer, into which we read portions of the telegram stream, and from which we can pick off characters. If this buffer becomes empty during execution, we must refill it. The telegrams are reprinted on the output medium in lines of 100 or more characters. Presumably, the lines can hold at most 120 characters, and we only expect words with fewer than 20 characters. Finally, extra blanks in the telegram stream are to be deleted on output, and the word count and overcharge message (if necessary) are printed after each telegram. An overcharge occurs in a telegram containing one or more words of length greater than 12 characters.

$P_0$

process the telegrams

$P_1$

PROCESS (stream)

$P_2$

set initial conditions of program

A:     if not at end of telegram then process current word;  
  update conditions;  
  repeat (A);

if end of telegram but not at end of stream;  
          then process end of telegram;  
                  reset initial conditions;  
                  repeat (A);

if end of telegram and end of stream  
          then exit;

Figure 2. Initial refinements of the telegram problem.

The assumed primitives for the program are not clearly stated in the paper, although I think they should be. We shall state them here:

refill (BUFFER)

next-char (BUFFER)

print (LINE)

length (STRING)

Now we shall write the program. It is an easy program, but not one that can be written in 20 minutes. First, let's consider the error made by Henderson. In his program there was a procedure called ADJUST that removes blanks from the buffer until a non-blank is encountered or the buffer becomes empty. After ADJUST is invoked, the words are counted. The mistake here is that ADJUST is not re-invoked if the buffer becomes empty. As a result, if the buffer is refilled with a string containing initial blanks, the first blank is (erroneously) counted as a word. I am not suggesting that the error in Henderson's program was so easy to find. It was not. But the real question is whether an error like the one made is an intrinsic folly of the stepwise refinement programming process. I claim that it is not, and that formalizing and debugging each level is the key issue.

I am going to try to write a program to do the same computation. For my part of the experiment, I wrote the program after only a cursory reading of the paper. I spent about seven hours on the program, which is probably a lot for a small program like this, but I was keeping explicit track of my approach.

The first level  $P_0$  (see Figure 2) is just "process the telegrams". Here we immediately stop and formalize. It is significant to write that the refinement  $P_1$  should be PROCESS (stream). PROCESS is a procedure that takes the input stream as an argument and whose value is the side effect of producing the annotated and respaced stream on the output medium.

At the next level,  $P_2$ , I was most careful. Initially I was thinking of  $P_2$  something like:

$P_2$  (rejected)

- (1) initialize the program
- (2) get the next character
- (3) if the character is a blank then repeat (2)
- (4) if the character is Z, then count it,

etc.

In my thinking, I went down to  $P_3$  (see Figure 3). The character orientation produced somewhat of a tangle, and I decided to (See page 30)



P<sub>3</sub>

refill (BUFFER);  
CURRW ← Λ;

INITIALIZE (OVERCHARGE, WCOUNT, LINE, PREVW, CURRW);

- A.    if CURRW ≠ ZZZ then PRINT1 (LINE, CURRW);  
              UPDATE (OVERCHARGE, WCOUNT, PREVW, CURRW);  
              goto A;
- if CURRW = ZZZ and PREVW ≠ ZZZ  
              then PRINT2 (LINE, OVERCHARGE, WCOUNT);  
                  INITIALIZE (OVERCHARGE, WCOUNT, LINE, PREVW,  
                  CURRW); goto A;
- if CURRW = ZZZ and PREVW = ZZZ  
              then exit

P<sub>4</sub>

refill (BUFFER);  
CURRW ← Λ;

- B.    INITIALIZE (OVERCHARGE, WCOUNT, LINE, PREVW, CURRW);
- if CURRW ≠ ZZZ then PRINT1 (LINE, CURRW);  
              UPDATE (OVERCHARGE, WCOUNT, PREVW, CURRW);  
              goto A;
- if PREVW ≠ ZZZ then PRINT2 (LINE, OVERCHARGE, WCOUNT);  
              goto B;  
              else exit;

Figure 3. Intermediate refinements of telegram problem.

Procedures for  $P_5$

(1) INITIALIZE (OVERCHARGE, WCOUNT, LINE, PREVW, CURRW)

```
OVERCHARGE ← false ;  
WCOUNT     ← 0 ;  
LINE       ←  $\Lambda$  ;  
PREVW      ← CURRW ;  
CURRW      ← GET_NEXT_WORD ( )
```

---

(2) PRINT1 (LINE, CURRW)

```
LINE ← LINE || ' ' || CURRW ;  
  
if length(LINE) > 100 then print (LINE) ;  
                        LINE ←  $\Lambda$  ;
```

---

(3) UPDATE (OVERCHARGE, WCOUNT, PREVW, CURRW)

```
if length(CURRW) > 12 then OVERCHARGE = true ;  
  
WCOUNT ← WCOUNT + 1 ;  
PREVW  ← CURRW ;  
CURRW  ← GET_NEXT_WORD ( ) ;
```

---

(4) PRINT2 (LINE, OVERCHARGE, WCOUNT)

```
if LINE  $\neq$   $\Lambda$  then print (LINE) ;  
if OVERCHARGE then print ('OVERCHARGE') ;  
print ('NO WORDS IS', WCOUNT) ;
```

Figure 4: Refinements of unspecified procedures of  $P_4$ .

P<sub>6</sub>

```
Procedure GET_NEXT_WORD ( );  
  
WORD ← Λ;  
Find first non-blank character;  
read in characters up to next blank;  
return (WORD);
```

P'<sub>6</sub>

```
procedure GET_NEXT_WORD ( );  
  
WORD ← Λ;  
  
A: CHAR ← next-char (BUFFER);  
  if CHAR = Λ then refill (BUFFER);  
    goto A;  
  if CHAR = ' ' then goto A;  
  
  repeat WORD ← WORD || CHAR ;  
    CHAR ← next-char (BUFFER);  
  until CHAR = Λ or CHAR = ' '  
  
  return (WORD);  
  
end GET_NEXT_WORD
```

Figure 5: Refinements of the GET\_NEXT\_WORD Procedure.

```

begin /* Declaration of GET_NEXT_WORD */

  procedure GET_NEXT_WORD ( ); local WORD,CHAR; global BUFFER;

    WORD = Λ;

  A: CHAR ← next-char(BUFFER);
    if CHAR = Λ then refill(BUFFER); goto A;

    if CHAR = ' ' then goto A;

    repeat WORD ← WORD || CHAR;
      CHAR ← next-char(BUFFER);
    until CHAR = Λ or CHAR = ' ';

    return(WORD);
  end GET_NEXT_WORD;

/* Main Program */

local OVERCHARGE, WCOUNT, LINE, PREVW, CURRW; global BUFFER;

refill(BUFFER);
CURRW ← Λ;

B: OVERCHARGE ← false;
  WCOUNT   ← 0;
  LINE      ← Λ;
  PREVW     ← CURRW;
  CURRW     ← GET_NEXT_WORD( );

A: if CURRW ≠ ZZZ then LINE ← LINE || ' ' || CURRW;
  if length(LINE) > 100
    then print(LINE); LINE ← Λ;
  if length(CURRW) > 12
    then OVERCHARGE ← true;
  WCOUNT ← WCOUNT + 1;
  PREVW   ← CURRW;
  CURRW   ← GET_NEXT_WORD ( );
  goto A;

if PREVW ≠ ZZZ then if LINE ≠ Λ then print(LINE);
  if OVERCHARGE then print('OVERCHARGE');
  print ('NO.WORDS IS ', WCOUNT);
  goto B;
  else exit

```

end

Figure 6: The Final Program

```

procedure GET_NEXT_WORD(BUFFER);      local WORD,CHAR;
    WORD ← Λ;

    repeat CHAR ← next-char (BUFFER);
        if CHAR = Λ then refill(BUFFER);
    until CHAR ≠ Λ and CHAR ≠ '␣';

    repeat WORD ← WORD || CHAR;
        CHAR ← next-char (BUFFER);
    until CHAR = Λ or CHAR = '␣';

    return (WORD);
end GET_NEXT_WORD;

local OVERCHARGE,WCOUNT,LINE,CURRW; external BUFFER;
refill (BUFFER);
CURRW ← GET_NEXT_WORD(BUFFER);

repeat   OVERCHARGE ← false;
        WCOUNT   ← 0;
        LINE       ← Λ;

        repeat LINE ← LINE || '␣' || CURRW;
            if length(LINE) > 100 then print(LINE); LINE ← Λ ;
            if length (CURRW) > 12 then OVERCHARGE ← true;
            WCOUNT ← WCOUNT + 1;
            CURRW ← GET_NEXT_WORD(BUFFER);
        until CURRW = 'ZZZZ';

        CURRW = GET_NEXT_WORD(BUFFER);
        if CURRW ≠ 'ZZZZ' then if LINE ≠ Λ then print(LINE);
            if OVERCHARGE then print('OVERCHARGE');
            print (WCOUNT);

until CURRW = 'ZZZZ'

```

Figure 7. Final Program without GOTO's

reject the approach. This is important.

Understand, I had not yet written  $P_2$ . I was looking ahead, trying to see how a decision would look at lower levels. I was concerned with such things as initializing the word count and the output lines, counting words, and updating variables. After an hour I came up with a different  $P_2$  as shown in Figure 2. Here I committed myself. We shall see later that this too probably wasn't the best overall choice.

Having decided on  $P_2$ , I again formalized each of the statements. Figure 3 shows the result. PRINT1 takes care of adding a new word to the print line, and outputting it if its length is over 100. PRINT2 outputs the last line of a telegram, the word count, and the overcharge if pertinent.

Now there are several refinements that we can make right away. We have two calls to the procedure INITIALIZE, so we can keep just one call and use a simple goto for the other. Furthermore, we can later copy in the code for INITIALIZE, because it will be called only once. This is for efficiency. The order of the boolean conditions can be checked to see that the average number of tests is likely to be minimal. It makes sense to test for the end of the telegram before you test for the end of the stream. On the other hand, there are several redundant tests that can be eliminated. So now we can move forward to  $P_4$  with just a simple refinement.

Figure 4 shows the detailed refinements of the procedures of  $P_4$ . Note that rather than output the telegrams word by word or save the whole telegram in one string, we output our strings when they exceed 100 characters.

Figure 5 shows the development of GET-NEXT-WORD. In the Henderson and Snowdon paper, the error essentially arose here, although my program has a different structure. The procedure refills the buffer if appropriate and obtains the next word. I really don't know how the original error cropped up. There doesn't seem to be any problem.

Figure 6 shows the final program. I believe that it is correct.

#### 4. CONCLUSION

Although fairly efficient, the program of Figure 6 has a couple of small deficiencies. The previous word PREVW is saved everytime a new word is requested, even though the value is only needed at the end of a telegram. This seems to me an unnecessary bit of computation. Second, once there is an overcharge, there seems to be no need to continue testing for words that cause an overcharge. Commitments affecting these



points were at a very high level. One pays a high price for initial decisions. This I consider the most serious danger of the top-down methods. That is, I think that the danger of not being aware of the full consequences of a decision is really more significant than that of making errors.

For those who prefer the elimination of GOTO's Figure 7 shows the final program rewritten so that the only control structures are repeat-until and do-while. In addition, the use of variable PREVW is eliminated. The resulting program is somewhat shorter and clearer, and may more easily be compared with the program of Henderson and Snowden.

One point often made about top-down methods is that the structure of all higher levels is represented in the final program. This is very nice. The documentation of the program can then also be a series of levels. We don't need flow charts to document, although they may be useful. We can simply keep a copy of each level. One can use  $P_1$  or  $P_2$  for broad descriptions,  $P_4$  or  $P_5$  for detailed descriptions.

Finally, some errors will always arise even in the best of programming methods. The important issue is to write programs that are so well-structured and modularized that errors, even when they arise, will be easy to detect.

## 5. DISCUSSION

- Q. What if you make a bad design choice in programming top-down?
- A. You have a good point. Say you find that  $P_n$  is bad while working down at  $P_{n+2}$ . You may have to scrap everything below  $P_n$ . The method is not perfect. It is a suggested way of programming, and generally does seem to work well.
- Q. I wonder if that danger might be preferred to the danger of starting bottom up and having to scrap something at a middle level.
- A. I think it is much more likely to scrap everything using the bottom up approach. In the top-down approach I think it is very important that we insist on formalizing interfaces at each level. We should stop and decide exactly what the choice of  $P$  is, decide on the interfaces, and formalize them as much as possible before you go any deeper. This should sharply reduce the scrapping process.
- Q. Can we really specify the interfaces with programs at this level? I fear that people start making premature decisions on their data structures based on their knowledge of the programming language that they know they are going to use. You want to specify the interfaces at every level, but perhaps you don't want to commit yourself to the data structures that are implied by the

programming language until you also get your program refined to that level.

- A. You can formalize input and output without resorting to language dependent data structures. For example, given a collection of numbers, you may want to find the 10 highest numbers or the average. You don't have to say you have a one dimensional array. You can describe the abstract properties of objects, operations, and data structures. That leaves you as much room as you can possibly have.
- Q. I think you are saying that you should formalize interfaces from the top-down the same way as the program itself.
- A. Yes. It is less likely that formalized interfaces will later influence each other and cause you to scrap them. The sons may have to be scrapped due to a bad design choice in their father, but if you formalize the interfaces, the sons will not kill each other off.
- Q. You are talking about keeping the brothers from feuding?
- A. Yes, right.
- Q. Isn't it true that if you design the interface between the upper level and the lower levels then you may be able to salvage the lower levels even if you scrap the upper level?
- A. I just don't think that is necessarily true. But I haven't played this game strict enough or long enough to know.

# STRUCTURED PROGRAMS

S. Rao Kosaraju

*Powers of various structured control constructs are compared within a framework of weak and strong program equivalence. Results include a demonstration that Dijkstra's D-programs are strongly equivalent to programs built from functions and one-input/two-input predicates.*

## 1. INTRODUCTION

In this lecture we will discuss a systematic programming technique for improving the quality of computer programs. Most of what we are going to say seems to be non-controversial and not completely out of phase with current practice. Some of you might even ask - "Don't we already program that way?" Perhaps so, but we are not concerned only with inventing new principles; once in awhile it pays to isolate and stress the good principles we already practice.

Systematic programming has recently been the subject of much critical introspection. The central idea is to avoid unnecessary complexity by decomposing a problem into an interconnection of more tractible subproblems. By iterating this process for each of the subproblems, we develop the control structure of the final program gradually. This approach seems to be the key to breaking down the complexity of large problems, but it should be understood that we do not consider this a complete solution to the problem of program development.

Let us investigate this method in detail.

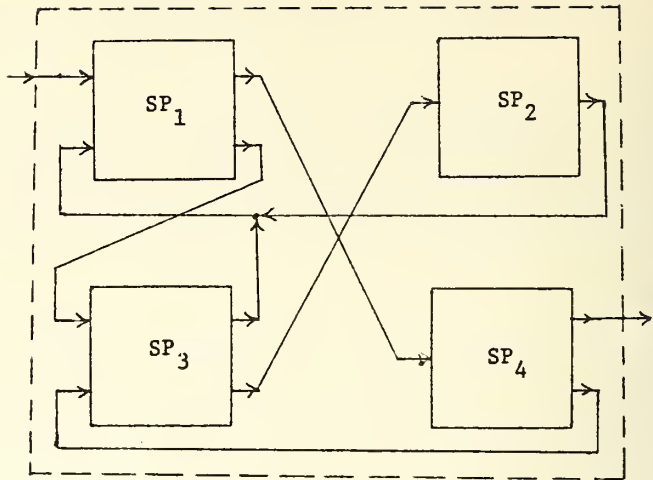
## 2. BASIS FOR STRUCTURED PROGRAMS

Program development can be classed as either top-down or bottom-up. These techniques are discussed in detail in another lecture in this collection (Chapter 2). A brief review follows.

### 2.1 Top-down Programming

In top-down programming, the original problem is decomposed into a number of interconnected subproblems. We iterate the process on each one of these subproblems, until we come to the basic instruction level of the programming language. The below figure illustrates one step of decomposition.

Problem:



$SP_2$  : (1,1) sub-problem

$SP_4$  : (1,2) sub-problem

$SP_1, SP_3$  : (2,2) sub-problems

## 2.2 Bottom-up Programming

In bottom-up programming we do just the reverse: we take some basic instructions and assemble them into a module which is used as a sub-module for a bigger module. We continue this process until the problem is realized.

## 2.3 A Thesis

For "Reliability in Programming" - at each step in the development of a program:

- I. A module should be decomposed into as few non-trivial, non-overlapping sub-modules as possible, in the top-down case; or composed from only a few sub-modules in the bottom-up case. This is particularly important for the multi-exit sub-modules.
- II. The number of entries (Inputs) and exits (Outputs) of each sub-module should be as small as possible.

The comment that any program can be trivially considered as the result of top-down (or bottom-up) programming by viewing each instruction as a sub-module, illustrates the importance of I. Part of our

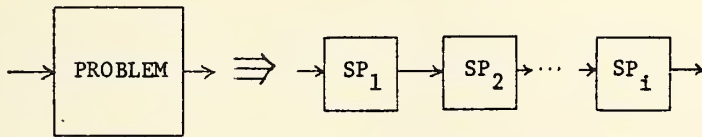
motivation for II stems from the fact that if a sub-program has 20 entry and exit points, then it may be difficult to understand the final program.

Q. What definition of entry and exit are you using?

A. We are not counting the number of input or output variables; but the number of input or output points, such as PL/1 multiple entry points.

There is no formal proof of these theses, but they seem to be intuitively reasonable. Accepting these theses, we may investigate the minimum number of entries and exits needed for a sub-module.

The first possibility is that each sub-module has only one entry and one exit. Then the only construct allowed is:



This construct alone is obviously not powerful enough to constitute the entire control structure for any real programming language; i.e. the one entry, one exit constraint is too restrictive.

Q. Why is one entry and one exit not sufficient?

A. The constraint does not allow conditional statements, and thus in any program, we are constrained to use only functional instructions. It can be shown that we cannot realize each and every problem using just a finite sequence of these instructions. We are not considering "super-languages" which contain an infinite number of basic instructions.

Q. I do not understand whether you are submitting this argument as a thesis or as a proof. As a proof it seems to be false. You can have an instruction-set in the old 3 address machine, in which each instruction has one entry and one exit.

A. The third address serves as a jump location and thus each instruction is a function followed by a multi-way branch.

It can further be shown that n entry and one exit sub-modules are not sufficient. We may ask if a general problem can be realized using one entry and two exit modules?

We will show later on that one sub-module with one entry and two exits, combined with a finite number of one entry and one exit sub-



modules is sufficient for any decomposition. This assumes that the language has certain minimal power [Bohm and Jacopini, 1966]. Let us denote the class of programs we obtain by restricting the decomposition of a module to allow only a single one entry two exit sub-module and a finite number of one entry and one exit sub-modules by  $TD_1$ -programs. We shall investigate the close relationship between  $TD_1$ -programs and the so-called class of GOTO-less programs.

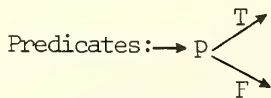
Q. I do not see how you relate (1,2) programs and GOTO-less programs, since a GOTO instruction is nothing but Add N to the P Register which has one entry and one exit.

A. But there is an implied 'Go to the location given by the contents of P' which is a multi-way branch.

### 3. TERMINOLOGY

Let us first discuss, more precisely, how we define programs. Any programming language has some basic instructions: predicates (conditional statements) and functions. A function is a one input/one output store to store transformation and its execution changes control from input point to output point. A predicate is a one input/two output, labeled T and F, (1,2) conditional test on the store, which does not modify the store. If the condition is satisfied, then the control changes from input to the output labeled T, otherwise to the output labeled F. We could consider more general constructs like one input/n-output (1,n) predicates, CASE statements or SELECT statements. The above types of instructions form a normalized class; many questions about other forms can be answered by considering only the above normalized class. Thus, we will restrict our discussion to the two basic elements: functions and (1,2) predicates. The notations are:

Functions:  $\rightarrow a \rightarrow$



A program is any one input (called IN)/one output (called OUT) deterministic interconnection of these basic elements. Deterministic refers to the usual constraint that the out of a basic element can become the in of only one basic element. For simplicity, we also assume that for any basic element, there exists a path from IN to OUT passing through that element. Mills calls such programs proper programs (Mills, 1972)

A program, P, operating on any input, x, starts with control at IN, and if the control passes to OUT with value z, then we say that P outputs z on input x. We write this  $z = P(x)$ .

Now we define how to compare two programs. There are basically two types of equivalence, strong and weak.

Two programs,  $P$  and  $Q$ , are strongly equivalent ( $P \equiv_s Q$ ) if and only if every valid input passes through the same basic element at every step within  $P$  and  $Q$ , i.e. have the same computational history.

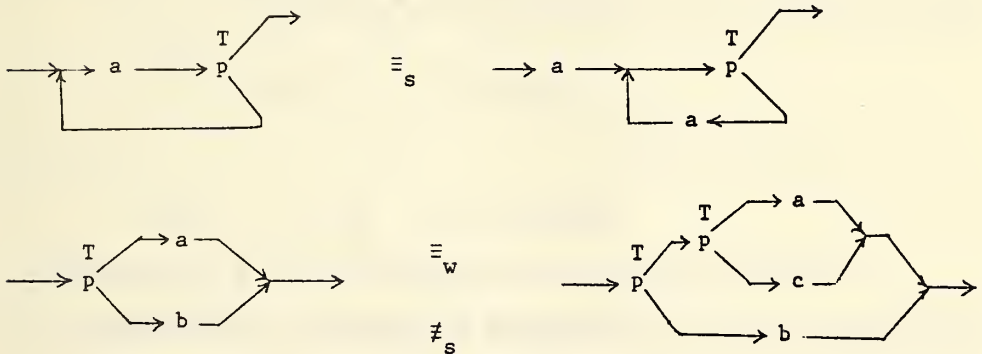
Two programs,  $P$  and  $Q$ , are weakly equivalent ( $P \equiv_w Q$ ) if and only if they have identical input to output transformation for all valid inputs.

Thus, weak equivalence is only concerned with the store to store transformation performed by the programs and not their internal structures.

### 3.1 Remark

If  $P \equiv_s Q$ , then  $P \equiv_w Q$ .

### 3.2 Examples



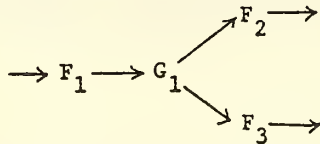
## 4. $TD_1$ -PROGRAMS

Let us come back to the general class of  $TD_n$ -programs. Since we also need  $(1,2)$ -programs to use as sub-modules, we will define  $TD_n$ -programs and  $(1,2)_n$ -programs together as follows.

1. Any basic function is a  $TD_n$ -program.
2. Any basic predicate is a  $(1,2)_n$ -program.
3. Any deterministic interconnection of at most  $n$   $(1,2)_n$ -programs, and any number of  $TD_n$ -programs is a  $(1,2)_n$ -program if it is a  $(1,2)$ -program, and is a  $TD_n$ -program if it is a  $(1,1)$ -program.

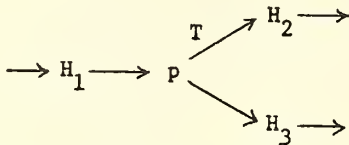
What types of reductions can be performed on  $TD_1$  constructs? Observe that if  $F_1$  and  $F_2$  are  $TD_1$ -programs, then  $\rightarrow F_1 \rightarrow F_2 \rightarrow$  is a  $TD_1$ -program. Let us denote it as BLOCK rule. Applying the block rule, we can easily see that the general construct for  $(1,2)_1$ -programs can be replaced by:

If  $F_1, F_2$  and  $F_3$  are  $TD_1$ -programs, and  $G_1$  is a  $(1,2)_1$ -program, then



is a  $(1,2)_1$ -program.

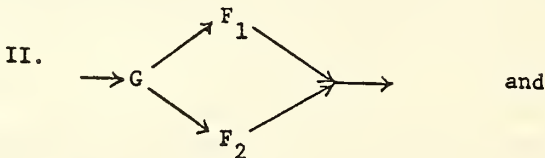
Applying this rule again to  $G_1$  and continuing this process, we could easily see that any  $(1,2)_1$ -program,  $G$ , is of the form:

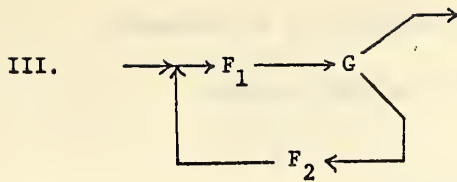


where  $H_1, H_2$  and  $H_3$  are  $TD_1$ -programs and  $p$  is a basic predicate.

The general construct for  $TD_1$ -programs is covered by the following 3 constructs:

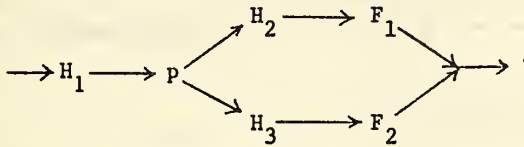
I.  $\rightarrow F_1 \rightarrow F_2 \rightarrow$



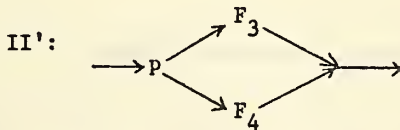


where  $F_1$  and  $F_2$  are  $TD_1$ -programs and  $G$  is a  $(1,2)_1$ -program.

Substituting the previous derived form for  $G$ , construct II reduces to:

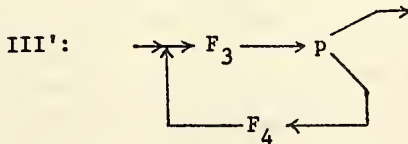


By construct I (BLOCK rule), it can be further reduced to:



where  $F_3$  and  $F_4$  are  $TD_1$ -programs.

By similar reasoning, we can reduce construct III to

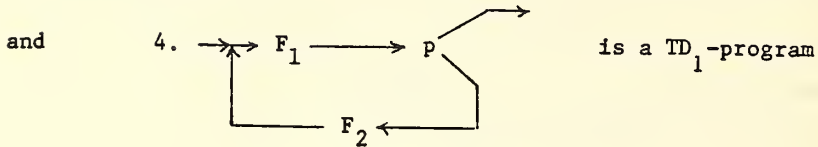
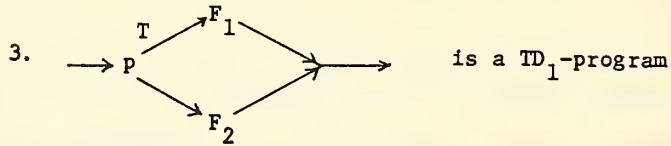


where either labeling (of T and F) is allowed for p.

Thus, the class of  $TD_1$ -programs can be equivalently defined as follows:

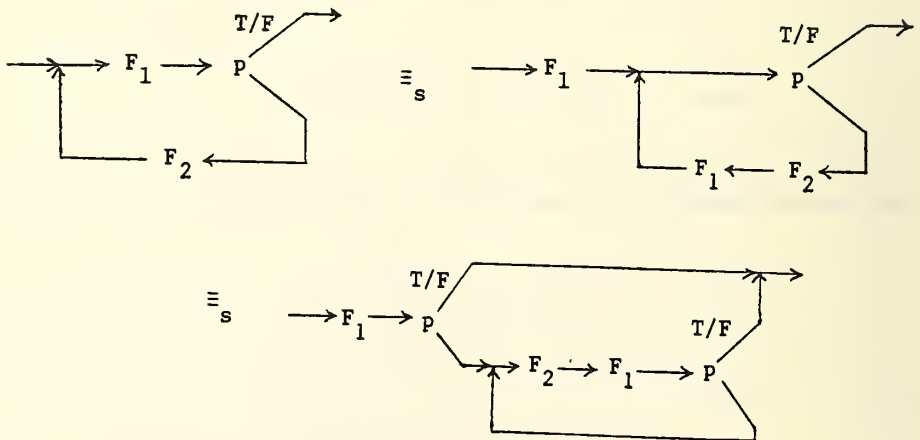
1. Any basic function is a  $TD_1$ -program.

2.  $\rightarrow F_1 \rightarrow F_2 \rightarrow$  is a  $TD_1$ -program.



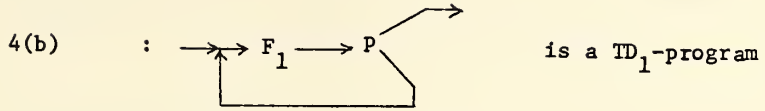
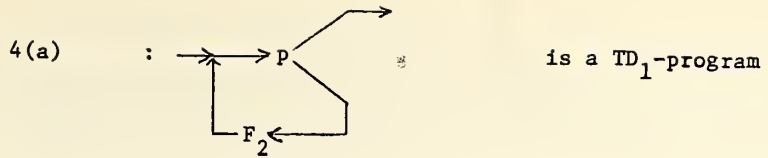
where  $F_1$  and  $F_2$  are any  $TD_1$ -programs, and  $p$  is any basic predicate.

If we are interested in only strong equivalence, then



Thus, preserving strong-equivalence, we could replace construct 4 by 4(a) or 4(b) given below:





Construct 3 is known as IFTHENELSE construct, since we can write it in the linear form:

IF  $p$  THEN  $F_1$  ELSE  $F_2$

Construct 4(a) is known as DOWHILE construct, since it can be written as:

WHILE  $q$  DO  $F_2$  where  $q = p$  or  $\sim p$

Construct 4(b) is known as DUNTIL construct, since it can be written as:

DO  $F_1$  UNTIL  $q$  where  $q = p$  or  $\sim p$

The class of programs defined by constructs 1,2,3 and 4(a)(or 4(b)) is known as the class of D-programs (D for Dijkstra). The same class is known as the class of GOTO-less programs, since we do not need the unconditional GOTO as a control structure.

Thus, by the above analysis, the class of  $TD_1$ -programs is strongly equivalent to the class of D-programs. It would be interesting to analyze the general class of  $TD_n$ -programs.

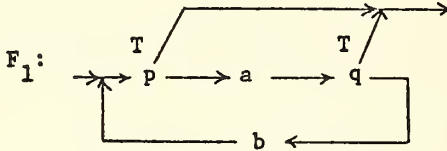
In these classes of programs we are imposing restrictions on the structure of the programs. Hence each class is known as a class of structured programs. Formally, if we denote the class of all programs by  $\hat{p}$ , then any class  $S \subset \hat{p}$  is a class of structured programs and any  $G \in S$  is a structured program with the structure implied by  $S$ . The very act of defining a sub-setting rule makes the subset  $S$  a structured class. Thus, the class of D-programs is only one of the many structured classes possible.

## 5. COMPUTATIONAL POWER OF D-PROGRAMS

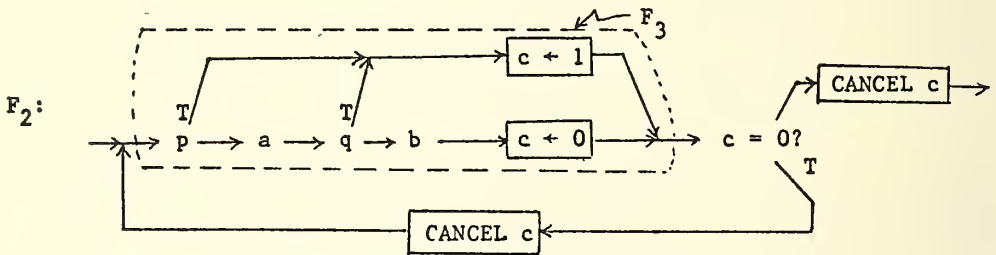
It is not very hard to verify that all partial recursive functions are computable by D-programs. Böhm and Jacopini [1966] have proved that for every program there exists a weakly equivalent D-program. Their technique is rather easy and we illustrate it with an example;

### 5.1 Example.

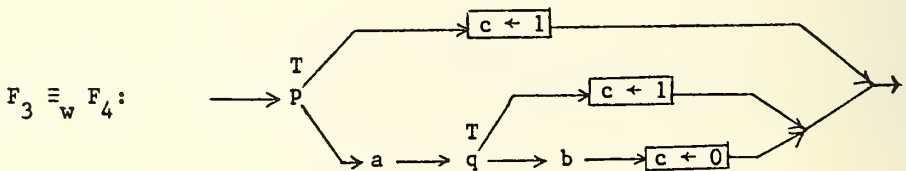
Convert  $F_1$  into a D-program:



$F_1 \equiv F_2$  given by:



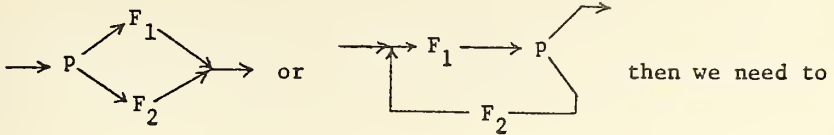
If  $F_3$  can be converted into a D-program, then  $F_2$  can in turn be converted into a D-program by the DO and BLOCK constructs.



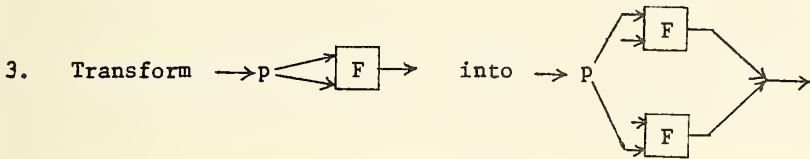
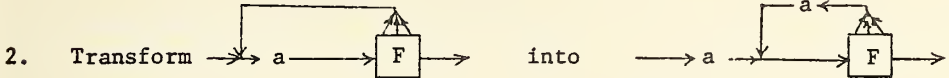
which is a D-program by IFTHENELSE and BLOCK constructs.

## 5.2 General Böhm and Jacopini Reduction

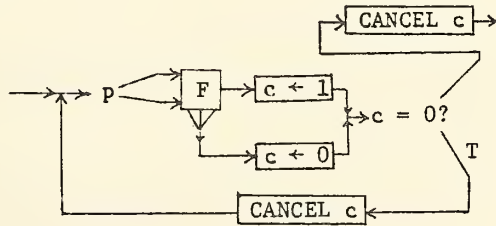
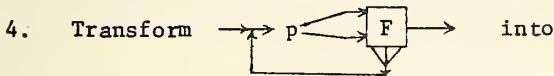
1. If the program has the form  $\rightarrow a \rightarrow F_1 \rightarrow$  or



reduce  $F_1$  (and  $F_2$ ) only.

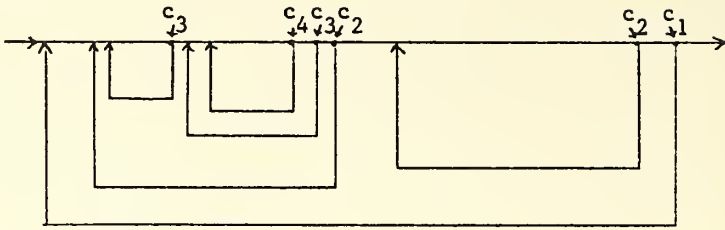


and delete, from each copy of  $F$ , basic elements not on any path from the corresponding IN to OUT.



You can easily verify that by applying these rules any program can be reduced to an equivalent D-program. Observe that the control variable  $c$  associated with a loop is not needed when control passes out of that loop. When control is within a loop, control variables associated with loops enclosing that loop will not be tested. Hence we can introduce  $m$  control variables  $c_1, c_2, \dots, c_m$  such that if  $c_1$  is assigned to the outer-most loop and  $c_2$  to the immediate inner loops and so on, then the control variables could be implemented as a stack.

### 5.3 Example (Control Variable Structure)

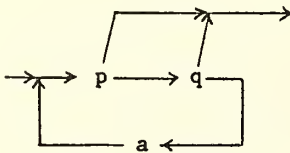


Assigning 0 or 1 to a variable is equivalent to pushing the corresponding number on the stack. Testing a variable is equivalent to testing the top of the stack. Cancelling a variable is equivalent to popping off the stack.

Cooper [1967] showed that only one DO loop is sufficient. Ashcroft and Manna [1971] gave some other reduction techniques.

We should not expect to gain much by writing any program and then converting into an equivalent D-program. One should write "natural" programs directly as a D-program. So part of the advantage seems to be psychological.

Note that we have to introduce extra control variables and tests on those variables. They tend to obscure the program, rather than shed new light on it. Let us consider decomposition techniques which do not allow the introduction of extra variables; i.e. the reduced program must consist of only the basic elements of the original program. Then not all programs are reducible to equivalent D-programs. For example:



is not reducible to D form. Elsewhere

[Kosaraju, 1972] we treat this problem in more detail.

## 6. PRACTICE

Structured programming discipline can be enforced by voluntarily restricting the control structures one uses in an existing programming language or by restricting the language itself.

An example of a system based on the latter approach is BLISS

[Wulf, 1972] which has basically six control structures. Four of them (Function, Condition, Loop and Case/Select) are BLOCK, IFTHENELSE and DO constructs or their generalizations. The fifth construct (ESCAPE or EXIT) allows jumping out of  $n$  levels of the above constructs. This construct seems to be a very powerful one. Elsewhere [Kosaraju, 1972] we investigate the power and limitation of this construct. The last one (co-routine) essentially consists of two programs which "call" each other. When a program is "called" control returns to the place it left off earlier. This construct can become extremely complicated and should be used with care.

- Q. Do you advocate restricting programmers to flow-chartable programs only; and avoiding concepts like recursive procedures and all the power we have in present day languages?
- A. We should systematically study the various recursive procedures. If it so happens that some of the features are "hard" we might not want to use them. More and more powerful languages with "interesting features" have been developed, the only limitation being personal fancy and imagination. Maybe it is time to sort out the useful from the interesting ones. It is also true that many processes can be expressed most simply, concisely and clearly as recursive procedures.

## 7. CONCLUSIONS

The principle of using only simple constructs in the decomposition of a problem into subproblems seems to be a very significant one in writing quality software. We should spend more time in understanding the "complexity" of various control structures, either theoretically or by experimentation, rather than debating whether or not avoiding GOTO's is the ultimate solution for all the evils in programming.



Henry F. Ledgard

*After a review of Quine's notion of referential transparency, the author examines elements of good and bad programming practice. A series of examples demonstrate various points. In addition, a table of programming proverbs provides guidance to a programmer, and should be especially useful to a novice.*

## 1. INTRODUCTION

The nature of "quality" software is a vague one; if two of us write a program to do a specific job we each might claim that ours is the "better" program, but in general we cannot prove or validate our claims. One of the things I want to get to here is some idea for doing just that. There seem to be two major problems:

- (1) The formalization of the intuitive notion of "quality" in software.
- (2) The development of mechanical procedures to insure "quality" software.

As to the first objective, the term formalize is well understood; it means to develop a basic set of concepts from which we can derive properties of and theorems about the objects in question, here - programs. What we need are well defined metrics that can be applied to a program to obtain a measure of its quality. Once we have achieved this set of metrics, then we can say something definite about the quality of a program. As to the second objective, we need to devise means for assuring that the software we produce is indeed quality software. Now I don't just mean attending a good programming course; I mean some specific mechanical constraints which a programmer or programming team can apply to insure the quality of the output. For example, we might restrict the amount of computer time available, place a programmer in a closed room, provide a programmer with languages or compilers that restrict the mechanisms available to him (e.g. no GOTO's), or completely restructure the programming process.

Our major interest today will be on objective (1). My interest in this objective started quite casually. Working with several students, I have been writing a small text to help student programmers improve the quality of their programs. The text is based on a collection of 26 "Programming Proverbs", which are listed in Table 2, p 63. Two of these proverbs, "Avoid Side-Effects" and "Avoid Tricks", were the original motivation for today's work. More generally, we shall focus on two important issues:

- (1) An attempt to formalize the notion of program modularity.
- (2) An outline of a model for writing natural, readable programs.

## 2. REFERENTIAL TRANSPARENCY

My first major topic was one upon which I spent considerable time. My objectives were the following:

1. To somehow get a hold on context dependent features of programs. It is well known that languages can be classified as context-free or context sensitive. While most programming languages are in fact context sensitive, context free languages are simpler to understand, implement, and define. It appeared to me that perhaps we could get at a similar result for programs.
2. To define a precise measure that could be applied to a program to give some indication of its "modularity".
3. To use as little apparatus from the semantics of programming languages as possible.

The concept that I thought would do the trick was Quine's notion [Quine, 1960] of "referential transparency". My investigation into referential transparency has to a large degree failed. Nevertheless, I would like to review Quine's notion, give some programs that I thought would be embraced by this concept, and at least point out the properties that I think are worthy of future study.

### 2.1. Definition 1 (Quine)

Let  $e$  be an expression with value  $v$  ; and let

$$E = \{e_1, \dots, e_n\}$$

be a set of expressions each of which has the same value  $v$ . Now let  $\psi(e)$  be a sentence containing an occurrence  $c$  of the expression  $e$ . Then the occurrence  $c$  of  $e$  in  $\psi(e)$  is purely referential if and only if  $c$  can be replaced by any  $e_i, e_i \in E$

For example, let

$$e = \underline{\text{sq}}(2)$$

$$v = \underline{\text{value}}(e) = 4$$

$$E = \{4, 40*10^{-1}, \underline{\text{sq}}(1+1)\}$$

$$\psi(e) = X \leftarrow Y * \underline{\text{sq}}(2)$$

The occurrence of  $e$  in  $\psi$  is purely referential.

## 2.2 Definition 2 (Quine)

Let  $\Phi$  be a mode of containment, i.e. any fixed method of constructing a composite expression from a given number of component expressions  $\psi_1, \psi_2, \dots, \psi_n$ .  $\Phi$  is referentially transparent if, whenever an occurrence of a sub-expression  $e$  is purely referential in some component expression  $\psi_i(e)$ , the occurrence is also purely referential in  $\Phi$ .

Consider the following example:

$e$	Cicero
$E$	$\{\frac{e_1}{\text{Marcus Tullius}}, \frac{e_2}{\text{Tully}}\}$
$\psi_1$	Most people
$\psi_2$	... was an orator
$c$ and $\psi_2(e)$	$\frac{c}{\text{Cicero was an orator}}$
$\Phi$	... believe that ...
$\Phi(\psi_1, \psi_2(e))$	Most people believe that Cicero was an orator
$\Phi(\psi_1, \psi_2(e_1))$	Most people believe that Marcus Tullius was an orator

Now in the above example,  $\psi_2(e)$  is a true statement and the corresponding state  $\Phi(\psi_1, \psi_2(e))$  is (probably) true. However, while  $\psi_2$  remains true if we substitute  $e_2$  (Marcus Tullius, one of Cicero's other names) for  $c$ ,  $\Phi(\psi_1, \psi_2(e_2))$  (probably) does not remain true.

The importance of this concept lies in the fact that we often want to replace a piece of program text by an "equivalent" one. It is this concept which allows us to "draw a circle" around a piece of text and refer to "its value" independent of the larger context in which it occurs.

I attempted to generalize this notion to programming languages.

This included distinguishing between free and bound variables, the effect of assignment, and the difference between formal parameters of pure sub-routines and formal parameters of pure functions. I further tried to define "referential transparency with respect to a procedure" and the notion of "purely referential procedure calls". For example, it would be nice to say in some definitive sense that a program P is referentially transparent with respect to, say, a scheduling algorithm or its data structure. By this we would mean that the scheduling algorithm or data structure could be independently changed and still leave the rest of the program correct. Rather than go into these gory details, I will present some simple examples and discuss some relevant issues.

I will ask the reader's indulgence in the sequel and use the terms "transparent" and "opaque". But in no case should these terms be considered as well-defined or correctly used.

#### Example 1. (OPAQUE)

```
dcl F(X); X ← X + 1; return (X)
```

One generally undesirable feature of programs exhibited in Example 1, is the assignment of new values to the arguments of pure functions. The problem with this feature is that, for example,  $F(A) + F(A)$  does not equal  $2 * F(A)$ .

#### Example 2 (OPAQUE)

```
dcl F(X); return (X+Z)
```

The problem with Example 2 is the reference to the free variable, Z. This again should be avoided. Consider the following:

```
A ← F(1)
Z ← Z+1
B ← F(1)
```

Here we see two calls to F each of whose returned values are different, even though the arguments of F are the same. A similar case holds for assignment to free variable in pure sub-routines:

#### Example 3 (TRANSPARENT)

```
dcl SUB(X); X ← X+1
```

#### Example 4 (OPAQUE)

```
DCL SUB(X); Z ← X+1
```

In some sense, assignment to the arguments of sub-routines is fine (Example 3), but assignment to free variables is not (Example 4). I pause again to state that I wished to develop a formal notation that could be applied on a "yes" or "no" basis to determine whether a program did or did not have these undesirable effects. I wanted Examples 1,

2, and 4 to be opaque; Example 3 to be transparent.

Part of the reason for not wishing to couch the discussion in terms of functions, sub-routines, store-to-store transformations, and the like, is the following. Namely, many languages allow these same violations without explicit recourse to the notion of functions and sub-routines. Consider the following SNOBOL-like example.

Example 5 (OPAQUE)

```
Pl = *H ', ' BREAK (', '). H
dcl DEAL ( ); 'N,S,E,W' Pl;

H = 'S';
DEAL ( )
```

The call to DEAL, a function of no explicit argument, results in updating the value of H.

Next consider Examples 6 and 7.

Example 6 (TRANSPARENT)

```
I ← 1

Loop: I ← I + 1

GO TO LOOP
```

Example 7 (TRANSPARENT)

```
dcl F(X) = X + 1
Z ← 1
A ← F(Z)
Z ← 2
B ← F(Z)
```

Whatever criteria we use for determining the transparency or opacity of a linguistic construct, each of these examples should be transparent. In line 2 of Example 6, it should not be possible to replace "I + 1" by "2". In Example 7, we should not insist that the two calls to F(Z) return the same value.

Some thought was given to the notions of Input/Output and random numbers. For example consider the following statements:

Example 8 (OPAQUE)

```
X ← read(N) + read (N)

Y ← random ( ) + random( )
```

The claim is that the uses of read and random as "functions" are opaque. They both cause side effects. Furthermore the two statements above could not be replaced by

X ← 2 \* read(N)

Y ← 2 \* random( )



When read and random are used as sub-routines the case is different.

Example 9 (TRANSPARENT)

<u>read</u> (N)	<u>random</u> (A)
N1 $\leftarrow$ N	A1 $\leftarrow$ A
<u>read</u> (N)	<u>random</u> (A)
N2 $\leftarrow$ N	A2 $\leftarrow$ A
X $\leftarrow$ N1 + N2	X $\leftarrow$ A1 + A2

Here we fully expect different calls to give different values. Claim: Input/output and random number calls should only be used as sub-routines.

As we stretch our imagination, we come to more diverse examples. Consider the following:

Example 10 (OPAQUE)

```
declare X BASED (P),  
        Y BASED (Q);  
ALLOC X;  
ALLOC Y;  
X  $\leftarrow$  2;  
Y  $\leftarrow$  3;  
Z1  $\leftarrow$  X + Y;  
P  $\leftarrow$  Q  
Z2  $\leftarrow$  X + Y;
```

Here we have two occurrences of the same expression "X + Y". While neither X nor Y is changed between the two occurrences, the values of the two occurrences are different.

The point of all this discussion was the attempt to develop a property of programs. I had hoped that a program with this property would have the following characteristics.

1. Given any piece of text within the program, only those variables that are explicitly stated within that piece of text may have their values changed.
2. The effect of any sub-program can be defined solely by its input/output characteristics, and hence its meaning is independent of the variable names used, declarations, sequencing, etc.
3. Replacing any sub-program with another (referentially transparent) sub-program with the same input/output characteristics will not cause an error in any other part of the program.

I, didn't quite make it.



### 3. TRICKY PROGRAMMING

The final issue that I want to consider was motivated by a question attributed to Christopher Strachey. Namely, "What is the difference between a number and a date?" For example, you can perform all the usual operations on numbers, but while you can subtract two "dates" to get an "age", it doesn't make sense to "add" two dates or take the "square root" of a date.

Programmers often work with various kinds of abstract objects, for example, dollars, social security numbers, measurements, people's names, etc. These objects have abstract properties that differ significantly from objects like numbers and strings. Yet, numbers and strings are the primitive objects of today's programming languages. Part of the programmer's job is to map the abstract objects he is working with into the objects of a particular programming language. In PL/1 for example, these objects would include numbers, strings, arrays, structures, pointers, offsets, labels, files, etc. This allows the programmer to make an exact choice of a particular representation for an abstract object but not necessarily to describe the abstract object itself or to reflect its own peculiar properties. More importantly, it does not guarantee that an operation that can be performed in a programming language corresponds to any meaningful operation with respect to the original abstract objects.

Consider the following piece of code.

$$(1a) \quad X \leftarrow (Y < Z) * 4 + (Y > Z) * 6$$

where the value of the expression " $(Y < Z)$ " is 1 if the value of  $Y$  is less than the value of  $Z$ , and otherwise is 0. Consider also an equivalent piece of code:

$$(1b) \quad \text{if } (Y < Z) \text{ then } X \leftarrow 4 \text{ otherwise } X \leftarrow 6$$

Claim: Example (1a) is an "unnatural" method of performing the desired computations while example (1b) is "natural". The motivation here is that in example (1a) it makes no sense to multiply 4 times the result of comparing  $Y$  with  $Z$ , i.e. it makes no sense to multiply numbers by truth values.

More generally, consider Figure 1. Here we have two spaces: the space  $A$  that corresponds to a space of "abstract" objects and operations in some external world, and the space  $C$  of objects and operations of a given "concrete" programming language  $L$ . In order to perform some computation over an abstract space, a programmer must map the operations in his abstract space to those in the given programming language. This mapping is indicated by the morphism  $M$ . The programmer must then write a program  $P$  that is a composition of operations given in his programming language. The result computed by the program presumably corresponds to some result in the abstract space.

Our claim here is that a program P has a "natural" interpretation if and only if the following condition holds:

Property X

Given A, C, M, P, and L,

$\forall g, b_i \in C$  where  $g$  is applied to  $b_1, b_2, \dots, b_n$  in P

$\exists f, a_i \in A$  where  $f$  is applied to  $a_1, a_2, \dots, a_n$  in A

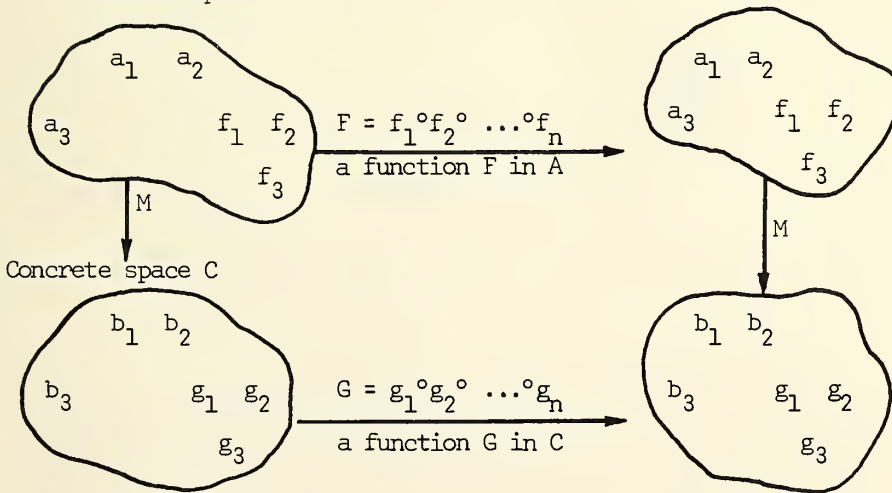
such that

$$M\langle f \rangle = g$$

$$M\langle a_i \rangle = b_i$$

$$\begin{aligned} M\langle f \rangle(M\langle a_1, a_2, \dots, a_n \rangle) &= M\langle f(a_1, a_2, \dots, a_n) \rangle \\ &= g(b_1, b_2, \dots, b_n) \end{aligned}$$

Abstract space A



Property X

$$\begin{aligned} M\langle f \rangle &= g \\ M\langle a_i \rangle &= b_i \\ M\langle f \rangle(M\langle a_1, a_2, \dots, a_n \rangle) &= M\langle f(a_1, a_2, \dots, a_n) \rangle = g(b_1, b_2, \dots, b_n) \end{aligned}$$

Figure 1: The Programming Problem

This equation means that:

- (1) any operation  $g$  performed in a programming language must correspond to some abstract operation  $M\langle f \rangle$  in the abstract space;
- (2) any object  $b_i$  manipulated in the programming language must also correspond to some object  $M\langle a_i \rangle$  in the abstract space;
- (3) the application of the programming language operation  $g$  to objects  $b_1, b_2, \dots$ , and  $b_n$  must correspond to some manipulation  $M\langle f \rangle(M\langle a_1, a_2, \dots, a_n \rangle)$  in the abstract space;
- (4) the result  $g(b_1, b_2, \dots, b_n)$  computed in the programming language must correspond to the desired result  $M\langle f(a_1, a_2, \dots, a_n) \rangle$  in the abstract space.

Given some abstract problem, there is some question as to what the morphism  $M$  should be. At some risk, I will rely on the reader's intuition to conjure up some conventional definition of  $M$  for a given problem.

Now consider the example (1a) and the morphism  $M$  such that

$M\langle \text{true} \rangle = 1$   
 $M\langle \text{false} \rangle = 0$   
 $M\langle \text{number} \rangle = n$       where  $n$  is a number in  $L$   
 $M\langle \text{times} \rangle = *$

Here the comparison of  $X$  with  $Y$  yields a truth value true or false. Hence

$M\langle \text{times} \rangle(M\langle \text{true}, \text{number} \rangle) = *(1, n) \neq M\langle \text{times}(\text{true}, \text{number}) \rangle$

since times applied to a truth value and a number is undefined.

Consider next examples (2a) and (2b). In this case the input into the program is an array called DECK that contains 51 elements, corresponding to 51 cards from a conventional deck of 52 cards. The object of the program is to determine the missing card needed to complete a normal deck of 52 cards. The function CONVERT maps an "Ace" into the number 1, a "deuce" into a number 2,  $\dots$ , and a "King" into number 13. DECONVERT is the inverse of CONVERT and maps these corresponding numbers into their corresponding card rank. The reader is invited to read these corresponding pieces of code and to try to determine why each of the programs computes the missing card.

The claim here is that the program of Example (2a) is not natural. In particular, consider the statement

COUNT  $\leftarrow$  CARD + COUNT

/\*Find the rank of the missing card in DECK \*/

COUNT ← 0

varying I from 1 by 1 until I = 51 do

CARD ← CONVERT(DECK[I])

COUNT ← CARD + COUNT

if COUNT > 13 then COUNT ← COUNT - 13

end do

RANK ← DE CONVERT(13-COUNT)

print (RANK)

Example (2a) "Unnatural" Card Count

/\*Find the rank of the missing card in DECK \*/

dcl COUNT[1:13] array initial value (0);

varying I from 1 by 1 until I = 51 do

CARD ← CONVERT(DECK[I])

COUNT[CARD] ← COUNT [CARD] + 1

end do

varying J from 1 by 1 until 13 do

if COUNT[J] ≠ 4 then RANK ← DECONVERT (J)

end do

print(RANK)

Example (2b) "Natural" card count

Here CARD corresponds to some card in the deck and COUNT corresponds to some total which is kept during the course of the program. The program is unnatural in the sense that, in the abstract space of cards, for example a king, and a COUNT, for example 10, cannot in any meaningful sense be "added", i.e.

$M < +(\text{card}, \text{number}) >$

is undefined. To "add" a number to a King makes no sense that in the corresponding space of cards, ranks, and suits. Hence, the equation for Property X is violated. In Example 2, no violation occurs. The number of cards of a given rank are simply counted, and the results are stored in an array called COUNT. The rank of the missing card is then determined by simply scanning through the array COUNT to find that element for which there is a count of 3. The rank of the missing card is then computed by applying the function DECONVERT to the index of the array denoting the missing card.

Consider now the bridge hand deal of the SNOBOL4 program\* [11] of example (3). The program generates bridge hands for four players NORTH, SOUTH, EAST, and WEST. From a quality software point of view, this is a programming disaster.

First let's look at the operation of the program. It starts off (line 1) with some OUTPUT routine definitions that will print headings when invoked. More routines are then defined via the DEFINE function, and control is passed (line 8) to the statement labeled CONSTANT (line 49). After this point, several variables are initialized. Note, for example, that the variable NXTHAND (line 67) is initialized to a value which is a pattern, not a string.

Starting at line 77 an array NEWDECK is loaded with "cards" to give a "flat deck." Empty arrays for each player (NORTH, SOUTH, EAST, and WEST) are created using the (primitive) function COPY. Finally, the two user defined functions DEAL and DISPLAY are called.

It is with DEAL that we will be most concerned, so let's briefly go through its operation. First, a pattern match occurs which picks the next dealer. DECK is then initialized to be a "flat deck." Another pattern match (line 12) is used to obtain the next recipient of a card. The card to be dealt is chosen by picking a random number CARD,  $0 \leq \text{CARD} \leq N$ , where N is the number of cards remaining in DECK. Once the randomly selected card has been dealt (using ITEM, a primitive function which accesses a specified element in the array given as the first argument), the last card in the deck replaces the one just dealt (unless they are the same) and the length N of the DECK is decremented. This process continues until the DECK is empty, at which point control returns to the calling point. The function DISPLAY prints the results of DEAL, as shown at the end of example 3.

\* Griswold, Poage, and Polonsky, THE SNOBOL 4 PROGRAMMING LANGUAGE, 2nd ed., (c) 1971. Reprinted by permission of Prentice Hall, Inc., Englewood, Cliffs, New Jersey.



```

*
* THIS PROGRAM USES ARRAYS, PROGRAMMER-DEFINED
* FUNCTIONS, AND A VARIETY OF OUTPUT FORMATS TO
* PRODUCE SETS OF BRIDGE HANDS. THE FUNCTION OF
* DEAL() USES A PSEUDO-RANDOM NUMBER GENERATOR
* [13] TO DEAL CARDS INTO FOUR ARRAYS NORTH, EAST,
* SOUTH, AND WEST. THE FUNCTION DISPLAY() PRINTS
* THE HANDS, ONE TO A PAGE.
*

OUTPUT('TITLE',6,'(14H1THIS IS HAND ,110A1)')
OUTPUT('DEALER',6,'(11H DEALER IS ,110A1)')
OUTPUT('SKIP',6,'(A1)')

*
* FUNCTIONS
*

DEFINE('DEAL()')
DEFINE('DISPLAY()')
DEFINE('LINE(STR1,COL1,STR2,COL2)BL1,BL2')
DEFINE('RANDOM(N)')
DEFINE('SUITL(HAND,SUIT)N')      : (CONSTANT)

*
DEAL    DEALSEQ  DEALHAND
        DECK    =  COPY(NEWDECK)
        N       =  51
NLOOP   DEALSEQ  NXTHAND
        CARD    =  RANDOM(N + 1)
        DECK<CARD> SUITRANK =  NE(CARD,N)  DECK<N>
        ITEM($HAND,SUIT,RANK) = $RANK
        N       =  GT(N,0)  N - 1          : S(NLOOP)F(RETURN)

*
DISPLAY TITLE =  NTHDEAL
        DEALER =  DEALR
        SKIP =
        OUTPUT =  LINE('NORTH',40)
        OUTPUT =
        OUTPUT =  LINE(SUITL(NORTH,'S'),40)
        OUTPUT =  LINE(SUITL(NORTH,'H'),40)
        OUTPUT =  LINE(SUITL(NORTH,'D'),40)
        OUTPUT =  LINE(SUITL(NORTH,'C'),40)
        SKIP =
        OUTPUT =  LINE('WEST',20,'EAST',60)
        OUTPUT =
        OUTPUT =  LINE(SUITL(WEST,'S'),20,
+                SUITL(EAST,'S'),60)
+        OUTPUT =  LINE(SUITL(WEST,'H'),20,
+                SUITL(EAST,'H'),60)
+        OUTPUT =  LINE(SUITL(WEST,'D'),20,
+                SUITL(EAST,'D'),60)
+        OUTPUT =  LINE(SUITL(WEST,'C'),20,
+                SUITL(EAST,'C'),60)
        SKIP =

```



```

OUTPUT = LINE('SOUTH',40)
OUTPUT =
OUTPUT = LINE(SUITL(SOUTH,'S'),40)
OUTPUT = LINE(SUITL(SOUTH,'H'),40)
OUTPUT = LINE(SUITL(SOUTH,'D'),40)
OUTPUT = LINE(SUITL(SOUTH,'C'),40)
+
*
*
LINE BL LEN(COL1 - 1) . BL1
BL2 = DIFFER(STR2) DUPL(' ',COL2 - (COL1 +
+ SIZE(STR1)))
LINE1 LINE = DUPL(' ',COL1 - 1) STR1 BL2 STR2
+
*
*
RANDOM RAN.VAR = RAN.VAR * 1061 + 3251
RAN.VAR RTAB(5) =
RANDOM = (RAN.VAR * N) / 100000 :(RETURN)
*
*
SUITL SUITL = SUIT ' '
SUITL1 SUITL = SUITL HAND<$SUIT,N> :F(RETURN)
N = N + 1 :(SUITL1)
*
*
CONSTANTS
*
CONSTANT S = 1
H = 2
D = 3
C = 4
$0 = 'A'
$1 = 'K'
$2 = 'Q'
$3 = 'J'
$4 = '10'
$5 = '9'
$6 = '8'
$7 = '7'
$8 = '6'
$9 = '5'
$10 = '4'
$11 = '3'
$12 = '2'
DEALSEQ = 'NORTH,EAST,SOUTH,WEST,NORTH,'
NXTHAND = *HAND , , BREAK( , , ) . HAND
DEALHAND = *DEALR , , BREAK( , , ) . HAND . DEALR
SUITRANK = LEN(1) . SUIT REM . RANK
EMPTYHAND = ARRAY('4,0:12')
NEWDECK = ARRAY('0:51')
RAN.VAR = 157

```

```

DEALMAX = 3
NTHDEAL =
DEALR = 'WEST'
N = 0
BLDDEK I = I + 1
R = 0
BLDDK1 NEWDECK<N> = I R
N = LT(N,51) N + 1 :F(NEWDEAL)
R = LT(R,12) R + 1 :S(BLDDK1)F(BLDDEK)
*
*
NEWDEAL NTHDEAL = LT(NTHDEAL,DEALMAX) NTHDEAL + 1
+ :F(END)
NORTH = COPY(EMPTYHAND)
EAST = COPY(EMPTYHAND)
SOUTH = COPY(EMPTYHAND)
WEST = COPY(EMPTYHAND)
*
DEAL()
*
DISPLAY()
END

```

Example (3) SNOBOL 4 - Bridge hand deal

Consider now a detailed analysis of the bridge hand deal. Here the morphism  $M$  maps an Ace into 0, a King into 1, etc. First observe the statement

$$\text{NEWDECK}\langle N \rangle = I R$$

Here  $R$  is used as a string whose numerical value corresponds to one of the ranks (Ace, King, etc). This use of 0, 1, ..., 12 to represent ranks is perfectly natural, i.e., a suit  $I$  and a rank  $R$  are concatenated to form a card. Second, observe Table 1, which explains the use of the indirect addressing operation  $\$$ . Here for example we see that if  $e$  is an expression whose value is 0, then  $\$$  applied to 0 will yield the string "ACE". Here  $M\langle \text{Ace} \rangle = \underline{0}$ , and  $M\langle \text{convert-to-print-name} \rangle = \$$ .

Now consider the statement

$$\text{ITEM}(\$ \text{HAND}, \text{SUIT}, \text{RANK}) = \$ \text{RANK}$$

Consider first the right hand side of the assignment. If say, the value of  $\text{RANK}$  is 0, we have

$$M\langle \text{convert-to-point-name}(\text{Ace}) \rangle = \$(\underline{0}) = \text{"ACE"}$$

$$\text{i.e.} \quad \text{convert-to-print-name}(\text{Ace}) = \text{"ACE"}$$

Now consider the left side of the assignment, using the string "WEST" as the value of  $\text{HAND}$ , the array name  $\text{WEST}$  as the value of  $\$ \text{HAND}$ , 1 as the value of  $\text{SUIT}$ , and 0 as the value of  $\text{RANK}$ .

In SNOBOL, the left-hand expression is then equivalent to the array reference.

$$\text{WEST}(\text{SUIT}, \text{RANK})$$

Then we have

$$\begin{aligned} M^{-1}\langle \text{WEST}(\underline{1}, \underline{0}) \rangle &= \text{WEST}(\text{SUIT}, \text{RANK}) \\ &= \text{WEST's Array } (\underline{1}, \underline{0}) \\ &\neq \text{WEST}(\underline{\text{Spades}}, \underline{\text{Ace}}) \\ &= \text{Undefined} \end{aligned}$$

Here there is an attempt to use the suit spades and the rank ace as subscripts of an array, which fails Property X in the space of cards. The use of 0 to represent an Ace, and the subsequent use of 0 as a subscript with its attendant numeric or string properties simply results in tricky programming.

Another violation of naturality also occurs in the same statement. Here the direct value of  $\text{HAND}$  is one of the strings "NORTH," "SOUTH," "EAST," and "WEST." These strings are used as the names of the dealer and names associated with a dealt hand. As names, these values can of course be printed. On the other hand, the indirect value of  $\text{HAND}$  is the internal (to the program) name of a 52 element array, 13 of whose elements

Table 1: Meaning of the \$ Operator in SNOBOL

Let  $v$  be the value of  $e$ .

(1) \$ operator on the left-hand side of an assignment statement

if  $v \in \text{NAME}$  then convert-to-name o look-up-string-value ( $v$ )

$\$_{\text{LHS}}(e) =$  if  $v \in \text{STR}$  then convert-to-name ( $v$ )

if  $v \in \text{NUM}$  then convert-to-new-name ( $v$ )

(2) \$ operator on the right-hand side of an assignment statement

$\$_{\text{RHS}}(e) =$  lookup-string-value o  $\$_{\text{LHS}}(e)$

(3) Meaning of \$ in bridge deal program

if  $v = \underline{0}$  then Ace

$\$(e) =$  if  $v = \underline{1}$  then King

·  
·  
·

if  $v = \underline{12}$  then Deuce

contain the cards dealt to a player. This is a violation of Property X. First, M<WEST> equals the string "WEST" which is used in the program to name a particular hand (with its attendant print value, the name of a player), and which is subsequently used as the internal name or location of an array containing the cards within that hand.

There are several other violations of Property X in the SNOBOL program. For example, the indirect value, v, of HAND is a 52 element array, not a 13 element array. This is a violation of Property X in the sense that in the abstract space there is no object corresponding to a bridge hand that contains 52 elements, some of which are blank. Rather, a complete bridge hand should contain 13 elements, none of which is blank.

You may ask then, when is indirect addressing "natural"? The answer is simple, when the \$ operator is applied to an abstract object that is, in fact, an address, or the location of a place which contains a "value".

What about interpreting the strings "NORTH," "SOUTH," "EAST" and "WEST" as addresses whose contained values are hands of cards? This is fine, except for one serious violation. If these strings represent internal names, addresses, or locations, then they should not be used as external players' names, nor should they be printed or used in patterns. So the violation of Property X under this interpretation occurs in the output routines or in the pattern matching statements.

Other violations of Property X can be found in various ways. For example, consider the following two pieces of text.

(4a)         $P \leftarrow 1 + P * \text{SIGN}(4 - P)$

(4b)        if  $P < 4$         then  $P \leftarrow 1 - P$

else if  $P = 4$  then  $P \leftarrow 1$

else if  $P > 4$  then  $P \leftarrow 1 + P$

Example (4a) is confusing due to the trick of multiplying the value of the sign function times the value of P. The value of the sign function should be the string "+" or "-". Conceptually, multiplication by a "sign" has no natural analog in the traditional space of abstract arithmetic operations.

A slightly different violation of Property X arises when operations performed in a program correspond in fact to operations in the abstract space, but not those operations expected in practice. In the SNOBOL program for example, a call to DEAL results in an unusual algorithm for "dealing". This algorithm performs a shuffle and deal by randomly picking a card from the deck, dealing it to the next player, taking the

Table 2: The Programming Proverbs

### APPROACH TO THE PROGRAM

1. Clearly define your problem.
2. Think first, program later.
3. Use the top-down approach.
4. Beware the "imitation" approach.

### CODING THE PROGRAM

5. Construct your program in logical units.
6. Use functions and subroutines.
7. Avoid unnecessary GOTO's.
8. Avoid side effects.
9. Get the syntax right now, not later.
10. Use Mnemonic identifiers.
11. Use intermediate variables properly.
12. Leave loop variables alone.
13. Do not recompute constants within a loop.
14. Avoid implementation dependent features.
15. Avoid tricks.
16. Build in debugging techniques.
17. Never assume the computer assumes anything.
18. Use comments.
19. Prettyprint.
20. Provide documentation.

### RUNNING THE PROGRAM

21. Check your program before running it.
22. Get your program correct before trying to produce good output.
23. When your program is correct, produce good output.

### GENERAL

24. Reread your manual.
25. Consider another language.
26. Don't be afraid to start over!



top card from the deck and inserting it in the space vacated by the dealt card. While this is not a strict violation of naturality, the "deal" does not really correspond to the way cards are dealt at a bridge table. However, on most machines this "deal" is efficient in speed.

This leads us to a fundamental point about Property X. There are cases where "unnatural" methods are in fact justified, for example for efficiency of execution or economy of storage. However, before we resort to unnatural programming, we should have a clear reason for doing so. Otherwise the programmer should stick to operations and objects that have a natural analog in the abstract space.

## DISCUSSION

Q - I've noticed that a lot of programmers wait until the end to add comments. Does that make any difference to you?

A - I don't know. Most experts claim for example, that it's better to read a book several times before annotating it.

Comment - When time and money are running out, the comments may not get in.

Q - There's another problem--if you comment first and then change the program without changing the comments you're in worse shape than with no comments at all.

A - When you're writing a program, comments could be of the kind "N = 18 at the end of the loop," whereas you would use an entirely different set while documenting. We'll notice that many of our principles are conflicting and this is one of the problems we will have to consider.

Comment - Parnas in the latest CACM, pushes the view that you should hide the semantic roles of identifiers to force better understanding.

Q - To say "avoid side effects" sounds like instructing the enlisted man not to drop a hammer on his toe.

A - Yes, but the programmer did just this and wasn't even aware of side effects.

Comment - Even so, if he had thought about it he should have declared "I" local.

Q - Where should the control be? Should the language be such that you can't hang yourself up that way, or should it be up to the programmer, or should it be up to the supervisor?

A - Yes.

Q - Perhaps the compiler, instead of making implicit assumptions, should force the programmer to declare everything. In this way the programmer would have made the right choice automatically.

Comment - There's a compromise--the compiler could tell you what you've done and what it has done for you.

A - Those are awfully verbose solutions.

Comment - It could be an option which could be turned off.

A - My own masters thesis is the ultimate side effect. In two boxes of cards I have over 50 variables in COMMON.

Q - Some side effects are good. For example, most random number generators are pure side effect.

A - That's true. My point is that side effects should be present because of a conscious decision to use them.

## 5. ACKNOWLEDGEMENT

I am grateful to Lee Hoeval, Rao Kosaraju, Amos Gileadi, and Eliot Solloway, for many insightful discussions on the issues contained herein.

S. Rao Kosaraju

*Discussions on problem and program specification provide an introduction to a review of proof-of-correctness techniques. Then, noting some practical limitations on proof-of-correctness, the author goes on to examine selected facets of program synthesis.*

## 1. PROBLEM AREAS

In the general area of theory of computer programming we can isolate some problems of practical importance:

1. Is a given program syntactically good? (Recognition Problem). A significant amount of work was done in this area. There are well-known efficient algorithms for recognition of programming languages.

2. If we know a program to be syntactically good, what does it compute? That is, can we give a suitable functional description of what it computes? (Semantic Recognition and Elaboration Problem).

3. Is a given program the "best program" for the function it computes; if not, can we write a "significantly better" program? That is, can it be optimized in terms of some measure like time or storage? This is known as Optimization or Speed-up.

4. Given any program P and any function f (suitably described), does P compute f? That is, does a program solve the problem it is claimed to solve? This is one of the major problems we will be addressing in this lecture and is known as establishing "Correctness of Programs."

5. Does a given program halt for every one of the allowable inputs? We might call it "Termination Problem" or "Uniform Halting Problem."

6. Given any two programs, do they compute the same function? This we call establishing "Equivalence of Programs."

There are many other problems of utmost significance in the theory of computer programming. From a purely theoretical point of view most of the above problems are undecidable in the general sense. But that should not deter us from considering such problems. Because of their practical importance, any partial results which could isolate some significant decidable sub-sets are worth the effort. In this lecture we shall concentrate on problems 4, 5, and 6 and the following problem:

7. Given a suitable functional description of a problem, can we write a program which solves that problem? This we call "Writing Correct Programs." This is entirely different from problem 4. In correctness of programs we are given the function  $f$  and also the program  $P$ , whereas in this problem we are given the function  $f$  only, and we have the freedom to synthesize a suitable  $P$ . We might not be able to decide whether any given program computes  $f$ ; but we might be able to establish that a suitably selected program computes  $f$ .

## 2. PROBLEM SPECIFICATION

### 2.1 Static Versus Dynamic

All along we have assumed that we are given a "suitable" functional description of the problem. What are the actual implications of such an assumption? Consider the following functions:

1.  $f$  is defined for all natural numbers and

$$f(x) = 2x$$

2.  $g$  is defined for all pairs of natural numbers and

$$g(x, y) = \begin{cases} 2y & \text{if } x \text{ considered as a program string} \\ & \text{halts on input } x \\ 0 & \text{otherwise (i.e. } x \text{ is not a syntactically} \\ & \text{good program or does not halt on } x) \end{cases}$$

I am sure nobody will object to the description of the problem  $f$ . What about  $g$ ? For any constant  $x$ ,  $g$  defines either the function  $f$  or the constant function 0; but we cannot see, except by simulation for instance, whether or not the program  $x$  halts on data  $x$ . We denote this difference by saying that  $f$  is defined statically, while  $g$  is defined dynamically.

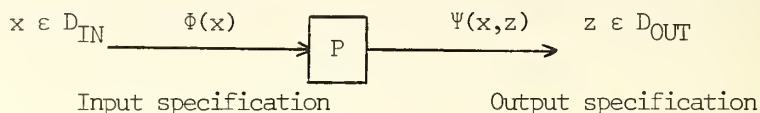
### 2.2 Informal Versus Formal

We may be concerned with the degree of formality with which a problem is described.

In a gross sense, most problems are first described informally, and in a natural language rather than a precise formal language. In particular, programmers are usually asked to solve problems with informal, static definitions. Thus before beginning to write a program, programmers normally have a general picture of the overall problem. In discussing such problems, however, we will want to be able to test whether a particular description is acceptable or not. For this, we will have to describe the problem in a precise formal language.

There are various formal languages, e.g. Turing Machines, Markov Algorithms, Predicate Calculus, Lambda Calculus, Post Normal Forms, Programming Languages. Computationally all these languages describe exactly the same class of problems.

Conceptually, we can make use of the following diagram, which shows the general form of a "problem description."



input vector =  $x$

output vector =  $z$

input domain =  $D_{IN}$  (non-empty)

output domain =  $D_{OUT}$  (non-empty)

$\phi$  is a total predicate over  $D_{IN}$ , i.e. for every  $x \in D_{IN}$ ,

$\phi(x)$  has the value T(True) or F(False)

$\psi$  is a total predicate over  $D_{IN} \times D_{OUT}$

Intuitively, for any input  $x$  from  $D_{IN}$  and satisfying  $\phi$  (i.e.  $\phi(x) = T$ ), the output  $z$  satisfies  $\psi$  (i.e.  $\psi(x,z) = T$ ). Any input  $x$  from  $D_{IN}$ , and satisfying  $\phi$ , is a valid input.

The problems we are interested in transform any valid input to a unique output; that is, we assume that if  $\psi(x,z_1) = T$  and  $\psi(x,z_2) = T$ , then  $z_1 = z_2$ . Thus we could also describe  $\psi$  by a function  $f$ :

$$f(x) = \begin{cases} z & \text{if } \psi(x,z) = T \\ \text{undefined} & \text{if there exists no } z \text{ such that } \psi(x,z) = T \end{cases}$$

## 2.3 Example

$D_{IN}, D_{OUT} = \mathbb{N}$  (the set of natural numbers)

$\phi(x) \equiv (x \in \mathbb{N})(x = 2i + 1)$

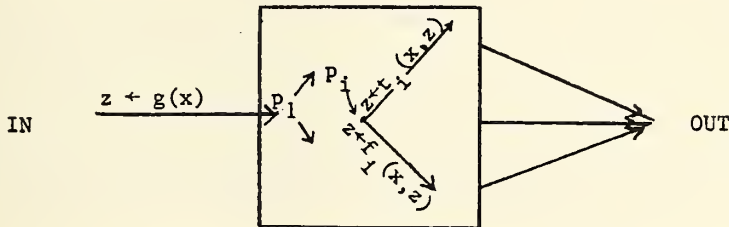
$\psi(x,z) \equiv (z = x^2)$

In this problem, the valid inputs will be the set of odd natural numbers, and  $f$  is a representation for a function which squares its argument.



### 3. PROGRAM SPECIFICATION

Let us say that a program is composed of a set of instructions in some programming language. We can assume that each of these instructions is either a functional assignment statement (basic function) or a conditional statement (basic predicate). Without loss of generality we can assume that each conditional is a binary predicate. We want to consider flow chart representations for such programs in which each conditional in the textual form corresponds to a unique predicate node (many in, 2 out), and assignment statements are collapsed, by composition, into functions associated with the lines in the flow chart. There will be a finite number of nodes, one for the IN (0 in, 1 out) of the program, and one for the OUT (many in, 0 out), and one for each of the  $n$  predicates  $P_i$ ,  $i = 1, \dots, n$ . It is easily seen that there are  $2n + 1$  branches in a flow chart with  $n$  predicate nodes. For simplicity, we will assume that internal variables are part of the output vector of the program; the modification of this treatment to agree with the more complicated conventions in the literature should be obvious [Manna, 1969]. Thus, any program can be represented as follows:



where  $z \leftarrow t_i(x, z)$  is the assignment associated with the "true" branch of the  $i$ -th predicate;  $z \leftarrow f_i(x, z)$  is the assignment associated with the "false" branch of the  $i$ -th predicate;  $x$  and  $z$  are the input and the output vectors respectively.  $z \leftarrow g(x)$  is an initialization step, which we will assume transfers control to node  $P_1$ .

Note that the input vector  $x$  never gets changed. A computation starts with control at IN, and when control passes to the node OUT, we say that the computation is terminated.

### 4. PROOF OF CORRECTNESS VERSUS WRITING CORRECT PROGRAMS

#### 4.1 Proof of Correctness (POC)

( $\forall p$ , a program) ( $\forall f$ , a problem) (Does  $P$  satisfy the input-output specifications of  $f$ ?)



## 4.2 Writing Correct Programs (WCP)

( $\forall f$ , a problem) (Find a program  $P$ , which satisfies the input-output specifications of  $f$ )

There is a central theoretical distinction between POC and WCP:

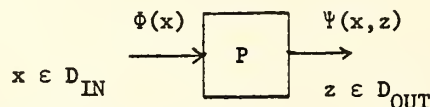
If the problem  $f$  is defined in a formal system, then

- a) POC is undecidable                      and
- b) WCP is possible

I hasten to add that it is a trivial theoretical result and has no practical significance. The trouble is in the assumption that the problem is described formally; the main problem in WCP is formalizing an informal specification of the problem.

- Q: How could you be sure you could write a correct program, if in fact you could not prove a program to be correct?
- A. For any problem  $f$ , if I randomly select a program  $P$ , I might not be able to show that  $P$  computes  $f$ . For some  $P$ 's I will be able to verify correctness, however, so I write only such a program. More specifically, I will give a mechanism for converting descriptions in any given formal system into equivalent descriptions in the given programming language (another formal system); further, I will show that the mechanism works correctly. Thus, we can convert any problem in the formal system into a correct program.
- Q. POC problem seems to be too general. We would not go to this broad class of programs. POC problem amounts to determining whether any given screwball program is correct. But we are usually interested only in programs that look to be almost correct.
- A. I completely share your view. I do not know any investigations along the lines you suggest. In POC we consider a program without assuming to know the intuition behind its design. In WCP the person writing the program takes on himself the responsibility that every "step" he takes is "safe."

## 5. PROOF OF CORRECTNESS



If  $x$  is the input to program  $P$ , then  $P(x)$  represents the output of the program.

Def: Correctness: P is correct for the given input-output specifications iff for any valid input x, P(x) is defined and  $\Psi(x, P(x)) = T$ .

Def: Partial Correctness: P is partially correct for the given input-output specifications iff for any valid input x, if P(x) is defined, then  $\Psi(x, P(x)) = T$ .

Thus in partial correctness we verify the correctness of the output only if the program halts. Note that if the program never halts for any valid input, then it is partially correct, by default.

Def: Termination: Program P terminates iff for every valid input x, P(x) is defined. Thus if P is partially correct and terminates, then P is correct.

There are various methods to establish the correctness of programs. In another lecture we give an outline of the main techniques, but today we restrict ourselves to one of them - Proof by Inductive Assertions.

Proving correctness can be traced to Goldstine and von Neumann [1961]. But the first treatment in any great detail is by Gorn [1968], who gives two principles; one for proving partial correctness, and the other for establishing termination.

### 5.1 Principle of Command Recursion (Partial Correctness)

"Any function or property of the contents of a storage area which is left invariant by every sub-processor of a processor is left invariant by the whole processor."

### 5.2 Principle of Monotony (Termination)

"If a processor with a single exit has a sub-processor (in fact, a discriminator) which must be invoked again after each use not leading to the exit, and if there is a function of the storage whose range is the set of natural numbers and such that its value at each application of the discrimination is less than at the preceding, then the processor must conclude."

Let us apply these principles to the following program (Gorn):

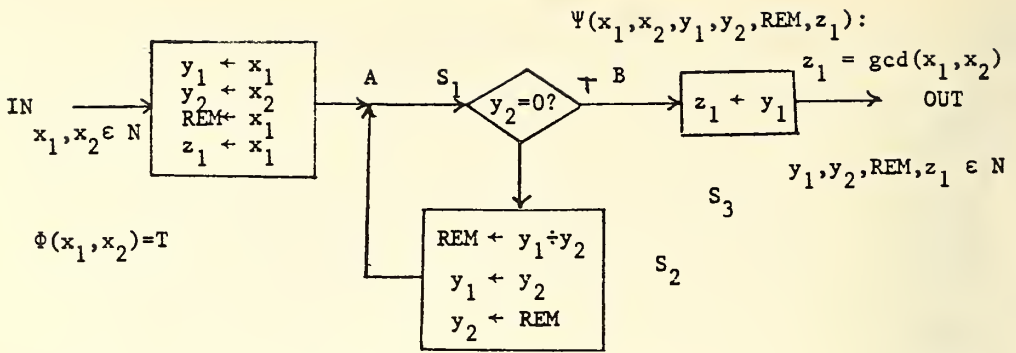


Figure 1

Input variables:  $x_1, x_2$

Output variables:  $y_1, y_2, \text{REM}, z_1$

( $\text{REM} \leftarrow y_1 \div y_2$  results in assigning remainder of  $y_1 \div y_2$  to  $\text{REM}$ )

We want to prove that the program is correct for the above specifications.

### Partial Correctness:

It is easy to verify that the condition " $\text{gcd}(y_1, y_2) = \text{gcd}(x_1, x_2)$ " is left invariant by both  $S_1$  and  $S_2$ . Hence, by Command Recursion, the same condition is invariant for the program from A to B. If we substitute the initial assignment, the condition  $y_2 = 0$  at B, and the final assignment, we get " $z_1 = \text{gcd}(x_1, x_2)$ " at the output. This establishes partial correctness.

### Termination:

For the program from A to B, " $y_2 = 0?$ " is the test condition (discriminator) and the sub-program  $S_2$  always results in decrementing  $y_2$  (but always results in a natural number). Hence the program from A to B terminates ( $y_1$  and  $y_2$  are natural numbers initially).

Notice that if we replace  $S_2$  by the single assignment statement  $y_2 \leftarrow y_2 + y_1$ , the above partial correctness proof remains valid ( $\text{gcd}(y_1, y_2) = \text{gcd}(y_1, y_1 + y_2)$ ); but the proof of termination breaks down ( $y_2$  non-decreasing).

This method uses a global property which has to be left invariant by

all applicable local transformations. In practice, it is often very tricky to find this global "invariant" property.

### 5.3 Inductive Assertions Method

This method, advanced by Floyd [1967] and Naur independently, is another major technique in this area. There is a tendency to credit it to Goldstine and von Neumann [1961].

The principle is quite simple and you should easily be able to convince yourself of its validity. We select suitable check points, including IN and OUT, in the program. At each check point assert a "suitable" relationship between the variables. Verify that at any instant, if the assertion at the point of control is satisfied, then when the next check point is reached the corresponding assertion will be satisfied. If the assertions at the IN and the OUT nodes are  $\Phi(x)$  and  $\Psi(x,z)$  respectively, then this verification procedure will establish partial correctness; so in order to take advantage of the simplifications possible using this approach, we need local assertions instead of the global assertion of Gorn.

Some obvious ways of selecting the check points are:

- 1) Select all the nodes (predicate) of the program.
- 2) Select nodes such that no closed path exists without a check point.

Q. Theoretically you do not need so many check points.

A. In fact, we need only two check points--the IN and the OUT nodes; in which case we gain nothing by this method!

Let us apply this technique to the program given in Figure 1. In addition to IN and OUT, select A as a check point. Let the assertion at A,  $\Phi_A$ , be:

$$\Phi_A(x_1, x_2, y_1, y_2, REM, z_1): \quad "gcd(x_1, x_2) = gcd(y_1, y_2)".$$

If control is at IN then we need to verify that

$$T \supset \Phi_A(x_1, x_2, x_1, x_2, x_1, x_1)$$

i.e. we have to verify that  $gcd(x_1, x_2) = gcd(x_1, x_2)$ .

This is obviously true.

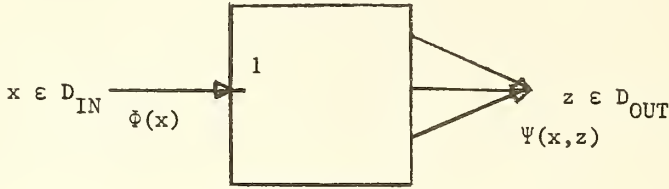
If control is at A, then we need to verify that

Path  $S_1S_2$ :  $\Phi_A(x_1, x_2, y_1, y_2, \text{REM}, z_1) \supset (y_2 \neq 0 \supset \Phi_A(x_1, x_2, y_2, y_1 \div y_2, y_1 \div y_2, z_1))$

Path  $S_1S_3$ :  $\Phi_A(x_1, x_2, y_1, y_2, \text{REM}, z_1) \supset (y_2 = 0 \supset \Psi(x_1, x_2, y_1, y_2, \text{REM}, y_1))$

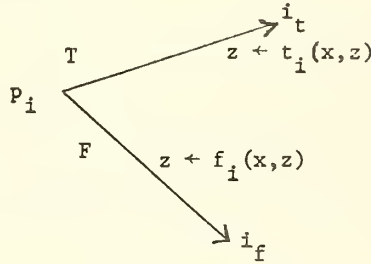
Both are easy to verify. Thus the program is partially correct.

In the following we give a complete discussion of partial correctness and termination. We use the program model developed earlier. The program has  $n + 2$  nodes: IN, OUT and nodes  $1, 2, \dots, n$ .



Select the  $n + 2$  nodes as check points and associate assertion  $q_i$  with node  $i$ ,  $i = 1, \dots, n$

If the  $i$ th node is of the form



then the verification condition,  $W_i$ , for node  $i$  is:

$$W_i: (\forall z)(q_i(x, z) \supset (p_i(x, z) \supset q_{i_t}(x, t_i(x, z)) \ \& \ (\sim p_i(x, z) \supset q_{i_f}(x, f_i(x, z))))$$

(If  $i_t$  or  $i_f$  is the OUT node, then replace the corresponding  $q_{i_t}$  or  $q_{i_f}$  by  $\Psi$ .)

Thus we obtain the verifications conditions  $W_1, W_2, \dots, W_n$ .

For the node IN the verification condition  $W_0$  is  $\Phi(x) \supset q_1(x, g(x))$ .

Let  $[P, \Psi](x)$  be  $W_0 \wedge W_1 \wedge \dots \wedge W_n$ .

Theorem:  $P$  is partially correct for  $\Phi$  and  $\Psi$  iff

$$(\forall x \in D_{\text{IN}}) ([P, \Psi](x)) \text{ is satisfiable.}$$

Proof:  $\Leftarrow$  :

If the condition is satisfiable (i.e. we are able to select suitable assertions -  $q_i$ 's) by trivial induction along the path,  $P$  is partially correct.

$\Rightarrow$ :

If  $P$  is partially correct, we give an interpretation for each  $q_i$ , such that the above condition is satisfied. For any valid input  $x$ , control might be at node  $i$  at any instant. At such instants let  $z$  have values  $z_1, z_2 \dots$

Let  $q_i(x, z_1) = q_i(x, z_2) = \dots = T$ , and for any other  $z$

$q_i(x, z) = F$ , be the interpretation for  $q_i$ .

Verify trivially that this interpretation satisfies the above condition.

Theorem:  $P$  is correct for  $\Phi$  and  $\Psi$  iff

$$(\exists x \in D_{\text{IN}}) (\Phi(x) \wedge [P, \sim\Psi](x)) \text{ is not satisfiable. [Manna, 1969].}$$

Proof:  $\Rightarrow$  :

We want to show that if  $P$  is correct, then the condition cannot be satisfied. Assume that the condition is satisfiable. The  $x$  satisfying the condition is a valid input, since  $x \in D_{\text{IN}}$  and  $\Phi(x) = T$ . Hence  $P(x)$  is defined. Hence, from the condition we have  $\sim\Psi(x, P(x)) = T$ , which gives  $\Psi(x, P(x)) = F$ . Thus  $P$  cannot be correct, a contradiction.

$\Leftarrow$ :

We want to show that if the condition is not satisfiable, then  $P$  is correct. Let  $P$  be incorrect. It could be because 1) there exists a valid input for which  $P(x)$  is not defined or 2) for some  $x$ ,  $P(x)$  is defined and  $\Psi(x, P(x)) = F$  i.e.  $\sim\Psi(x, P(x)) = T$ .

In either case it is easily seen that the interpretations given in the previous theorem satisfy the above condition, a contradiction. QED. Observe that if  $P$  does not terminate, then  $(\exists x \in D_{\text{IN}})(\Phi(x) \wedge [P, \sigma](x))$  is satisfiable for any output assertion  $\sigma$ .

This technique could become very involved if you have even a moderately large program. There is no guarantee that we will not make a mistake in a correctness proof. At the end of the whole experience you might be more at ease with the program than the proof.



## 6. WRITING CORRECT PROGRAMS

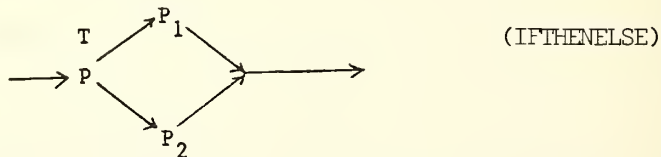
Proof of correctness seems to be very complex even for moderate size programs. Is there any other way of handling the "complexity of large problems?"

Dijkstra [1970] advocates top-down programming and a highly structured discipline of program synthesis. In one step of a program synthesis we decompose the given problem into a small number of sub-problems, with suitable input-output specifications for each sub-problem. We establish correctness of the given problem, assuming the correctness of the sub-problems, and repeat the procedure for each of the sub-problems until the basic instruction level is reached. The structure of the final programs is highly characterized; hence the name structured programming.

Some people advocate specifically, restricting the decompositions to the following:

1. Decompose  $\rightarrow P \rightarrow$  into  $\rightarrow P_1 \rightarrow P_2 \rightarrow$  (BLOCK)

2. Decompose  $\rightarrow P \rightarrow$  into



3. Decompose  $\rightarrow P \rightarrow$  into



The programs synthesized through the above decompositions are known as D-programs (D for Dijkstra). At this time there is no convincing evidence that shows this severely restricted decomposition to be of much pragmatic value. It seems that we need more powerful decompositions for "facility" in programming, although theoretically, the above decompositions have sufficient computing power. There is, however, a great deal of merit in the general philosophy of the structured approach to programming.

We could derive partial correctness and termination conditions for any decomposition by using the assertions method. For simple structures like the above, however, we can write correctness criteria by observation as follows [Mills, 1972]:

Assume that the problem is specified by a function  $P$  and input specifications  $\Phi$ , with input domain = output domain =  $D$

$$x \in D \xrightarrow{\Phi(x)} P \longrightarrow$$

We also assume that there is only one vector  $x$ , which gets transformed by the problem. (A notable difference from the previous approach.) Thus for any valid input  $x$ , the output is defined, and is  $P(x)$ .

## 6.1 Structure 1

$$x \in D \xrightarrow{\Phi} P \longrightarrow \text{is decomposed into} \quad x \in D \xrightarrow{\Phi} P_1 \xrightarrow[\Phi_1]{x \in D} P_2 \longrightarrow$$

Then the only conditions to be satisfied are

$$\Phi(x) \supset \Phi_1(P_1(x))$$

$$\text{and} \quad (x) \supset P(x) = P_2(P_1(x)).$$

there is a lot of flexibility in selecting  $P_1$  and  $P_2$ . Each of them should have "less complexity" than  $f$ .

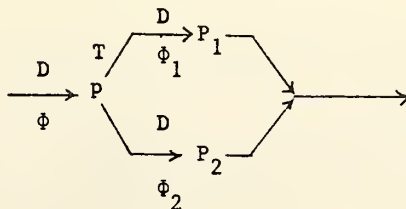
Q. Is there any strategy in the selection of  $P_1$  and  $P_2$ ?

A. Not in an absolute sense.

Q. What do you mean by "less complexity?"

A. I do not have a precise definition. Qualitatively the less complex problem is easier to realize. If we select  $P_1 = \text{id}$  function and  $P_2 = P$  then the correctness conditions are satisfied. But  $P_2$  has the same complexity as  $P$ .

## 6.2 Structure 2



The correctness conditions are

$$\Phi(x) \wedge p(x) \supset \Phi_1(x)$$

$$\Phi(x) \wedge \sim p(x) \supset \Phi_2(x)$$

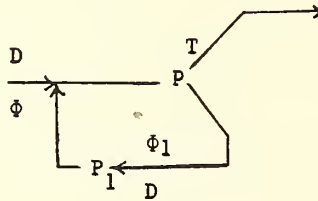
$$\Phi(x) \wedge p(x) \supset P(x) = P_1(x)$$

$$\Phi(x) \wedge \sim p(x) \supset P(x) = P_2(x)$$

Thus for a valid input  $x$ , if  $p(x) = T$ , then  $P_1(x) = P(x)$  and if  $p(x) = F$ , then  $P_2(x) = P(x)$ .

$P_1$  and  $P_2$  may be defined for other values in order to simplify  $P_1$  and  $P_2$  as much as possible.

### 6.3 Structure 3



The partial correctness conditions are:

$$\Phi(x) \wedge \sim p(x) \supset \Phi_1(x)$$

$$\Phi(x) \wedge \sim p(x) \supset P(x) = P(P_1(x))$$

$$\Phi(x) \wedge p(x) \supset P(x) = x$$

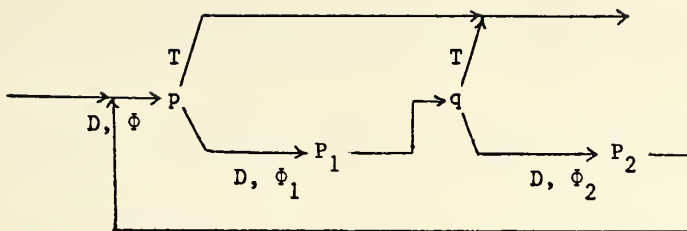
The termination condition is:

$$(\forall x) (\Phi(x)) \supset (\exists i \geq 0) (p(P_1^i(x)) = T)$$

where  $P_1^0(x) = x$ .

The termination condition derived here is similar to the principle of monotony of Gorn.

We should be extra careful with this structure, since it requires a non-trivial termination condition to be satisfied. If we measure the complexity of a control structure by the "complexity" (in a quality sense) of the termination condition, then the following structure seems to have almost "the same complexity" as the previous structure:



Termination condition:

$$(\forall x)(\phi(x) \supset (\exists i \geq 0)(p(P_2 P_1)^i(x)) \vee q(P_1(P_2 P_1)^i(x))).$$

#### 6.4 Conclusions

It might prove to be fruitful to systematically study the problem of complexity of structures. This approach might prove to be useful in the design of control structures in a programming language.

Co-routine construction seems to be much more complex than the structures we have considered above. The Repeat-Exit constructs of Knuth and Floyd [1970], deserve further study because they are useful in a pragmatic sense, and yet do not seem to complicate the techniques developed for analyzing D-programs.

## BIBLIOGRAPHY

1. Ashcroft, E. and Manna, Z., The Translation of "Go To" Programs, Booklet TA-2, Preprints of the IFIPS Congress 1971, Ljubljana, Yugoslavia, August 1971 (North Holland Publishing Co., Amsterdam, 1971).
2. Baker, F. T., Chief Programmer Team Management of Production Programming, IBM Syst. J. 11, No. 1, pp. 56-73 (1972).
3. Bohm, C. and Jacopini, G., Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, CACM 9, No. 5, pp. 366-371 (May 1966).
4. Buxton, J. N. and Randell, B. (Editors), Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, Oct. 27-31, 1969 (available from Scientific Affairs Division, NATO, Brussels 39, Belgium) (1970).
5. Computer Science Research Review, 1970-71 (Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1971).
6. Cooper, D. C., Bohm and Jacopini's Reduction of Flow Charts, CACM 10, No. 8, pp. 463, 473 (Aug. 1967).
7. Dijkstra, E. W., A Constructive Approach to the Problem of Program Correctness, BIT 8, pp. 174-186 (1968).
8. Dijkstra, E. W., Go To Statement Considered Harmful, CACM 11, No. 3, pp. 147-148 (Mar. 1968).
9. Dijkstra, E. W., Notes on Structured Programming, T. H. Report 70-WSK-03 (Technological University, Eindhoven, Netherlands, 1970).
10. Floyd, R. W., Assigning Meanings to Programs, Proceedings of the Symposium on Applied Mathematics 19, pp. 19-32 (American Math. Society, Providence, R. I., 1967).
11. Goldstine, H. H. and Von Neumann, J., Planning and Coding Problems for an Electronic Computing Instrument, Collected works of John Von Neumann, Vol. 5, pp. 80-104 (Pergamon Press, New York, 1961).
12. Gorn, S., The Identification of the Computer and Information Sciences: Their Fundamental Semiotic Concepts and Relationships, Foundations of Language 4, pp. 339-372 (1968).
13. Griswold, R. E., Poage, J. F., and Polansky, I. P., The SNOBOL 4 Language (Prentice-Hall, Inc., Englewood Cliffs, N. J., 1968).

14. Henderson, P. and Snowdon, R., An Experiment in Structured Programming, BIT 12, pp. 38-53 (1972).
15. Knuth, D. E. and Floyd, R. W., Notes on Avoiding "Go To" Statements, Report No. SC-148 (Computer Science Department, Stanford University, 1970).
16. Kosaraju, S. R., Analysis of Structured Programs, Technical Report (The John Hopkins University, Nov., 1972).
17. Lyon, G. E., Least Errors Recognition of Mutated Context Free Sentences in Time  $n^3 \log n$ . Proceedings, Sixth Princeton Conference on Information Systems and Sciences, pp. 115-118 (Mar. 1972).
18. Manna, Z., The Correctness of Programs, JCSS 3, No. 2, pp. 119-127 (May 1969).
19. Mills, H. D., Mathematical Foundations for Structured Programming, FSC 72-6012 (Federal Systems Division, IBM Corp., Gaithersburg, Md., 1972).
20. Parnas, D. L., A Technique for Software Module Specification with Examples, CACM 15, No. 5, pp. 330-336 (May 1972).
21. Peach, P., Quality Control for Computer Programming, A Final Report on an Initial Study (SDC Tech. Memo for the Defense Documentation Center, 1965); AD642303.
22. Quine, W., Word and Object (Technology Press of the Massachusetts Institute of Technology, Cambridge, 1960).
23. Sammet, J. E., Programming Languages: History and Fundamentals (Prentice-Hall, Inc., Englewood Cliffs, N. J., 1969).
24. Smith, Don, Jr., An Organization for Successful Project Management, AFIPS Conference Proceedings, Vol. 40, Spring Joint Computer Conference, Atlantic City, N. J., May 16-18, 1972, pp. 129-140 (AFIPS Press, Montvale, N. J., 1972).
25. Tou, J. T. (Editor), Software Engineering, COINS III, Vol. I, Proceedings of the Third Symposium on Computer and Information Sciences, Miami Beach, Florida, Dec. 1969 (Academic Press, New York, 1970).
26. Weinberg, G. M., The Psychology of Computer Programming (Van Nostrand Reinhold Company, New York, 1971).
27. Wirth, N., Program Development by Stepwise Refinement, CACM 14, No. 4, pp. 221-227 (Apr. 1971).



28. Wirth, N., The Programming Language PASCAL, Acta Informatica 1, pp. 35-63 (1971).
29. Wulf, W. A., A Case Against the GOTO, Proceedings of the ACM Annual Conference, Boston, Aug. 1972, pp. 791-797 (Association for Computing Machinery, New York, 1972).
30. Wulf, W. A., ELISS Reference Manual (Computer Science Department, Carnegie-Mellon University, Pittsburgh, rev. Oct., 1971).

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET		1. PUBLICATION OR REPORT NO.  NBS TN-842	2. Gov't Accession No.	3. Recipient's Accession No.
4. TITLE AND SUBTITLE  CONCEPTS IN QUALITY SOFTWARE DESIGN			5. Publication Date  August 1974	
			6. Performing Organization Code	
7. AUTHOR(S) S. L. Stewart (editor)			8. Performing Organ. Report No.	
9. PERFORMING ORGANIZATION NAME AND ADDRESS  NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE WASHINGTON, D.C. 20234			10. Project/Task/Work Unit No. 6401121	
			11. Contract/Grant No.	
12. Sponsoring Organization Name and Complete Address (Street, City, State, ZIP)  Same as 9			13. Type of Report & Period Covered  Interim 1972	
			14. Sponsoring Agency Code	
15. SUPPLEMENTARY NOTES  Library of Congress Catalog Card Number: 74-600130				
16. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)  A seminar series on quality software, sponsored by the Systems and Software Division, was held at the National Bureau of Standards during the summer of 1972. This Note includes five of these seminars in edited form. (I) A brief background provides motivation for studies in software quality. The authors mention some factors which influence software manufacture, and propose measures which might quantify concepts of "software quality". Several approaches to establishing program correctness receive attention. (II) Elements of top-down programming are sketched out and then examined in detail. An extended critique of another top-down experiment provides example material. (III) Powers of various structured control constructs are compared within a framework of weak and strong program equivalence. Results include a demonstration that Dijkstra's D-programs are strongly equivalent to programs built from functions and one-input/two-output predicates. (IV) After a review of Quine's notion of referential transparency, the author examines elements of good and bad programming practice. In addition, a table of programming proverbs provides guidance to a programmer, and should be especially useful to a novice. (V) Discussions on problem and program specification provide an introduction to a review of proof-of-correctness techniques. Then, noting some practical limitations on proving correctness, the author goes on to examine selected facets of program synthesis.				
17. KEY WORDS (six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons) Control structures; GOTO-less programming; program validation; programming; proofs of correctness; referential transparency; software quality; structured programming; top-down programming.				
18. AVAILABILITY  <input type="checkbox"/> For Official Distribution. Do Not Release to NTIS  <input checked="" type="checkbox"/> Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13. 46:843  <input type="checkbox"/> Order From National Technical Information Service (NTIS) Springfield, Virginia 22151		19. SECURITY CLASS (THIS REPORT)  UNCLASSIFIED		21. NO. OF PAGES  89
		20. SECURITY CLASS (THIS PAGE)  UNCLASSIFIED		22. Price  \$1.25







# NBS TECHNICAL PUBLICATIONS

## PERIODICALS

**JOURNAL OF RESEARCH** reports National Bureau of Standards research and development in physics, mathematics, and chemistry. Comprehensive scientific papers give complete details of the work, including laboratory data, experimental procedures, and theoretical and mathematical analyses. Illustrated with photographs, drawings, and charts. Includes listings of other NBS papers as issued.

Published in two sections, available separately:

### • Physics and Chemistry (Section A)

Papers of interest primarily to scientists working in these fields. This section covers a broad range of physical and chemical research, with major emphasis on standards of physical measurement, fundamental constants, and properties of matter. Issued six times a year. Annual subscription: Domestic, \$17.00; Foreign, \$21.25.

### • Mathematical Sciences (Section B)

Studies and compilations designed mainly for the mathematician and theoretical physicist. Topics in mathematical statistics, theory of experiment design, numerical analysis, theoretical physics and chemistry, logical design and programming of computers and computer systems. Short numerical tables. Issued quarterly. Annual subscription: Domestic, \$9.00; Foreign, \$11.25.

**DIMENSIONS/NBS** (formerly *Technical News Bulletin*)—This monthly magazine is published to inform scientists, engineers, businessmen, industry, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on the work at NBS.

**DIMENSIONS/NBS** highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, **DIMENSIONS/NBS** reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing.

Annual subscription: Domestic, \$6.50; Foreign, \$8.25.

## NONPERIODICALS

**Monographs**—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

**Handbooks**—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

**Special Publications**—Include proceedings of high-level national and international conferences sponsored by NBS, precision measurement and calibration volumes, NBS annual reports, and other special publications appropriate to this grouping such as wall charts and bibliographies.

**Applied Mathematics Series**—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

**National Standard Reference Data Series**—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a world-wide program coordinated by NBS. Program under authority of National Standard Data Act (Public Law 90-396). See also Section 1.2.3.

**Building Science Series**—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

**Technical Notes**—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

**Voluntary Product Standards**—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The purpose of the standards is to establish nationally recognized requirements for products, and to provide all concerned interests with a basis for common understanding of the characteristics of the products. The National Bureau of Standards administers the Voluntary Product Standards program as a supplement to the activities of the private sector standardizing organizations.

**Federal Information Processing Standards Publications (FIPS PUBS)**—Publications in this series collectively constitute the Federal Information Processing Standards Register. The purpose of the Register is to serve as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations). FIPS PUBS will include approved Federal information processing standards information of general interest, and a complete index of relevant standards publications.

**Consumer Information Series**—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

**NBS Interagency Reports**—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service (Springfield, Va. 22151) in paper copy or microfiche form.

Order NBS publications (except Bibliographic Subscription Services) from: Superintendent of Documents, Government Printing Office, Washington, D.C. 20402.

## BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

**Cryogenic Data Center Current Awareness Service** (Publications and Reports of Interest in Cryogenics). A literature survey issued weekly. Annual subscription: Domestic, \$20.00; foreign, \$25.00.

**Liquefied Natural Gas.** A literature survey issued quarterly. Annual subscription: \$20.00.

**Superconducting Devices and Materials.** A literature survey issued quarterly. Annual subscription: \$20.00. Send subscription orders and remittances for the pre-

ceding bibliographic services to the U.S. Department of Commerce, National Technical Information Service, Springfield, Va. 22151.

**Electromagnetic Metrology Current Awareness Service** (Abstracts of Selected Articles on Measurement Techniques and Standards of Electromagnetic Quantities from D-C to Millimeter-Wave Frequencies). Issued monthly. Annual subscription: \$100.00 (Special rates for multi-subscriptions). Send subscription order and remittance to the Electromagnetic Metrology Information Center, Electromagnetics Division, National Bureau of Standards, Boulder, Colo. 80302.



**U.S. DEPARTMENT OF COMMERCE**  
**National Bureau of Standards**  
Washington, D.C. 20234

OFFICIAL BUSINESS

Penalty for Private Use, \$300

POSTAGE AND FEES PAID  
U.S. DEPARTMENT OF COMMERCE  
COM-215

