Teaching CS/1 Courses in a Literate Manner

Bart Childs, Deborah Dunn, and Wm. Lively
Texas A&M University

Abstract

The first course in Computer Science is often called 'CS/1' based upon the designation in curriculae recommendations. The content of CS/1 courses often shows that it should include a significant amount of documentation, problem solving, problem formulation, ... Experience has shown that instructors often slide into almost total emphasis on language syntax. Ask the student who has taken such a class as to its content and the answer usually comes back like "It was a C (or Pascal or ...) course."

We will report on an experiment of teaching the honors section of our first course at Texas A&M University using Knuth's `WEB`. The primary advantage we saw in the use of the system is that the `WEB` system would enable the progression through the problem solving methodology by editing and extending the same document.

Our analysis of data obtained by tracking the students in later semesters shows significant benefit from the use of literate programming. We found little or no problem using emacs, TEX, `WEB`, and requiring **documentation** after the initial scares in the course. We will describe how we taught the course, present performance statistics, and outline our recommendations for pursuit of similar goals. Finally, we will outline our longer range goals with the use of similar systems.

1 Introduction

We embarked on a project to teach the first computer science course (CS/1) (2; 14) using literate programming (5) and still covering all the topics covered in the usual sections (3). The parallel sections used Turbo Pascal and its supporting environment.

CPSC 110 is entitled "Programming I." The catalog description does not specify the languages to be used, but we normally use English and Pascal. A few years ago we tried C instead of Pascal but have never tried any substitutes for English except TEXian (although some students may argue the point.) The reasons that the C experiment were a failure will not be addressed in this paper.

An inherent part of these CS/1 courses is to develop the student's skills in *problem solving*. Indeed, in many course outlines, that is part of the title and the main emphasis in the description of the course contents. A problem solving methodology is often stated in CS/1 courses which generally has steps like:

1. State the problem completely!
2. Develop all necessary assumptions.
3. Develop an algorithm and test data set(s).
4. Code the problem.
5. Analyze the results (and iterate?)

Literate programming is a style in which the design of the code reflects that the human reader is as important as the machine reader. The human reader is often associated with the expensive process of maintenance and the machine reader is the compiler/interpreter. Literate programming is a process which should lead to more carefully constructed programs with better, relevant 'systems' documentation. We think that the first sentence in this paragraph should be particularly relevant to students because the human reader (the one who assigns grades) is obviously the most important reader.

The features of literate programming that gave us the confidence to expect positive results are:

1. top-down and bottom-up programming since it is a structured pseudo-code,
2. programming in small sections where most sections of code and documentation (section in this use is similar to a paragraph in prose) are approximately a screen or less of source,
3. typeset documentation (after all, Knuth was rewriting TEX),
4. pretty-printed code where the keywords are in bold, user supplied names in italics, ..., and
5. extensive reading aids are automatically generated including a table of contents and index.

We offer these comments about the above list We repeat the item numbers for clarity:

1. these topics are usual in CS/1 books but they generally lack the integration to make them really effective for the student,
2. divide and conquer is also espoused but the larger examples that are furnished in many books forsake the principle,
3. it may be argued that this is 'feeding pearls to the swine' but we like the cognitive emphasis

that comes from logical substitution of words for key-words ...,

4. the fact that `weave` breaks lines based on its parsing is another cognitive reinforcement,
5. encouraging/requiring students to review their programs as documents makes them **think** about readability.

2 Problems with 'Problem Solving'

Researchers have found that many of the difficulties experienced by novice programmers are not a result of misunderstanding the language constructs, but a result of problems with "putting the pieces together" (13). Thus, the process by which programs (and documentation) are developed should be examined.

Linn and Clancy (6) state that a good programmer needs both a knowledge of the programming language *and* good problem solving skills. Introductory courses tend to emphasize programming; that is, the *product* of good design and development (6). Although this is obviously an important aspect of programming, the real problems exist in the design of problem solutions (8). Few textbooks used in the introductory courses actually emphasize teaching the student how to develop good design solutions (6), regardless of the university catalog description.

Linn, Sloane, and Clancy (8) found, in teaching program design, that teachers who discuss how they solve problems, including their interpretation of the problem statement, are more effective than those who present just the subject matter. Studies have shown that explicit teaching of problem solving strategies greatly influences learning (7; 8).

Soloway (11) states that *goals* and *plans* are the two key components in the task of representing problems and solutions to a problem. Problem solving, and hence learning to program, requires that students learn to construct *mechanisms* and *explanations* for those mechanisms. Students are led to believe that programs are the output from the programming process. Rather, they must be made to understand that programming is a design discipline. Instead of the programming process being viewed as a program, it should be viewed as "an artifact that performs some desired function" (11).

Soloway and colleagues (12; 13) have studied *bugs* – errors in programs – and *misconceptions* – misunderstanding in the minds of novice programmers – in an attempt to identify the needs of novice programmers by understanding the kinds of mistakes they are likely to make. Because there are many ways to solve a given problem, bugs are identified using a *goal/plan analysis*. Goals are what is to be accomplished and plans are those stereotypical sections of code that are used to achieve the goal. Thus, bugs are the differences between the correct plans and the incorrect implementations used by novices (13).

Soloway believes a program has two audiences (11), as shown in Figure 1. Soloway uses this to conclude: "learning to program amounts to learning how to construct mechanisms and how to construct explanations" (11).

The Audiences for a Program
◇ *The computer*, which, based on instructions is a *mechanism* for *how* a problem is solved.
◇ *The human reader*, who needs an *explanation* for *why* the program solves the problem.

Figure 1: Program Audience

3 "Are You Crazy!?"

The title of this section was frequently shouted at us because, *everybody in the world knows:*

- emacs is impossible to learn and use,
- TeX is impossible to learn,
- `WEB`'s steps make it too many steps to learn, and
- there is a reason for all those Aggie jokes.

and **therefore** our project was doomed!

Well, we are Aggies and we decided to try teaching CS/1 using all those horrible things. We exercised a little judgement and did it on the smallest sections of the course, namely the honors sections.

The next four subsections are the things that we did that are different from our usual CS/1 course. They are presented in the order that the students saw them.

3.1 Testing

The first meeting of the laboratory included some quizzes that did not affect the grade but were done to determine the students' backgrounds.

During the semester there were some different questions on tests that addressed problem solving more than usual.

Introductory computer science students have difficulty viewing programming as a means by which we solve problems. Computer science instruction, at the introductory level, tends to emphasize programming, which is the *product* of problem solution design (6). Most textbooks give examples of programs, rather than demonstrate the method by which the given solution was derived (6).

It has been said the use of literate programming allows us to associate a given design step with its consequences; that is, the resulting code (15). Students should be taught that problem solution *design* leads directly to the result, which is the program. The use of literate programming encourages the inclusion of the design step in the source of the resulting program.

The results of the research were used to determine whether improvements in problem solving and programming skills can be attributed to the use of literate programming. An evaluation of the teaching methodology was made based on several factors:

1. Completion of a pre-test which was developed to indicate the students' problem solving ability and computing background as they entered the course.
2. Periodic tests which were designed to indicate the change in problem solving ability and programming skills.
3. An evaluation of the programs and documentation produced and the consistency between code and its corresponding documentation.
4. Completion of a post-test which indicates the students' ability to solve problems and write programs at the end of the test period.
5. An evaluation of the students' performance in the subsequent Programming II course.
6. An evaluation of the students' performance in the subsequent Data Structures course.

The results were expected to indicate an increase in problem solving ability over time. Programmers who use the literate programming paradigm were expected to be more *problem-oriented* rather than *program-oriented.*

3.2 Emacs and `web-mode`

We decided to use Mark Motl's `web-mode` environment (1; 10). This keeps the neophyte (and expert) user from making a number of simple mistakes that are easily committed. It relieves the user of knowing how and where to insert that necessary TEX mumbo-jumbo that `WEB`s start with; it ensures matched `@<` and `@>` pairs (complete with the "`=`" when needed); and allows selection of existing section names (rather than having to type it identically).

At a later time, it will help them in other ways by allowing navigation in terms of `WEB` terminology.

Emacs was introduced with a one page handout and a modified version of the 'Emacs Reference Card.' The reference card is printed on 8.5`"`×`11"` paper, two sided, with three panels on each side. The modifications were to remove some of the more advanced features of Emacs and replace them with `web-mode` features. Emacs was covered in one hour of laboratory time with some questions and answers at the start of subsequent periods.

3.3 Knuth's `WEB`

The subsequent courses in our curriculum are based upon having some use of the Pascal language in CS/1. Thus, we selected a current implementation of Knuth's original `WEB` (4), which is Pascal based.

The current implementation means that we did not convert the output of **`tangle`** to all upper case, shorten variable names, nor remove underscores. The resulting output of **`tangle`** is still relatively "unfit for human consumption."

The rules of `WEB` were covered by the use of a five page "memo" from the `WEB` distribution, Knuth's "`WEB` User's Manual."

3.4 How The Course Was Taught

The focus of the semester was on problem solving. The students were taught Pascal syntax, but the emphasis was on problem solving using the `WEB` style of programming. A portion of the class was spent on learning (and evaluating) problem solving skills for the design and development of programs. One method by which problem solving was taught was by example. The students were given several examples of how to design solutions to a problem. This technique of problem solving with examples was used throughout the semester as the difficulty of the problems increased.

An important part of learning problem solving was to practice iteration in the design of a solution. An iteration of the students' problem solution was evaluated by the teaching assistant. The students received feedback regarding their iterative process, such as whether they were approaching the details of the problem at an acceptable level and whether they were considering all aspects of the problem.

The final measurement in the design and development phase was made upon completion of the program assignment. Each program was examined and an evaluation made as to the correctness of the solution, the consistency of documentation and code, and the quality of the documentation. The intent was to determine if the documentation portion of a section was, in fact, an explanation of the code.

3.5 Do All Labs Twice

We were particularly fortunate that when the curriculum was revised and a formal laboratory was added, the professor in charge decided that the laboratory meeting would not be one extended period, but two one hour periods with a day in between.

We used this to great advantage to require that each lab to be turned in for grading twice:

- Do the first three parts of the problem solving procedure outlined above **without any code!** We wanted the student to document that the problem was understood! `WEB` can be considered to be a structured system of pseudo-code and is therefore ideal for this purpose.
- The 47 hour lapse between laboratory meetings enabled the grading of those important first steps of problem solving to assist the student in understanding what is to be converted to code.

4 Initial Reactions and Thoughts

Students in first year courses are often rather intimidated but generally ready for any challenges that might arise. Of course we have that same experience. It is interesting to note those that were not 'the usual.'

CS/1 courses will often have a number of students who have had one to five years of computer experience, much of it unstructured. We certainly had our share.

There were a number of students who had at least one year of the use of Turbo Pascal in secondary school and who had obviously used it significantly outside that educational environment. Testing showed these students to have two general characteristics:

- a lack of understanding of how to state a problem.
- a great desire to do nothing other than use Turbo Pascal.

It was also common for these students to react in rather vigorous ways. We think that it is a characteristic of those in the programming professions to resist change unless it is change that they tried to start.

Nearly half the class indicated no programming background from their secondary training. (They may have had word processing, spread sheets, and computer math; but they indicated no programming. Further, they were frequently not CS majors.)

Thirty-eight students enrolled in the honors class during the Fall 1993 semester. The administration of a pre-test provided information regarding the general background and experience of the participants. The purpose of the pre-test was to establish that these were, in fact, novice programmers. The results of the problem solving portion of the test provided a basis for measuring the initial problem solving skills of the participants.

The students entered the course with a variety of backgrounds in computer science. Only one student had never taken a computer science course and one student had taken only a computer literacy/computer history course. Few of the students had any background in computer science at the college level. Table 1 is a summary of the college level experience of the participants.

Table 1: Unusual or Exceptional Computer Experience of Subjects

Count	Exceptional Experience
1	C course at a Junior College
4	University level Fortran course

The majority of the students had some type of computer science class in high school. Table 2 is a summary of the high school experience of the participants. Although there were thirty-eight students enrolled in the class, many of the students had experience in more than one of the areas listed.

Despite the appearance of having a significant background in computers, these students must still be considered novice programmers. Although a significant number had some background in Pascal programming, fifteen felt they could program without the use of a reference manual. Even so, their knowledge of advanced Pascal constructs

Table 2: High School Computer Experience of Subjects

Count	Computer Experience
8	Microcomputer applications, typically including DOS, WordPerfect, Lotus 1-2-3, and/or dBase
8	Computer Math, which may or may not include some experience in BASIC and/or Pascal
12	BASIC course
21	One or more semesters of Pascal

cannot be considered to be comprehensive. None of the students had experience as a professional programmer.

One student had limited experience with the emacs editor. The remaining thirty-seven had no experience with emacs. None of the students had heard of `WEB` programming; therefore, none of the test study participants had previous experience with literate programming.

The pre-test included a question designed to provide some measurement of the students' initial problem solving ability. The students were asked to state the steps necessary to solve a given problem. They were instructed to give detailed answers in complete English sentences and paragraphs. The problem was stated as follows:

> You are the manager of Aggie Lawn Service. Alvin is your new employee. You must explain to Alvin the process of calculating an estimate for a potential customer. (Of course, in the future this may use a hand-held computer.) The quote will include a cost statement and estimated time to complete the job.
>
> This estimate is based upon the area of the lawn and a standard (confidential) charge per square foot. Grass can be cut at the rate of 2 square feet per second. You may assume that a rectangular house is situated in a rectangular yard. Give the details of the process and itemize all assumptions you have made.

It is difficult to measure a person's problem solving ability. For example, if it is easily seen that the problem is a basic input-process-output problem, then each subject should receive points if the necessary inputs and the required outputs were described. In terms of the processing, many students felt it was sufficient to merely give the formula for the area of a rectangle. They then subtracted the area of the house from the area of the lawn (sometimes shown, again, as a formula).

In general, most of the students were able to give an answer which solved the problem. However, several exceptions were noted as follows:

- some participants simply gave the necessary formula(s), omitting any description of the inputs and/or outputs;
- some participants failed to describe their solution using complete English sentences and paragraphs;
- some participants described the necessary inputs and the required processing, but failed to produce a result; and
- some participants made and described additional assumptions or expressed a need for additional information regarding items such as driveways, sidewalks, trees, flower beds, etc.

The students' solutions were scored based on their ability to solve the problem. Table 3 is a summary of the minimal set of problem solving issues that should have been addressed or noted, with their associated point value.

Table 3: Problem Solving Issues

Points	Problem Solving Issue
2	Obtain dimensions of yard
2	Obtain dimensions of house
2	Calculate area for house and yard
2	Calculate area for lawn to be cut
3	Calculate total cost to cut the lawn
3	Calculate the time for completion
2	Convert the time to minutes or hours
2	Produce the final cost for cutting lawn
2	Produce the time for completion

A final score of twenty indicates that the student adequately described the required inputs, calculations, and necessary outputs. A student lost points for omitting information or not describing the process in sentence form. A student could earn extra points by addressing issues that were not explicitly mentioned, but might be a factor in solving the problem.

Table 4 is a summary of the results of measuring the students' initial problem solving ability. There are 47.4% above and only 31.6% below average. The grade of "C" is described as average, yet it is rare that a class will have as many D's and F's as A's and B's. The distribution of the data in Table 4 is consistent with grade

preliminary draft, July 17, 1995 11:30

Table 4: Initial Problem Solving Ability

Percent of Students	Problem Solving Ability
31.6	Excellent (18+ points)
15.8	Above average (16-17 points)
21.1	Average (14-15 points)
13.2	Below average (12-13 points)
18.4	Poor (below 12 points)

distributions for the CS/1 course over the last few years.

5 Results

We feel the background of the students is not atypical of many CS/1 type courses. The majority of the class are majoring in computer science, but a significant number are using the course as a minor elective, a basis for deciding if they want CS as a major, or other reasons.

Some results will be presented with this diversity as an identifying factor. Results will also reflect the tracking of the students in subsequent courses and differences between other semesters of the same course.

5.1 Performance during the semester

The actual scores received by the test group on the problem solving portion of each test are included in Appendix D. The mean of the scores for the problem solving portion of each test are shown in Table 5.

Table 5: Mean Problem Solving Scores – Tests (Percent)

Test	Overall	Majors	Non-Majors
Pre-Test	72.6	74.0	68.3
Test 1	78.8	79.7	76.1
Test 2	66.6	65.7	71.6
Test 3	80.9	80.3	82.7
Post-Test	76.6	76.2	77.8

It is difficult to determine whether or not the problem solving skills for the test group increased over the course of the semester. The class, as a whole, experienced a decrease in scores on the second test, although there was a greater decrease for computer science majors. This decrease in scores for the second test may be attributed to the fact that the problem for that test was significantly different and more difficult than any of the problems encountered previously during the lab or on a test. The scores also decreased on the post-test, or final exam, as compared to the third test; however, they still improved as compared to the scores on the pre-test.

The problem solving scores, as a whole, were higher on the labs than they were for the exams. This was to be expected since the problem solving portion of the lab was not developed under stressful situations, as in the test-taking scenario. Another reason for having higher scores in the lab is that measuring problem solving skills is not something we are used to doing on a test. It is much easier to evaluate someone's problem solving skills developed through iteration during lab than it is to evaluate one-time problem solving skills on a test.

Table 6 is a summary of the overall grade distribution for students completing the CS/1 course for the subject and comparison classes (in percent form).

Table 6: Overall Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	20.6	50.0	14.7	5.9	8.8
Fall 92-H	51.3	20.5	20.5	2.6	5.1
Fall 93-H	24.3	40.5	21.6	5.4	8.1

The percentage of students that passed the CS/1 course was similar for each of the classes. A grade of "A", "B", or "C" is considered passing. The Fall 1990 and the Fall 1992 comparison groups had 85.3% and 92.3% of the students, respectively, pass the course. The test group had 86.4% of the students pass the course.

5.2 Performance in CS/2

Approximately 65-70% of the honors CS/1 students enrolled in the CS/2 course (73.5% of the Fall 1990 class, 66.7% of the Fall 1992 class, and 67.6% of the Fall 1993 class).

Table 7 is a summary of the overall grade distribution for the subsequent CS/2 course for those students in the subject and comparison classes in percent form.

Table 7: Overall CS/2 Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	68.0	28.0	4.0	0.0	0.0
Fall 92-H	73.1	19.2	7.7	0.0	0.0
Fall 93-H	52.0	40.0	4.0	0.0	4.0

At first glance it appears that the students in the Fall 1990 honors and the Fall 1992 comparison classes performed much better than the students in the test study group in the CS/2 class. Both of the comparison groups had a higher percentage of students make "A"s in the subsequent course. However, all of the classes had over 90% of the students make an "A" or a "B" in the course.

Table 8 is a comparison of the average grades in the CS/1 class and the subsequent CS/2 class for those students in the subject and comparison classes. The grade point shown is out of a total possible grade of 4.0. The Mann-Whitney U-test was used to conclude that there is not a significant difference in average grade point ratio for any of the groups.

Table 8: Average Grade for CS/1 and CS/2 Courses

Semester	CS/1	CS/2
Fall 90-H	2.676	3.640
Fall 92-H	3.103	3.654
Fall 93-H	2.676	3.360

This may still not be a good representation of how the students in the subject and comparison classes performed in the subsequent course. These grades can be evaluated in terms of the particular section and semester the class was taken and the instructor that taught the class.

Table 9 is a summary of the average difference in grades between the subject class, the comparison classes, and the other CS/2 classes. This summary is itemized by section, instructor, and semester.

Table 9: Average Difference in Grade for CS/2 Classes

Semester Semester	Difference in Section	Difference in Instr.	Difference in Semester
Fall 90-H	+0.02	+0.01	+0.01
Fall 92-H	+0.03	+0.01	+0.01
Fall 93-H	+0.06	+0.05	+0.09

With these figures, it is shown that the students in the CS/1 comparison classes scored somewhat higher than their peers in the same section of the CS/2 course. However, those students in the CS/1 test group scored even higher than their peers in the same sections of the CS/2 course. This data was also analyzed including the CS/2 instructors and semester. The same results held.

When the performance of the students in the test study group was compared with the performance of their peers, it was determined that the students in the test study group actually scored higher than the students in the comparison groups (and the other students) in the CS/2 course.

5.3 Performance in Data Structures

Approximately 45-55% of the honors CS/1 students enrolled in the Data Structures course (55.9% of the Fall 1990 class, 56.4% of the Fall 1992 class, and 45.9% of the Fall 1993 class).

Table 10 is a summary of the overall grade distribution for the Data Structures course for those students in the subject and comparison classes in percent form.

Table 10: Overall Data Structures Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	21.1	63.2	15.8	0.0	0.0
Fall 92-H	50.0	13.6	22.7	9.1	4.5
Fall 93-H	52.9	35.3	11.8	0.0	0.0

Not only did the test study group have a larger percentage of students make an "A" in the course, but a larger percentage of students made an "A" or a "B" in the course.

Table 11 is a comparison of the average grades in the CS/1 class, the CS/2 class, and the Data Structures class for those students in the subject and comparison classes. Again, the grade point shown is out of a total possible grade of 4.0. Using an unpaired t-test, with $\alpha = 0.10$, it was concluded that there is a significant difference in average grade for the Data Structures course between the Fall 1993 test group and both the Fall 1990 and the Fall 1992 comparison groups.

Table 11: Average Grade for CS/1, CS/2, and Data Structures Courses

Semester	CS/1	CS/2	DS
Fall 90-H	2.676	3.640	3.053
Fall 92-H	3.103	3.654	2.955
Fall 93-H	2.676	3.360	3.412

A chi-square test of independence was conducted to determine if the grade and CS/1 semester variables are related (or dependent). The critical value of X^2 for $\alpha = 0.10$ and degrees of freedom = 8 is 13.36. The computed value, 15.368, exceeds 13.36, so we conclude that the two variables are dependent. That is, the proportion of students receiving a particular grade varies depending on the semester in which they took CS/1.

When the performance of the students in the test study group was compared with the performance of their peers in a course which requires extensive problem solving skills, it was determined there is a significant difference in the performance of the students in the test study group compared with the performance of the students in the comparison groups.

5.4 Student Evaluation of CPSC 110 Teaching Methodology

Upon nearing completion of the CS/1 course, the students were asked to submit a paper reflecting their feelings and attitudes towards the WEB programming methodology. It was stressed that statements made would in no way affect their grade in the course. This was to be written as a typical one-page technical note.

Three people evaluated the reaction of the test subjects. None of the people had prior training in rating. A rating scale (9) was developed and the reports were evaluated in order to appraise the students' reactions to the WEB programming process. The scale consisted of five categories, rated 1-5 and a 0 value that was taken to mean no response.

Below is a summary of the results of the rating process. The mean of the scores (columns R-1, ..., R-3) for each of the three volunteers who rated are shown in Table 12. Kendall's coefficient of concordance (9) was used to test agreement between the ratings. The result was a value of 0.673, which indicates there was a modest level of agreement between the evaluations of the questionaire results.

Table 12: Evaluation of Fall 1993 CPSC 110H Students' Reactions

Question	R-1	R-2	R-3	Overall
1	3.21	2.81	2.52	2.85
2	3.20	1.60	3.25	2.67
3	2.69	2.64	1.87	2.43
4	3.44	3.14	1.67	2.88
5	3.41	3.28	2.90	3.20
6	3.54	2.87	3.57	3.31

The questions and some comments of interpretation were:

1. What was your original reaction to being told you were going to learn something called WEB?

   Although a few of the students were enthusiastic about the idea, many were unhappy with the fact that they were going to be using a different methodology. Much of the unhappiness was due to the fact that many of the students entered the course with prior expectations about what is taught in the class.
2. What was your expectation of the course (answers varied from 'unknown ≡ 0' to 'Turbo Pascal ≡ 3' to 'Problem Solving and Programming ≡ 5').

   Most students entered the course under the impression that CPSC 110H was a course in Turbo Pascal, despite the course description.

   The answers to the rest of the questions were: 'not discussed'; poor; fair; average; good; and excellent receiving numeric values of 0, ..., 5 respectively.
3. What was your reaction to emacs?

   Many of the students objected to the use of the emacs editor. This may be due to the fact that the user interface is not extremely user-friendly, especially to the novice user. The students were required to use predefined keystrokes, rather than pull-down menus. (This was apparently because the question was asked. It did not show up on the course evaluation.)
4. What was your reaction to TeX?

   Although a minimal amount of TeX knowledge is required, the students seemed to find the language difficult. Although several examples were provided, with a variety of TeX commands, they students did not seem to adapt well to the use of TeX. Despite the lack of TeX knowledge, the students seemed to adapt to the WEB environment. (Same comment?)
5. What was your reaction to WEB programming?

   The evaluators of the students' reaction seem to believe this response was a bit above average. The lack of enthusiastic response may have been due to their overall difficulty in understanding the WEB process and concepts.
6. What was your reaction to the overall WEB process/concepts.

   Generally, the students' understanding was average to good. Many of the students continued to have difficulty separating the concepts of editor, WEB files, TeX commands, etc. Some seemed overwhelmed in the beginning with having to learn more than just 'a language.'

There were certainly a number of students who can code but did not catch the 'big picture.' Comments like why document when you have the code to read were not uncommon.

6 Conclusions

We taught an honors section of a CS/1 course in a different manner than usual, namely using literate programming. The students used an editor, a formatting system, and a coding style that was new to all. The students' performance in subsequent courses was not hurt and may have been helped with the different methodology. The results of using the program development methodology in the CS/1 course indicate that the methodology is successful in teaching novice programmers good problem solving skills.

These are the results of the experiment:

- The students showed an increase in their problem solving skills.
- Those students unfamiliar with the Pascal programming language, or any other programming language, were more successful then those familiar with Pascal at using the literate programming paradigm to capture and document their problem solving process.
- The students were able to learn the `WEB` rules, the `web-mode` environment, GNU Emacs, and TeX rules, as well as the Pascal syntax and constructs.
- Those students exposed to the program development methodology utilizing the literate programming paradigm were as successful in the subsequent CS/2 course as those not exposed to the methodology.
- Those students exposed to literate programming were significantly more successful in the Data Structures course than those not exposed to the methodology.
- The subject program development methodology may lead to an improved software development process; however, more tests should be conducted.

Norman Ramsey (author of Spider and NoWEB) has recently presented a position paper at an ICSE '95 workshop entitled "Literate Programming should be a model for Software Engineering and Programming Languages," dated March 1995. While we are in agreement with the obvious philosophy, we are concerned that it is too late because we have observed first year students are already like the professionals: "No, I do not want to learn anything new if I already have some knowledge in the area." We think it should appear early in the curriculum and repeatedly.

7 Recommendations and Future Use

In teaching a CS/1 course, you can do darned near anything and succeed. You just have to keep your eye on the goal, don't apologize, and push! We have known of real problems because the professor (assigned to the class at the last minute) often is less familiar with the specifics of 'Turbo Pascal' than many of the students. The lesson from that is "give them a new challenge." We feel that there is a real benefit to the use of literate programming and requiring that students practice writing, using of pseudo-code, and documenting their programs. It is a new challenge to them.

The following is a list of things we wish we had or recommend to similar projects:

1. It sure would be nice if we have some video training on how to do some of these things, particularly simple TeX, emacs, and DOS.
2. Now that we have METAPOST and other drawing packages, we think that more diagrams should be included in the first lab attempts.
3. Make students write! Repeat "**Make students write!**"
4. This is a natural for "cooperative learning;" have the students do **extensive** peer review.
5. It would be easier if we had some tools that would extract a component of the grade based upon "user supplied" index entries, long variable names, variable names made of words from the dictionary, ... Then the person should add significant markup from reading the English.
6. Assign some labs that are extensions of previous work to let them see what maintenance is. If you have time to let them work on programs done with and without literate programming. Otherwise, they will never learn! This can be difficult in a first course, but should always be part of later courses.

8 References

References

[1] **Cameron, D., and Rosenblatt, B.** *Learning GNU Emacs.* O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[2] **Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R.** Computing as a discipline. *Communications of the ACM 32*, 1 (January 1989), 9–23.

[3] **Dunn, D. L. B.** *Literate Programming As A Mechanism for Improving Problem Solving Skills.* PhD thesis, Texas A&M University, College Station, TX, May 1995.

[4] **Knuth, D. E.** The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, Sept. 1983.

[5] **Knuth, D. E.** Literate programming. *Computer Journal* (May 1984), 97–111.

[6] **Linn, M. C., and Clancy, M. J.** The case for case studies of programming problems. *Communications of the ACM 35*, 3 (March 1992), 121–132.

[7] **Linn, M. C., and Dalbey, J.** Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist 20*, 4 (1985), 191–206.

[8] **Linn, M. C., Sloane, K. D., and Clancy, M. J.** Ideal and actual outcomes from precollege pascal instruction. *Journal of Research in Science Teaching 24*, 5 (1987), 467–490.

[9] **Meister, D.** *Behavioral Analysis & Measurement Methods.* John Wiley & Sons, Inc., New York, 1985.

[10] **Motl, M. B.** *A Literate Programming Environment Based on an Extensible Editor.* PhD thesis, Texas A&M University, College Station, TX, December 1990.

[11] **Soloway, E.** Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM 29*, 9 (September 1986), 850–858.

[12] **Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J.** What do novices know about programming? In *Directions in Human-Computer Interaction*, B. Shneiderman and A. Badre, Eds. Ablex Publishing Corp., Norwood, NJ, 1982, pp. 27–54.

[13] **Spohrer, J. C., and Soloway, E.** Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM 29*, 7 (July 1986), 624–632.

[14] **Tucker, A. B.**, Ed. *Computing Curricula 1991 – Report of the ACM/IEE-CS Joint Curriculum Task Force* (New York, NY, December 1990), Association for Computing Machinery.

[15] **van Ammers, E. W.** Communication on July 16, 1993 at 7:05 CDT. Literate Programming Mailing List. e-mail: `ammers@rcl.wau.nl`.

9 Permission to Distribute