TEACHING PROBLEM SOLVING IN AN INTRODUCTORY COMPUTER SCIENCE CLASS

David D. Riley
Department of Computer Science
University of Wisconsin - La Crosse

Abstract

This paper deals the difficuties of teaching problem solving in an introductory level computer science course where the majority of students are not computer science majors. An approach is suggested using top-down design techniques. The specific pseudo language, problem definition form, and design procedure taught in this course are described.

Keywords

I/O specifications, introductory computer science course, problem definition, problem solving, pseudo-instruction, software engineering, stepwise refinement, structured programming, top-down design.

Introduction

Perhaps Donald Knuth said it best in 1974 [14], "A revolution is taking place in the way we write programs and teach programming, because we are beginning to understand the associated mental processes more deeply. It is impossible to read the recent book Structured Programming [4] without having it change your life." However, Knuth never elaborated on how the precepts of structured programming should be taught to students in an introductory level computer science course.

The essence of the problem facing the University of Wisconsin-La Crosse Computer Science Department in the spring of 1980 was how to teach the appropriate programming concepts to 600 students per semester. Of these 600 students, approximately 90% are not computer science majors. Staffing problems require that the four semester hour course be taught in two distinct sections. Twice a week, all students view 50 minute videotaped lectures; and twice a week, students meet in classes of 30 in a more traditional classroom presentation.

Background

As early as 1965, Edsger W. Dijkstra [5] advocated the construction of programs in a structured manner. The phrase "structured programming" was introduced and more carefully delineated by Dijkstra [6] in 1972. The particular version of program design known as "top-down programming" is credited to Zurcher and Randell [21] and was later refined by H. D. Mills [16, 17]. The techniques of structured programming were first shown to be of considerable value by Harlan Mills and F. Terry Baker [1]. Datamation proclaimed structured programming as a "programming revolution" in December 1973.

Acceptance of the precepts of structured programming have permeated the computer software industry. New languages such as PASCAL [11], ADA [19], and FORTRAN [3] emphasize the use of control structures suggested by structured programming. Further justification for structured programming comes from similar languages used for systems development work. Languages such as SDL, PL/S and PASCAL underline the importance assigned to structured programming by Burroughs Corporation, IBM Corporation, Control Data Corporation and Texas Instruments, Incorporated.

History and Content of the Course

The course for which all the previously mentioned work in software engineering became highly significant at the University of Wisconsin-La Crosse (UW-L) was CPTS 110 - Introduction to Computer Science. CPTS 110 is a four semester hour course taught by members of the Computer Science Department at UW-L and supporting an enrollment

 0-89791-036-2/81/0200/0244 $00.75

averaging more than 600 students per semester. The composition of CPTS 110 is about 10% computer science majors and 90% students satisfying university basic studies requirements. Additionally, it must be noted that 85% of the computer science majors at UW-L enroll in CPTS 110 as their first computer science course.

The most recent revision of CPTS 110 was initiated in the fall of 1979. After an extensive course review, the Computer Science Department voted to adopt, as a course outline, Figure 1 in November, 1979. As an integral portion of the outline, a percentage was assigned to each major topic to indicate the approximate percentage of class time devoted to the topic. The particular portion of the outline in Figure 1 relevant to this paper is the topic entitled "Programming in a High Level Language."

The selection of PASCAL as the programming language to be taught in CPTS 110 deserves a brief explanation. It is more common in a course such as CPTS 110 to use BASIC as the vehicle to teach programming. In fact, the textbook [9] used currently in CPTS 110 contains only a brief reference to PASCAL, but the third of its three parts is devoted soley to BASIC.

In the fall of 1979, BASIC was being taught in CPTS 110. The choice of using PASCAL was made largely for two reasons:

1. The notions of software engineering suggest that a language with control structures similar to PASCAL would be amenable to currently accepted design techniques.

2. The Computer Science Department at UW-L has a strong commitment to the use of PASCAL as the programming language to be used centrally in the curriculum. As a result, it was felt that the computer science majors in CPTS 110 would benefit more from the exposure to PASCAL than to BASIC.

This choice to use PASCAL also had an effect upon the mechanism used to teach programming.

In November, 1979, it was also decided that, due to various constraints, CPTS 110 must be taught in a divided mode. For two fifty-minute periods per week, students would view video-taped lectures. These lectures would be interleaved with two fifty-minute classroom sessions in groups of no more than 30 students meeting with an instructor.

Problem Solving at UW-L

It is clear that the act of programming is a special type of problem solving. It is relatively easy to teach a student the syntax rules for a programming language; but it is extremely difficult to teach the same individual how to select from all the sequences of characters representing syntactically correct programs a sequence that performs the desired task. Dijkstra [7] said, "It seems vain to hope-to put it mildly-that a book could be written that we could give to young people, saying 'Read

5% Computer Science History

10% Introductory Computer Architecture (including machine language concepts)

10% A Survey of Computer Applications

- -Medical
- -Data Processing
- -Scientific (simulations, O.S., etc.)
- -Word Processing
- -Educational Applications
- -Artificial Intelligence
- -Personal Computing

5% A Survey of Various Languages, (COBOL, FORTRAN, ADA, SNOBOL, LISP, APL, BASIC, PASCAL, PL/1, ALGOL)

5% A survey of Various Computer Science Disciplines & Topics (Data Structures, Architecture, Simulaton, System Analysis, Numerical Analysis, Data Processing, C.A.I., Information Retrieval, Artificial Intelligence, Operating Systems, Compiler Construction, Microcomputers, Discrete Structures, Software Engineering)

10% Social Implications of Computers

- -Privacy & Security
- -Automation & Power
- -Future

5% Program Design Methodologies (Survey)

- -Flowcharts
- -Top-down algorithmic development
- -Hipo diagrams

50% Programming in a High Level Algorithmic Language (PASCAL)

- -Instruction Set
  - assignment statement
  - IF-THEN-(ELSE) control
  - I/O statements (simple-free I/O)
  - A looping control structure
- -Data Structures
  - fixed point scalars
  - floating point scalars
  - character scalars
  - arrays
- -Program Design

FIGURE 1 - CPTS 110 COURSE OUTLINE

this, and afterwards you will be able to think effectively'...."

Due to the overwhelming acceptance of top-down design [16] using stepwise refinement [20], this technique was selected to provide the basic mechanism for problem solving in CPTS 110.

Having made this choice, three basic questions still remain:

1. What is the particular pseudo language to be used?
2. How is a "step" defined?
3. How can the choices in 1. and 2. be made workable in CPTS 110?

What is the Particular Pseudo-language to be Used?

Two major qualities seem most important in the choice of a pseudo-language for CPTS 110. They are 1) simplicity and 2) conformity with PASCAL.

The set of pseudo-language instructions should be simple in terms of form, and the different pseudo-instructions should be few in number. In the terminology of Randall Jensen [12] it is felt that several "sequence" statements, a "selection" statement and a "iteration" (repetition) statement would be appropriate. A single repetition and a single selection statement, if appropriately chosen, are clearly sufficient control structures [2]. The set of sequence statements selected include a comment statement, an assignment statement, an input statement, and an output statement.

In order to promote ease of teaching top-down design, it was determined to use pseudo-instruction syntax that was English-like and clearly implied pseudo-instruction semantics. Additionally, the choice of pseudo-instruction syntax was influenced by the choice of PASCAL as programming language. It was felt that using pseudo-instructions with syntax close to the syntax of the corresponding PASCAL instructions would simplify the process of translating an algorithm in the pseudo-language into a PASCAL program. The final syntax of the six pseudo-instructions to be used in CPTS is shown below:

COMMENT syntax:

{a} where "a" may be any sequence of characters;

ASSIGNMENT STATEMENT syntax:

var ← expression

where "var is a variable and "expression" is some expression that can be evaluated to yield a value consistent in type with "var";

INPUT STATEMENT syntax:

READ (varlist)

where "varlist" is a list of variables separated by commas;

OUTPUT STATEMENT syntax:

WRITE (explist)

where "explist" is a list of expressions separated by commas;

SELECTION STATEMENT syntax:

IF condition THEN
  then clause

or

IF condition THEN
  then clause

ELSE
  else clause

where "condition" is a logical expression that can be evaluated to true or false, and "then clause" and "else clause" consist of one or more pseudo-instructions; each begins on a separate line;

REPETITION STATEMENT syntax:

WHILE condition DO
  loopbody

where "condition" is a logical expression that can be evaluated to true or false, and "loopbody" consists of one or more pseudo-instructions; each begins on a separate line

Specific details are omitted from the descriptions of "expression" and "condition". This is done to allow students flexibility without syntax overhead. Of course, a syntax for these expressions must be imposed when a programming language is taught.

This pseudo-language also incorporates indentation as an integral feature of the syntax of selection and repetition statements. No compound statements, "ENDIF", "FI", or "ENDLOOP" appear in the pseudo-lanugage and none are necessary. THEN clauses, ELSE clauses, and bodies of WHILE loops are specified in this pseudo-language by their indentation from "IF", "THEN", "ELSE" or "WHILE".

How is a "Step" Defined?

Having determined the method of problem solving used in CPTS 110 to be top-down design with stepwise refinement, it still remains to specify this process in more detail in order to use it in the classroom. The process of proceeding from one step to the next has to be clarified. Also, a determination is required of the content of the initial step of the design.

"The first and most important step in the design process is the formulation or definition of the problem." [13] Convinced of the truth of the above statement, the specification of an initial step problem is reduced to specification of the problem definition. Research in the area of problem definition is devoted largely to I/O specifications [18, 8,

10]. The difficulty with these approaches is that the typical student enrolled in CPTS 110 does not have adequate sophistication to deal with formal I/O specifications.

The problem definition technique finally adopted for use in CPTS was based on an example from Henry Ledgard [15] (page 8, example 2.36). The technique involves presenting a definition using five distinct parts: a general description, input specifications, output specifications, error or unusual conditions, and an example. Figure 2 is an example definition used in CPTS 110.

The general description is expected to be a broad statement of the problem. Details of input and output specifications need not be included in this general description. The general description is included to serve as an introductory comment unifying the remainder of the definition.

Input and output specification parts of the problem definitions for CPTS 110 are intended to describe the details of the form of "expected input" and the corresponding form of output. In each of these parts, specification is done via English description.

The example part of the CPTS 110 problem definition form consists of a particular input set and the particular output that should be produced. The input of the example should consist only of "expected input", but should also contain as many different variations in "expected input" as is possible.

The final part of CPTS 110 problem definition chosen is the error or unusual conditions. This part is included to fill in gaps in the input and output specificatons. Error or unusual conditions are intended to include all cases of possibilities for input that are not covered by "expected input". Just as in the case of input and output specifications, errors and unusual conditions are described in English.

This choice of problem definition is not as formal as many of those suggested in the literature. However, it was felt that the lack of formality and formal notation is the factor that made this approach useful in an introductory level course.

In the top-down design technique used in CPTS 110, the problem definition serves as the initial step. In order to express the problem definition in pseudo-language, it is treated as a single comment. The remainder of the process of stepwise refinement is described to CPTS 110 students in the following definition of top-down design:

> "Stepwise refinement of a program such that each step is a complete and correct program resulting from refining comment(s) from the previous step. The first step is an adequate problem definition in the form of one large comment. The final step is a program free of comment pseudo-instructions."

The particular technique of replacing comments with sequences of pseudo-instructions is further clarified by the following refinement guideline:

> "To proceed from one step in a top-down design to the next, refine all comments into sequences of pseudo-instructions, but within these sequences all THEN clauses, ELSE clauses, and bodies of WHILE loops should consist of a single READ, WRITE, assignment, or comment (to be refined at the next step)."

GENERAL DESCRIPTION

Write a program to input an initial checkbook balance then accept withdrawals and deposits and calculate the final balance. Withdrawals are subtracted from the balance and deposits are added to it.

INPUT SPECIFICATIONS

The first input line contains a single positive number that is the initial checkbook balance. Every input line after the first contains a transaction. A transaction line consists of 2 parts:

1) the first column of the line contains a "W" for withdrawal, a "D" for deposit, or an "E" to indicate end of input.
2) the remainder of the line contains a single positive number (representing the amount to be withdrawn or deposited - this number is meaningless on the "E" line).

There is only one "E" transaction and it is the last input line.

OUTPUT SPECIFICATIONS

The output consists of 3 parts in the following order:

1) INITIAL BALANCE: b where "b" is the value of the initial balance. This is followed by a blank line.
2) Each transaction, excepting the last, causes a blank line followed by the line below to be output:
   t AMOUNT: a NEW BALANCE: n
   where "t" is either WITHDRAWAL or DEPOSIT as appropriate, "a" is the amount of the transaction and "n" is the value of the new balance as calculated after the transaction.
3) Two blank lines are output followed by the line below:
   FINAL BALANCE: f
   where "f" is the balance at the time of the "E" transaction processing.

ERROR OR UNUSUAL CONDITIONS

1) Any time the balance becomes negative (i.e. balance was previously positive and this transaction caused a negative balance) after the normal transaction, the following additional line of output is output produced:
   WARNING - NEGATIVE BALANCE
   There is a $5 charge imposed when the balance becomes negative.
2) No attempt is made by the program to verify that the amount of a transaction is positive.
3) Any invalid input (a character other than "W", "D", or "E" in the first column or any non-numeric values when numeric values are expected) cause undefined results.
4) Any additional input (more than the appropriate number of values per line or additional lines after the "E" line) is ignored.

EXAMPLE

```
Input  → 133.26
         W  100
         W 23.16
         D  10
         W 50.10
         D  15
         D  200
         E  0

Output → INITIAL BALANCE:  133.26
         WITHDRAWAL AMOUNT:  100        NEW BALANCE:  33.26
         WITHDRAWAL AMOUNT:  23.16      NEW BALANCE:  10.1
         DEPOSIT AMOUNT:  10            NEW BALANCE:  20.1
         WITHDRAWAL AMOUNT:  50.10      NEW BALANCE:  -35
         WARNING - NEGATIVE BALANCE
         DEPOSIT AMOUNT:  15            NEW BALANCE:  -20
         DEPOSIT AMOUNT:  200           NEW BALANCE:  180
         FINAL BALANCE:  180
```

FIGURE 2 - EXAMPLE CPTS 110 DEFINITION

This comment specifies the process of progressing from one step to the next. Figures 3, 4, 5 and 6 show an example top-down design used in CPTS 110.

It must be mentioned that while the top-down design process used in CPTS 110 is defined to be one of eliminating comments, this is not done to discourage the use of comments. Great care is taken to encourage students to use comments in final algorithms, as well as programs in CPTS 110. Students are required to submit designs with programming assignments and encouraged to integrate comments from the various steps of the design into the final program.

How Can This Top-Down Design Technique Become Workable in CPTS 110?

As mentioned earlier, CPTS 110 is taught using both 50-minute video-taped lectures and 50-minute traditional classroom sessions. The structuring is such that students typically alternate between the video-taped lectures and traditional classroom meetings. Additionally, the traditional classroom meetings are taught by a variety of instructors, although each student will be exposed to only one of these instructors throughout his or her classroom meetings. It was determined to present as much of the mechanics of the top-down design process in three video-taped lectures. Traditional classroom meetings would necessarily have to manage the task of involving students in the problem solving process.

In the first video-taped lecture, students are exposed to the general notion of problem solving as it relates to programs. This first lecture defines the notion of a program and describes an algorithm as, a program using pseudo-instructions (those instructions not specifically belonging to any known programming language). Without defining the concept of top-down design, this first lecture presents two very simple stepwise refinement examples using instructions from the pseudo-language described earlier. Finally, the lecture presents the syntax of the six pseudo-instructions. Additionally, this lecture discusses the concepts of variables, expressions, conditions, and flow of control (sequence, selection and repetition) as they relate to the pseudo-instructions. The traditional class immediately following this first video-tape does not yet deal with design. Rather, this traditional class meeting is used to cover binary numbers.

The second video-tape on design deals exclusively with problem definition. The lecture begins by pointing out that the first step in problem solving must be definition of the problem. As an initial attempt at delineating the problem definition process, this second design video-tape presents a series of three problem definitions. Figures 7, 8, and 9 illustrate this development. Figure 7 shows what is termed as an example "poor problem definition". This definiton is expanded slightly to yield the "better problem definition" seen in Figure 8. Figure 9 shows the final version of a definition for this problem. The lecture emphasizes the continuum of qualities of problem definitions for a given problem.

STEP 1

GENERAL PROBLEM DESCRIPTION
Identify the character from any set of 3 input characters that would appear first (lowest) alphabetically.

INPUT SPECIFICATIONS
Input consists of 3 characters on a single line.

OUTPUT SPECIFICATIONS
Each of the 3 input characters is output one per line in the order they were input. Following this echo of input a blank line is output, followed by the line below:

THE LOWEST CHARACTER FROM ABOVE IS c

Where "c" is the input character that would appear first alphabetically.

Example

input →XBT

output →X
B
T
THE LOWEST CHARACTER
FROM ABOVE IS B

UNUSUAL OR ERROR CONDITIONS
1. If there are too few characters input then undefined results occur.
2. If any of the input characters are not upper case alphabetic characters then the following message is output after all characters are echoed:
   INVALID CHARACTER ENCOUNTERED-WARNING!
3. If too many characters are input then the first 3 are processed and the others ignored.
4. If 2 or all of the input characters have the same value and it is the least value alphabetically then that value is output as usual.

FIGURE 3-EXAMPLE TOP-DOWN DESIGN (STEP 1)

STEP 2

{input and echo 3 characters}

{identify the input character that is alphabetically first and store it in LOWCHAR}

WRITE (a blank line)

WRITE ('THE LOWEST CHARACTER FROM ABOVE IS', LOWCHAR)

FIGURE 4-EXAMPLE TOP-DOWN DESIGN (STEP 2)

STEP 3

```
READ (FIRSTCHAR)
WRITE (FIRSTCHAR)
READ (SECONDCHAR)
WRITE (SECONDCHAR)
READ (THIRDCHAR)
WRITE (THIRDCHAR)
IF (FIRSTCHAR is not uppercase alphabetic) OR
   (SECONDCHAR is not upper case alphabetic) OR
   (THIRDCHAR is not uppercase alphabetic) THEN
     WRITE ('INVALID CHARACTER
     ENCOUNTERED')
IF FIRSTCHAR < SECONDCHAR THEN
   {store smaller of FIRSTCHAR & THIRDCHAR in
     LOWCHAR}
ELSE
   {store smaller of SECONDCHAR & THIRDCHAR
     in LOWCHAR}
WRITE (  a blank line  )
WRITE ('THE LOWEST CHARACTER FROM ABOVE
     IS', LOWCHAR)
```

FIGURE 5-EXAMPLE TOP-DOWN DESIGN (STEP 3)

STEP 4

```
READ (FIRSTCHAR)
WRITE (FIRSTCHAR)
READ (SECONDCHAR)
WRITE (SECONDCHAR)
READ (THIRDCHAR)
WRITE (THIRDCHAR)
IF (FIRSTCHAR is not uppercase alphabetic) OR
  (SECONDCHAR is not upper case alphabetic) OR
  (THIRDCHAR is not uppercase alphabetic) THEN
   WRITE ('INVALID CHARACTER ENCOUNTERED')
IF FIRSTCHAR < SECONDCHAR THEN
     IF FIRSTCHAR < THIRDCHAR THEN
        LOWCHAR ← FIRSTCHAR
     ELSE
        LOWCHAR ← THIRDCHAR
     ELSE
        IF SECONDCHAR < THIRDCHAR THEN
           LOWCHAR ← SECONDCHAR
        ELSE
           LOWCHAR ← THIRDCHAR
WRITE (  a blank line  )
WRITE ('THE LOWEST CHARACTER FROM ABOVE
     IS', LOW CHAR)
```

FIGURE 6-EXAMPLE TOP-DOWN DESIGN (STEP 4)

POOR PROBLEM DEFINITION

Reorder input words so that the first is swapped with the last, the second is swapped with the second from the last, etc.

FIGURE 7 - EXAMPLE POOR PROBLEM DEFNITION

BETTER PROBLEM DEFINITION

Assuming that words are non-blank sequences of characters, reorder 5 input words so that the first is exchanged with the last and the second is exchanged with the fourth. Input words will appear one per line and output words should all be together on the same line.

FIGURE 8-EXAMPLE BETTER PROBLEM DEFINITION

ADEQUATE PROBLEM DEFINITION

GENERAL PROBLEM DEFINITION

Reorder 5 input words so that the first word is exchanged with the last and the second word is exchanged with the fourth.

INPUT FORM SPECIFICATIONS

Input consists of 5 words (non-blank sequences of characters) types on 5 consecutive lines.

OUTPUT FORM SPECIFICATIONS

The 5 input words will be output with one blank separating each pair. The first word output will be the last one input, the second output will be the four input, etc.

EXAMPLE

```
input →  INK
         GREEN
         EATS
         COMPUTER
         BLUE
output → BLUE COMPUTER EATS GREEN INK
```

UNUSUAL OR ERROR CONDITIONS

1. No attempt is made to verify that input words are valid English words.
2. If fewer than 5 words are input, the following message is output:
   INSUFFICIENT INPUT!
3. If more than 5 words are input, the first 5 are processed and all others are ignored.

FIGURE 9 - EXAMPLE ADEQUATE PROBLEM DEFINITION

The adjective "adequate" rather than "good" is used for the course. The lecture then presents and describes the five part definition form to be used in the course and concludes by presenting two more definitions in the specified form. In addition, students are supplied two additional examples as part of the printed course notes they receive.

In the traditional class meeting following the second design video-tape, instructors review the notion of problem definition and answer student questions. In addition, instructors develop with class assistance additional problem definitions. This class meeting also contains the first design assignment-writing an adequate definition for a poorly stated one.

The final video-tape on design formalizes the notion of top-down design by presenting the definition quoted earlier. This lecture also presents the "refinement guideline" and carefully traces three different top-down designs through the process from definition to final algorithm. Care was taken to see that one of these designs dealt with character processing, one with purely numeric processing and one with a business oriented problem. Also included in the student's course notes were three more complete designs for students to study.

While the video-tapes following the third design lecture deal with other issues of computer science such as operating systems, history, etc.; the next five traditional classroom meetings are used to exercise top-down design. In these meetings, students are shown completed algorithms asked to trace their execution, shown example designs from poor problem definition to completed algorithm, and encouraged to participate in these processes. The particular amount of time spent on any particular issue varies from class to class. During the period of time for these five class meetings, all students are asked to perform one complete top-down design as a homework problem.

Student testing over the topic of design in CPTS 110 is done in two parts. As part of a 40 question multiple choice exam, students are asked questions pertaining to problem definition, algorithm execution tracing, and top-down design. Additionally, students take a 50-minute design quiz requiring them to formulate an adequate problem definition for a poorly state problem and also to perform a top-down design given an adequate problem definition.

Conclusions

There are a few general conclusions that can be drawn from the experiences of teaching this material for the first time. First, it is evident that many students entering college have problem solving skills that are woefully inadequate. Expanding upon these skills can in some cases be somewhat like teaching the concept of a fraction to an individual that does not understand the concept of an integer. It is also obvious that a saturation of examples is useful. Students draw on past examples to tackle new situations. It would also appear that the acquisition of problem solving skills for many students comes only after considerable repetition of involvement in the process.

Having completed the instruction of this material only a few weeks ago, it is difficult to draw concrete conclusions regarding the effectiveness of the methodology. It is possible, however, to compare CPTS 110 this semester with is predecessor. CPTS 110 prior to this semester used BASIC as programming language and flowcharting as design tool.

Specific comparison of exam results is of questionable value due to the fact that it is difficult to suggest that questions using flowcharting and pseudo-language algorithms are of similar complexity. Still, looking at somewhat similar questions from previous exams, the results of this semester would appear to demonstrate little change in student performance. There is no evidence to indicate that students are finding top-down design more difficult to grasp than flowcharting.

Seven instructors have taught in the traditional classroom setting this semester. Their reactions to the change in design presentation from earlier semesters have been very similar in nature. The instructors do not find this new approach more or less difficult to teach. They also find little difference in student performance excepting a general feeling that students who could not grasp the diagramatic nature of flowcharting appear to have a better feeling about the English-like nature of pseudo-instructions.

The key issue is that all instructors uniformly support this technique of top-down design. They support it because it is "state-of-the-art" software development, presenting students with a more accurate view of computer software development. They also support it because it provides students who will subsequently enroll in other software courses the basic tools required to manage the material. Instead of feeling as though students have been taught questionable design techniques that may even have to be "untaught", instructors believe students are being provided with exactly the skills necessary to solve problems in the best manner currently known.

References

1. Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM Systems Jounal, Vol. 11, No. 1, January, 1972, pp. 56-73.
2. Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Communications of the ACM, Vol. 9, No. 5, May, 1966, pp. 366-371.
3. Brainerd, W., "FORTRAN 77", Communications of the ACM, Vol. 21, No. 10, October, 1978, pp. 806-820.
4. Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R., Structured Programming, Academic Press, London, England, 1972.
5. Dijkstra, E. W., "Programming Considered as a Human Activity", Proceedings of the IFIP Congress, 1965, pp. 213-217.
6. Dijkstra, E. W., "Notes on Structured Programming", Structured Programming, Academic Press, London, England, 1972.
7. Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.
8. Floyd, R. W., "Assigning Meanings to Programs", Proc. American Math Society Symposium in Applied Mathematics, Vol. 19, 1967, pp. 19-31.

9. Graham, Neill, The Mind Tool, Second Edition, West, St. Paul, 1980.
10. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12, No. 10, 1969, pp. 576-583.
11. Jensen, Kathleen and Wirth, Niklaus, PASCAL User Manual and Report, Springer-Verlag, New York, 1976.
12. Jensen, R. W., "Structured Programming", Software Engineering, Prentice-Hall, Englewood Cliffs, N.J., 1979, pp. 221-328.
13. Jensen, R.; Randall, W.; and Tonies, C., Software Engineering, Prentice-Hall, Englewood Cliffs, N.J., 1979.
14. Knoth, Donald E., "Structured Programming with GO TO Statements", Computing Surveys, Vol. 6, No. 4, December, 1974.
15. Ledgard, H. F., Programming Proverbs, Hayden, Rochelle Park, N.J., 1975.
16. Mills, H. D., "Top-Down Prgramming in Large Systems", Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 41-55.
17. Mills, H. D., "On the Development of Large Reliable Programs", Proc. 1973 IEEE Symposium on Computer Software Reliability, IEEE, New York, 1973, pp. 155-159.
18. Naur, P., "Proof of Algorithms by General Snapshots", Bit, Vol. 6, No. 4, 1966, pp. 310-316.
19. U.S. Department of Defense, Reference Manual for the Ada Programming Language, Proposed standard document, July, 1980.
20. Wirth, Niklaus, Systematic Programming, Prentice-Hall, Englewood Cliffs, N.J., 1973.
21. Zurcher, F. and Randell, B., "Interative Multi-Level Modelling - A Methodology for Computer System esign", Proc. IFIB Congress 1968, Booklet D, North-Holland, Amsterdam, 1968, pp. 138-142.