Just the Numbers: An Investigation of Contextualization of Problems for Novice Programmers

Ellie Lovellette
Southern Illinois University
Edwardsville, USA
elovell@siue.edu

John Matta
Southern Illinois University
Edwardsville, USA
jmatta@siue.edu

Dennis Bouvier
Southern Illinois University
Edwardsville, USA
dbouvie@siue.edu

Roger Frye
Southern Illinois University
Edwardsville, USA
rfrye@siue.edu

ABSTRACT

Contextualization of problems is widely studied in mathematics education. In computer science it is taken for granted that authentic, contextualized programming assignments will increase student interest and therefore enhance performance in programming assignments. This paper examines whether contextualization is, in fact, beneficial for students. We present a study that compares novice programmers' ability to code a solution given two versions of a problem. One version is contextualized, the other is non-contextualized, using "just the numbers." The results presented indicate that there is no difference in success rates for the two types of programming assignments.

Keywords

Novice programming; problem solving; CS1; Soloway; rainfall

1. INTRODUCTION

Learning to program – that is, learning to express solutions to problems in a computer programming language – is difficult for a significant percentage of students. Much research has attempted to help educators understand the difficulty students face when learning programming. Some explanations for these difficulties are rooted in Cognitive Load Theory [9].

In short, Cognitive Load Theory posits that a person's learning degrades as the person is required to remember more items than the capacity of their working memory [11]. A similar effect is seen for problem solving when a person reaches the capacity of their working memory [3].

Computer programming shares many of the characteristics of computation procedures in mathematics [8]. From this, it follows that education research in mathematics could apply to computer science as well.

The skill of symbolizing – translating the concepts of real-life problems into workable entities – has been the subject of much mathematics education research [5]. It is thought that the ability of students to solve story problems depends not only on comprehending the problem, but also on being able to express it as mathematical symbols. One interesting question is whether the details in the context (backstory, or story) of a problem help or hinder the novice programmer in comprehending a problem or symbolizing it. One might expect that contextualization of a problem helps with comprehending, but that it creates an additional cognitive load in the symbolizing step making solving a problem more difficult.

Elliot Soloway, one of the earliest computing education researchers, viewed student learning through an idea based in cognitive theory [8]. In cognitive theory a schema is a chunk of information that has been successfully learned and is contained in memory. It is basic knowledge that a student or programmer can recall and use in a constructive way. Soloway's schemas are called goals and plans. In this context, a programmer's job is to know how to combine the appropriate goals and plans into workable programs. A step beyond symbolizing, the use of goals and plans requires a programmer to interpret a problem in a way that involves choosing the correct plans, and combining them in a way that leads to the solving of the problem. For example, students who learned the plan for finding the average of a list of numbers could apply that plan to any problem in which a goal is to find the average of some numbers. This idea is not unique to computer science. Related ideas, such as categorizing mathematical story problems based on their corresponding plans have been proposed in mathematical education [7].

To assess students' programming ability, Soloway used relatively small programming assignments. Though the problems were small, students did not always succeed in solving them. In one 1983 study Soloway [13] reported that only 38% of students could solve the Averaging Problem shown in the box below.

SIGCSE '17, March 08-11, 2017, Seattle, WA, USA

DOI: http://dx.doi.org/10.1145/3017680.3017726

The Averaging Problem
Write a program that repeatedly reads in integers, until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999.

In reflecting on his students' performance in writing programmatic solutions for the Averaging Problem, Soloway wrote "This problem is neither tricky nor esoteric; one would expect this problem to be easy for students at the end of a semester course on Pascal. In fact, we find that students do surprisingly poor on this and related problems." [12]

The Rainfall Problem, as stated (inset) below, is very similar to the Averaging Problem. Indeed, the Rainfall Problem is analogous to the Averaging Problem with the addition of a data validation condition. Thus, a student who recognizes the goal of finding the average of daily rainfall values can combine the plan for finding the average of numbers with the appropriate plan for validating input values to arrive at a solution for the Rainfall Problem. One aspect of the problem solving process is for the student to recognize the Averaging Problem in the context of the other problem.

The Rainfall Problem
Read in integers that represent daily rainfall, and print out the average daily rainfall; if the input value of rainfall is less than zero, prompt the user for a new daily rainfall value.

Soloway's publications report poor performance by students on the Rainfall Problem as well. For example, in 1983 Soloway reported that only 14% of one group of novice programmers implemented a correct solution to the Rainfall Problem [13].

Whether Soloway intended it or not, his Rainfall Problem has become a pseudo-standard for measuring novice programmers' ability and has been mentioned by Lister [6] and Guzdial [4] in calls for rigorous computing education research.

An additional aspect of the current research lies in the constructivist school of learning. One of the hallmarks of constructivism is authenticity [10]. Authenticity means that learning sources and assignments should be as original and real as possible. The Averaging Problem, which is simply computing a running total and deriving further information from its results, is made (more) authentic when contextualized as the Rainfall Problem. According to constructivist theory, students should do better on the contextualized Rainfall Problem because it is more authentic.

The issue of contextualization of programs is therefore interesting from at least three different points of view. First is the question of its effect on the programmer's ability to comprehend and symbolize the problem. Second is whether the cognitive process of choosing plans is affected by the contextualization of the problem. Third, is whether contextualization triggers the increased capacity to learn that is predicted by constructivist learning theory.

The question of contextualization is thus threefold: 1) (authenticity) does wrapping assignments in context help students' understanding and performance by presenting problems from a real-world perspective as opposed to forcing them to look for a general solution to a generic problem; 2) (cognitive load) does context distract and hinder performance; or 3) does it have no influence at all?

Your program should open and read from the file "monthlyRainfall.txt." This file contains data about the amount of rain (in inches) that fell, each day, for a month. The data are stored as doubles. When your program finishes, it should print the following results to the screen:

(a) How many days were in the month analyzed

(b) The total amount of rainfall during that month

(c) The average daily rainfall during that month

(d) The amount of rain that fell on the rainiest day

(e) The number of days that it rained

Figure 1: Task R, the Rainfall Problem

Your program should open and read from the file "numbers.txt." This file contains several doubles. When your program finishes, it should print the following results to the screen:

(a) How many numbers have been read

(b) The sum of the numbers in the file

(c) The average value of the numbers in the file

(d) The largest value in the file

(e) The number of values that are greater than zero

Figure 2: Task N, a "just the numbers" Task

2. EXPERIMENT

2.1 Research Question

To explore the role of contextualization of a problem in a programming assignment, two programming tasks were designed. One task, Task R, is a variation of Soloway's Rainfall Problem. The other task, Task N, is the same problem in terms of input, output, and computation, but explained only in terms of the numbers. The following null-hypothesis is proposed: Contextualization of a problem in a programming assignment will have no effect improve students' success in completing the assignment. Stated another way, students will perform better on Task R than on Task N.

2.2 Methodology

Two versions of the exercise were created. Figure 1 presents Task R, the Rainfall Problem. Task R requires students to write a program to compute the average of daily rainfall values, the count of rainy days, and the maximum daily rainfall. Figure 2 presents Task N, a generic data analysis exercise, involving keeping a count of items, a running total, and other common data manipulations. Task N is a "just the numbers" version of Task R.

We wanted our students to spend their time on the problem itself and not get caught up in the syntax necessary to access the data they were to work with, so, in addition to

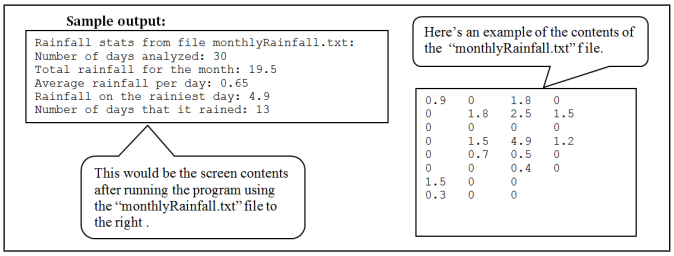


Figure 3: Example Input & Output Given in TaskR

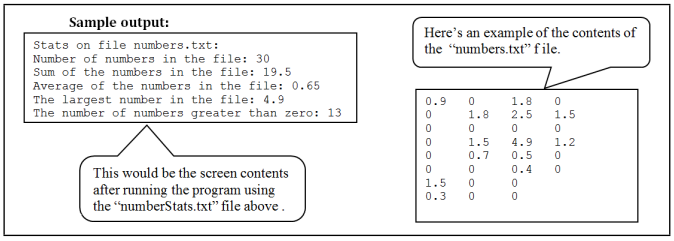


Figure 4: Example Input & Output Given in TaskN

the problem description, each task included an example of the Java code necessary to open and read data from a file. The tasks also included separate corresponding input and output examples. Figure 3 shows the example input and output for Task R. The difference between what is shown in it and the input and output example in Task N in Figure 4 was contextualization. For example, instead of "the number of days" represented in the data, students completing Task N were asked to compute the "Number of numbers in the file."

The two tasks are equivalent in terms of work. In fact, a successful program for either task would be a success for both tasks.

2.3 Setting

All participants in the experiment were enrolled in CS 140 Introduction to Computing I, a CS1 course, at Southern Illinois University Edwardsville. CS 140 is a requirement for majors in Computer Science and Computer Engineering, and minors in Computer Science. It also satisfies core course requirements for majors in Electrical, Industrial, and Mechanical Engineering, and is a prerequisite for some upper division courses in Mathematics. Thus, even though students taking the course are predominantly majoring or minoring in CS, enrolled students' majors vary widely.

The experiment was conducted in two consecutive semesters. CS 140 is structured as 150 minutes per week of "lectures section" augmented by a weekly 110 minute "lab section" in 15 academic weeks. Students can enroll in any one of the lecture sections and any one of the lab sections. The "lecture" sections varied in format from extemporaneous lecture, to prepared lectures, to peer instruction [2] with electronic response devices and were taught by different instructors. In each lab session, students were tasked with programming problems, which they solved in pairs. Students employed pair programming, taking 10 minute turns to solve problems and develop programs using an IDE.

The Rainfall experiment was conducted during the 15th, and last, lab session in both the Fall and Spring semesters.

Table 1: Class Level Breakdown of Participants

Class Level	Participants	
Freshmen	90	36.3%
Sophomores	80	32.3%
Juniors	49	19.8%
Seniors	20	8.1%
Seniors w/degree	7	2.8%
Graduate students	2	0.8%

At this point students were fairly experienced, having received 14 weeks of instruction in Java, and having completed 14 previous lab exercises as well as a number of structured programming projects.

Two of the previous lab assignments were similar to the Rainfall Problem in that they involved the skills of reading data from a file and keeping a running total. In order to encourage participation in the Rainfall experiment, the lab session was promoted as a "lab test." Students were informed that the lab would count as part of their final exam grade. It was hoped that students would approach the experiment with the same degree of seriousness with which they would approach an exam. All students were awarded points for participation regardless of their performance.

For the experiment, Task R and Task N were distributed as printouts. Students seated next to each other were given different tasks, with the goal that the versions would be distributed evenly in each lab section and overall. The paper task descriptions were collected from each student as they completed their submission. Not a single paper task description was retained by a student.

Students had 1 hour and 50 minutes to complete their task. They were not allowed to use outside resources such as previous labs or the Internet. The labs were supervised, but no programming help was offered. The finished programs were submitted electronically.

2.4 Participants

In the first semester of the experiment (Fall) initial enrollment of CS 140 was 164 students. Of the original 164, 24 students, some of which had D or F averages, had either dropped the class or were in the process of withdrawing, and did not take part in the experiment. Thus the participation group (in both semesters) is somewhat self-selected towards better students. Seven students were absent from the lab in the week of the experimental task for miscellaneous reasons. One student was disqualified because of attempted cheating, and two students' work was lost because it was submitted incorrectly. This left 130 students completing one of the experimental tasks in the Fall semester. The enrollment headcount in the Spring semester was 136 with 12 withdrawals by the time the experiment took place. Three students were absent, and 2 submissions were improperly recorded. Thus the total number of participants in the Spring was 118, bringing the total population to 248.

Of the 248 students taking part in the experiment, 209 (84.3%) were male, 39 (15.7%) were female. The breakdown of other demographic characteristics of the population is shown in Tables 1 and 2.

Table 2: Major Breakdown of Participants

Major	Participants	
Computer Science	74	29.8%
Electrical Engineering	22	8.9%
Mechanical Engineering	52	21.0%
Computer Engineering	31	12.5%
Undeclared	51	20.6%
Other	18	7.2%

Students were predominantly male, and of the 248 total participants, 204 (82%) self-identified as Caucasian. Approximately 70% of the population consisted of freshmen and sophomores. A third of the population was students majoring in Computer Science. The "other" category included students in Psychology, Pharmacy, Physics, Art, Mathematics, and other Engineering disciplines.

3. RESULTS

3.1 Overall Performance

As mentioned above, there were five values to be computed and output in each task. The submissions were compiled (if possible) and run with two different input files to check for correct execution behavior. The first file contained a valid dataset, the second file was empty and designed to check for edge cases and division by zero errors. Submissions were compiled, executed, and the resulting output was collected in report files by a script. The correctness of the output for each submission was assessed by a member of the research team.

A submission could receive a maximum of 9 points – 4 for structure and 5 for output correctness. One structure point each was awarded for the following - compile, successfully open and read the data file, identify a loop condition able to handle an arbitrary number of values in the file. Structure points awarded for Tasks N and R are shown in Table 3.

Correctness points were awarded for correct outputs. Each correct output for sum and count of values, count of values greater than zero, and the maximum value in the dataset received a point. To avoid over-penalizing errors, the output for the average received a point if the student correctly divided the sum by the count, even if the output was not equal to the expected number. Actual output correctness points awarded are shown in Table 4. The final structure point was awarded when the student explicitly or implicitly utilized a guard against division by zero.

Submitted programs predominantly computed all five output values or none of them. 105 submissions produced 0 correct values, however, 48 of those were non-compiles. 75 submissions produced all five correct values.

Out of a total of 248 students with completed submissions, 121 students worked on Task N, while 127 received Task R.

3.2 The Possibility of Cheating

With the same two programming tasks repeated for subsets of participants spread over multiple days, and the ease with which students could have propagated a solution to either task, the possibility of a solution being shared among students was considered. Naturally, the teaching team reminded the participants of the expectation of academic honesty. Equally naturally, the research team looked for evidence that the participants did not remain honest. The research team used MOSS [1] (Measure Of Software Similarity) developed at UC Berkeley to compare participants' submissions. MOSS had been utilized by the teaching team for several semesters and this experience with the software gave them confidence the results could be used to detect shared solutions.

Table 3: Overall Performance - Structure

Task	Task N	Task R	Population	
Compile	98	102	200	81%
Open file	76	81	157	63%
Read file of arbitrary length	73	79	152	61%
Guard against division by zero	6	6	12	5%

Table 4: Overall Performance - Output Correctness

Number of Correct Outputs	Task N	Task R	Population	
0	53	52	105	42%
1	21	21	42	17%
2	3	6	9	4%
3	3	6	9	4%
4	2	6	8	3%
5	39	36	75	30%

The majority of pairs flagged for similarity were between programs submitted for different tasks, which is evidence that the assignments are interchangeable. MOSS compares code regardless of its ability to compile.

High similarity scores in both iterations of the experiment were caused by the try-catch segment used to open the input file, the `System.out.print` statements used to output the 6 required results, and the Java package, class and main method declarations. Additionally, the I/O functionality required import statements and declarations of the same number and types of variables needed to complete the task.

Students were provided the import statements, and the syntax for opening the input file, which artificially drove up the similarity findings. Likewise, the necessity for six `System.out.print` statements also inflated the similarity. In addition, the solution to the problem is relatively short and allows for minimal variation from the obvious approach. MOSS reacted to that as well. An in-depth look at high similarity pairs led to the conclusion that we had no actual cases of plagiarism.

3.3 Rejecting the Null-Hypothesis

With the reasonable validity of the data thus established, an investigation of the possible statistical significance of the

Table 5: Course Grade vs Output Subtask Completion for Task N

Subtasks Completed	Final Course Grade A	B	C	D	F	Total	%
0	15	11	20	5	2	53	43.8
1	5	9	5	1	1	21	17.4
2	1	1	1			3	2.5
3	1		2			3	2.5
4	1	1				2	1.6
5	24	12	1	2		39	32.2
Total	**47**	**34**	**29**	**8**	**3**	**121**	

Table 6: Course Grade vs Output Subtask Completion for Task R

Subtasks Completed	Final Course Grade A	B	C	D	F	Total	%
0	14	21	11	5	1	52	40.9
1	5	7	7	2		21	16.5
2	4	2				6	4.7
3	3	3				6	4.7
4	3		3			6	4.7
5	24	10	2			36	28.3
Total	**53**	**43**	**23**	**7**	**1**	**127**	

Figure 5: Course Grade vs Total Performance on Task N

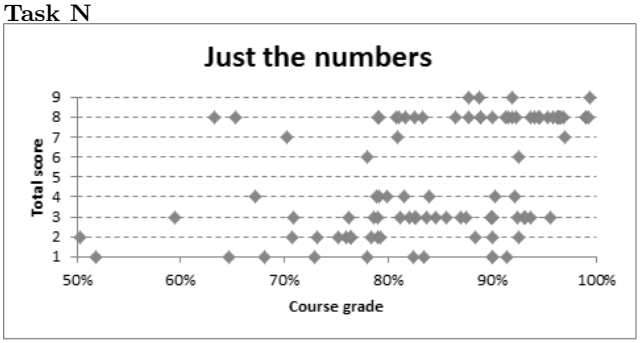
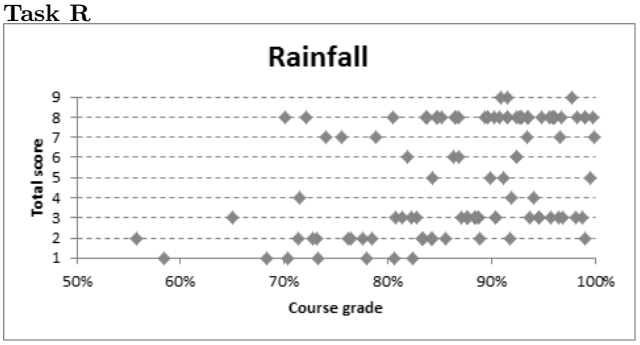


Figure 6: Course Grade vs Total Performance on Task R



difference in the performance of the two groups is investigated in this section.

3.3.1 The Sub-Populations

An important question is whether any difference in results is due to contextualization or to other factors. For example, it is possible that, by chance, the better students received one of the tasks, while the weaker students received the other. To test for this possibility, the final course grades for each group were analyzed.

The average final course grades for the R and N task populations are about the same: 85.3% for Task R and 83.8% for task N. An F-Test was performed to determine the appropriate T-Test. The 2 tailed T-Test on course grade yielded p=0.267, indicating there is not a significant difference in the strength of the two groups.

3.3.2 Performance

The population outcomes do not show a high degree of variability. The distribution of correctly completed output subtasks versus course grades is shown for Task N in Table 5, and for Task R in Table 6. The total received points (out of 9) for each task group after removing the non-compiling submissions is shown as plots in Figures 5 and 6. Approximately half of the students in each of the two task sub-populations didn't produce a well structured program. If the code didn't compile the submission received 0 points. If the code compiled but couldn't open the input file or couldn't set a loop condition to read a dataset of variable length it was considered structurally incomplete. The structurally incomplete submissions delivered at most 2 correct output values. The other half of the submissions, which received 3 points for structure by compiling, properly setting up access to the input file and including the correct loop condition, repeated the pattern of generally either delivering all or none of the correct numeric outputs. The breakdown of these results is shown in Table 7. The students with structurally sound programs, which produced some but not all 5 correct outputs struggled the most with identifying the number of values greater than zero and finding the maximal number in the dataset. A 2 tailed T-Test on output success (0-5 correct numeric output) was performed to compare performance on Task R and Task N. The test yielded p=0.907, indicating that there is virtually no difference in the performance of the two task groups. The similarity was even more stark,

Table 7: Output Correctness of Structurally Correct Submissions

Number of Correct Outputs	Task N	Task R
0	14	13
1	4	1
2	0	4
3	3	6
4	2	6
5	39	36

giving p=0.985, when performing the same test on the combined results on the total 9 point scale for both structural and output correctness.

4. ANALYSIS AND DISCUSSION

4.1 Experimental Results

4.1.1 On Contextualization

The experience and intuition of some of the authors was that the contextualization of the problem would provide a distraction to the students and result in lower success rates and significant differences in performance between the two tasks. Our findings however, indicate that novices perform at the same degree of success on both generic "just the numbers" problems and contextualized problems. These results point to the cautious conclusion that contextualizaion of problems does not hinder novice's ability to program, so the increased cognitive load is of no consequence. Context however, also does not seem to aid students' performance, going against constructivist theory, which posits that authenticity increases understanding and success.

4.1.2 Students' Ability to Program

The relative success students had with both tasks (30.2% of participants wrote programs that computed all five output values) was, on one hand satisfying because they didn't perform as poorly as some historic student groups, but, on the other hand, disappointing. 37.1% (92 of 248) of participants who earned a course passing grade (A, B, or C) submitted a program that computed none of the specified outputs.

4.2 Methodology

We believe the methodology developed and presented here is sound. While there is room for minor improvements in the presentation of the "just the numbers" task, the basic methodology could be adopted by other researchers.

4.3 Threats to Validity

Naturally, these results should not be taken as the final word on this topic. The data presented here is from a limited population of student participants; from a single institution, in one course, in two semesters, programming in an object-oriented programming language, using one programming language, and in a curriculum that does not emphasize problem solving techniques in any particular way.

Students' final course grades were used to judge if the two sub-populations were equal; however the course grading criteria were set by individual instructors, so the course grade is not a standardized measure. Neither is it a measure of problem solving or programming ability alone, but frequently includes penalties for late or missing submissions and might be artificially inflated by students utilizing the help of instructors, peers, or tutors to varying degrees.

It is possible that the two types of task completed by students were not different enough to affect student performance. A more complex degree of contextualization with a stronger effect on cognitive load, or a different context topic might have also altered the outcome.

The choice of timing may have played a role as well. Should the tasks have been assigned shortly after students completed the module on looping instead of at the end of the semester, the results may have been different.

5. CONCLUSION AND FUTURE WORK

The evidence presented here indicates that novice programmers' success is not influenced by the contextualization of programming assignments. In our study, students performed comparably on generic "just the numbers" programming problems. Assuming these results are generalizable, then instructors in introductory programming courses may safely assign contextualized programming tasks to boost motivation, authenticity, and students' ability to relate to the material, without negatively affecting their performance. Generic assignments, where context may be too distracting, should allow students to concentrate on identifying and combining goals and plans without jeopardizing their success.

The authors intend to replicate the study to provide further evidence and invite external replications as well. An international multi-institutional study examining the issue is currently underway.

6. REFERENCES

[1] A. Aiken. Moss. http://theory.stanford.edu/~aiken/moss/. Accessed: 2016-04-29.

[2] G. Braught, T. Wahls, and L. M. Eby. The case for pair programming in the computer science classroom. *Trans. Comput. Educ.*, 11(1):2:1–2:21, Feb. 2011.

[3] T. de Jong. Cognitive load theory, educational research, and instructional design: some food for thought. *Instructional Science*, 38(2):105–134, 2010.

[4] M. Guzdial. From science to engineering. *Commun. ACM*, 54(2):37–39, Feb. 2011.

[5] N. T. Heffernan and K. R. Koedinger. The composition effect in symbolizing: The role of symbol production vs. text comprehension. *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, pages 307–312, 1997.

[6] R. Lister. Ten years after the mccracken working group. *ACM Inroads*, 2(4):18–19, Dec. 2011.

[7] R. E. Mayer. Frequency norms and structural analysis of algebra story problems into families, categories, and templates. *Instructional Science*, 10(2):135–175, 1981.

[8] R. E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, Mar. 1981.

[9] B. B. Morrison, B. Dorn, and M. Guzdial. Measuring cognitive load in introductory cs: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 131–138, New York, NY, USA, 2014. ACM.

[10] E. Murphy. Constructivism: From philosophy to practice.

[11] J. L. Plass, R. Moreno, and R. Brunken. *Cognitive load theory*. Cambridge University Press, 2010.

[12] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, Sept. 1986.

[13] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM*, 26(11):853–860, Nov. 1983.