THE USES OF PROGRAM READING

Lionel E. Deimel, Jr.
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-8206

Abstract

It is argued that program reading is an important programmer activity and that reading skill should be taught in programming courses. Possible teaching methods are suggested. The use of program reading in test construction and as part of an overall teaching strategy is discussed. A classification of reading comprehension testing methods is provided in an appendix.

The Importance of Program Reading

We tend to think of programming as the writing of programs and programming instruction as the teaching of how to write programs. I would like to suggest that it is more illuminating to think of programming as "what programmers do." This definition is necessarily all-inclusive, encompassing not only program writing, but also problem analysis, program design, testing, debugging, and so forth. This alternative point of view makes us realize that programming requires a variety of skills, many of which we make no conscious effort to teach.

One of the most overlooked programming skills is the ability to read a program, an activity the programmer is called upon to do with surprising frequency. I will explain why I believe program reading is important, discuss how reading skills may be taught, and suggest how program reading may be useful in the teaching and evaluation of programming in the most general sense.

There are many circumstances in which the programmer must read programs. To begin with, he must often read programs in the technical literature, whether for professional development or in seeking a design for a particular program he has been called upon to write.

The programmer learning a new language often finds that reading programs written by others is an effective way to learn not only basic syntax, but also the subtleties of style associated with the language.

Programs are read at many stages in the software life cycle. In particular, the programmer must read his own code in the course of writing and debugging a program. Significant programs are written over an extended period, measured in days, weeks, even months. While writing one part of the program, it is often necessary to study a previously-written part, the details of which have been forgotten and must be retrieved through reading. Debugging provides similar opportunities to read one's own code.

Programmers seldom work completely alone, often working in teams or relying upon each other for assistance in writing and debugging. (We usually ignore or pretend to be unaware of this phenomenon among our students. We might be well advised to acknowledge it and attempt to utilize it.) The ability to examine and understand a colleague's program is critical to being able to offer help. Reading is also a needed skill for code walkthroughs and certain management reviews.

Finally, we come to the most obvious opportunity for program reading--in maintenance work. Maintenance and enhancement of existing software consume a major portion of time and resources of the typical data processing organization--estimates run from 40 to 70% [9, 10, 18]. Most maintenance work involves modifying programs written by someone other than the maintenance programmer. In fact, the original author is usually unavailable, and the maintenance programmer, having exhausted the resources of local myth and legend, has no alternative but to actually read the code he is required to fix.

Need Program Reading Be Taught?

Program reading may be a necessary professional skill, but need it be taught explicitly in courses? The programming student has ample opportunities to read programs in textbooks, to read his own

code during development, and to read the programs of others while assisting fellow students. Is it not possible that he naturally develops reading skill without intervention of the instructor? Although hard empirical evidence is lacking, I am convinced the answer to that question is "no." Let me illustrate with an example.

Figure 1 shows a question reproduced from an examination I gave recently to an introductory programming class. The question presents seven fragments of UCSD Pascal code which perform various character-processing operations. In part (a), the student is required to tell which fragments do the same thing. No hints are given as to whether all fragments are different, all are equivalent, or whether there are several groups of equivalent fragments. The question is further complicated by the request in part (b) to specify the effect of each fragment, given a particular set of initial values.

```
1.  i := pos(chstr, st);
    IF i>0 THEN
       st := concat( copy(st, i+1,
          length(st)-i), copy(st, i, 1),
          copy(st, 1, i-1) );
```

```
2.  str1 := '';
    FOR i:=1 TO length(st) DO
       BEGIN
       chst := copy(st, i, 1);
       str1 := concat(chst, str1)
       END;
    st := str1;
```

```
3.  i := 1;
    sought := true;
    WHILE (i<=length(st)) and sought DO
       IF st[i]=chstr[1] THEN
          sought := false
       ELSE
          i := succ(i);
    IF i<=length(st) THEN
       BEGIN
       FOR j:=1 TO length(st)-i DO
          BEGIN
          FOR k:=i+j-1 DOWNTO j+1 DO
             st[k] := st[k-1];
          st[j] := st[i+j]
          END;
       FOR j:=length(st) DOWNTO
          length(st)-i+2 DO
          st[j] := st[j-1];
       st[length(st)-i+1] := chstr[1]
       END;
```

```
4.  FOR i:=1 TO length(st) DO
       BEGIN
       temp := st[length(st)-i+1];
       st[length(st)-i+1] := st[i];
       st[i] := temp
       END;
```

```
5.  i := 1;
    sought := true;
    WHILE (i<=length(st)) and sought DO
       IF st[i]=chstr[1] THEN
          sought := false
       ELSE
          i := succ(i);
    IF i<=length(st) THEN
       BEGIN
       delete (st, i, 1);
       insert(chstr, st,
          1 + length(st) DIV 2)
       END;
```

```
6.  FOR i:=1 TO length(st)-1 DO
       st := concat( copy(st, 1, i-1),
          copy(st, length(st), 1),
          copy(st, i, length(st)-i) );
```

```
7.  str1 := '';
    FOR i:=1 TO length(st) DO
       BEGIN
       chst := copy(st, i, 1);
       str1 := concat(str1, chst)
       END;
    st := str1;
```

a.) Considering only the effect on variables CHSTR and ST, indicate which of the fragments above do the same thing. (Assume that LENGTH(CHSTR)=1.) For example, if all the fragments do the same thing, write something like:

1, 2, 3, 4, 5, 6, 7

If the first five programs do the same thing, but the 6th and 7th each do different things, write

1, 2, 3, 4, 5
6
7

and so forth. You do not have to say what any of the fragments actually do, merely which do the same thing.

b.) For each of the fragments above, give the value of ST after execution if, upon entry, CHSTR='c' and ST='abacaba'. If the answer cannot be determined from the information given, write "unknown."

Figure 1. Reading question.

The question proved to be highly unreliable. Students' answers were mostly wrong, and many students who scored highest seemed to have had the least understanding of the question. (In grading, a meticulously constructed system of partial credit was employed. This system was, I suspect, thoroughly inscrutable to the class, which viewed the question as an unmitigated disaster.) The students' universal complaint was that insufficient time had been allowed. From discussions with students, it became clear what the problem was. The question-answering strategy generally employed was to trace the execution of each fragment with the test data of

part (b). The rationale for this was that the tracing was necessary anyway for part (b), so one might as well do it first and use the results to answer part (a). Of those students who did not employ this strategy, most tried to answer part (a) by tracing the code for test cases of their own devising.

Why were my students fundamentally wrong in their approach to this question? They were wrong because they were operating at an inappropriate level of abstraction. They concentrated their attempts at understanding on the statement level, never grouping statements into meaningful units whose functions could then be abstracted into high-level descriptions which could further be abstracted into overall discriptions of what each fragment accomplished, a process Linger, Mills, and Witt have called "stepwise abstraction" [15]. A competent reader would have employed this more effective strategy immediately. Interestingly, most students recognized the futility of their approach, but failed to question whether another more efficient method was available to them. Program reading had not been taught in the course, and students obviously had failed to develop reading strategies, at least for situations like the one presented in the examination question, on their own. It is reasonable to expect that had reading been taught, student performance on this question and in other circumstances requiring it (in programming assignments, for instance) would have been better.

As an aside, I should mention that the class was justified in considering the question a nasty one. For the test data provided in (b), all the fragments had the same effect, even though they are not all functionally equivalent. It occurred to few members of the class that the string assigned to ST is highly idiosyncratic and therefore unlikely to contribute to an illuminating test case. (The equivalence classes sought in (a) are (1, 3), (2, 6), (4, 7), and (5). The answer to each part of (b) is 'abacaba'.)

The techniques required to answer the question in Figure 1 are a subset of all reading skills. Poor performance on this question does not therefore imply that students developed no reading proficiencies on their own. It does suggest that we cannot assume all important reading skills automatically develop alongside writing abilities.

How We Should Think about Programs

In order to teach programming, and particularly in order to teach program reading, it is helpful to have something more than a naive idea of what a program is and how it is interpreted. By "program," I refer not to some mathematical abstraction, but to a naturally-occurring artifact fulfilling a real need. This is the kind of object most of our students will be called upon to use, write, understand, and modify in their future careers.

It is appropriate that computer codes are called languages. Each programming language has its own syntax, semantics, pragmatics, style, and idiom. A program, therefore, besides providing instructions for controlling a machine, represents a kind of literary genre. It is, in a sense, a narrative essay which specifies some computational process. The programmer has it within his power to make this essay more or less comprehensible by attending to issues which are often lumped together under the term "style." A good program is remarkable in that it is really an essay intertwined with a second essay in natural language--documentation in the form of comments--which is a paraphrase of or commentary on the primary one. Both essays have a role in communicating the meaning of the program.

Programs or parts of programs are not always read with the intention of gaining total comprehension of their meaning. For example, the maintenance programmer trying to fix a bug probably only wants to discover the source of the error and enough about the program to assure himself that a proposed fix will have no unpleasant side effects. It is helpful, however, to have a model of what complete comprehension entails. Ruven Brooks has suggested such a model [3, 4]. When someone completely understands a program, he suggests, that person understands the program at a variety of levels and can relate the levels to one another. (Brooks speaks of "knowledge domains" and their interrelations.) These levels begin with the problem domain and include the mathematical model used in the problem analysis, the algorithms of the program, the data structures of the program, the program organization, the source language statements, and perhaps even the underlying operations of the host machine. The process of gaining understanding of a program is one of devising and revising hypotheses about what the program does. Obviously, this process involves all of the relevant levels or knowledge domains. A highly readable program facilitates our understanding of the relationships among domains through clear expression and well-chosen documentation.

A good program is necessarily a well-documented one, yet the world contains both bad programs and poorly explicated fragments. The would-be programmer needs to learn to read both well- and poorly-documented code. In fact, one can easily argue that he needs to be able to read structured and unstructured programs as well. Although we may choose not to ad-

dress the general reading problem in our teaching, we must at least recognize that very different reading strategies may be required for programs which differ in documentation level and structuredness.

Teaching Program Reading

So how do we teach program reading? This is a question for which little guidance is available in either the empirical or theoretical literature. I will make some intuitively appealing suggestions based on my own experience and that of my colleagues who have been studying the reading process. Readers are encouraged to pursue this question on their own and to share their results with the rest of us.

There are three necessary components, I think, to teaching program reading. The first is an expository component--students need to be sensitized to a number of reading issues and taught a variety of specific approaches to reading. Second, students must actually practice reading through reading exercises for which objectives and guidance are provided by the instructor. Finally, students need to be given specific program writing standards and taught to produce their programs with the human reader in mind. Let us consider each of these components in turn.

What can we tell our students about reading? To begin with, we must explain its importance. Students should be encouraged to view programs at different levels of abstraction and in different frames of reference. We can show how a statement or group of statements may be understood in terms of the effect on particular variables, in terms of a change in a data structure, in terms of effecting part of an algorithm, or in terms of the problem which the program is supposed to solve. (To reinforce these ideas, we should assign exercises in which the students must interpret code at different levels, a part of the practice component of reading instruction described in the next paragraph.) We must explain that programs may be read top-down or bottom-up, depending on one's reading objective, the program structure, and the nature of the comments. (Chapter 5, "Reading Structured Programs," of Structured Programming by Linger, Mills, and Witt [15] is a helpful reference on this subject for the instructor.) We need to warn that although comments are often very helpful, they are subject to being vague, misleading, or even wrong. Students should not be discouraged from tracing code in order to understand it, but they should be made to realize that doing so is a means to an end, a source of data for the real task of interpretation.

The second component of teaching program reading is practice. Students should be presented with programs to read, and their reading must be guided by specific instructor-supplied objectives or questions. They have to be given something to do with the program other than simply take it in. This technique mirrors the workplace environment. Asking students to answer questions about a program is one possibility, although questions need to be well thought out and usually very specific. Another alternative is asking students to modify or debug a program or to evaluate or improve its documentation. (Do not overlook the virtues of showing students realistic--that is, less than perfect--programs.) Supplying sample program solutions for an assignment the class has just completed provides incentive for program reading, although even here additional incentives will likely be needed. It may be possible to give out programs for study about which test questions will be asked, but do not expect students to be able to handle this too early. Similarly, asking for open-ended evaluations of distributed programs is possible, but not likely to be productive until students become moderately experienced readers.

Finally, we should provide specific guidelines for making programs more readable. We should explain why these guidelines say what they do, that is, how we think they help the human reader. Students should be encouraged to put themselves in the place of the reader as they write programs--especially should they ask themselves what they would want to know if they were reading for the first time the program they are writing. In the Computer Science Department at North Carolina State University, we distribute a Pascal "style sheet" designed to enforce a uniform style of formatting and to encourage the writing of meaningful comments. An excerpt from this handout is shown as Figure 2. The section shown specifies information to be displayed at the beginning of a program. The use of standards and guidelines not only produces better programs, but makes students more aware of what they should be looking for when reading a program.

Uses of Program Reading in the Classroom

Besides being of interest in its own right, the reading of programs has possible applications in the classroom. One of the most important of these is in the construction of examination questions. I, along with others, have advocated the use of questions keyed to selected reading passages on examinations as an alternative to program writing questions [5, 11, 19]. The obvious advantage of reading questions, especially multiple-choice ones, is that they are easier to grade, more objective, and likely to be more reliable measures of student performance in the technical sense of being more consistent measures. (Student program-writing performance on examinations is notoriously

II. Header Comments for Programs

In general, modules (programs, units, procedures, and functions) should begin with a block of information in the form of a comment. This "header" provides a reader's overview of the module. The standard header for a program is illustrated below. (Minor variants are acceptable. Irrelevant material may be omitted.) The header should tell all important facts about the behavior of the program. Therefore, a modification of any part of the program has the potential for requiring changes in the program header.

```
PROGRAM EXAMPLE;
{------------------------------------------------------------------------------

PURPOSE: Very brief statement of the purpose of the program.

PROGRAMMER: name                              COURSE: course #
DATE: date of last revision                   SECTION: section #
VERSION: optional, but helpful                LAB SECTION: lab #
                                              LAB INSTRUCTOR: name

INPUT: The order, layout, types, and meaning of input from each file.  The
  description should be adequate from which to prepare correct input without
  reading the program.

OUTPUT: Briefly, each file's expected output.

UNITS USED: List of units used, if any.

ASSUMPTIONS & LIMITATIONS: The expected quantities and quality of data, errors
  not caught, and other simplifying assumptions. (These are the programmer's
  disclaimers.)

ERROR CHECKS & RESPONSES: Erroneous values or situations that will be caught,
  and (briefly) what is done about them. (These are the programmer's
  warranties.)

ALGORITHM/STRATEGY: Notes about special algorithms, techniques, data
  structures, and so on, and/or a description or informal outline of the
  program--whatever is needed. (These notes explain the program as a whole.)

------------------------------------------------------------------------------}
```

Figure 2. Excerpt from NCSU Pascal style sheet.

variable.) If we truly believe reading is important, of course, reading questions may be justified on their own terms, but their attractiveness invites use for more general evaluation purposes.

Although there are few systematic studies relating program reading and writing competencies, it is reasonable to assume that both depend somewhat on common knowledge and skills. Any correlation one may wish to hypothesize clearly will be affected by the type of reading considered and the manner in which it is measured, however. Lemos in [14], discussing the results of two experiments reported earlier [12, 13], concludes that mere grammatical tests do not correlate strongly with the ability to write programs, but that there does exist a moderate correlation between the results of reading tests and the ability to write programs on examinations. As we have already seen from the question discussed above, however, this correlation cannot be 1. For any question we write, we must ask ourselves what exactly it is we are testing and whether our students can reasonably be expected to answer correctly, given their experience.

Because questions keyed to reading passages are attractive for examinations and because, as I have suggested, programs for reading should be accompanied by specific questions, it is useful to consider how to generate good questions. Often a "reading" question merely asks the student to trace code execution. Although there is nothing wrong with such an exercise, it is of limited usefulness, as it concentrates on the statement level, which generally requires only the most fundamental knowledge of the programming language. At the other extreme, it is a temptation to frame very general questions about programs, requiring more analytical ability than the

average student in an introductory course can muster. Such general questions are not very useful, particularly if they are meant to encourage serious reading of a program.

Psychologist Lois Makoid and I have discussed program reading comprehension questions in [5] and [6]. In our papers, we suggest a methodology for question generation and offer a taxonomy of questions. The taxonomy is useful for assessing question difficulty and for generating questions for evaluating specific skills and knowledge. Our classification is essentially a two-dimensional one. To begin with, questions may be asked in any of Brooks's domains--we can ask questions on the statement level, data structures level, and so forth. In fact, questions can be phrased in terms of several levels or can be made to require students to work across levels. Generally, the farther we abstract from the statement domain and the more cross-level relationships are required, the harder the question, although the presence of comments which provide high-level commentary may cause this relationship to break down. At any of Brooks's levels, we may ask questions requiring more or less demanding cognitive processing. For this dimension, we chose to adapt Bloom's taxonomy of behavioral objectives in the cognitive domain, a classification often used in defining educational objectives [1, 2]. Bloom's levels are knowledge, comprehension, application, prediction, analysis, synthesis, and evaluation. Bloom's idea is that these levels are hierarchical--the latter levels requiring the skills of the former ones. Whether this is strictly true or not, it provides a reasonable framework for addressing question construction. Within each of Bloom's levels, we have defined types of reading exercises and suggested possible forms and variations. Our complete taxonomy, expanded and corrected from [5], is shown in the appendix. I have found it to be of great assistance, not only in the construction of questions about programs, but also in reminding me of the many aspects of programs to which I must make my students sensitive. Our classification is tentative, by the way, and could benefit greatly from feedback from users in the field.

In Figure 3 are shown two questions designed to illustrate the taxonomy and suggest that reading questions, even multiple-choice ones, can be varied and challenging. Both questions are based on a fully-documented procedure named UPDATE, which is not shown. Question 1 is an analysis-level question involving inference. (See appendix.) Students are asked to infer unstated assumptions of the procedure. All the answers are written in the problem domain, so the code must be related to higher-level concepts. Question 1 is not easy, nor is question 2, a synthesis-level question requiring that students specify how the program could be modified. I believe a question such as this does bear on program writing ability, not just reading ability. Several knowledge domains must be related to answer this question.

---

1. Which of the following are apparently assumptions of the procedure?

   (a) If two courses (each with students enrolled) have the same name, their sections must be different.
   (b) All sections for the same course are represented consecutively.
   (c) Every section contains at least one student.
   (d) No student is in two different courses.

2. Which one of the following statements best describes the effort required to change the WHILE loop beginning at line 85 to a REPEAT loop, with the understanding that the function of UPDATE is to remain unchanged?

   (a) The change is easily made by substituting REPEAT for lines 85 and 86 and following the body of the loop with UNTIL (ERCODE<>OKUPDATE) OR (STUDPTR^.NAME=SNAME).
   (b) The change requires the introduction of a new Boolean variable.
   (c) The change requires the introduction of an IF statement.
   (d) The change requires the introduction of a new Boolean variable and an IF statement.
   (e) The change requires restructuring the code in ways other than those described above.

Figure 3. Sample reading comprehension questions.

To illustrate other possible uses of program reading I would like to mention a course-structuring idea advanced, though not yet well-tested by David Moffat and me [7, 8]. We have referred to our plan as a more analytical (as opposed to synthetic) approach to teaching programming, as it emphasizes the study of existing programs before requiring construction by students of new ones. We believe that students often fail to understand just what programming is all about, in part because they are, in effect, asked to compose unfamiliar literary forms in what amounts to a foreign language. According to our plan, an introductory course would proceed in four phases. Students first become users of programs. They begin to understand what programs can do, as well as appreciate the need for clear documentation and human-engineered man/machine interaction. Stu-

dents then begin to read and analyze programs while learning Pascal. This is more like the learning of natural languages, where writing skills follow upon reading skills. Ideally, by doing significant reading before writing, students develop intuition about what programs should look like--they will avoid many silly writing errors later on because those errors just will not look right. Program documentation, instead of representing an onerous chore, provides helpful guidance. It is thus seen in a positive light. In the third phase, programs are modified and enhanced. This becomes a gentle, phased introduction to what we usually call "programming," all done in a highly structured and supportive environment. Only in the fourth and final phase of the course do students do significant amounts of original programming. Although this plan of study seems to get off to a slow start, we believe that by the time the student is writing his own complete programs, he has a good understanding of what programs are, why documentation is needed, and why the user is more important than the programmer.

An exciting aspect of our analytical approach is the way the various phases may be tied together with the same software. Students can use a program, later read it, modify it, and finally write a new, improved (or perhaps simplified) version of it. Also, since understanding a program is much easier than writing it, students have an opportunity to deal with more complex programs than they do in a more conventional class. Learning language features is better motivated, as students come to understand Pascal constructs in realistic contexts before having to compose with them.

Unfortunately, our plan requires a large volume of materials which are largely unavailable. Instructors can start to build their own collections of useful programs. The implementation of departmental standards for programs can facilitate the exchange of suitable materials among the faculty. Certainly few textbooks provide the kind of programs required, namely, well-constructed, thoroughly documented ones. Dave Moffat's book, Common Algorithms in Pascal with Programs for Reading [16], although not ideal, is probably the best sourcebook of program reading materials available. Moffat's forthcoming UCSD Pascal Examples and Exercises [17] will be more useful to those using UCSD Pascal who are interested in our four-phase analytical approach.

Research Agenda

Program reading offers many possibilities for research. How do programmers read programs? What factors affect reading comprehension and how? (Program style, program documentation, reader experience, and medium--paper vs. screen, are only a few of the possibilities.) What techniques are effective for teaching reading? What are the effects of exposing students to reading before writing? Answers to all these questions are needed. Readers are encouraged to begin finding them.

Acknowledgements

I would like to thank Alexander Z. Warren of Phillips Academy, Andover, Mass. for providing me with the opportunity to combine a number of ideas into what has become this paper. Thanks are also in order to David V. Moffat and Jo E. Perry both for general inspiration and specific help with this manuscript, and to James E. Miller for his generous and prompt assistance in solving a troublesome problem. Lois Makoid had a major hand in bringing coherence to the final draft, in part because she was instrumental in shaping many of the ideas expressed.

References

[1] Bloom, B., et al., eds. Taxonomy of Educational Objectives: Handbook I: Cognitive Domain. New York: David McKay, 1956.

[2] Bloom, B., Hastings, J., and Madaus, G. Handbook of Formative and Summative Evaluation of Student Learning. New York: McGraw-Hill, 1971.

[3] Brooks, R. "Using a Behavioral Theory of Program Comprehension in Software Engineering." Proc. 3rd International Conf. on Software Engineering. New York: IEEE, 1978, 196-201.

[4] Brooks, R. "A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs." Proc. Conf. Human Factors in Computer Systems. Washington, D. C.: Washington, D. C. Chapter ACM, 1982, 125-129.

[5] Deimel, L. E. and Makoid, L. A. "Measuring Program Reading Comprehension: The Search for Methods." NECC '84: 6th Annual National Educational Computing Conference. Dayton, O.: Univ. of Dayton, 1984, 142-146.

[6] Deimel, L. E. and Makoid, L. A. "Developing Program Reading Comprehension Tests for the Computer Science Classroom." To appear in Proc. World Conf. on Computers in Education. Norfolk, Va., 1985.

[7] Deimel, L. E. and Moffat, D. V. "A More Analytical Approach to Teaching the Introductory Programming Course." Proc. National Educational Computing Conference '82. Columbia, Mo.: The

Curators of the Univ. of Mo., 1982, 114-118.

[8] Deimel, L. E., Moffat, D. V., and Hodges, L. "Restructuring the Introductory Programming Course." AEDS Monitor 21, 7-8 (Jan./Feb. 1983), 11-15.

[9] Hugo, I. St. J. "A Survey of Structural Programming Practice." AFIPS Conf. Proc., Vol. 46, 1977 National Computer x Conf. Montvale, N. J.: AFIPS Press, 1977, 741-747.

[10] Glass, R. L. and Noiseux, R. A. Software Maintenance Guidebook. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

[11] Haas, M. and Hassell, J. "A Proposal for a Measure of Program Understanding." SIGCSE Bulletin 15, 1 (Feb. 1983), 7-13.

[12] Lemos, R. S. "FORTRAN Programming: An Analysis of Pedagogical Alternatives." J. Educational Data Processing 12, 3 (Aug. 1975), 21-29.

[13] Lemos, R. S. "An Implementation of Structured Walk-Throughs in Teaching COBOL Programming." Comm. ACM 22, 6 (June 1979), 335-340.

[14] Lemos, R. S. "Measuring Programming Language Proficiency." AEDS J. (Summer 1980), 261-273.

[15] Linger, R. C., Mills, H. D., and Witt, B. I. Structured Programming: Theory and Practice. Reading, Mass.: Addison-Wesley, 1979.

[16] Moffat, D. V. Common Algorithms in Pascal with Programs for Reading. Englewood Cliffs, N. J.: Prentice-Hall, 1984.

[17] Moffat, D. V. UCSD Pascal Examples and Exercises. Englewood Cliffs, N. J.: Prentice-Hall, in press.

[18] Munson, J. B. "Software Maintainability: A Practical Concern for Life-Cycle Costs." Proc. Compsac 78. New York: IEEE, 1978, 54-59.

[19] Trombotta, M. "On Testing Programming Ability." SIGCSE Bulletin 11, 4 (Dec. 1979), 56-60.

Appendix: Techniques for Measuring Program Reading Comprehension

The following list enumerates techniques for measuring program reading comprehension, whether for the classroom or for experimental purposes. Except as noted, we assume in these descriptions that the student or subject has been given a program or program fragment to study. The code may or may not be documented; other material such as external documentation or output may also be available. Bloom's taxonomy is used to organize this list, but note that the level under which a procedure is classified is flexible; characteristics of each specific question need to be examined to see where the question properly belongs. Note that such factors as the inclusion of comments in the code can affect level; to the degree that comments are present, true, and interpret the code, questions about the function of that code are made easier. Exercises at any level may be phrased, require answers, or require reasoning in one or more knowledge domains.

1. Knowledge-level techniques.

   A. Recall. The subject is asked to recall, after the program has been removed, either the program itself of features of it.

   B. Recognition. The subject is asked to identify whether several program fragments were in fact contained in the original program.

   C. Reconstruction. The subject is asked to reconstruct the program from memory with the help of supplied fragments.

   D. Identification. The subject is asked to give the meaning of certain program elements. Questions of this type can be used to check a subject's knowledge of basic syntax and semantics of the programming language.

   E. Correction. The subject is asked to correct (usually straightforward syntax) errors in a program. This is a common type of exercise in opening chapters of programming texts, but one which should be used with care. Error repair presupposes knowledge of the programmer's intention, so that correction questions, in the absence of explicit specifications or unambiguous cues, are often ambiguous.

2. Comprehension-level techniques.

   A. Interpretation. The subject is asked to explain the function of designated portions of the program. Simple questions of this type may represent knowledge-level testing.

   B. Interpolation. The subject is asked to supply elements of a program which has been subjected to occasional or periodic deletions (cloze procedure).

C. Translation. The subject is required to paraphrase or to translate the program into another programming language or system of notation (for example, flowcharts). Success on this task is dependent on competence in the target notation, of course. A translation exercise may also be very much more complex. Translating a program from APL to Pascal, for instance, is surely a synthesis-level activity.

3. Application-level techniques.

A. Prediction. The subject is given code and initial conditions and is asked to predict its run-time behavior. He may be asked to indicate what output is printed, what the values of variables are at certain points, etc. In this form, prediction questions probably represent the most commonly used form of reading comprehension measurement and are often best classified under the comprehension level. Alternatively, the subject might be asked the effect on behavior of a change in the program. Yet another possibility is that of asking the subject to specify program behavior when the program, procedure, etc. is transferred to a different environment.

B. Simulation. The subject is required to trace program execution and show the simulation on paper. Presumably, most subjects predict using simulation. Unless a step-by-step simulation is demanded, however, it cannot be known whether the subject is simulating or not. The simpler examples of simulation are properly classified under the comprehension level.

C. Application. The subject is given one or more subprograms and is asked how they might be used to accomplish some goal.

4. Analysis-level techniques.

A. Inference. The subject is asked to infer the objectives, prejudices, or assumptions of the programmer. He may be asked to determine if the program was written with primary concern for space economy versus run-time efficiency, to state the apparent primary and secondary computational objectives of the programmer, to evaluate the degree of concern for error handling, and so forth.

B. Extrapolation. The subject is given code and final conditions and is asked to supply initial conditions sufficient to explain the program's behavior. The subject may be asked to supply program input or variable values. Note that what we have called "prediction" and "extrapolation" have elsewhere in the literature been called "forward comprehension" and "backward comprehension." We prefer our more differentiated terms, as the activities described are only superficially similar and are not equally difficult. A variant is to ask the subject the circumstances under which a section of the program is executed.

C. Deduction. The subject is given part of a program and is asked to determine other program elements which must be present. For example, he might be asked to deduce necessary or plausable declarations based on the presumed validity of a supplied fragment.

D. Abstraction. The subject is asked to create a higher-level abstraction of what he has read. An outline or pseudocode version might be requested. The subject might be asked to insert section-level comments into an undocumented program. (Note how this differs from translation.)

E. Location. The subject is asked to identify the location where a certain function is performed. Locating bugs is a particularly complex form of location, as the subject is not even told what he is looking for.

F. Justification. The subject is asked to tell why some particular program function is performed or why some sequence of statements is present.

G. Explanation. The subject is asked to explain how some function is performed.

H. Analysis. The subject is asked to analyze the behavior of the program. He may be asked to compute the number of times a statement is executed as a function of the input, determine the range of values taken on by a variable, explain the relationships among variables, identify implicit restrictions on input values, or otherwise characterize the range of program behavior.

5. Synthesis-level techniques.

A. Modification. The subject is required to supply changes which will modify program behavior. The modification could be a change or an enhancement. The subject also could be required to repair a bug. Such a task is increased in difficulty, of course, if the bug itself is not identified in advance.

B. Rewriting. The subject is asked to rewrite a program or code fragment with particular objectives in mind. The use of different control or data structures might be required, for instance. Note that this and other synthesis-level techniques may lack a major advantage of most other comprehension measurements--ease of evaluation.

C. Test Planning. The subject is given a program and is required to devise testing strategies for it. A debugging plan might be called for, based on a program and error report. A test plan for the program and/or appropriate test data might be requested.

6. Evaluation-level techniques.

A. Criticism. The subject is asked to evaluate a program with respect to some software quality metric. He might be asked to comment upon the appropriateness of its structure, the selection of identifiers, its consistency of style, or its modifiability. Alternatively, two programs could be compared in these respects.

B. Evaluation. The subject is asked to evaluate the design decisions implicit in a program with respect to another program, supplied or hypothetical. For example, the subject might be asked to evaluate the appropriateness of using one algorithm as against another with regard to space or time efficiency.

C. Appraisal. The subject is asked to assess the correctness of a program or fragment. A formal or informal correctness proof might be required. Alternatively, the subject might be asked to determine the conditions under which the program does not perform properly.

18th EDITION OF BIBLIOGRAPHY OF COMPUTER-ORIENTED BOOKS

More than 300 new books are listed in the 18th edition of the Annual Bibliography of Computer-Oriented Books, recently released by the University of Colorado.

Significant in this year's listing is the increased quality of books on advanced programming (14 new books) and management of data processing (11 new books). In the system design area, 18 new books were published. The sections on micros and PCs erupted with 82 new books in the last two years.

All introductory-type books published prior to 1982 were deleted. Despite the deletions, the bibliography still contains more than 1,250 books from 155 publishers. The bibliography separates the books into 86 categories and catalogs them according to type (reference, textbook, handbook) and style of presentation (programmed instruction, case study or narrative).

Copies of the bibliography are available for $4 from Computing Newsletter, Box 7345, Colorado Springs, CO 80933. The cost is $6 if an invoice is required.