

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

**LITERATE PROGRAMMING AS A MECHANISM FOR
IMPROVING PROBLEM SOLVING SKILLS**

A Dissertation

by

DEBORAH LYNN BYRUM DUNN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 1995

Major Subject: Computer Science

UMI Number: 9534331

UMI Microform 9534331

Copyright 1995, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

**300 North Zeeb Road
Ann Arbor, MI 48103**

LITERATE PROGRAMMING AS A MECHANISM FOR
IMPROVING PROBLEM SOLVING SKILLS

A Dissertation

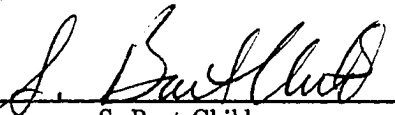
by

DEBORAH LYNN BYRUM DUNN

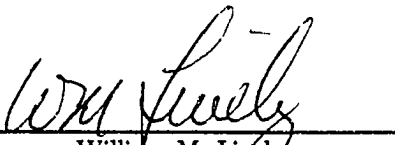
Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

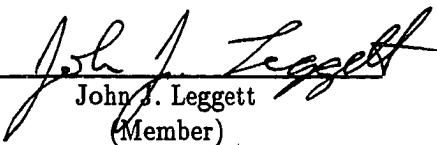
Approved as to style and content by:



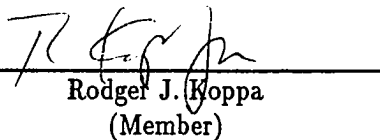
S. Bart Childs
(Co-Chair of Committee)



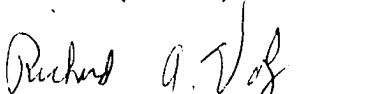
William M. Lively
(Co-Chair of Committee)



John J. Leggett
(Member)



Rodger J. Koppa
(Member)



Richard A. Volk
(Head of Department)

May 1995

Major Subject: Computer Science

ABSTRACT**Literate Programming as a Mechanism for
Improving Problem Solving Skills. (May 1995)****Deborah Lynn Byrum Dunn, B.B.A., Stephen F. Austin State University;****M.S., Stephen F. Austin State University****Co-Chairs of Advisory Committee: Dr. S. Bart Childs
Dr. William M. Lively**

Software maintenance is becoming an increasingly important factor in determining software costs. Researchers are investigating methodologies for improving program development and documentation which may, in turn, reduce maintenance costs. The result of the design phase of software development affects the quality of the code which is written and implemented. The ability with which a programmer solves a given problem directly affects the quality of the program developed for the solution.

An increasing amount of research is being performed in the area of problem solving and methods by which we teach novice programmers to solve problems. Many of the difficulties experienced by novice programmers are not a result of misunderstanding the language constructs, but a result of problems with forming a solution to the problem. The manner in which a novice programmer solves a problem will directly affect the program that is produced.

Knuth coined the phrase "literate programming" to refer to programs which are meant to be read by humans, as well as executed by a computer. His WEB programming methodology was designed to encourage pseudocode development, stepwise refinement,

and documentation of code, including the design rationale, prior to the actual writing of code. The methodology should also produce programs in which the documentation is highly correlated with the code.

This research involved the adaptation of a methodology which can be used to improve the software development process and the evaluation of programs which are developed in introductory computer science courses. The methodology combines literate programming with the concepts of problem solving to capture, document, and emphasize the problem solving process. The production of well-designed, readable, maintainable software for the solution of problems is the goal.

The methodology was tested and the results compared with previous introductory computer science classes. A group of novice programmers with limited programming experience utilized the methodology successfully in the development of problem solutions. The design solutions were then successfully used in the implementation of the accompanying programs. Since the implementation of the methodology was successful for the study, we feel the adaptation of the methodology is viable and should be tested in successive classes.

To my husband Henry and my parents Burton and Aura. Their constant love and support
allowed me to complete this work.

ACKNOWLEDGMENTS

I want to thank Dr. Bart Childs for providing me the guidance to accomplish this research. His patience and encouragement gave me the strength to complete this work. I feel privileged to call him a friend and mentor. I also want to thank Dr. William Lively for providing me the guidance and support to complete this research. I will never forget these two men.

I also sincerely thank the members of my committee, Dr. John Leggett and Dr. Rodger Koppa. Both members have provided guidance and encouraged me and I am proud to have worked with such fine researchers. A special thanks goes to Dr. Stephen Smith as Graduate Council Representative for providing support and encouragement.

Others deserving a very special note of gratitude are Peter Nuernberg for assisting with the test study and providing feedback on my ideas; Marie Legare for keeping me sane; Michael Vidlak for his constant friendship and support; Hiroko Fujihara for always being there; my father for never letting me lose sight of my goal; and especially my mother for always encouraging me and never letting me forget to laugh.

Finally, I thank my husband Henry for his love, support, encouragement and, most of all, his patience.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
DEDICATION	v
ACKNOWLEDGMENTS	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	x
LIST OF FIGURES	xii
 CHAPTER	
I INTRODUCTION	1
I.A Background	1
I.B Research Objectives	3
I.C Overview	3
II LITERATURE SURVEY	4
II.A The Software Development Process	4
II.B Problem Solving	6
II.C Literate Programming	13
II.D Software Engineering Concepts	19
II.E Using Literate Programming to Teach Good Programming Practices	21

TABLE OF CONTENTS (CONTINUED)

CHAPTER	Page
II.F Using Literate Programming to Improve Problem Solving Skills . .	22
III DESIGN OF A TEST STUDY	23
III.A Selection of Participants	23
III.B Teaching Introductory Students	29
III.C Literate Programming and the Design of Solutions	30
III.D Design	51
III.E Participants	52
III.F Methods of Measurement	52
IV IMPLEMENTATION OF A TEST STUDY	57
IV.A Test Study	58
IV.B Teaching Assistant	61
V RESULTS	62
V.A Background/Experience of Test Study Participants	62
V.B CS/1 Class Information	66
V.C Student Classification Distribution	67
V.D Problem Solving Performance	68
V.E Programming Performance	71
V.F Exam Performance	75
V.G Course Performance	78

TABLE OF CONTENTS (CONTINUED)

CHAPTER	Page
V.H CPSC 120 Performance	82
V.I CPSC 210 Performance	85
V.J Student Evaluation of CPSC 110 Teaching Methodology	90
VI SUMMARY, CONCLUSION, AND FUTURE WORK	93
VI.A Summary	93
VI.B Conclusion	95
VI.C Extensions and Future Research	96
REFERENCES	98
APPENDIX	
A COURSE MATERIALS	103
B OVERALL COURSE STATISTICS	189
C INDIVIDUAL COURSE STATISTICS	192
D INDIVIDUAL PROBLEM SOLVING TEST STATISTICS	215
E SUMMARY CS/2 COURSE STATISTICS	217
F STUDENT REACTIONS TO WEB	222
VITA	268

LIST OF TABLES

TABLE		Page
1	Unusual or Exceptional Computer Experience of Subjects	63
2	High School Computer Experience of Subjects	63
3	Problem Solving Issues	65
4	Initial Problem Solving Ability	66
5	Student Distribution by Classification (Percent)	68
6	Student Distribution by Major (Percent)	68
7	Mean Problem Solving Scores – Labs (Percent)	69
8	Standard Deviation of Problem Solving Scores – Labs (Percent)	69
9	Mean Problem Solving Scores – Tests (Percent)	70
10	Standard Deviation of Problem Solving Scores – Tests (Percent)	70
11	Mean Program Scores	74
12	Standard Deviation of Program Scores	74
13	Mean Exam Scores	77
14	Standard Deviation of Exam Scores	77
15	Overall Grade Distribution (Percent)	79
16	Grade Distribution for CPSC/CSEN Majors (Percent)	79
17	Grade Distribution for Other Majors (Percent)	80
18	Overall CS/2 Grade Distribution (Percent)	82
19	Average Grade for CS/1 and CS/2 Courses	84
20	Average Difference in Grade for CS/2 Classes	84

LIST OF TABLES (CONTINUED)

TABLE		Page
21	Overall Data Structures Grade Distribution (Percent)	86
22	Average Grade for CS/1, CS/2, and Data Structures Courses	88
23	Evaluation of Fall 1993 CPSC 110H Students' Reactions	91
24	Standard Deviation of Rating Scale	91
25	Student Distribution by Classification (Actual)	189
26	Student Distribution by Major (Actual)	189
27	Overall Grade Distribution (Actual)	190
28	Grade Distribution for CPSC/CSEN Majors (Actual)	190
29	Grade Distribution for Other Majors (Actual)	190
30	Overall CS/2 Grade Distribution (Actual)	191
31	Overall Data Structures Grade Distribution (Actual)	191
32	Problem Solving Statistics (Actual)	216
33	CS/2 Course Statistics - 1991	218
34	CS/2 Course Statistics - 1992	219
35	CS/2 Course Statistics - 1993	220
36	CS/2 Course Statistics - 1994	221

LIST OF FIGURES

FIGURE	Page
1 Contents of a Case Study	9
2 Contents of an Expert's Template	10
3 Knowledge Not Possessed by Novice Programmers	11
4 Program Audience	12
5 The WEB Process	16
6 Defining a Problem	20
7 Design Example – Table of Contents	33
8 Design Example – Problem Statement	34
9 Design Example – Inputs Required	35
10 Design Example – Outputs Required	36
11 Design Example – Processing Required	37
12 Design Example – Algorithm Development	38
13 Design Example – Testing	39
14 Design Example – Index	40
15 Program Example – Table of Contents	41
16 Program Example – Problem Statement	42
17 Program Example – Inputs Required	43
18 Program Example – Outputs Required	44
19 Program Example – Processing Required	45
20 Program Example – Algorithm Development	46

LIST OF FIGURES (CONTINUED)

FIGURE		Page
21	Program Example - The Actual Program	47
22	Program Example - Testing	48
23	Program Example - Index	49
24	Program Example - List of Sections	50
25	CS/1 Course Grade Distribution	81
26	CS/2 Course Grade Distribution	83
27	Data Structures Course Grade Distribution	87
28	Grade Progression	89

CHAPTER I

INTRODUCTION

I.A Background

The software crisis has been described as the inability to develop reliable, maintainable software in a timely, economical manner [3, 4, 6, 22, 26]. It is the focus of much attention and concern by software engineers. A method by which we can reduce the cost of developing software has been a goal for researchers in software engineering.

One of the predominant goals in developing software is to reduce both development and maintenance costs. Software maintenance is a factor in determining software costs as additional software is being developed. Typically, maintenance costs are approximately seventy percent of the total software life-cycle costs [22]. The development cycle of a software product may span one or two years, while the maintenance cycle can span five to ten years [22]. Therefore, ease of maintenance is an important consideration in software development.

Researchers are investigating methodologies for improving program development and documentation which may, in turn, reduce maintenance costs. Soloway, et. al. [49, 68, 69] explored the “design of software documentation for maintenance” in order for the maintainer of a program to be better able to understand the design rationale. The goal of *literate programming*, a concept introduced by Donald Knuth [28], is “instead of imagining that our task is to instruct a *computer* what to do, let us concentrate rather on explaining This dissertation was prepared in the format of *Communications of the ACM*.”

to *human beings* what we want a computer to do.” He stated “documentation is at least as important as programming” [28].

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” [69]. Thus, the process by which programs (and documentation) are developed should be examined.

An examination of the methods by which students are taught to program has led to the conclusion that there is a failure to provide explicit instructions in the area of problem solving [38]. Introductory computer science classes emphasize language features and general programming practices, although course syllabi often emphasize problem solving techniques.

Soloway states the major stumbling block is not the syntax of a language, but the composition and construction of a program. He also suggests the way to overcome the problem is to shift the method in which our introductory computer science students are taught. He uses the concept of *goals* and *plans* to emphasize design rather than syntax [64].

The focal point of this research is the adaptation of a methodology which can be used to improve the software development process and the evaluation of the programs which are being developed. The methodology combines literate programming with the concepts of problem solving to capture, document, and emphasize the problem solving process. The production of well-designed, readable, maintainable software for the solution of problems is the goal.

I.B Research Objectives

The objectives of this research are to determine the effects of:

1. emphasizing software engineering concepts of problem specification and design;
2. including the design rationale and expected test results in the program as documentation;
3. developing more readable, understandable software as a result of the iterative problem solving process utilizing the framework of literate programming;
4. using the literate programming paradigm in emphasizing problem solving by iteration, review, and feedback; and
5. peer review as part of the design and development process.

This dissertation is the result of the development of a problem solving methodology which may improve the manner in which software is designed and developed.

I.C Overview

A survey of the literature in the areas of problem solving and literate programming is in Chapter II, including:

- the research that has been performed in the area of learning to program;
- problem solving and where it fits into the program development cycle; and
- Knuth's WEB system and the concept of web programming.

The design of a test study to determine the effects of literate programming on problem solving is presented in Chapter III.

A discussion on the implementation of the test study is in Chapter IV.

A description of the comparison groups for the test study and the results of the test study are presented in Chapter V.

Finally, a summary and the conclusion for the experiment may be found in Chapter VI. Extensions to the test study and possible future studies are presented.

CHAPTER II

LITERATURE SURVEY

The first three steps in the software development process are requirements, specification and design [19]. The result of the design phase affects the quality of the code which is written and implemented. Therefore, the ability with which a programmer solves the given problem directly affects the quality of the program developed for the solution. Extensive research has been performed in the area of learning to program [12, 23, 25, 34, 35, 37, 38, 58, 64, 69]. A competent programmer requires a knowledge of programming language syntax and constructs, as well as good problem solving skills [35].

The literate programming methodology was introduced by Knuth upon his second writing of \TeX [28, 29]. The methodology encourages:

- correlation of internal code and documentation;
- pseudocode development;
- stepwise refinement; and
- the documentation of code, including the design rationale, prior to the writing of code.

This documentation of the design rationale has become an important factor in the development and maintenance of software [13, 14, 15, 33, 41].

II.A The Software Development Process

Software engineers have a goal of reducing software costs and increasing productivity [4, 6]. In 1976, Boehm [3] predicted that software costs would escalate to

over 80% of the total cost of a system (hardware and software). His prediction included the statement that the portion of the effort spent on software maintenance was (and would continue to be) greater than that spent on software development.

One of the techniques used for developing software is to adhere to a specific life-cycle model. There are a variety of software life-cycle models from which to choose [3, 5, 22, 24, 26, 60]. The purpose of a model is to provide some type of guidance on the order in which major tasks should be carried out in the development and maintenance of software. Fairley [22] states that "different models emphasize different aspects of the life cycle, and no single life-cycle model is appropriate for all software products." Regardless of the model name, all life-cycle models consist of some form of problem definition, analysis, and design.

The first step in software development (or maintenance, for that matter) is to clearly define the problem. Although this seems like an obvious and simple task, it sometimes takes a great deal of time and many iterations. It is important that the users of the system be involved during this phase, as well as subsequent phases.

The purpose of the analysis phase of software development is to determine the requirements of the proposed system [26]. The output from this phase is a problem specification which can then be used in the design of the system. A variety of tools and techniques, such as data dictionaries, data flow diagrams, and flowcharts, may be produced [22, 26, 60].

The design phase of the life-cycle model utilizes the outputs from the previous phases to create a system which fulfills the users' requirements [19, 22, 26, 60]. These preliminary phases (in all of the models) greatly affect the quality of the software that is developed.

And, the quality of the design greatly affects the amount of maintenance that may be performed on the system.

Boehm [22] estimates that 40% of the effort in the software life cycle is spent on development, while the remaining 60% is spent on maintenance. Of that 40%, only 8% is spent on implementation. The remaining 32% is spent equally on analysis/design and testing. It can be seen that a significant amount of time is spent on determining how the problem will be solved and testing the solution. In order to improve the quality of our software, we need to improve the methods by which we solve problems and develop comprehensive tests for our solutions.

II.B Problem Solving

A computer system is simply a problem solving tool [31]. We should concern ourselves with teaching students good approaches to using this tool; that is, better approaches to problem solving. The first step in problem solving is actually understanding the problem. Few of the problems presented in introductory textbooks are extensive [35]. Therefore, it is relatively simple to define the problem. It could be said that a precise understanding of the problem definition is itself a solution to the problem.

There are a variety of approaches to problem solving and programming. Wirth believes programming can be introduced “as the art or technique of constructing and formulating algorithms in a systematic manner, recognizing that it is a discipline in its own right” [79]. Introductory programming textbooks normally discuss some kind of approach to software development, whether it be structure charts, top-down design, divide-and-conquer, and/or pseudocode [30, 46, 57]. The students may be taught to verbalize the problem. One way

this may be accomplished is by peer review and feedback. Students may also be taught to visualize the problem. This may be accomplished with the use of pseudocode or English-like expressions. Flowcharts, hierarchical diagrams, structure charts, and sketches are used to give a better understanding of the problem [30, 46, 57]. Textbooks typically emphasize the top-down design techniques of breaking a problem down into parts [35].

Linn and Clancy [35] 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 [35]. Although this is obviously an important aspect of programming, the real problems exist in the design of problem solutions [38]. Few textbooks used in the introductory courses actually emphasize teaching the student how to develop good design solutions [35], regardless of the university catalog description.

Soloway [64] 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” [64].

Many disciplines at the university level are beginning to require their students take at least one computer science course. As a result, a large number of students enroll in the introductory computer science courses. There is some controversy as to whether or not all students should be taught to program [65]. The reason for the controversy stems from the

difficulty with which novice programmers learn to program. It is for this reason there is a growing amount of research in the field of learning to program.

II.B.1 Solving Problems by Example

Several studies have been performed in the area of learning by example. Pirolli and Anderson [50] found that examples can play an important part for students learning recursion. Reder, Charney, and Morgan [54] found that the most effective manuals for teaching students how to use a personal computer were those containing examples. The work of Chi, et. al. [12] focuses on the theory that differences in students' ability to solve problems may stem from the differences in the way they understand examples. They found that students learn well from example, provided they explain the examples to themselves while they are learning.

As a result of some of this research, a software tool called EXPLAINER has been developed [55]. The purpose of EXPLAINER is to help programmers solve problems by exploring previously worked-out examples. The software tool combines examples of code with knowledge about how the examples were solved.

Some related work has been performed in the area of determining the mental representations of programs that are formed by programmers [23]. Two sets of programmers, novice and expert, were asked to study a program and later recall certain information about the program. The results of the experiment showed that experts scored significantly higher than novice programmers. The study tends to support the fact that the experts have developed skills which help them develop better mental representations. This difference in mental representations may be attributed to the difference in programming knowledge and in program comprehension strategy.

II.B.2 Solving Problems With the Use of Case Studies and Templates

Linn, Sloane, and Clancy [38] 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 [37, 38].

Linn and Clancy have performed significant research in using programming case studies or *templates* [34, 35, 36, 38]. Figure 1 is a summary of the information contained in a case study [35].

A Case Study
◇ A programming problem statement.
◇ A narrative description of the process by which an expert solved the problem, written so a novice can understand the approach.
◇ The expert's source code.
◇ Study questions for practice in program design, analysis, and problem solving.
◇ Test questions designed to assess the students' understanding of the solution.

Figure 1. Contents of a Case Study

Expert programmers hold certain templates that may be communicated to novice programmers. The contents of these templates is summarized in Figure 2 [35].

Novice programmers often organize their knowledge in terms of syntax, rather than in terms of a conceptual algorithm. This may be a result of how the students are taught [35]. A case study can be used as a method to provide explicit instructions on how to combine the template knowledge with program design skills to solve a problem. The students must then practice generalizing their new skill to new problems. If too much emphasis is placed on explicit strategies, students will not learn problem solving skills [25].

An Expert's Template
◇ A general representation of the action of a component, in pseudocode form.
◇ Sample programs which use the component.
◇ A pictorial representation of the action of the component.
◇ Verbal descriptions that facilitate communication about the component.
◇ Implementation steps for incremental development of the component.
◇ Testing information, including possible difficulties that should be covered in the tests.
◇ Debugging information which includes possible bugs, their symptoms, and ways to anticipate them.
◇ Connections of the component to related components, to subtemplates, and to supertemplates.

Figure 2. Contents of an Expert's Template

The idea of templates has been extended to include on-line template libraries and case studies [58]. A study was devised in which three groups of subjects (novice, intermediate, and expert) were given access to an on-line template library. The template library contains a variety of templates of algorithms that are typically taught in the introductory programming courses. The subjects were then observed in order to ascertain how they organized, learned and applied the various templates [58].

As expected, the experts organized their templates in a more abstract manner than novice programmers. The case studies helped the novice programmers reuse templates and all subjects found that the code and pseudocode representations helped them write the code to solve new problems.

II.B.3 Solving Problems With the Goals and Plans

Soloway and colleagues [67, 69] 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 [69].

Two observations were made based on the goal/plan analysis of novice programmers [69]:

1. some bugs occur repeatedly in novices' programs, while others rarely occur; and
2. most bugs occur because students do not fully understand the semantics of certain programming language constructs.

Spohrer and Soloway also determined that novices have difficulty *composing* plans [69].

Soloway and Ehrlich [66] believe expert programmers harbor at least two types of knowledge not possessed by novice programmers. These types of knowledge are shown in Figure 3.

Expert Knowledge
◇ <i>Programming plans</i> , which are code fragments that represent typical action sequences in programs.
◇ <i>Rules of programming discourse</i> , which are rules that specify programming conventions.

Figure 3. Knowledge Not Possessed by Novice Programmers

Programs are created by using programming plans which are modified to fit the needs of the specific problem. The rules of programming discourse are used to govern the composition of the plans [66]. This lack of knowledge regarding plans and the rules of programming discourse is the reason for the difficulty experienced by novice programmers.

Soloway's research [64, 69] suggests that it is not the language constructs which prove difficult for novices, but actually composing the program. Experts possess knowledge of language syntax, semantics, *and* strategies for coordinating and composing the components of a program.

Soloway believes a program has two audiences [64], as shown in Figure 4.

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

Figure 4. Program Audience

Therefore, Soloway believes that “learning to program amounts to learning how to construct mechanisms and how to construct explanations” [64].

Soloway's proposed curriculum has two underlying assumptions [64]:

1. *Tacit Knowledge*. Although experts are not necessarily conscious of the strategies they employ to solve a given problem, scientists must “make explicit that which was implicit.” We (as scientists) must tease out the tacit knowledge.
2. *Whorfian Hypothesis*. Benjamin Whorf suggested that “language determines thought”; that is, a person can only think of something if they have a word for it. We can use a weaker claim that “language aids thought” to say that students cannot learn what is necessary unless it is explicitly taught to them.

Goals and plans, therefore, are key components in solving problems [64, 69]. Goals are the requirements for a problem and plans are those “canned” solutions for solving the problems. In teaching problem solving and programming, a key objective must be to teach methods of abstraction such that every problem does not appear to be new. Novices should be taught that every new problem can be viewed in terms of old problems [64].

Students must also be taught that programming has something in common with other problem solving tasks [64]. Programming should be thought of as a “design discipline” with the output being an artifact or mechanism that, “when set in motion, produces some desired effect” [64]. Programmers must provide some trail as to how and why an artifact was designed a particular way. This trail of information, or explanation, can then be used by the next programmer who is required to modify the artifact. The product of the programming process can be viewed as mechanisms and explanations. In an introductory course, the students must be taught to construct these mechanisms and explanations [64].

II.C Literate Programming

Literate programming is Knuth’s solution for better documentation and readability of programs [28]. Literate programming using the WEB system is concerned with writing programs as “works of literature” [28]. Knuth [28] developed the WEB style of programming for writing systems programs. However, there is some evidence that literate programming, if used frequently, might be able to reduce the problems faced by beginning programmers. Smith [62, 63] states once a programmer becomes familiar with the WEB style of programming, the process of understanding and developing programs is simplified. Literate programming facilitates problem solving by “streams of consciousness” [28] to minimize the intimidation for beginning programmers.

Brown and Childs [8] believe the WEB style of programming has several advantages over the traditional style of programming. The WEB system:

- encourages the organization of code based on psychological, rather than syntactic constraints;
- makes the structure of the program more visible to the reader; and

- encourages an explanatory style of writing, leading to more careful consideration as to the details of the program.

Some of the basic references for literate programming are [8, 17, 28, 45, 52, 53, 62, 63].

Research relevant to literate programming has proceeded along the lines of tool development and development of WEB-like systems. The majority of the work in the tool area has dealt with constructing an environment in which WEB programs may be more easily developed [2, 8, 45].

II.C.1 Organization of a WEB

The WEB system incorporates two languages, a formatting language and a programming language. The two languages are combined to document a program, as well as express an algorithm in a manner suitable for a computer [45]. Thus, in order to write a WEB program, it is necessary to know a high-level language, a formatting language, and the WEB rules [8]. Knuth [27, 28] selected T_EX as the formatting language and Pascal as the high-level programming language for his WEB system.

A WEB source file is made up of program statements written in the programming language and documentation written in the formatting language. A WEB program is made up of groups of statements, called *sections*. Each section has three parts [27, 28, 40, 45, 52, 59]:

1. *Explanatory Material*. This part provides a description of the section. It should include the purpose of the section, along with (possibly) the design rationale. It is written in the document formatting language.
2. *Definitions*. This part contains any macro or format definitions.
3. *Program Code*. This part contains small pieces of the program. It is written in the high-level programming language and is processed by a compiler. Ideally, the code part should be no longer than about twelve lines so that it is easily comprehended [27].

The three parts must be in the above order for each section, and any part may be empty. An example of a WEB source file may be found on pages 121-123 in Appendix A.

A WEB section begins with '@*' or '@␣', where ␣ denotes a space. The '@*' denotes the beginning of a major section or a chapter. A section ends at the beginning of a new section or at the end of the file.

Sections are numbered automatically, with the first being section 1. The programmer does not have to be concerned about the section numbers. Programmers assign names to each section and can then refer to a section by name, rather than number. The section name should be a short description indicative of the contents of the section. Every WEB program has one unnamed section which designates the main program.

II.C.2 Processing of a WEB

The procedure for processing a WEB is shown in Figure 5 [28]. The WEB source file is used to produce a typeset document suitable for the human reader and a high-level program suitable for compiling and executing by a computer [39, 45].

TANGLE takes as input the WEB source and produces as output the high-level source code which can then be input to a compiler. TANGLE completely ignores the documentation in each section. The source code produced by TANGLE is not meant to be read by humans [28]. Therefore, TANGLE does not go to great pains to format the resulting source code. An example of TANGLED output may be found on page 124 in Appendix A.

WEAVE takes as input the WEB source and produces as output a T_EX file. The documentation part of the code is copied directly to an output file; the definition and code parts are pretty printed [45]. WEAVE automatically generates a table of contents, an alphabetized cross-reference index, and an alphabetized list of section

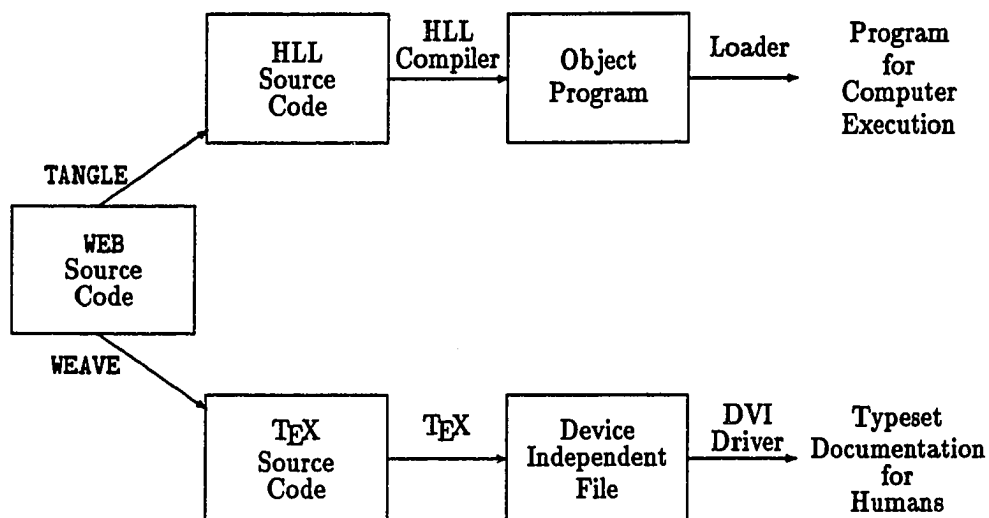


Figure 5. The WEB Process

names [27, 28, 40, 45, 52, 59]. The resulting TeX file may then be processed to produce a device independent (DVI) file which may be viewed on the screen or used as input to a printer driver for hardcopy. An example of WEAVED and TeXd output may be found on pages 125-130 in Appendix A.

II.C.3 Documentation

The literate programming paradigm encourages more than just documentation, it is designed for “explaining to human beings what we want a computer to do” [28]. It has

been said that real world programs are not developed to be read, but rather to be executed [1]. However, real world programs are modified. The documentation which is produced using literate programming should benefit the programmer developing the software, as well as the programmer maintaining the software. Some believe the critical characteristic of a WEB program is that the exposition is independent of the code [32]. This allows the explanations to be directed toward the human reader [32]. It is also a general belief that the WEB style of programming makes debugging easier, primarily due to the expository residing with the code [70, 71, 77].

II.C.4 Software Maintenance

As some have discovered, one of the benefits of using literate programming is the ability to associate a given design step with its code [76]. Williams states a literate program is more than just a typeset document. It allows the programmer to produce higher quality programs by “unfolding program code in English” [78]. This “in your face” (sic) presentation of documentation [43] can simplify the maintenance process when design rationale and implementation decisions are presented with the code.

The reason for improved maintenance with the use of literate programming is due to the fact the documentation is presented *with* the code. It has also been reported that the code produced using WEB is more readable [80]. Those familiar with program maintenance state that experience has shown “even mediocre literate code is easier to modify than good non-literate code” [80]. The benefits realized in the maintenance of literate programs is more than likely attributable to the quality of the design documentation, which is an integral part of the WEB style of programming.

II.C.5 Expressed Concerns

There are those who have other concerns regarding literate programming. Some are concerned with the documentation created during the development of a program.

Mehringer [43] says “I’m a software developer – not a typesetter.” While the use of \TeX provides the ability to typeset mathematics, a minimum amount of \TeX knowledge is actually required to produce a readable, professional-looking document. The primary hindrance seems to be that programmers are not necessarily authors. Ramsey [51] says his findings indicate it is “difficult to teach people to write.”

Both of these quotes offer proof that programmers tend to view themselves as people who work with computers, rather than people who produce information that is used by other people. Programmers *should* view themselves as authors of documentation. They must realize that a documented program is not for personal benefit; rather, it should be a well-written document (free of grammar and spelling errors) that is produced for the benefit of others.

II.C.6 A Literate Programming Environment

Both Knuth [28] and Thimbleby [73] envisioned an interactive programming environment for \WEB ; however, neither made steps toward the implementation of such an environment. The Literate Programming Tool (LPT) created by Brown [7, 8] was a step in that direction; however, it is a display-only viewer which shows the control flow of a \WEB and provides no interactive editing capabilities.

A prototype environment, `web-mode`, developed by Motl [45], provides a facility which can support the *goal/plan analysis* type of development suggested by Soloway [64, 66, 69].

Motl's environment provides interactive editing capabilities utilizing the aids provided by WEAVE. The user can view the table of contents, the cross-reference index, and the list of section names. The user may select a section name or an index entry and traverse the list(s) associated with the entry [45].

A Literate Programming Environment could easily support development of programs in the format of *goals* being expanded into *plans*. The web-mode environment is sensitive to the creation and use of section names. When a section name is created, it is added to the list of existing section names. A "stub" is placed in the WEB file to remind the programmer that the section has been referenced and, therefore, must be expanded, or written. In essence, it creates the section in order for the programmer to not receive a compiler error if the section has not yet been written. This allows for incremental development and testing. It also serves as a reminder to the programmer that the section must still be developed.

II.D Software Engineering Concepts

Fairley [22] states that "the primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of software engineers." There seems to be little emphasis on these goals in the introductory computer science classes. It appears that the majority of the emphasis is on programming.

Most computer science students eventually take a course which teaches software design principles and problem solving skills. Typically, this course is taken during the junior or senior year. The course description of the senior software engineering course at Texas A&M University (CPSC 431) is [72]:

Application of engineering approach to computer software design and development; life cycle models, software requirements and specification; conceptual model design; detailed design; validation and verification; design quality assurance; software design/development environments and project management.

One of the major pitfalls, then, is that our students are not exposed to problem solving disciplines until they reach the software engineering class. As a result, students are writing programs before they learn to solve problems.

The steps for defining a problem are shown in Figure 6 (taken from [22]).

Problem Definition
◇ Develop a definitive statement of the problem to be solved. Include a description of the present situation, problem constraints, and a statement of the goals to be achieved.
◇ Justify a computerized solution strategy for the problem.
◇ Identify the functions to be provided by, and the constraints on, the hardware subsystem, the software subsystem, and the people subsystem.
◇ Determine system-level goals and requirements for the development process and the work products.
◇ Establish high-level acceptance criteria for the system.

Figure 6. Defining a Problem

It is Pierce's belief that we should focus on preparing students to enter industry with "the theme of process improvement for quality and productivity" [48]. To this end, he "subscribed to the conventional wisdom" that the students produce a system from scratch in the software engineering course in order to experience all of the phases in the life-cycle. However, he was finding that few students succeeded in anything but a high-level design, and those systems that were implemented were poorly designed, documented, and

tested [48].

A major reason for the lack of success seems to be that the students, even at this level, focus on completing the project, rather than defining the problem. In an attempt to make the course (and the students) more successful, Pierce implemented a maintenance-based project [48]. In addition to having the potential for completion, it probably better prepares the students for the “real-world” of maintenance. Those projects that are not completed are carried over into the next semester of students.

There has been some attempt to introduce software engineering techniques at the introductory levels in the curriculum [21, 56, 75]. The Rochester Institute of Technology introduces students to the foundations of software engineering during the second year [21]. Others have used laboratories in the introductory course in an attempt to focus on problem decomposition, data abstraction, documentation, design specification, testing, and code review [56]. In each case, attempts are being made to expose students to software engineering concepts earlier in their academic career.

II.E Using Literate Programming to Teach Good Programming Practices

Shum and Cook [61] incorporated the literate programming paradigm into the teaching of a junior-level course. Their goal was to encourage students to write informative and usable documentation in order to facilitate program maintenance. They used literate programming to emphasize good documentation practices; that is, writing programs to be read by humans.

Introductory computer science instructors, too, are interested in good programming (and documentation) practices. However, an emphasis on problem solving should be the

first step. Good programming practices will naturally follow.

II.F Using Literate Programming to Improve Problem Solving Skills

Knuth envisioned literate programming as a tool for systems programmers to develop well-documented, maintainable programs [28]. Literate programming can also be used as a tool for novice programmers to begin to develop good problem solving skills.

As shown in Figure 4, Soloway believes a program has two audiences: the computer and the human reader. Knuth would (probably) concur. Literate programs are designed, not only to solve a problem using the computer, but for the human to understand the thought process behind the design solution.

Figure 1 and Figure 2 contain discussions of techniques that have been used to improve the programming skills of novice programmers. The literate programming methodology can be used to capture and represent the information contained in case studies and templates.

The use of literate programming encourages the programmer to solve a problem using the goal/plan analysis techniques presented by Soloway. The use of literate programming provides novice programmers the opportunity to develop the expert knowledge, described in Figure 3, that is typically not available to them.

One of the primary functions of literate programming, however, is to capture a programmer's thoughts about the problem solution. A major task in software engineering is to define the problem. Figure.6 contains a list of the steps involved in problem definition. The literate programming development methodology can be used to explicitly address each of the issues involved in planning a software project.

CHAPTER III

DESIGN OF A TEST STUDY

A program development methodology was proposed for use in introductory computer science course(s) to determine the effects of literate programming on program design and development by novice programmers. The goal of the use of this methodology is to develop and enhance problem solving and (therefore) program design skills for beginning programmers. Results were expected to show that use of the methodology will develop problem solving skills along with the usual programming skills. The methodology included developing measures which were used to assess the success of the methodology.

Knuth's style of literate programming was used as a framework to assist students with program design and development. Literate programming makes use of *sections* which include code and documentation. Sections should be small, simple thoughts, similar to paragraphs in literary works. These sections are linked through a system of structured pseudo-code.

Literate programming gives the programmer the ability to piece together the design (and corresponding code) in the manner in which it is conceptualized. The student is not required to adhere strictly to the common top-down, procedure-oriented approach.

III.A Selection of Participants

Literate programming should be usable at any level in the curriculum to enhance the problem solving skills of students involved in software development. It can be considered a

structured means of doing what programmers should be doing anyway; that is, pseudo-code, top-down and bottom-up design, and documentation.

The potential problems in teaching literate programming techniques to beginning programmers (freshmen) and to programmers with experience (seniors or graduate students) will be discussed in the following sections.

III.A.1 Literate Programming for Introductory Students

Freshmen computer science students arrive in the introductory course with little or no computer experience. The term *computer experience* includes programming languages, text editors, and software design and development techniques. The advantage of teaching introductory students is they have not had ample opportunity to develop any programming habits, poor or otherwise.

Pascal is a popular language in the introductory computer science class [75]. Teaching literate programming to freshmen requires that the syntax of a particular programming language be taught. In addition, the use of various language constructs must also be introduced. These factors may contribute to the difficulty in teaching the use of literate programming at the introductory level.

The proposed literate programming paradigm includes the utilization of *web-mode* for editing WEB documents. Introductory students must learn the GNU Emacs editor and the *web-mode* editing environment. The topics may at first prove to be overwhelming to a beginning computer science student, although each topic is intended for only cursory coverage reflected by a one-page handout.

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 [35]. Most textbooks give examples of programs, rather than demonstrate the method by which the given solution was derived [35].

Introductory computer science students have acquired few problem solving skills, either through experience or from instruction. The programming problems encountered in an introductory course are relatively simple and students see little benefit in program design at the introductory level.

III.A.2 Literate Programming for Advanced Students

By the time a student reaches senior status, the student may be familiar with several programming languages, including Modula-2, Ada, Pascal, C, and FORTRAN. Each of these languages is supported by a WEB system. Thus, teaching literate programming to seniors is not constrained to using one specific language and no time is spent teaching language syntax. Advanced students are already familiar with language syntax and that will not have to be taught along with the rest of the literate programming concepts.

One of the reasons for developing the literate programming style of programming is to concentrate on “explaining to human beings what we want a computer to do” [28]. Advanced students have completed a minimum of four classes in which they write programs. They have had ample opportunity to develop a coding style. However, seniors are in the habit of instructing the computer what to do. They must be taught to approach programming in a different manner.

Typical programming classes teach the student to solve problems using a top-down approach [35]. WEB programming makes use of *sections* of code, along with using the more common procedure-oriented approach. WEB sections are not subprograms, they are

expanded inline. The use of WEB sections is a technique by which programmers design a program in a “stream of consciousness” order [28]. Buyukisik [9] states literate programming can be used as a program design language and these modules are not procedures or functions in the traditional sense. Changing the way in which a person writes programs, then, may be more difficult than teaching them to program with no prior experience.

An advanced computer science student is confronted with difficult and extensive problems. Typically, in order to solve the problem in a reasonable manner, the student must utilize some type of problem solving technique. Thus, the problem solving experience obtained by advanced computer science students has been acquired by practice rather than learned through instruction.

The difficulty in changing the manner in which a person approaches a problem may also be applied to the use of an editor. Senior computer science students have experience using at least one editor. However, this does not necessarily mean the senior-level student will learn emacs and web-mode faster or more easily than the freshman-level student. Motl [45] found that although students had no difficulty in learning emacs and web-mode, many of the features available were not utilized. This is apparently due to the tendency of a person to use only what is required to solve the given task.

It may be possible that a novice user will tend to use more of the available features because the novice is not constrained by any preconceived ideas of what features an editor is supposed to provide. However, many of the files that are maintained by web-mode would have to be examined in order to determine the possibility.

People tend to approach problem solving in a variety of ways. The fact that a senior

computer science student has some experience in this area may make it easier to teach literate programming at an advanced level. The use of literate programming allows the programmer to focus on higher levels of abstraction without being concerned about the details [71]. This ability to delay the focus on lower level, or program code, details is developed over time and after much practice. It may be difficult to dispel the belief held by introductory students that programming is different from problem solving.

III.A.3 Discussion

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 [76]. 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. In all likelihood, the senior level student would have realized more immediate benefit with the use of literate programming than the novice programmer.

It may appear that literate programs take longer to write than non-literate programs [78]. One of the reasons for this may be that the documentation is being developed *with* the program. Knuth stated that the total time for writing and debugging a WEB program is no greater than that for a non-WEB program, even though the WEB program is better and contains more documentation [28]. One other experience shows the debugging time for literate programs is much less than for non-literate programs [71]. This may be attributed to the fact that literate programs are better designed.

The development time required for literate programming did become a substantial factor in teaching freshmen versus seniors. A freshman typically has not yet been forced to develop any time management skills. The tendency of introductory students is to

prolong the start of program development. The use of literate programming sometimes caused the students to become frustrated at the time involved or caused them to overlook the goal of literate programming and revert to merely instructing the computer what to do. The solution to a problem is typically arrived at over a period of time. Advanced computer science students have learned this by experience over the years and may have realized the benefit of using literate programming techniques to document the solution to the problem, as well as develop the code necessary to solve the problem.

One of the most important uses of literate programming is design by iteration. A solution for overcoming the perceived difficulties in the use of literate programming seems to be an emphasis on iteration and peer review. This technique should also be included in the problem solving category because it is the method by which problems are solved and solutions are derived. Design by iteration takes time, however. The solution to a problem may be revised a number of times. The novice programmer typically does not practice design by iteration for two reasons: lack of time and lack of experience. There is a difference in emphasizing the use of literate programming for iterative design to senior level students because they have experience in problem solving and have developed time-management skills.

Knuth did not intend for literate programming to be used by novice programmers. He designed the WEB system of programming for those "computer scientists" who were comfortable with the use of several different languages [28]. However, literate programming has been used successfully by computer science students [45, 61, 63], rather than systems programmers. Novice programmers (that is, introductory students) can utilize the literate programming paradigm in order to develop good problem solving skills.

III.B Teaching Introductory Students

The ACM/IEEE-CS Joint Curriculum Task Force presented a new framework for the discipline of computing and a new basis for computing curricula [18, 74]. The task force addresses the role of programming in the discipline. Although it is clear that programming must be a part of the curriculum and every computer science student should demonstrate competence in it, “this does not, however, imply that the curriculum should be based on programming or that the introductory courses should be programming courses” [18]. The task force notes that programming languages are merely “tools” for the discipline.

The course description of the CS/1 course at Texas A&M University (CPSC 110) is [72]:

“Basic concepts, nomenclature and historical perspective of computers and computing; internal representation of data; software design principles and practices; structured programming in a high-level language; use of terminals, operation of editors and execution of student-written programs.”

This course description, as well as the implementation of the course, is quite likely representative of the introductory computer science courses being taught at many universities today. The description does not specify that students be taught problem solving. However, problem solving techniques may be included in the principles of software design and are, therefore, implied. Although the students are exposed to the topic of software design principles, in reality, the majority of the time is being spent on programming language syntax and execution of student-written programs.

III.C Literate Programming and the Design of Solutions

The theme of this test is that literate programming can be used in the introductory computer science course to emphasize problem solving. This use will result in better designed programs, and documentation will not be *ad hoc* as seems to be the norm.

The students are typically instructed to produce the following information in the design of their solution:

- problem statement;
- inputs required;
- outputs produced;
- processing required;
- algorithm; and
- testing.

Although this information is requested, it is rarely produced as an integral part of the code. It is probably produced in handwritten form or with the use of a word processor, but rarely with the same set of tools as the actual code. Therefore, it is rather simple to produce a code in which the design may easily be forgotten or ignored and the reasons for these changes are lost.

A natural use of literate programming is to create a model for the design of problem solutions. The use of sections allows the programmer/author to outline the solution to a problem. The programmer will define a set of goals that are to be expanded into plans through the structured use of pseudo-code.

The initial design solution for the problem can be produced as a literate program without any code. The design process may then be iterated several times before arriving at a solution. Once the final design is produced, the programming task can be performed.

Programming the problem merely involves writing the code to implement each of the design steps. The literate programming paradigm provides a mechanism by which the programmer explains to the human reader his/her solution to the problem before explaining the solution to the computer.

For example, assume the problem to be solved is the quadratic equation. Although the problem may be a familiar one, the design process should be used to arrive at an acceptable solution. An example of the iterations in solving this standard mathematical problem using this methodology are available via anonymous ftp from `ftp/pub/tex-web/web/DOCs`. The files `wm*` are available in WEB and PostScript form. The mathematics involved are easily represented in handwritten documentation; however, they cannot be as easily represented using typical programming languages and accompanying documentation. The literate programming paradigm enhances the representation of complex (as well as not so complex) mathematical solutions. And the implementation of the mathematical solutions require minimal knowledge of `TEX`.

The inclusion of graphics should also be a standard part of this approach. However, it was not included in the CS/1 course because the students have not been exposed to an appropriate drawing package.

The six design points (problem statement, input, output, processing, algorithm, and testing), in conjunction with the literate programming paradigm, provide an outline with which problems may be solved. The programmer may use literate programming to design a problem solution much in the way an author designs a piece of literature. The first step is to develop an outline and piece it together in the manner in which it is conceptualized. The outline is expanded and refined through many iterations. Literate programming

provides the framework necessary for solving problems. The solution is outlined using sections, which are similar to paragraphs. These paragraphs are iterated until a final solution to the problem is reached. Suppose we have the following problem:

A rectangular house is situated on a rectangular yard. Given that the lawn may be mowed at a rate of 2 feet per second and there exists a standard charge per square foot, determine the cost of mowing the yard and the length of time the job will take.

An example using literate programming to develop a solution to this problem follows. Figures 7 through 14 contain the initial design, which is merely an outline that contains no code. The WEB program contains sections which address each of the previously stated design topics. Figures 15 through 24 contain the final version of the program. This final version of the WEB program contains the documentation and the corresponding code for solving the problem.

Lawn Service

December 2, 1994

	Section	Page
Problem Statement	1	1
Problem Inputs	2	2
Processing Requirements	3	3
Problem Outputs	6	4
Algorithm	7	5
Testing	8	6
INDEX	9	7

Abstract. *Aggie Lawn Service is a business which provides lawn care for the citizens of Bryan-College Station. An estimate for a potential customer is provided which includes a cost statement and an estimated time to complete the job.*

Deborah Dunn
December 2, 1994
9:51

Figure 7. Design Example – Table of Contents

§1 Lawn Service**PROBLEM STATEMENT 1**

1. Problem Statement. Aggie Lawn Service is a business which provides lawn care for the citizens of Bryan-College Station. An estimate for a potential customer is provided which includes a cost statement and an estimated time to complete the job. The 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. It is assumed that a rectangular house is situated in a rectangular yard. This estimate ignores any obstacles in the lawn, bare spots or driveways, and does not account for breaks, slower/faster mowers, or the mower running out of gas.

Figure 8. Design Example – Problem Statement

2 PROBLEM INPUTS**Lawn Service §2**

2. Problem Inputs. In order to provide an estimate for the customer, several items must be received. It is assumed that a rectangular house is situated on a rectangular yard. The following must be provided to solve the problem:

- length of yard
- width of yard
- length of house
- width of house

Figure 9. Design Example – Inputs Required

4 PROBLEM OUTPUTS**Lawn Service §6**

6. Problem Outputs. The customer will be provided with the final estimate which includes the following items:

- the cost of mowing the lawn
- the estimated time (in minutes) to mow the lawn

Figure 10. Design Example - Outputs Required

§3 **Lawn Service****PROCESSING REQUIREMENTS 3**

3. Processing Requirements. Two items must be calculated for the estimate. The cost of mowing the lawn and the estimated time to complete the job must be calculated.

4. Givens (or Knowns). There is a standard (confidential) charge per square foot of lawn. It is also assumed that grass can be cut at the rate of 2 square feet per second.

5. Formulas Needed. The following is a list of the formulas that will be needed in order to provide the estimate.

$$\text{Area of Yard} = \text{Lenth of Yard} \times \text{Width of Yard}$$

$$\text{Area of House} = \text{Length of House} \times \text{Width of House}$$

$$\text{Area of Lawn} = \text{Area of Yard} - \text{Area of House}$$

$$\text{Cost Estimate} = \text{Area of Lawn} \times \text{Charge per Square Foot}$$

$$\text{Time Estimate} = \frac{\text{Area of Lawn}}{2}$$

$$\text{Minutes Time Estimate} = \frac{\text{Time Estimate}}{60}$$

Figure 11. Design Example – Processing Required

§7 Lawn Service**ALGORITHM 5**

7. Algorithm. The following steps must be taken to solve the problem:

1. Get length and width of the yard.
2. Get length and width of the house.
3. Calculate the area of the house and the yard.
4. Calculate the area of the lawn.
5. Calculate the cost of mowing the lawn.
6. Calculate the time needed to mow the lawn.
7. Present the estimate to the customer.

Figure 12. Design Example – Algorithm Development

6 TESTING**Lawn Service §8**

- 8. Testing.** The program will be tested with the following scenarios:
1. The house is a 25 foot square house situated on a 50 foot square yard. The charge per square foot is \$0.01. The area to be mowed is 1875 square feet. Therefore, the cost of mowing the lawn is \$18.75. The estimated time for completion is 15.63 minutes.
 2. The house is 1' by 20' situated on a 5' by 25' yard. The charge per square foot is \$0.005. The area to be mowed is 105 square feet. Therefore, the cost of mowing the lawn is \$0.53. The estimated time for completion is 0.875 minutes.

Figure 13. Design Example – Testing

§9 Lawn Service

INDEX 7

9. INDEX.

Aggies: 1.

confidential charge: 1, 4.

strange yard: 8.

Figure 14. Design Example -- Index

Lawn Service

December 2, 1994

	Section	Page
Problem Statement	1	1
Problem Inputs	2	2
Processing Requirements	4	3
Problem Outputs	9	4
Algorithm	10	5
The Actual Program	11	6
Testing	12	7
INDEX	13	8

Abstract. *Aggie Lawn Service is a business which provides lawn care for the citizens of Bryan-College Station. An estimate for a potential customer is provided which includes a cost statement and an estimated time to complete the job.*

Deborah Dunn
December 2, 1994
9:51

Figure 15. Program Example – Table of Contents

§1 Lawn Service**PROBLEM STATEMENT 1**

1. Problem Statement. Aggie Lawn Service is a business which provides lawn care for the citizens of Bryan-College Station. An estimate for a potential customer is provided which includes a cost statement and an estimated time to complete the job. The 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. It is assumed that a rectangular house is situated in a rectangular yard. This estimate ignores any obstacles in the lawn, bare spots or driveways, and does not account for breaks, slower/faster mowers, or the mower running out of gas.

Figure 16. Program Example - Problem Statement

2 PROBLEM INPUTS

Lawn Service §2

2. **Problem Inputs.** In order to provide an estimate for the customer, several items must be received. It is assumed that a rectangular house is situated on a rectangular yard. The following must be provided to solve the problem:

- length of yard
- width of yard
- length of house
- width of house

(Get Input Information 2) ≡

```
procedure Get_Input;
begin
  write('Please enter the length of the yard: '); readln(yard_length);
  write('Please enter the width of the yard: '); readln(yard_width);
  write('Please enter the length of the house: '); readln(house_len);
  write('Please enter the width of the house: '); readln(house_width)
end;
```

This code is used in section 11.

3. At this point I realize that I need to declare some variables in order to accomplish the above input operations.

(Variable Declarations 3) ≡
 "yard_length, yard_width: real;
 "house_len, house_width: real; ""

See also sections 7 and 8.

This code is used in section 11.

Figure 17. Program Example – Inputs Required

4 PROBLEM OUTPUTS

Lawn Service §9

9. **Problem Outputs.** The customer will be provided with the final estimate which includes the following items:

- the cost of mowing the lawn
- the estimated time (in minutes) to mow the lawn

(Provide Statement 9) ≡

```

procedure Print_Statement;
begin "writeln('Aggie_Lawn_Service' : 25); "writeln;
      "writeln('The_area_of_the_lawn_is:UU', area_of_lawn : 5); "writeln;
      "writeln('The_estimated_cost_is:UU$', billing_amount : 5 : 2);
      "write('The_estimated_time_for_completion_is:UU');
      "writeln(cutting_time_estimate : 4, 'minutes');"
end; "
```

This code is used in section 11.

Figure 18. Program Example – Outputs Required

§4 Lawn Service

PROCESSING REQUIREMENTS 3

4. **Processing Requirements.** Two items must be calculated for the estimate. The cost of mowing the lawn and the estimated time to complete the job must be calculated.

5. **Givens (or Knowns).** There is a standard (confidential) charge per square foot of lawn. It is also assumed that grass can be cut at the rate of 2 square feet per second.

{Constant Declarations 6} ≡

"confidential_charge = 1.25; "rate_of_cutting = 2; ""

This code is used in section 11.

6. **Formulas Needed.** The following is a list of the formulas that will be needed in order to provide the estimate.

$$\text{Area of Yard} = \text{Lenth of Yard} \times \text{Width of Yard}$$

$$\text{Area of House} = \text{Length of House} \times \text{Width of House}$$

$$\text{Area of Lawn} = \text{Area of Yard} - \text{Area of House}$$

$$\text{Cost Estimate} = \text{Area of Lawn} \times \text{Charge per Square Foot}$$

$$\text{Time Estimate} = \frac{\text{Area of Lawn}}{2}$$

$$\text{Minutes Time Estimate} = \frac{\text{Time Estimate}}{60}$$

{Calculations 6} ≡

```
procedure Calculate_Area; "
  begin "area_of_yard ← yard.length * yard.width;
    "area_of_house ← house.len * house.width;
    "area_of_lawn ← area_of_yard - area_of_house"
  end; ""
procedure Calculate_Cost_and_Time; "
  begin "billing_amount ← area_of_lawn * confidential_charge;
    "cutting_time_estimate ← area_of_lawn / (rate_of_cutting * 60.0)"
  end; ""
```

This code is used in section 11.

7. I need some more variables declared in order to complete the above calculations.

{Variable Declarations 3} +≡

"area_of_lawn, area_of_yard, area_of_house: real; ""

8. Oops, I almost forgot the last of the variable declarations. I need to declare the two areas for my output.

{Variable Declarations 3} +≡

"billing_amount, cutting_time_estimate: real; ""

Figure 19. Program Example – Processing Required

§10 Lawn Service**ALGORITHM 5**

10. Algorithm. The following steps must be taken to solve the problem:

1. Get length and width of the yard.
2. Get length and width of the house.
3. Calculate the area of the house and the yard.
4. Calculate the area of the lawn.
5. Calculate the cost of mowing the lawn.
6. Calculate the time needed to mow the lawn.
7. Present the estimate to the customer.

Figure 20. Program Example – Algorithm Development

6 THE ACTUAL PROGRAM

Lawn Service §11

11. The Actual Program. This is where the actual program begins. This could appear anywhere in the WEB program. The only rule is that the program actually begin with the 'at-p'.

```

--
program Lawn_Service; "
  const "(Constant Declarations 5)
  "
  var "(Variable Declarations 3)
    "(Get Input Information 2)
    "(Calculations 6)
    "(Provide Statement 9)
  "
  begin "Get_Input;
    "Calculate_Area;
    "Calculate_Cost_and_Time;
    "Print_Statement
  "
  end."

```

Figure 21. Program Example – The Actual Program

§12 Lawn Service

TESTING 7

12. Testing. The program will be tested with the following scenarios:

1. The house is a 25 foot square house situated on a 50 foot square yard. The charge per square foot is \$0.01. The area to be mowed is 1875 square feet. Therefore, the cost of mowing the lawn is \$18.75. The estimated time for completion is 15.63 minutes.
2. The house is 1' by 20' situated on a 5' by 25' yard. The charge per square foot is \$0.005. The area to be mowed is 105 square feet. Therefore, the cost of mowing the lawn is \$0.53. The estimated time for completion is 0.875 minutes.

Figure 22. Program Example – Testing

8 INDEX

Lawn Service §13

13. INDEX.

Aggies: 1.
area.of.house: 6, 7.
area.of.lawn: 6, 7, 9.
area.of.yard: 6, 7.
billing.amount: 6, 8, 9.
Calculate.Area: 6, 11.
Calculate.Cost.and.Time: 6, 11.
confidential charge: 1, 5.
confidentialCharge: 5, 6.
cutting.time.estimate: 6, 8, 9.
Get.Input: 2, 11.
house.len: 2, 3, 6.
house.width: 2, 3, 6.
Lawn.Service: 11.
Print.Statement: 9, 11.
rate.of.cutting: 5, 6.
readln: 2.
real: 3, 7, 8.
strange yard: 12.
write: 2, 9.
writeln: 9.
yard.length: 2, 3, 6.
yard.width: 2, 3, 6.

Figure 23. Program Example – Index

§13 Lawn Service**NAMES OF THE SECTIONS 9**

- (Calculations 6) Used in section 11.
- (Constant Declarations 5) Used in section 11.
- (Get Input Information 2) Used in section 11.
- (Provide Statement 9) Used in section 11.
- (Variable Declarations 3, 7, 8) Used in section 11.

Figure 24. Program Example - List of Sections

III.D Design

The test study was implemented during the Fall 1993 semester at Texas A&M University. The test was performed while teaching all topics normally taught in the CS/1 course. The students were scheduled for 3 hours of lecture and 2 one-hour supervised labs per week, for a duration of 15 weeks. The equipment used by each student was an IBM 486 33 MHz PC compatible with 4MB on a Novell network. The equipment, computer labs, and classroom were also used by the regular CS/1 course.

The students used an editing environment called `web-mode` [45]. The environment is based on GNU Emacs [10]. The following topics were taught during the course:

- problem solving techniques, including top-down design, divide and conquer, and hierarchical development;
- the syntax of the Pascal programming language;
- use of the `web-mode` editing environment and the GNU Emacs editor;
- an introduction to the `TEX` formatting language and the `WEB` rules and constructs.

A pre-test was administered at the beginning of the course which was designed to evaluate the students' computing background and problem solving skills as they entered the course. They were periodically tested throughout the semester on problem solving, Pascal, `web-mode`, emacs, and `WEB` rules and constructs.

The majority of the classroom lecture was spent on problem solving and Pascal syntax, similar to the manner in which the course is regularly taught. Each of the remaining topics, which were specific to the test study, were covered with the use of *reference cards*. Most of the information on `web-mode`, emacs, `TEX`, and `WEB` was conveyed during the lab time with the use of handouts and examples. The time spent on any one part of the edit, compile, link, and debug process was reduced because of the extra `WEB` steps.

III.E Participants

The group of novice programmers selected for the study were those enrolled in the honors section of the introductory computer science course (CPSC 110H) at Texas A&M University. This group was selected for three reasons: class size, computing background of the students, and the author was an experienced teacher of this course.

The honors class was a relatively small test group. The regular introductory course has an average enrollment of 175 students, while the honors section typically has about 40 students enrolled. In general, the students enrolled in the introductory course (both honors and regular) have a limited knowledge of personal computers and/or programming. Typically, many of the beginning computer science students have had at least one semester of a high school computer class. They arrive with some limited knowledge of programming language syntax (primarily Pascal and/or BASIC). These students are still considered novice programmers because they rarely have experience in data structures, solving large problems, or programming teams.

There are several qualifications for participation in the honors program at Texas A&M University [72]. New freshmen must graduate in the top 10% of their class and score 1150+ on the SAT or 28+ on the ACT. All other students must attain a cumulative GPR of 3.50 or above at Texas A&M to enroll in honors courses. Students must maintain a 3.25 GPR to continue in the honors program.

III.F Methods of Measurement

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*.

Three methods for measuring the effect of literate programming on problem solving were considered. The methods are described as follows:

1. The experimental group consists of students enrolled in CPSC 110H and the control group consists of students enrolled in CPSC 110.
2. The subject class (CPSC 110H) can be divided evenly into an experimental and control group.
3. An "expert" can be used for the evaluation of the students performance based on past and present performance of honors and non-honors students.

III.F.1 Method 1: CPSC 110H versus CPSC 110

This method employs the CPSC 110H class as the experimental group and the regular CPSC 110 class as the control group. An evaluation of the experiment would be based on the results of similar assignments given and similar tests administered throughout the semester.

The advantages of this method are as follows:

- both groups use the same textbook;
- the study utilizes an experimental and a control group; and
- the classes are taught during the same semester.

The disadvantages of this method are as follows:

- there is a difference in the quality of the students (honors versus non-honors);
- different instructors teach the two courses; and
- there is a difference in class size (175+ versus 35+).

It was decided that this would not be a feasible method of measurement, primarily due to the difference in instructor, class size, and the quality of the students.

III.F.2 Method 2: Dividing the CPSC 110H Section

This method divides the CPSC 110H class into an experimental and a control group based on the particular lab section. An evaluation of the experiment would be based on the results of similar assignments given and similar tests administered throughout the semester.

The advantages of this method are as follows:

- the test study utilizes an experimental and a control group;
- the classes are taught during the same semester;
- one instructor teaches the course;
- both groups are honors students;
- class size is not a factor; and
- both groups use the same textbook.

The disadvantages of this method are as follows:

- the development environment is sometimes discussed in lecture; thus both environments must be discussed during the same lecture;

- the students will discuss their respective environments and will determine there is a difference, which may affect the results; and
- if the environment is not discussed in lecture, lab time must be spent lecturing.

This would have been the preferred method but was not possible due to facilities and resources necessary to support this format. Also, a mechanism to ensure the two groups were created evenly is too costly and nearly impossible.

III.F.3 Method 3: CPSC 110H versus Previous CPSC 110/CPSC 110H

This method utilizes the CPSC 110H class in the experiment and relies on statistics and the author's "expert" opinion for the evaluation. An evaluation of the experiment was based on assignments given and tests administered throughout the semester by the author, past performance of honors and non-honors students, and performance of honors and non-honors students in the subsequent Programming II course.

The advantages of this method are as follows:

- the author has been the primary instructor for the CS/1 course at Texas A&M University for approximately 3 years;
- the author taught CPSC 110H at Texas A&M University during the Fall 1990 semester, therefore it may be used as a control group;
- differences in instructor, quality of student, class size, and textbook are not a factor;
- class lecture time may be spent discussing the development environment, leaving lab time for development work; and
- the author has access to the final exam results for CPSC 110H during the Fall 1992 semester, as well as results for the non-honors classes for previous semesters.

The disadvantages of this method are as follows:

- the classes may not have used the same textbook;
- the comparison groups were not taught during the same time frame as the experimental group;

- comprehensive records for the comparison groups may not be available;
- the results of the evaluation rely on the author's expert opinion; and
- the author may be biased.

III.F.4 Discussion

Method 3 was selected to be used due to the lack of resources and facilities. Rather than attempt to implement a controlled experiment, a quasi-experimental approach was utilized, from which it is possible to draw reliable inferences [20, 42]. The research resembles an experiment, but lacks the controls of experimental research, such as a control group [42].

After conducting the experiment, an extension to the evaluation procedure was made. The CS/2 course is primarily a programming language course in which students are taught C syntax. Therefore, it was decided that in order to evaluate problem solving skills, an evaluation of the Data Structures course might prove more beneficial.

CHAPTER IV

IMPLEMENTATION OF A TEST STUDY

The test was performed during the Fall 1993 semester and the participants were those students enrolled in the honors section of the CS/1 course. The course differed from other CS/1 courses only in the sense that the WEB style of literate programming was utilized in an attempt to enhance problem solving skills. All of the topics normally covered in the CS/1 course were presented to the test group.

Many of the students had preconceived ideas about the course being explicitly “a Turbo Pascal class.” Therefore, it was necessary to explain that a different environment was to be used. Several students then perceived themselves as being “guinea pigs” for this new development environment. However, at no time were the students told they were participating in an experiment.

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.

IV.A Test Study

This section is a detailed discussion about the test study. Tests and problems will be described. The method by which the students were graded is presented, as well as a description of what was expected from each of the labs. The course materials, including tests, labs, and grading guidelines for the labs are included in Appendix A.

IV.A.1 Course Materials

The required textbook for the course was *Pascal: Understanding Programming and Problem Solving, Third Edition* by Douglas Nance. The same textbook was used in the regular CS/1 course. The text had also been used in the course for the previous two semesters. The textbook provides information on Pascal syntax, problem solving using top-down design, and the Turbo Pascal system.

The test study participants were required to use the GNU Emacs editor, rather than the editor provided with Turbo Pascal. They were given a GNU Emacs Reference Card and were shown how to access the emacs tutorial. The reference card also includes

information on navigating in `web-mode` (pages 114-115).

The information on `TEX` and `WEB` was conveyed primarily by example. The participants were given an excerpt from the `WEB` user manual which contains a brief description on how to write programs in the `WEB` language (pages 116-120). They were also given several handouts to illustrate various `TEX` features (pages 131-139).

IV.A.2 Lab Assignments

The students received six programming assignments throughout the course of the semester. The quantity and degree of difficulty of each assignment was comparable to those given in the previous `CS/1` courses. Each assignment, except the first and the last, was graded based on the initial design and the final lab. The first lab assignment was designed merely to have the students use `emacs` and the `web-mode` environment. The last assignment was graded on design and execution; however, the students were not required to turn in an initial design, as the problem was merely a different implementation of a previous problem.

In previous semesters, students were taught problem solving and were strongly encouraged to prepare an initial design for their programs. In a few instances, they were asked to turn in their design, which was usually submitted in handwritten form. The students were also asked to document their programs and write well-styled programs. The documentation typically took the form of explaining the purpose of a procedure or a function.

The test study participants were required to turn in an initial design and were required to use this design as the starting basis for their program development. There was no predefined format to which the design had to conform. The only rule was that each of

the following topics be addressed:

- problem statement;
- inputs required;
- outputs generated;
- processing required;
- algorithm; and
- testing.

It should be noted that this same list was given as a guide for design in the CS/1 course in each of the previous semesters.

The test study participants were also given the requirement that they document their programs. Each student received feedback on their initial design and, if necessary, modified the design. Their documentation included the design requirements and provided more information on what a particular *step* was accomplishing, rather than addressing the purpose of a procedure or a function.

IV.A.3 Exams

The students were required to take three in-class exams and a comprehensive final exam. The quantity and degree of difficulty of each exam was comparable to those given in the previous CS/1 courses. Each exam, including the final, contained a question which was designed to test the students' problem solving ability. This was done in an attempt to measure the participants' problem solving ability over time. The questions increased in complexity over the course of the semester. The exams also tested the students' knowledge of Pascal syntax, emacs editor functions, features of the WEB language, and web-mode functions.

The final exam was designed to test the subjects in the same manner in which students had been tested in previous semesters. Exact or comparable questions were used such that an unbiased comparison could be made. Again, the students were tested on their knowledge of Pascal syntax, editing facilities, and their problem solving ability. The results of the test study, including performance on labs and exams, are discussed in detail in the next chapter.

IV.B Teaching Assistant

The performance of the students could also be affected by the teaching assistant assigned to the course. The duties of the teaching assistant included the following:

- assisting the students during the scheduled lab time and during scheduled office hours;
- grading and providing feedback on the initial design for each lab assignment;
- grading the final design and program for each lab assignment; and
- providing instruction on WEB, T_PX, and Pascal, when necessary.

The teaching assistant was not familiar with literate programming prior to the test study. However, he practiced literate programming techniques during the course in order to provide assistance to the students.

The grading policy adhered to by the teaching assistant was not dictated by the instructor, but was agreed upon by both the instructor and the teaching assistant. The grading practices were strict, due to the nature of the study and the personality of the teaching assistant. However, it is believed that the teaching assistant was fair, objective, and comparable to the other teaching assistants the author has directed in previous honors courses in his evaluation of the lab assignments.

CHAPTER V

RESULTS

The honors computer science introductory course for the Fall 1993 semester was selected as the subject class for the test study. The performance of the students enrolled in this particular class was compared with the following groups of students:

- the honors computer science introductory course (CPSC 110H) for the Fall 1990 semester; and
- the honors computer science introductory course (CPSC 110H) for the Fall 1992 semester.

This chapter contains a detailed discussion of each of the groups. A comparison was made between the performance of the test study participants and the students enrolled in the other classes.

V.A Background/Experience of Test Study Participants

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.

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

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 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, it is easily seen that the problem is a basic input-process-output problem. Each subject received 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 distributions for the CS/1 course over the last few years.

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)

V.B CS/1 Class Information

The CS/1 course is open to students of all majors in the university. A particular CS/1 class may be distinguished by the semester it is taught, the instructor, the quality of students, the textbook used, and the size of the class. Every attempt was made to address each of these issues and to minimize the side effects of each issue.

The class selected for the test study was taught by the same instructor that taught one of the comparison groups. The other comparison group was taught by a different instructor. Each instructor covered the same material regarding programming and problem solving in the comparison class.

The textbook selected for the CS/1 course normally changes every 1-2 years. An attempt is made to select the textbook which best presents problem solving techniques and Pascal syntax. The textbook used for the Fall 1990 semester was *Turbo Pascal 4.0/5.0* by Walter Savitch. The textbook for the Fall 1992 semester was *Pascal: Understanding Programming and Problem Solving, Second Alternate Edition* by Douglas Nance. The test study participants were taught using Nance's Third Edition of the Pascal text. The textbooks are all similar enough that differences attributed to the use of different texts are considered insignificant.

V.C Student Classification Distribution

The following is an analysis of the classification of students for each of the CS/1 classes. These distributions can be considered typical for the CS/1 course at Texas A&M University.

Each of the subsequent tables could be presented in the form of counts or percentages. The tables presented in this chapter are those deemed most informative. Each table is presented in Appendix B in the alternative form.

Table 5 is a summary of the student classification distribution for the CS/1 course for the subject and comparison classes (in percent form). The U1 classification indicates the student is a freshman, U2 indicates sophomore, U3 indicates junior, and U4 indicates senior. A chi-square test of independence was conducted to determine if the classification and semester variables are related (or dependent). The critical value of X^2 for $\alpha = 0.10$ and degrees of freedom = 6 is 10.64. The computed value, 10.96, exceeds 10.64, so we conclude that the two variables are dependent. That is, the proportion of students of a particular classification varies depending on the semester.

The honors classes typically have a large percentage of freshmen and sophomores. One reason for this is that freshman computer science majors usually enroll in CS/1 their first semester. The remaining students may be honors students in other departments that are taking the course to satisfy their computer requirement.

Table 6 is a summary of the student major distribution for the CS/1 course for the subject and comparison classes (in percent form). The honors classes typically have a large number of computer science (CPSC) and computer engineering (CSEN) majors. A chi-square test of independence was conducted to determine if the major and semester

Table 5. Student Distribution by Classification (Percent)

Semester	U1	U2	U3	U4
Fall 90-H	77.8	16.7	2.8	2.8
Fall 92-H	69.0	16.7	14.3	0.0
Fall 93-H	68.4	29.0	0.0	2.6

variables are related (or dependent). The critical value of X^2 for $\alpha = 0.10$ and degrees of freedom = 2 is 4.605. The computed value, 3.972, does not exceed 4.605, so we conclude that the two variables are not dependent. That is, the proportion of students of a particular major does not vary depending on the semester.

Table 6. Student Distribution by Major (Percent)

Semester	CPSC/CSEN	Other
Fall 90-H	55.6	44.4
Fall 92-H	59.5	40.5
Fall 93-H	76.3	23.7

V.D Problem Solving Performance

One of the primary motivations for conducting the study was to determine if the use of the literate program paradigm leads to improved problem solving skills. Traditionally, instruction in the introductory courses emphasizes the product of design (programs), but not the design process itself [35]. This emphasis is reinforced because teachers place a grade on the running program and not the process that produced it. Grades are assigned based on the success (or failure) of the program, given certain test cases, and not on the design of the program [35].

The introductory course at Texas A&M University can be considered typical and, in

the past, has been taught using this same grading mechanism. A portion of the grade has been assigned based on the program documentation, and the remainder of the grade has been determined based on if the program ran with selected test cases. For this reason, there are no comparison figures available for validating the test group's problem solving skills. However, the problem solving skills of the test group were measured periodically during the semester and it can be determined if these skills improved.

V.D.1 Problem Solving Performance of the Test Group

The actual scores received by the test group on the problem solving portion of each lab are included in Appendix C. The mean and the standard deviation of the scores for the problem solving portion of each lab are shown in Table 7 and Table 8.

Table 7. Mean Problem Solving Scores – Labs (Percent)

Lab	Overall	Majors	Non-Majors
Lab 2	83.1	80.6	91.3
Lab 3	83.6	81.9	89.8
Lab 4	88.4	87.5	91.3
Lab 5	89.8	88.3	94.8

Table 8. Standard Deviation of Problem Solving Scores – Labs (Percent)

Lab	Overall	Majors	Non-Majors
Lab 2	16.9	18.5	4.2
Lab 3	17.0	18.5	7.2
Lab 4	10.5	11.4	5.9
Lab 5	10.7	11.6	4.1

The problem solving skills for the test group (as well as the difficulty of the problems) increased over the course of the semester. This increase in problem solving skills was

experienced by both computer science majors and non-majors.

Notice that the non-computer science majors consistently scored higher on program design than the computer science majors. This may be attributable to the fact that many of the computer science majors had some experience in Pascal prior to the class. This experience may affect the students' problem solving skills for two reasons. Firstly, as stated earlier, it is sometimes difficult to change the way a person has learned to perform a particular task. Secondly, this previous programming skill may have detracted from their ability to separate problem solving from programming.

The actual scores received by the test group on the problem solving portion of each test are included in Appendix D. The mean and standard deviation of the scores for the problem solving portion of each test are shown in Table 9 and Table 10.

Table 9. 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

Table 10. Standard Deviation of Problem Solving Scores – Tests (Percent)

Test	Overall	Majors	Non-Majors
Pre-Test	0.25	0.27	0.15
Test 1	0.12	0.12	0.11
Test 2	0.18	0.20	0.12
Test 3	0.15	0.16	0.10
Post-Test	0.24	0.24	0.25

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.

V.E Programming Performance

Another motivation for conducting the study was to determine if the literate programming paradigm can improve program quality as a result of improved problem solving. Program quality, however, is difficult to define without studying maintenance of code over a period of several years.

V.E.1 Programming Specifications for the Comparison Groups

The programs for the Fall 1990 honors CS/1 comparison group were graded based on program style, or documentation, and program execution, or "correctness." For grading

purposes, 40% of the grade was based on style, with the remaining 60% of the grade based on execution.

Program style refers to how well the programs were documented. Two levels of documentation were defined:

- program/module level - where the programmer gives a general description of the intent of the program and/or module. Some modules implement rather complex algorithms, so their descriptions are more detailed than the others. Other modules might make important assumptions that should be mentioned.
- code level - where the programmer explains what the program is doing at a particular moment in time.

Style includes the use of meaningful names for identifiers, indentation and white space for program readability, and limited use of global variables. All of the above can be combined to create a well-styled program.

The first program assignment was a Pascal source code listing that the students entered using the Turbo Pascal editor. The program was then compiled, debugged (if necessary), and executed. The program was designed as a learning exercise for the edit, compile, test, and debug process.

The remaining programs were assigned in problem specification form. The students were given a problem to solve using specific constructs, such as `if-then-else`, `while-do`, and `case`. They were also told to use certain types of subprograms, parameter passing, file manipulation, and data structures.

There is no data on programs for the Fall 1992 honors comparison group. This is due to the fact that it was taught by a different instructor.

V.E.2 Programming Specifications for the Test Group

The program grading guidelines for the test group are included in Appendix A. The programs were graded based on program design, documentation, and program execution, or "correctness." For grading purposes, 50% of the grade was based on documentation and design, with the remaining 50% of the grade based on execution.

Program documentation and design addressed the design issues, such as problem statement, required inputs, outputs generated, processing required, algorithm development, and testing. Included in the grade was the degree of consistency between the documentation and the implementation. The two levels of documentation defined previously (program/module and code), and the specifications for a well-styled program were also included in the documentation and design portion of the grade.

The first program assignment was a WEB source code listing that the students entered using the emacs editor. The program was then WEAVED and T_EXed to produce a device-independent file which was then converted by a printer driver to produce a hardcopy listing of the program. The program was also TANGLED, compiled, debugged (if necessary), and executed. The program was designed as a learning exercise for the edit, WEAVE, T_EX, dvips, TANGLE, compile, test, and debug process.

Like the comparison groups, the remaining programs were assigned in problem specification form. The students were given a problem to solve and part of the process was to extract (from the instructor and/or the teaching assistant) the necessary information for solving the problem. Specific constructs, such as *if-then-else*, *while-do*, and *case*, were used for the assignments. Like the comparison groups, they were told to use certain types of subprograms, parameter passing, file manipulation, and data structures.

V.E.3 Programming Performance of Test Group versus Comparison Groups

The actual scores received by the test and comparison groups on each lab are included in Appendix C. The mean and standard deviation of the scores for each lab are shown in Table 11 and Table 12.

Table 11. Mean Program Scores

Semester	Lab 1	Lab 2	Lab 3	Lab 4	Lab 5	Lab 6	Lab 7
Fall 90-H	99.0	99.2	94.2	92.0	94.1	90.0	90.5
Fall 93-H	89.8	85.8	87.8	77.9	76.6	75.7	N/A

Table 12. Standard Deviation of Program Scores

Semester	Lab 1	Lab 2	Lab 3	Lab 4	Lab 5	Lab 6	Lab 7
Fall 90-H	3.16	2.06	6.66	8.44	4.72	9.92	14.91
Fall 93-H	14.75	22.29	22.99	23.67	32.45	34.71	N/A

Notice the scores received by the test group are much lower than those received by the comparison group and the standard deviation is much higher.

The primary reason for the test group receiving lower program scores than the comparison group is due to the grading mechanism used for the test study. The programs were graded using more strict guidelines and high quality design was expected of the students.

There may be several reasons for experiencing lower program scores using this program development methodology:

1. The students experienced trouble comprehending the steps necessary for processing a WEB program; however, the degree of difficulty should be reduced over a period of extended use.
2. The programs themselves were not more difficult; however, the difficulty of the program may have been emphasized because less information about solving the problem was given to the students.

3. The debugging facility which is available in Turbo Pascal was not an option for the students, thus the debugging task was more difficult and the debugging time was increased.
4. Several students submitted programs that did not run in order to receive credit for the design, thus lowering the average.

Although it appears the test study group did not perform as well on the programming assignments as the comparison groups, this is not necessarily the case. Several of the students in the test study group could not fully grasp the concept problem solving with the use of WEB programming. However, the majority of the students performed well.

The quality of the programs, including documentation, was much higher for the test study group. Their program documentation contained all of the necessary design steps, including design rationale and testing. The documentation produced by the comparison groups typically included only items such as identifier descriptions and the purpose of the subprograms. The higher quality of the documentation produced by the test study participants is certainly not reflected in the grades.

V.F Exam Performance

One of the methods by which the students were measured and compared was with the use of in-class exams. The purpose of the exams was to measure problem solving ability and knowledge of Pascal syntax.

V.F.1 Exam Structure for the Comparison Groups

The exam structure for the comparison groups was tailored more towards measuring knowledge of Pascal syntax and programming. The questions on the exams for the honors CS/1 classes were designed to test the students' Pascal knowledge and their ability to use

specific constructs rather than their problem solving ability.

The final exam for the Fall 1990 honors class consisted of 69 true/false and multiple choice questions about Pascal syntax and concepts. The remaining questions tested the students' ability to write Pascal programs. The final exam for the Fall 1992 honors class consisted of questions which tested the students' ability to write portions of programs (Pascal syntax) and implement specific data structures. Although the ability to write programs involves using some type of problem solving ability, the students in the comparison groups were not specifically tested on their problem solving ability.

V.F.2 Exam Structure for the Test Group

The exams for the test study group were designed to test the students' knowledge of Pascal as well as their problem solving ability. Approximately 25% of each exam tested problem solving skills. The remaining 75% of each exam tested the students' knowledge of Pascal concepts and their programming ability much in the same way the comparison groups were tested. The exams for the test group, including the pre-test and the final, are included on pages 112-113 and 146-188 in Appendix A.

The final exam for the test study participants was designed such that specific comparisons could be made between the test study group and the comparison groups. This was accomplished by using, where possible, the same (or similar) questions on the exam to test knowledge of Pascal syntax.

V.F.3 Exam Performance of Test Group versus Comparison Groups

The actual scores received by the test and comparison groups on each exam are included in Appendix C. The mean and standard deviation of the scores for each exam are

shown in Table 13 and Table 14.

Table 13. Mean Exam Scores

Semester	Exam 1	Exam 2	Exam 3	Final Exam	Pascal Concepts
Fall 90-H	83.1	77.7	70.6	74.4	74.4
Fall 92-H	N/A	N/A	N/A	73.5	N/A
Fall 93-H	78.6	74.7	75.4	75.0	77.3

Table 14. Standard Deviation of Exam Scores

Semester	Exam 1	Exam 2	Exam 3	Final Exam	Pascal Concepts
Fall 90-H	11.06	12.26	11.53	10.81	9.11
Fall 92-H	N/A	N/A	N/A	15.75	N/A
Fall 93-H	10.23	12.31	13.35	12.21	12.27

The Mann-Whitney U-test (also known as the Wilcoxon Rank Sum Test) can be used to determine if there is a significant difference between the exam performance of the test study group and the exam performance of the comparison classes. The Mann-Whitney U-test is a statistical test used to determine if there is a statistically significant difference between the performance of two independent groups [11, 16, 47]. This test is similar to the t-test and makes three assumptions:

1. Both samples are random samples from their respective populations.
2. In addition to independence within each sample, there is mutual independence between the two samples.
3. The measurement scale is at least ordinal.

If both sample sizes are 10 or larger (as is the case here), the sampling distribution of T is approximately normal, which allows us to use a z statistic.

The Mann-Whitney U-test was conducted to determine if there was a significant difference between the exam performance of the test study group and that of the Fall 1990

comparison class. The critical value of z for $\alpha = 0.10$ is 1.282. The computed value, 0.04708, does not exceed 1.282, so we conclude that the distributions of grades on the final exam for the two groups are not significantly different.

We can use the same test to compare the Fall 1992 honors comparison group with the test study group. The computed value, 0.1244, does not exceed 1.282, so again we conclude that the distributions of grades on the final exam for the two groups are not significantly different.

Each of the final exams are equivalent in nature and level of difficulty. Each final exam also contained questions designed specifically to test the students' knowledge of Pascal concepts and syntax. The test study group scored higher than both of the comparison groups when tested on their knowledge of Pascal. Therefore it can be concluded that the teaching methodology has no detrimental effect on the students' ability to perform well on exams testing both Pascal knowledge and problem solving ability.

V.G Course Performance

Another method by which the students were measured and compared was their overall performance in the CS/1 course as determined by their final grade. The actual figures for the grade distribution tables in this section are included in Appendix B. The percentages below include only those students that completed the course. The grade classification of Other (which is not included in the calculations below) are those of Q (dropped before the semester deadline), WP (withdrew passing), WF (withdrew failing), NG (no grade for the course), S (satisfactorily passed), and U (unsatisfactory).

V.G.1 Course Performance of Test Group versus Comparison Groups

Table 15 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 15. 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.

Table 16 is a summary of the grade distribution for computer science majors completing the CS/1 course for the subject and comparison classes (in percent form).

Table 16. Grade Distribution for CPSC/CSEN Majors (Percent)

Semester	A	B	C	D	F
Fall 90-H	20.0	55.0	10.0	5.0	10.0
Fall 92-H	50.0	16.7	20.8	4.2	8.3
Fall 93-H	28.6	32.1	25.0	7.1	7.1

The percentage of computer science students that passed the CS/1 course was comparable for the test study group and each of the comparison groups. The Fall 1990 and the Fall 1992 comparison groups had 85.0% and 87.5% of the computer science students, respectively, pass the course. The test group had 85.7% of the computer science students pass the course.

Table 17 is a summary of the grade distribution for non-computer science majors completing the CS/1 course for the subject and comparison classes (in percent form).

Table 17. Grade Distribution for Other Majors (Percent)

Semester	A	B	C	D	F
Fall 90-H	21.4	42.9	21.4	7.1	7.1
Fall 92-H	53.3	26.7	20.0	0.0	0.0
Fall 93-H	11.1	66.7	11.1	0.0	11.1

The percentage of non-majors that passed the CS/1 course was 100.0% for the Fall 1992 comparison group. The percentage of non-majors that passed the CS/1 course in the test study group was 88.9%, while the Fall 1990 comparison group had 85.7% pass the course.

Figure 25 shows the percentage grade distribution for each of the CS/1 classes. Notice that the grade distribution for the Fall 1992 comparison group is much different from the distributions for the Fall 1990 comparison group and the Fall 1993 test group. This difference in distributions is largely attributable to the difference in instructors.

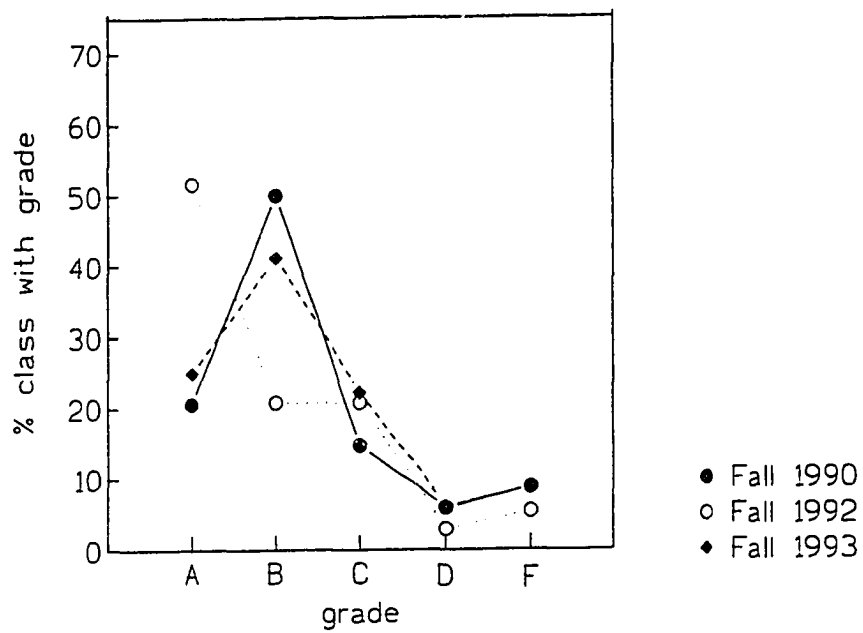


Figure 25. CS/1 Course Grade Distribution

V.H CPSC 120 Performance

Another method by which the students were measured and compared was their overall performance in the CS/2 course as determined by their final grade. The actual scores received by the test and comparison groups in the CS/2 course are included in Appendix C. The calculations given include only those students that completed the CS/2 course.

V.H.1 Subsequent Course Performance of Test Group versus Comparison Groups

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 18 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 18. 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.

Figure 26 shows the percentage grade distribution for each of the CS/2 classes. Notice

that there is no significant difference in the grade distributions for the test study group and both of the comparison groups.

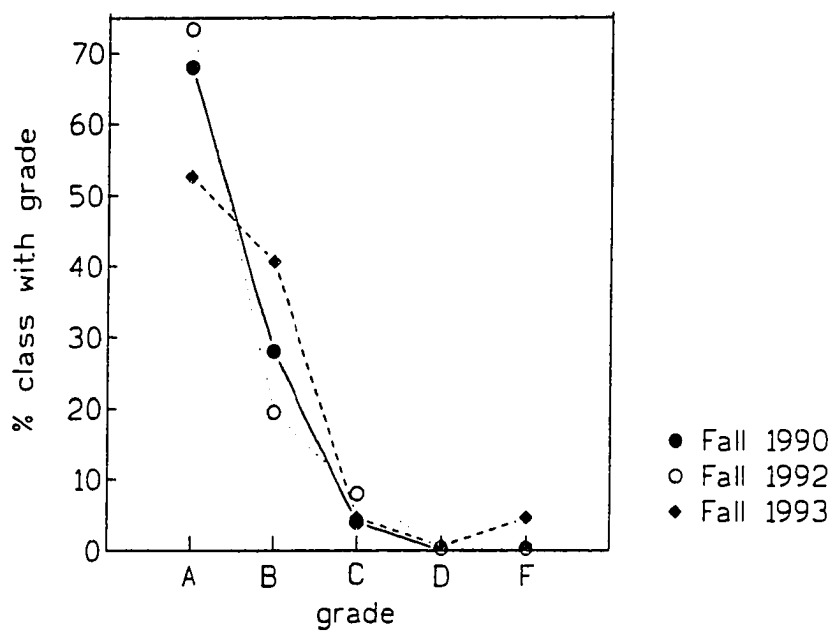


Figure 26. CS/2 Course Grade Distribution

Table 19 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 19. 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 20 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.

The actual figures for the grade distribution for each of the CS/2 classes are included in Appendix E.

Table 20. Average Difference in Grade for CS/2 Classes

Semester	Diff. in Section	Diff. in Instructor	Diff. 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.

V.I CPSC 210 Performance

The final method by which the students were measured and compared was their overall performance in the Data Structures course as determined by their final grade. The Data Structures course is the first course students take upon completion of the CS/1 and CS/2 courses. At this point, the students are no longer learning programming languages. Instead, they are using their programming and problem solving abilities in the “design of algorithms for efficient implementation and manipulation of data structures” [72]. In other words, this is the course where good problem solving skills take precedence over programming ability.

The actual scores received by the test and comparison groups in the Data Structures course are included in Appendix C. The calculations below include only those students that completed the Data Structures course.

V.I.1 Data Structures Course Performance of Test Group versus Comparison Groups

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 21 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 21. 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.

Figure 27 shows the percentage grade distribution for each of the Data Structures classes. Notice that there is a significant difference in the grade distributions for the groups.

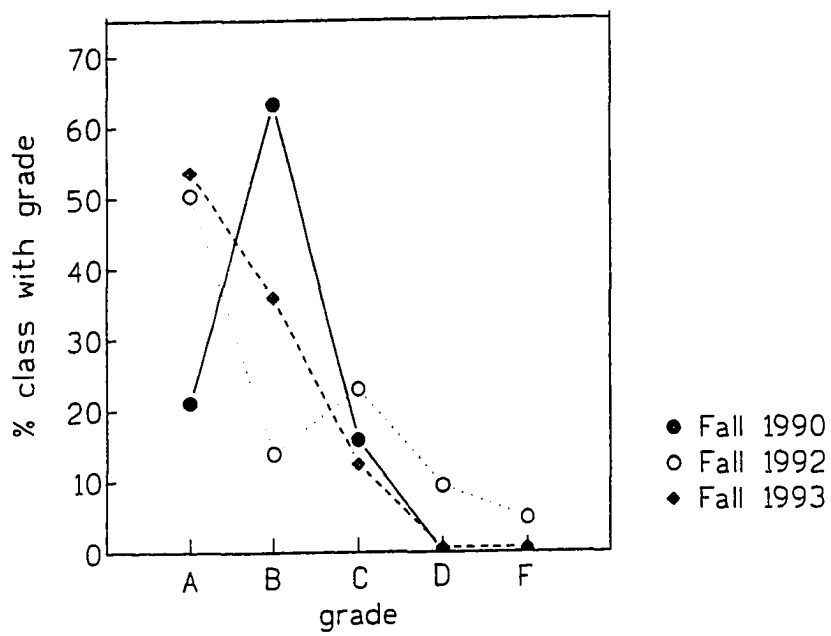


Figure 27. Data Structures Course Grade Distribution

Table 22 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 22. 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.

Figure 28 shows the progression of average grades for the Fall 1993 test study group and both the Fall 1990 and the Fall 1992 comparison groups. Notice that the test study group demonstrates an upward progression in terms of average grade. Both of the comparison groups increase in average grade from the CS/1 to the CS/2 course. However, the average grade for both groups drops significantly in the Data Structures course.

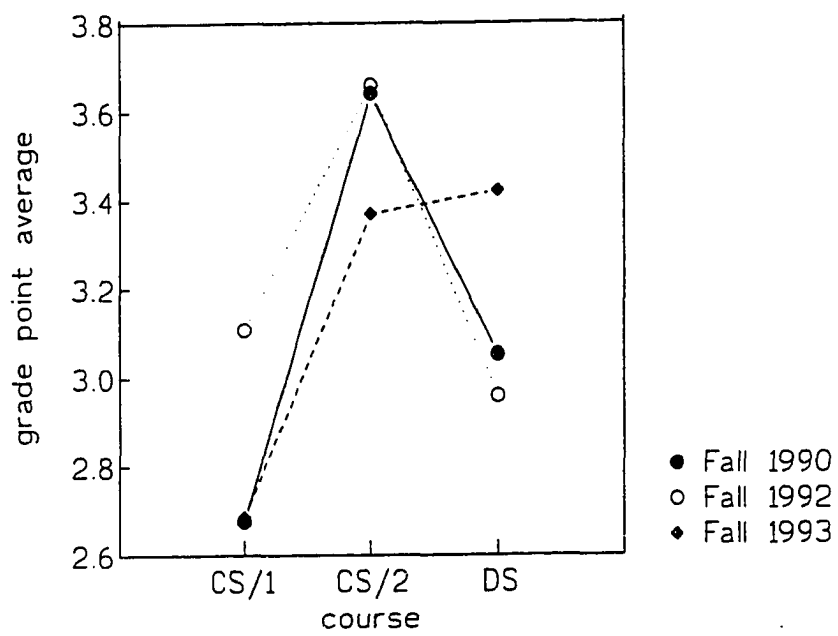


Figure 28. Grade Progression

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.

V.J 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.

A graphic rating scale [44] 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. In addition, the scale contained a "not discussed", rated 0, category. Three people evaluated the reaction of the test subjects. None of the people had prior training in rating. The rating scale and reproductions of the student reports can be found in Appendix F. Below is a summary of the results of the rating process. The "not discussed" selections were not included in the calculations. The mean and the standard deviation of the scores for each of the raters are shown in Table 23 and Table 24.

Kendall's coefficient of concordance [44] was used to evaluate the raters. The result was a value of 0.673, which indicates there was a modest level of agreement between the raters.

The first question required that the raters assess the students' original reaction to being told they were going to learn something called WEB programming. 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.

Table 23. Evaluation of Fall 1993 CPSC 110H Students' Reactions

Question	Rater 1	Rater 2	Rater 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

Table 24. Standard Deviation of Rating Scale

Question	Rater 1	Rater 2	Rater 3	Overall
1	0.94	1.00	1.13	1.07
2	1.05	1.17	1.02	1.33
3	1.26	1.26	1.31	1.32
4	1.17	1.22	1.25	1.38
5	1.33	1.36	1.47	1.41
6	1.35	1.09	0.68	1.12

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. The second question required that the raters assess the students' original expectation of the class. As shown by the ratings, most students entered the course under the impression that CPSC 110H was a course in Turbo Pascal, despite the course description.

The next three questions required that the raters assess the students' reactions to the GNU Emacs editor, \TeX , and WEB programming. 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.

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 T_EX. Despite the lack of T_EX knowledge, the students seemed to adapt to the WEB environment. The raters seemed to believe the students' reaction to the WEB programming process 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. The last question required that the raters assess the students' overall understanding of the WEB process. In general, the students' understanding was average to good. Many of the students continued to have difficulty separating the concepts of editor, WEB files, T_EX commands, etc. They seemed overwhelmed with having to learn more than just the Pascal language using the Turbo environment, despite the attempt to minimize the amount of material with the use of reference cards.

CHAPTER VI

SUMMARY, CONCLUSION, AND FUTURE WORK

VI.A Summary

The cost of software development is a subject of concern for software researchers and developers. The dominant portion of the lifetime cost of software is not in the development, but in the maintenance of the software. Research in the area of improving the quality of software documentation to reduce maintenance costs is increasing [49, 68].

Donald Knuth coined the phrase “literate programming” to refer to programs that are meant to be read by human beings, as well as executed by a computer. His original intent was that WEB programs be written and used by experienced programmers [28]. However, with practice, even less experienced programmers have had success with writing WEB programs [45, 62, 63].

A WEB consists of documentation, written in a formatting language, and program statements, written in a programming language. The WEAVE process prepares a combination of the documentation and the program to be read by humans. The TANGLE process extracts the program statements and creates a source program file to be executed by the computer.

The methods with which we teach programming and problem solving to our introductory students is an important research topic. Linn and colleagues have done an extensive amount of work using case studies and templates to teach programming. The basic premise is that students should be taught how an expert uses knowledge about a

previously solved problem in order to solve a new problem [34, 35, 37, 38, 58].

Soloway and colleagues have also performed research in the area of teaching novice programmers how to solve problems. They have found that the students have trouble “putting the pieces together” in order to solve the problem [64, 66, 69], rather than with Pascal syntax and constructs.

This research involved the use of the literate programming paradigm in the introductory computer science class in order to improve the software development process. The methodology combines literate programming with the problem solving process to capture, document, and emphasize the problem solving process.

The program development methodology was used in the introductory computer science course at Texas A&M University. The students enrolled in the Fall 1993 honors class were required to use the development environment, `web-mode`, to create `WEB` programs. The lecture time was spent discussing problem solving techniques and the syntax of the Pascal programming language. During the lab time, the students were required to use the editing environment (`web-mode`), which is based on GNU Emacs. They also received an introduction to the `TEX` formatting language and the `WEB` rules and constructs.

The students initially designed their problem solution using the `WEB` rules. They received feedback on their design and, using their design and any suggested changes, implemented their solution using the Pascal programming language.

The program development methodology was evaluated using several different measures:

1. The students were given a pre-test and were then tested periodically to evaluate their problem solving skills.
2. The students were compared with past introductory computer science course students to evaluate their performance on programming assignments, exams, and in the course.
3. The students were compared with past introductory computer science course students to evaluate their performance in the subsequent `CS/2` course.

4. The students were compared with past introductory computer science course students to evaluate their performance in the Data Structures course.

VI.B Conclusion

The Fall 1990 Honors CS/1 course was taught in a manner that differed somewhat from the traditional CS/1 course. 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. Therefore, it can be concluded that the use of literate programming in the introductory computer science class was successful. 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 than 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 the program development methodology utilizing the literate programming paradigm 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.

Negatives that are not felt to offset the positives:

- The program scores were not as high for those students using literate programming.

Other points of interest include:

- The use of two one-hour lab sections is recommended as an effective teaching design rather than the use of one two-hour lab section. This appears to reinforce iteration of the problem solving process.
- Those students familiar with the Pascal programming language, or another programming language, exhibited more resistance to change.

VI.C Extensions and Future Research

The literate programming development methodology should be used in the introductory computer science course repetitively to see if the performance experienced during the test study remains consistent, improves, or fades. More importantly, tests should be performed to verify that neophytes do, indeed, experience more success using the literate programming paradigm than those with some programming experience.

Tests should also be performed to compare the readability or, more importantly, the understandability of a WEB program to a “regular” program. This could be accomplished in several ways. The students could be given a “regular” program and then be required to answer questions about the program. They could then be given the WEB program, be required to answer questions about the program, and a determination could be made as to which program was easier to read. There may be some discrepancy as to whether or not prior understanding of the “regular” program affected the understanding of the WEB program.

This same concept of comparison could be used to test the maintainability of a program. Two sets of students could be required to modify a program. One group will be required to perform maintenance on a “regular” program and the second group be required to perform maintenance on a WEB program. The results could then be used to

determine the maintainability of a WEB program.

The tests regarding the readability and maintainability of a program could be conducted in the educational environment. However, a study regarding the maintainability of a WEB program versus a “regular” program should be performed over several years of the program’s lifetime.

It is believed that extended use of the literate programming methodology may lead to improved problem solving skills and, therefore, improve the software development process. For this reason, the program development methodology should continue to be tested throughout all levels of the undergraduate curriculum. A study should be performed in which the literate programming development methodology is used by a group of students over the course of their college career. This study might also compare students with no programming experience to those with some exposure to programming languages.

Extended use of the program development methodology can be used to improve the problem solving skills for novice programmers. It should prove to be an effective means for teaching problem solving and programming in the introductory computer science course. The improvement in problem solving skills should result in well-documented, higher quality software that is easier to read and maintain.

REFERENCES

1. Bentley, J. Programming pearls—literate programming. *Communications of the ACM* 29, 5 (May 1986), 364–369.
2. Bishop, J. M., and Gregson, K. M. Literate programming and the LIPED environment. *Structured Programming 13* (1992), 21–34.
3. Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
4. Boehm, B. W. Improving software productivity. *IEEE Computer* 21, 5 (September 1987), 43–57.
5. Boehm, B. W. A spiral model of software development and enhancement. *IEEE Computer* 21, 5 (May 1988), 61–72.
6. Brooks, F. P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer* 20, 4 (April 1987), 10–19.
7. Brown, M. E. *An Interactive Environment for Literate Programming*. PhD dissertation, Texas A&M University, College Station, TX, Aug. 1988.
8. Brown, M. E., and Childs, B. An interactive environment for literate programming. *Journal of Structured Programming* 11, 1 (1990), 11–25.
9. Buyukisik, O. F. Communication on June 1, 1993 at 9:49 CDT. Literate Programming Mailing List. e-mail: ae1181t@stnfor.ae.ge.com.
10. Cameron, D., and Rosenblatt, B. *Learning GNU Emacs*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
11. Cates, W. M. *A Practical Guide to Educational Research*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.
12. Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., and Glaser, R. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science* 13 (1989), 145–182.
13. Conklin, J. Design rationale and maintainability. Tech. Rep. STP-249-88, Microelectronics and Computer Technology Corporation, Austin, TX, June 1988.
14. Conklin, J., and Begeman, M. L. gIBIS: a hypertext tool for team design deliberation. In *Hypertext '87 Papers* (New York, NY, 1987), Association for Computing Machinery, pp. 247–251.
15. Conklin, J., and Begeman, M. L. gIBIS: a hypertext tool for exploratory policy discussion. Tech. Rep. STP-082-88, Microelectronics and Computer Technology Corporation, Austin, TX, March 1988.

16. Conover, W. J. *Practical Nonparametric Statistics*, 2 ed. John Wiley & Sons, Inc., New York, 1980.
17. Cordes, D., and Brown, M. The literate-programming paradigm. *IEEE Computer* 24, 6 (June 1991), 52–61.
18. 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.
19. Department of Defense – Ada Joint Program Office. Ada methodologies: Concepts and requirements. *Software Engineering Notes* 8, 1 (January 1983), 33–50.
20. Dyer, J. R. *Understanding and Evaluating Educational Research*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1979.
21. Etlinger, H. A., and Lutz, M. J. Professional literacy: Labs for advanced programming courses. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education* (Mar. 1994), vol. 26, pp. 102–105.
22. Fairley, R. E. *Software Engineering Concepts*. McGraw-Hill Publishing Company, Inc., New York, 1985.
23. Fix, V., Wiedenbeck, S., and Scholtz, J. Mental representations of programs by novices and experts. In *Proceedings INTERCHI '93 (Human Factors in Computing Systems)* (New York, NY, April 1993), Association for Computing Machinery, pp. 74–79.
24. Henderson-Sellers, B., and Edwards, J. M. The object-oriented systems life cycle. *Communications of the ACM* 33, 9 (Sept. 1990), 142–159.
25. Husic, F. T., Linn, M. C., and Sloane, K. D. Adapting instruction to the cognitive demands of learning to program. *Journal of Educational Psychology* 81, 4 (1989), 570–583.
26. Kendall, P. A. *Introduction to Systems Analysis and Design: A Structured Approach*, second ed. Wm. C. Brown Publishers, Dubuque, IA, 1989.
27. Knuth, D. E. The WEB system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, Sept. 1983.
28. Knuth, D. E. Literate programming. *Computer Journal* (May 1984), 97–111.
29. Knuth, D. E. *T_EX: The Program*, vol. B of *Computers & Typesetting*. Addison-Wesley, Reading, MA, 1986.
30. Koffman, E. B. *Pascal: Problem Solving and Program Design*, fourth ed. Addison-Wesley Publishing Company, Inc., Reading, MA, 1992.
31. Kreitzberg, C. B., and Shneiderman, B. *The Elements of FORTRAN style: Techniques for Effective Programming*. Harcourt Brace Jovanovich, Inc., New York, 1972.
32. Larkin, T. Communication on July 16, 1993 at 9:05 CDT. Literate Programming Mailing List. e-mail: ts11@cornell.edu.

33. Lease, M. W., Lively, W. M., and Leggett, J. J. Using an issue-based hypertext system to capture the software life-cycle process. *Hypermedia* 2, 1 (1991), 29–46.
34. Linn, M. C., and Clancy, M. J. Can experts' explanations help students develop program design skills? *International Journal of Man-Machine Studies* 36, 4 (1992), 511–551.
35. 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.
36. Linn, M. C., and Clancy, M. J. *Designing Pascal Solutions: A Case Study Approach*. W. H. Freeman, New York, 1992.
37. Linn, M. C., and Dalbey, J. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist* 20, 4 (1985), 191–206.
38. 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.
39. Lins, C. A first look at literate programming. *Journal of Structured Programming* 10, 1 (1989), 60–62.
40. Lins, C. An introduction to literate programming. *Journal of Structured Programming* 10, 2 (1989), 107–112.
41. Liu, L., and Horowitz, E. Object database support for a software project management environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (New York, NY, November 1988), Association for Computing Machinery, pp. 85–96.
42. Maruyama, G., and Deno, S. *Research in Educational Settings*. SAGE Publications, Inc., Newbury Park, CA, 1992.
43. Mehringer, V. Communication on April 14, 1993 at 17:19 CDT. Literate Programming Mailing List. e-mail: vince@eye.com.
44. Meister, D. *Behavioral Analysis & Measurement Methods*. John Wiley & Sons, Inc., New York, 1985.
45. Motl, M. B. *A Literate Programming Environment Based on an Extensible Editor*. PhD dissertation, Texas A&M University, College Station, TX, December 1990.
46. Nance, D. W. *Pascal: Understanding Programming and Problem Solving*, third ed. West Publishing Company, Inc., St. Paul, MN, 1992.
47. Ott, L. *An Introduction to Statistical Methods and Data Analysis*, 3 ed. PWS-Kent Publishing Company, Boston, MA, 1988.
48. Pierce, K. R. Rethinking academia's conventional wisdom. *IEEE Software* 10, 2 (March 1993), 94–95, 99.

49. Pinto, J., and Soloway, E. Providing the requisite knowledge via software documentation. In *Proceedings CHI '88 (Human Factors in Computing Systems)* (New York, NY, 1988), Association for Computing Machinery, pp. 257-261.
50. Pirolli, P. L., and Anderson, J. R. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology* 39, 2 (1985), 240-272.
51. Ramsey, N. Communication on June 28, 1993 at 12:11 CDT. Literate Programming Mailing List. e-mail: norman@bellcore.com.
52. Ramsey, N. Weaving a language-independent WEB. *Communications of the ACM* 32, 9 (Sept. 1989), 1051-1055.
53. Ramsey, N., and Marceau, C. Literate programming on a team project. *Software—Practice and Experience* 21, 7 (July 1991), 677-683.
54. Reder, L. M., Charney, D. H., and Morgan, K. I. The role of elaborations in learning a skill from an instructional text. *Memory and Cognition* 14 (1986), 64-78.
55. Redmiles, D. F. Reducing the variability of programmers' performance through explained examples. In *Proceedings INTERCHI '93 (Human Factors in Computing Systems)* (New York, NY, April 1993), Association for Computing Machinery, pp. 67-73.
56. Roberge, J., and Suriano, C. Using laboratories to teach software engineering principles in the introductory computer science curriculum. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education* (Mar. 1994), vol. 26, pp. 106-110.
57. Savitch, W. J. *Turbo Pascal*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.
58. Schank, P. K., Linn, M. C., and Clancy, M. J. Supporting pascal programming with an on-line template library and case studies. *International Journal of Man-Machine Studies* 38, 6 (1993), 1031-1048.
59. Sewell, E. W. *Weaving a Program: Literate Programming in WEB*. Van Nostrand Reinhold, New York, 1989.
60. Shelly, G. B., and Cashman, T. J. *Business Systems Analysis and Design*. Anaheim Publishing Company, Fullerton, CA, 1975.
61. Shum, S., and Cook, C. Using literate programming to teach good programming practices. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education* (Mar. 1994), vol. 26, pp. 66-70.
62. Smith, L. M. C. Measuring complexity and stability of web programs. Master's thesis, Oklahoma State University, Stillwater, OK, December 1990.
63. Smith, L. M. C., and Samadzadeh, M. H. Measuring complexity and stability of web programs. *Structured Programming* 13 (1992), 35-50.

64. Soloway, E. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (September 1986), 850–858.
65. Soloway, E. Should we teach students to program? *Communications of the ACM* 36, 10 (October 1993), 21–24.
66. Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 595–609.
67. 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.
68. Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. Designing documentation to compensate for delocalized plans. *Communications of the ACM* 31, 11 (November 1988), 1259–1267.
69. Spohrer, J. C., and Soloway, E. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM* 29, 7 (July 1986), 624–632.
70. Sylvan, K. Communication on April 14, 1993 at 5:36 CDT. Literate Programming Mailing List. e-mail: kayvan@satyr.Sylvan.COM.
71. Sylvan, K. Communication on June 5, 1993 at 2:11 CDT. Literate Programming Mailing List. e-mail: kayvan@satyr.Sylvan.COM.
72. Texas A&M University – *Undergraduate Catalog*, 1993-1994. No. 116.
73. Thimbleby, H. Experiences of ‘literate programming’ using `cweb` (a variant of Knuth’s `WEB`). *The Computer Journal* 29, 3 (June 1986), 201–211.
74. Tucker, A. B., Ed. *Computing Curricula 1991 – Report of the ACM/IEEE-CS Joint Curriculum Task Force* (New York, NY, December 1990), Association for Computing Machinery.
75. Tucker, A. B., and Wegner, P. New directions in the introductory computer science curriculum. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education* (Mar. 1994), vol. 26, pp. 11–15.
76. van Ammers, E. W. Communication on July 16, 1993 at 7:05 CDT. Literate Programming Mailing List. e-mail: ammers@rcl.wau.nl.
77. Wagner, Z. Communication on July 16, 1993 at 3:00 CDT. Literate Programming Mailing List. e-mail: WAGNER%2FCSEARN.BITNET@SHSU.edu.
78. Williams, R. N. Funnelweb User’s Manual. anonymous FTP at [sirius.itd.adelaide.edu.au](ftp://sirius.itd.adelaide.edu.au), May 1992. V1.0 for FunnelWeb V3.0.
79. Wirth, N. *Systematic Programming: An Introduction*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.
80. Wittenberg, L. Communication on July 18, 1993 at 13:37 CDT. Literate Programming Mailing List. e-mail: leew@pilot.njin.net.

APPENDIX A

COURSE MATERIALS

This appendix consists of materials that were distributed to the participants enrolled in the test study. The first 4 pages are information on how the students were graded and what was expected from each of their lab assignments. The students enrolled in the comparison classes received similar guidelines.

The next 3 pages are samples of the grading sheets for the lab assignments. The tentative class schedule is included and consists of a list of the textbook chapters for which each student was responsible, test dates, and the due dates for lab assignments.

The pre-test which was given to the test study participants appears next, followed by the GNU Emacs Reference Card.

The next document (5 pages) is the WEB User Manual Exerpt which was distributed to the test study participants.

The students' first lab assignment was to type in the quadratic equation problem, which appears next. It is immediately followed by a TANGLEd and a WEAVEd version of the program (10 pages total).

The next three documents are T_EX samples which were distributed to the students during the lab period (9 pages total).

The remainder of the appendix consists of all lab assignments (6 pages) and the exams which were given during the course (43 pages).

**CPSC 110H
PROGRAMMING I**

**Ms. Byrum
311B H. R. Bright
Office Phone: 845-5787**

Prerequisite: High School Algebra
Office Hours: 4:00-5:00 MTWR
Others by appointment

REQUIRED TEXTS: Nance, Douglas W., *Pascal: Understanding Programming and Problem Solving, Third Edition*, West Publishing Company, 1992.

OPTIONAL TEXT: Nance, Douglas W., *Student Solutions Manual to Accompany Pascal: Understanding Programming and Problem Solving, Second Alternate Edition*, West Publishing Company, 1992.

EXAMINATIONS: (60% of the course grade)

3 Class Examinations	12% each
Final Examination-Comprehensive	24%

Tentative Exam Schedule:

Exam 1	Friday, September 24
Exam 2	Friday, October 22
Exam 3	Friday, November 19
Final	Tuesday, December 14, 10:30 a.m. -12:30 p.m.

NOTE: There are no exemptions for the final examination. Check the final exam time. If the final exam time is a problem, you need to drop this course.

ASSIGNMENTS: (40% of the course grade)

30%	programming/homework assignments
10%	class participation/short, unscheduled quizzes

GRADING POLICY: Final grades will be assigned as follows:

90 - 100	A
80 - 89	B
70 - 79	C
60 - 69	D
below 60	F

CLASS INFORMATION AND POLICIES
Department of Computer Science, HRBB 311B, 845-5787

- ATTENDANCE:** Attendance will be taken each day in class. Attendance will not be used in calculating your final grade; however, class participation is a portion of your final grade. If you are absent from class please do not come by my office and ask me to repeat the class lecture. There will be no smoking, no chewing of tobacco, no bare feet, and no wearing of hats during class.
- EXAMINATION POLICY:** All class examinations are considered to be a major part of the course work upon which a large part of the course grade depends. There are **NO** make-up exams! Class examinations will be announced at least two classes prior to the examination. If you have a conflict with another university event, you must contact me well in advance of the examination. In case of an extreme emergency, contact me before the scheduled examination. Failure to do so will result in an examination grade of zero.
- ASSIGNMENT POLICY:** All assignments are due as specified by the lab assistant. Any assignment turned in after the assignments have been collected from the class is considered late. *Late labs will be penalized 10% for each calendar day late, beginning with the day on which the lab is due.* Under **NO** circumstances will any assignment be accepted for credit after the collected class assignments have been graded. If you are unable to turn in a lab during class time and are unable to find me or the lab assistant, place it under my door or under your lab assistant's door.
- IMPORTANT:** Every programming assignment **MUST** be turned in. Failure to turn in a programming assignment may result in the reduction of one (1) letter grade from the final course grade.
- CHEATING POLICY:** If in my judgement a student is found cheating on an examination, a grade of zero will be assigned as the examination grade and a minimum of one (1) letter grade will be lost in the course grade. A course grade of F may be assigned depending on the situation. A student found cheating on an examination may not drop the course.
- All other class assignments are to be done **INDEPENDENTLY**. Discussion is encouraged, but you are to do your own work. If in my judgement two or more people hand in assignments that I judge to be the same, a grade of zero will be awarded to all involved assignments and a minimum of one letter grade may be lost in the course grade. A recurrence of this by any individual will result in a grade of F in the course. Students should save all developmental copies of their programs so that individual program development can be verified to me if I think it is necessary. **DO YOUR OWN WORK!!!**

CPSC 110
Fall 1993
Laboratory Information

Lab Assistant: Peter Nuernberg
Lab Hours: TR 2:00 - 4:50
Office: HRBB 414B (inside the Hypermedia Research Laboratory)
Office Hours: MWF 1:45 - 2:45
Office Phone: 845-9980

Supplies Needed: at least two 3¹/₂" floppy diskettes.

Laboratory Guidelines:

For each lab, turn in both a disk with *all* source files, AND a printout of the WEAVE'd and T_EX'd WEB file.

Grade Breakdown:

I. Initial Design - 50 points

The initial design should address, at a minimum, the following points:

- A. Problem Statement
- B. Inputs Required
- C. Outputs Generated
- D. Processing Required
- E. Algorithm Development
- F. Testing

The design will be produced by WEAVE'ing and T_EX'ing the WEB source. Turn in only the printout during this phase. Spelling, grammar, style, etc. are important and will be factored into your grade.

II. Final Lab - 100 points

A. Documentation / Design - 50 points

This part of your final lab is a corrected and updated version of your initial design. It should address the same points as your initial design. Also included in your grade is the degree of correlation between your documentation and your implementation.

There are a few things I'd like to point out specifically about documentation:

- 1) You will be implementing some complex concepts in your programming assignments, so I expect your programs to be well-documented. There are two levels of documentation:
 - chapter / section level - at this level, you are to give a general description of what the chapter/section is intended to do. Note that the program description is accompanied by your Name, Class, etc. Some modules implement rather complex algorithms, so their descriptions are more detailed than the others. Other modules might make important assumptions that should be mentioned.
 - code level - at this level, you are to explain what the program is doing at that particular moment. This kind of comment is much more specific than the chapter/section description. If possible, make sure your comments don't "wander" all over the page.
- 2) You should give your identifiers meaningful names.
- 3) You should minimize the use of global variables. Any data item used in a module should be passed as a parameter or declared locally.

B. Implementation - 50 points

Implementation generally refers to how well your program solves the given problem. It is assumed that your program runs -- in fact, you automatically lose 50% if your program doesn't compile! This portion will cover any errors that exist in the compiled program. Failure to follow instructions will be reflected here.

A few words on cheating:

Don't do it.

Please read the section entitled Scholastic Dishonesty in the University Regulations if you are unsure what constitutes cheating. This section also details the disciplinary action which can be taken in scholastic dishonesty cases. These actions include grade penalty, probation, suspension, dismissal, and/or expulsion.

CPSC 110 - Lab 1 Grade Sheet

Name: _____

.WEB file: _____	/45	.TEX file _____	/20	.DVI file _____	/10
		.PAS file _____	/15	.EXE file _____	/10

Total: _____/100

CPSC 110 - Lab 1 Grade Sheet

Name: _____

.WEB file: _____	/45	.TEX file _____	/20	.DVI file _____	/10
		.PAS file _____	/15	.EXE file _____	/10

Total: _____/100

CPSC 110 - Lab 1 Grade Sheet

Name: _____

.WEB file: _____	/45	.TEX file _____	/20	.DVI file _____	/10
		.PAS file _____	/15	.EXE file _____	/10

Total: _____/100

CPSC 110 - Lab 1 Grade Sheet

Name: _____

.WEB file: _____	/45	.TEX file _____	/20	.DVI file _____	/10
		.PAS file _____	/15	.EXE file _____	/10

Total: _____/100

CPSC 110 - Lab 1 Grade Sheet

Name: _____

.WEB file: _____	/45	.TEX file _____	/20	.DVI file _____	/10
		.PAS file _____	/15	.EXE file _____	/10

Total: _____/100

CPSC 110 - Initial Design Grade Sheet

Name: _____

Lab: _____

Total grade: _____ /50

Problem Statement**Inputs Required****Outputs Generated****Processing Required****Algorithm Development****Testing****Writing Competence****Document Design****Other**

CP3C 110 - Lab Grade Sheet

Name: _____

Lab: _____ Total grade: _____ /100

Problem Statement

Compilation Errors

Inputs Required

Run-time Errors

Outputs Generated

Processing Required

Logical Errors

Algorithm Development

Testing

Document/Program Correspondence

Writing Competence

Elegance

Document Design

Other

Other

Design Grade: _____ /50

Implementation Grade: _____ /50

		Tentative Schedule		
WEEK	DATE	LECTURE TOPIC/CHAPTER	LAB TOPIC/CHAPTER	
1	8/30	Course Introduction		
	9/1	Chapter 1, 2		
	9/2	MEB, TANGLE, WEAVE	DOS	
2	9/6	Pre Test	Assign Lab 1	
	9/7	TeX, MEB, web-mode	emacs, WEAVE, TeX, dvips	
	9/9	Chapter 2, 3, 4	TANGLE, compile, Assign Lab 2	
3	9/10	Chapter 2, 3, 4	Lab 1 Due	
	9/13	MEB, Chapter 2, 3, 4		
	9/14	MEB, Chapter 2, 3, 4	emacs, TeX commands	
	9/15	MEB, Chapter 2, 3, 4	MEB commands, Lab 2 Design Due	
	9/17	MEB, Chapter 2, 3, 4		
4	9/20	MEB programming	Return Lab 2 Design	
	9/21	Chapter 5		
	9/22	Chapter 5		
	9/23	Test 1 - Chapters 3-4, MEB programming		
5	9/27	Chapter 5	Lab 2 Due, Assign Lab 2	
	9/28	Chapter 6		
	9/30	Chapter 6		
	10/1	Chapter 6		
6	10/4	Chapter 6	Lab 3 Design Due	
	10/5	Chapter 7	Return Lab 3 Design	
	10/7	Chapter 7		
	10/8	Chapter 7		
7	10/11	Chapter 8, 9		
	10/12	Chapter 8, 9		
	10/13	Chapter 8, 9		
	10/14	Chapter 9	Lab 3 Due, Assign Lab 4	
	10/15	Chapter 9		
8	10/18	Chapter 9	Lab 4 Design Due	
	10/19	Chapter 10	Return Lab 4 Design	
	10/20	Chapter 10		
	10/21	Test 2 - Chapters 5-9		
9	10/22	Chapter 10		
	10/26	Chapter 11		
	10/27	Chapter 11		
	10/28	Chapter 11		
	10/29	Chapter 11		
10	11/1	Chapter 11, 12	Lab 4 Due, Assign Lab 5	
	11/2	Chapter 12		
	11/3	Chapter 12		
	11/4	Chapter 12		
	11/5	Chapter 12		
11	11/8	Chapter 13, 14		
	11/9	Chapter 14	Lab 5 Design Due	
	11/10	Chapter 14	Return Lab 5 Design	
	11/11	Chapter 14		
	11/12	Chapter 14		
12	11/15	Chapter 15		
	11/16	Chapter 16		
	11/17	Chapter 16		
	11/18	Test 3 - Chapters 10-15		
13	11/19	Chapter 16	Lab 5 Due, Assign Lab 6	
	11/22	Chapter 16		
	11/23	Chapter 16		
	11/24	Chapter 16		
14	11/29	Chapter 16		
	11/30	Chapter 16		
	12/1	Chapter 16	Lab 6 Due	
	12/2	Chapter 16		
	12/3	Chapter 16		
15	12/6	Post Test		
	12/7	Review		
	12/8	Review		
	12/14	Final Exam		

September 6, 1993

Name _____

CPSC 110H

Mrs. Dunn

PreTest

1. What computer science courses have you taken? Give a description of any courses taken in high school and/or college(s).

2. What computer languages do you know?
 - * The languages I can program in without a reference manual.

 - * The languages I can program in with the help of the reference manual.

 - * I have previously programmed in these languages but would require some review and the use of a manual.

3. What experience do you have with emacs (prior to September 1)?

4. What experience do you have as a professional programmer? Give language and type of work.

5. What experience do you have with literate programming?

Classification _____ Major _____

September 6, 1993

Name _____

The purpose of this test is a preliminary evaluation of your problem-solving skills. State the steps necessary to solve this problem. Give detailed answers in complete English sentences and paragraphs.

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.

GNU Emacs Reference Card

(for version 18). web-made

Starting Emacs

To enter Emacs, just type its name: `emacs`

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (the usual way of leaving it) `C-z`
 exit Emacs permanently `C-x C-c`

Files

read a file into Emacs `C-x C-f`
 save a file back to disk `C-x C-s`
 insert contents of another file into this buffer `C-x i`
 replace this file with the file you really want `C-x C-v`
 write buffer to a specified file `C-x C-w`
 run Dired, the directory editor `C-x d`

Getting Help

The Help system is simple. Type `C-h` and follow the directions. If you are a first-time user, type `C-h t` for a tutorial. (This card assumes you know the tutorial.)

get rid of Help window `C-x 1`
 scroll Help window `ESC C-v`
 apropos: show commands matching a string `C-h a`
 show the function a key runs `C-h c`
 describe a function `C-h f`
 get mode-specific information `C-h m`

Error Recovery

abort partially typed or executing command `C-g`
 recover a file lost by a system crash `M-x recover-file`
 undo an unwanted change `C-x u` or `C-.`
 restore a buffer to its original contents `M-x revert-buffer`
 redraw garbaged screen `C-l`

Incremental Search

search forward `C-s`
 search backward `C-r`
 regular expression search `C-M-s`

Use `C-s` or `C-r` again to repeat the search in either direction.

exit incremental search `ESC`
 undo effect of last character `DEL`
 abort current search `C-g`

If Emacs is still searching, `C-g` will cancel the part of the search not done, otherwise it aborts the entire search.

© 1987 Free Software Foundation, Inc. Permission on back. v1.9

Motion

Cursor motion:

entity to move over	backward	forward
character	<code>C-b</code>	<code>C-f</code>
word	<code>M-b</code>	<code>M-f</code>
line	<code>C-p</code>	<code>C-n</code>
go to line beginning (or end)	<code>C-a</code>	<code>C-e</code>
sentence	<code>M-a</code>	<code>M-e</code>
paragraph	<code>M-[</code>	<code>M-]</code>
page	<code>C-x [</code>	<code>C-x]</code>
sexp	<code>C-M-b</code>	<code>C-M-f</code>
function	<code>C-M-a</code>	<code>C-M-e</code>
go to buffer beginning (or end)	<code>M-c</code>	<code>M-></code>

Screen motion:

scroll to next screen	<code>C-v</code>
scroll to previous screen	<code>M-v</code>
scroll left	<code>C-x <</code>
scroll right	<code>C-x ></code>

Killing and Deleting

entity to kill	backward	forward
character (delete, not kill)	<code>DEL</code>	<code>C-d</code>
word	<code>M-DEL</code>	<code>M-d</code>
line (to end of)	<code>M-0 C-k</code>	<code>C-k</code>
sentence	<code>C-x DEL</code>	<code>M-k</code>
sexp	<code>M-- C-M-k</code>	<code>C-M-k</code>

kill region `C-u`
 kill to next occurrence of `char` `M-x char`
 yank back last thing killed `C-y`
 replace last yank with previous kill `M-y`

Marking

set mark here `C-@` or `C-SPC`
 exchange point and mark `C-x C-x`
 set mark `any` words away `M-@`
 mark paragraph `M-h`
 mark page `C-x C-p`
 mark sexp `C-M-@`
 mark function `C-M-h`
 mark entire buffer `C-x h`

Query Replace

interactively replace a text string `M-Y`
 using regular expressions `M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next `SPC`
 replace this one, don't move `,`
 skip to next without replacing `DEL`
 replace all remaining matches `!`
 back up to the previous match `-`

exit query-replace `ESC`
 enter recursive edit (`C-M-c` to exit) `C-x`

Multiple Windows

delete all other windows	<code>C-x 1</code>
delete this window	<code>C-x 0</code>
split window in 2 vertically	<code>C-x 2</code>
split window in 2 horizontally	<code>C-x 5</code>
scroll other window	<code>C-M-v</code>
switch cursor to another window	<code>C-x o</code>
shrink window shorter	<code>M-x shrink-window</code>
grow window taller	<code>C-x ^</code>
shrink window narrower	<code>C-x {</code>
grow window wider	<code>C-x }</code>
select a buffer in other window	<code>C-x 4 b</code>
find file in other window	<code>C-x 4 f</code>
compose mail in other window	<code>C-x 4 m</code>
run Dired in other window	<code>C-x 4 d</code>
find tag in other window	<code>C-x 4 .</code>

Formatting

indent current line (mode-dependent)	<code>TAB</code>
indent region (mode-dependent)	<code>C-M-\</code>
indent sexp (mode-dependent)	<code>C-M-q</code>
indent region rigidly <code>any</code> columns	<code>C-x TAB</code>
insert newline after point	<code>C-o</code>
move rest of line vertically down	<code>C-M-o</code>
delete blank lines around point	<code>C-x C-o</code>
delete all whitespace around point	<code>M-\</code>
put exactly one space at point	<code>M-SPC</code>
fill paragraph	<code>M-q</code>
fill region	<code>M-g</code>
set fill column	<code>C-x f</code>
set prefix each line starts with	<code>C-x .</code>

Case Change

uppercase word	<code>M-u</code>
lowercase word	<code>M-l</code>
capitalise word	<code>M-c</code>
uppercase region	<code>C-x C-u</code>
lowercase region	<code>C-x C-l</code>
capitalize region	<code>M-x capitalize-region</code>

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	<code>TAB</code>
complete up to one word	<code>SPC</code>
complete and execute	<code>RET</code>
show possible completions	<code>?</code>
abort command	<code>C-g</code>

Type `C-x ESC` to edit and repeat the last command that used the minibuffer. The following keys are then defined.

previous minibuffer command	<code>M-p</code>
next minibuffer command	<code>M-n</code>

GNU Emacs Reference Card

web-mode version

Buffers

select another buffer C-x b
list all buffers C-x C-b
kill a buffer C-x k

Transposing

transpose characters C-t
transpose words M-t
transpose lines C-x C-t
transpose sexps C-M-t

Spelling Check

check spelling of current word M- $\$$
check spelling of all words in region M-x spell-region
check spelling of entire buffer M-x spell-buffer

Regular Expressions

The following have special meaning inside a regular expression.

any single character . (dot)
zero or more repeats *
one or more repeats +
zero or one repeat ?
any character in set [...]
any character not in set [^ ...]
beginning of line \$
end of line \e
quote a special character \c
alternative ("or") \|
grouping \(\ ... \)
nth group \n
beginning of buffer \'
end of buffer \'
word break \b
not beginning or end of word \B
beginning of word \<
end of word \>
any word-syntax character \w
any non-word-syntax character \W
character with syntax c \sc
character with syntax not c \Sc

Registers

copy region to register C-x x
insert register contents C-x g
save point in register C-x /
move point to saved location C-x j

Mode - web-mode

Navigation in web-mode

goto section named (completion) C-c g a
goto section # C-c g s
next section C-c a a
previous section C-c p a
which section C-c a s
goto chapter # C-c g c
next chapter C-c a c
previous chapter C-c p c
which chapter C-c v c
view index C-c v i
next index C-c a i
previous index C-c p i
view section names list C-c v s
next define C-c a d
next use C-c a u
previous define C-c p d
previous use C-c p a

Outline editing

hide body C-c h b
show all C-c s a
next visible heading C-c n h
previous visible heading C-c p h
forward same level C-c f l
backward same level C-c b l
up heading C-c u h
hide this entry C-c h e
show this entry C-c s e
hide subtree C-c h s
show subtree C-c a s
show children C-c c c
hide leaves C-c h l
show branches C-c a b

Miscellaneous

rename section C-c r s
insert index entry C-c i i
view chapter titles list C-c v c
view edited sections #s C-c v o
which edited section C-c w e
count edited sections C-c s e
count sections C-c s s
count chapters C-c s c
delimiter match check C-c d m
kill emacs from web-mode C-x C-c

Changing buffers in web-mode

goto buffer, change file C-c b c
goto buffer, include file C-c b i
goto buffer, web file C-c b w

Change file commands

edit section C-c e s
goto edited section # C-c g e
next edited section C-c n e
previous edited section C-c p e

Keyboard Macros

start defining a keyboard macro C-x (
end keyboard macro definition C-x)
execute last-defined keyboard macro C-x e
append to last keyboard macro C-a C-x (
name last keyboard macro M-x name-last-kbd-macro
insert lisp definition in buffer M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval sexp before point C-x C-e
eval current defun C-M-x
eval region M-x eval-region
eval entire buffer M-x eval-current-buffer
read and eval minibuffer M-ESC
re-execute last minibuffer command C-x ESC
read and eval Emacs Lisp file M-x load-file
load from standard system directory M-x load-library

Simple Customization

Here are some examples of binding global keys in Emacs Lisp. Note that you cannot say "\M-#"; you must say "\e#".

```
(global-set-key "\C-cg" 'goto-line)  
(global-set-key "\e\C-x" 'isearch-backward-regexp)  
(global-set-key "\e#" 'query-replace-regexp)
```

An example of setting a variable in Emacs Lisp:

```
(setq backup-by-copying-when-linked t)
```

Writing Commands

```
(defun (command-name) ((args))  
  "(documentation)"  
  (interactive "(template)")  
  (body))
```

An example:

```
(defun this-line-to-top-of-screen (line)  
  "Reposition line point is on to the top of  
the screen. With ARG, put point on line ARG.  
Negative counts from bottom."  
  (interactive "p")  
  (recenter (if (null line)  
                0  
                (prefix-numeric-value line))))
```

The argument to `interactive` is a string specifying how to get the arguments when the function is called interactively. Type `C-h f interactive` for more information.

Copyright © 1987 Free Software Foundation, Inc.
designed by Stephen Gildea. March 1987 v1.9
for GNU Emacs version 16 on Unix systems

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

For copies of the GNU Emacs manual, write to the Free Software Foundation, Inc., 675 Massachusetts Ave., Cambridge MA 02139.

This memo describes how to write programs in the WEB language; and it also includes the full WEB documentation for WEAVE and TANGLE, the programs that read WEB input and produce T_EX and Pascal output, respectively. The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like T_EX for formatting, and a language like Pascal for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a “web” that is made up of many interconnected pieces. To document such a program, we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by T_EX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by Pascal make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

General rules. A WEB file is a long string of text that has been divided into individual lines. The exact line boundaries are not terribly crucial, and a programmer can pretty much chop up the WEB file in whatever way seems to look best as the file is being edited; but string constants and control texts must end on the same line on which they begin, since this convention helps to keep errors from propagating. The end of a line means the same thing as a blank space.

Two kinds of material go into WEB files: T_EX text and Pascal text. A programmer writing in WEB should be thinking both of the documentation and of the Pascal program that he or she is creating; i.e., the programmer should be instinctively aware of the different actions that WEAVE and TANGLE will perform on the WEB file. T_EX text is essentially copied without change by WEAVE, and it is entirely deleted by TANGLE, since the T_EX text is “pure documentation.” Pascal text, on the other hand, is formatted by WEAVE and it is shuffled around by TANGLE, according to rules that will become clear later. For now the important point to keep in mind is that there are two kinds of text. Writing WEB programs is something like writing T_EX documents, but with an additional “Pascal mode” that is added to T_EX’s horizontal mode, vertical mode, and math mode.

A WEB file is built up from units called *modules* that are more or less self-contained. Each module has three parts:

- 1) A T_EX part, containing explanatory material about what is going on in the module.
- 2) A definition part, containing macro definitions that serve as abbreviations for Pascal constructions that would be less comprehensible if written out in full each time.
- 3) A Pascal part, containing a piece of the program that TANGLE will produce. This Pascal code should ideally be about a dozen lines long, so that it is easily comprehensible as a unit and so that its structure is readily perceived.

The three parts of each module must appear in this order; i.e., the T_EX commentary must come first, then the definitions, and finally the Pascal code. Any of the parts may be empty.

A module begins with the pair of symbols '@_ ' or '@*', where '_' denotes a blank space. A module ends at the beginning of the next module (i.e., at the next '@_ ' or '@*'), or at the end of the file, whichever comes first. The WEB file may also contain material that is not part of any module at all, namely the text (if any) that occurs before the first module. Such text is said to be "in limbo"; it is ignored by TANGLE and copied essentially verbatim by WEAVE, so its function is to provide any additional formatting instructions that may be desired in the TeX output. Indeed, it is customary to begin a WEB file with TeX code in limbo that loads special fonts, defines special macros, changes the page sizes, and/or produces a title page.

Modules are numbered consecutively, starting with 1; these numbers appear at the beginning of each module of the TeX documentation, and they appear as bracketed comments at the beginning of the code generated by that module in the Pascal program.

Fortunately, you never mention these numbers yourself when you are writing in WEB. You just say '@_ ' or '@*' at the beginning of each new module, and the numbers are supplied automatically by WEAVE and TANGLE. As far as you are concerned, a module has a name instead of a number; such a name is specified by writing '@<' followed by TeX text followed by '@>'. When WEAVE outputs a module name, it replaces the '@<' and '@>' by angle brackets and inserts the module number in small type. Thus, when you read the output of WEAVE it is easy to locate any module that is referred to in another module.

For expository purposes, a module name should be a good description of the contents of that module, i.e., it should stand for the abstraction represented by the module; then the module can be "plugged into" one or more other modules so that the unimportant details of its inner workings are suppressed. A module name therefore ought to be long enough to convey the necessary meaning.

We have said that a module begins with '@_ ' or '@*', but we didn't say how it gets divided up into a TeX part, a definition part, and a Pascal part. The definition part begins with the first appearance of '@d' or '@f' in the module, and the Pascal part begins with the first appearance of '@p' or '@c'. The latter option '@c' stands for the beginning of a module name, which is the name of the module itself. An equals sign (=) must follow the '@>' at the end of this module name; you are saying, in effect, that the module name stands for the Pascal text that follows, so you say '(module name) = Pascal text'. Alternatively, if the Pascal part begins with '@p' instead of a module name, the current module is said to be *unnamed*. Note that module names cannot appear in the definition part of a module, because the first '@c' in a module signals the beginning of its Pascal part. Any number of module names might appear in the Pascal part, however, once it has started.

The general idea of TANGLE is to make a Pascal program out of these modules in the following way: First all the Pascal parts of unnamed modules are copied down, in order; this constitutes the initial approximation T_0 to the text of the program. (There should be at least one unnamed module, otherwise there will be no program.) Then all module names that appear in the initial text T_0 are replaced by the Pascal parts of the corresponding modules, and this substitution process continues until no module names remain. Then all defined macros are replaced by their equivalents, according to certain rules that are explained later. The resulting Pascal code is "sanitized" so that it will be acceptable to an average garden-variety Pascal compiler; i.e., lowercase letters are converted to uppercase, long identifiers are chopped, and the lines of the output file are constrained to be at most 72 characters long. All comments will have been removed from this Pascal program except for the module-number comments that point to the source location where each piece of the program text originated in the WEB file.

If the same name has been given to more than one module, the Pascal text for that name is obtained by putting together all of the Pascal parts in the corresponding modules. This feature is useful, for example, in a module named 'Global variables in the outer block', since one can then declare global variables in whatever modules those variables are introduced. When several modules have the same name, WEAVE assigns the first module number as the number corresponding to that name, and it inserts a note at the bottom of that module telling the reader to 'See also sections so-and-so'; this footnote gives the numbers of all the other modules having the same name as the present one. The Pascal text corresponding to a module is usually formatted by WEAVE so that the output has an equivalence sign in place of the equals sign in the WEB file; i.e., the output says '(module name) \equiv Pascal text'. However, in the case of the second and subsequent appearances of a module with the same name, this ' \equiv ' sign is replaced by '+ \equiv ', as an indication that the Pascal text that follows is being appended to the Pascal text of another module.

The general idea of WEAVE is to make a TeX file from the WEB file in the following way: The first line of the

TEX file will be `\input webmac`; this will cause TEX to read in the macros that define WEB's documentation conventions. The next lines of the file will be copied from whatever TEX text is in limbo before the first module. Then comes the output for each module in turn, possibly interspersed with end-of-page marks. Finally, WEAVE will generate a cross-reference index that lists each module number in which each Pascal identifier appears, and it will also generate an alphabetized list of the module names, as well as a table of contents that shows the page and module numbers for each "starred" module.

What is a "starred" module, you ask? A module that begins with `'@*'` instead of `'@_` is slightly special in that it denotes a new major group of modules. The `'@*'` should be followed by the title of this group, followed by a period. Such modules will always start on a new page in the TEX output, and the group title will appear as a running headline on all subsequent pages until the next starred module. The title will also appear in the table of contents, and in boldface type at the beginning of its module. Caution: Do not use TEX control sequences in such titles, unless you know that the webmac macros will do the right thing with them. The reason is that these titles are converted to uppercase when they appear as running heads, and they are converted to boldface when they appear at the beginning of their modules, and they are also written out to a table-of-contents file used for temporary storage while TEX is working; whatever control sequences you use must be meaningful in all three of these modes.

Control codes. We have seen several magic uses of `'@'` signs in WEB files, and it is time to make a systematic study of these special features. A WEB control code is a two-character combination of which the first is `'@'` .

Here is a complete list of the legal control codes. The letters *L*, *T*, *P*, *M*, *C*, and/or *S* following each code indicate whether or not that code is allowable in limbo, in TEX text, in Pascal text, in module names, in comments, and/or in strings. A bar over such a letter means that the control code terminates the present part of the WEB file; for example, \bar{L} means that this control code ends the limbo material before the first module.

- | | |
|--|---|
| <p>@@ [<i>C, L, M, P, S, T</i>] A double @ denotes the single character '@'. This is the only control code that is legal in limbo, in comments, and in strings.</p> <p>@_ [$\bar{L}, \bar{P}, \bar{T}$] This denotes the beginning of a new (unstarred) module (or section). A tab mark or end-of-line (carriage return) is equivalent to a space when it follows an @ sign.</p> <p>@* [$\bar{L}, \bar{P}, \bar{T}$] This denotes the beginning of a new starred module, i.e., a module that begins a new major group (or chapter). The title of the new group should appear after the @*, followed by a period. As explained above, TEX control sequences should be avoided in such titles unless they are quite simple. When WEAVE and TANGLE read a @*, they print an asterisk followed by the current module number, so that the user can see some indication of progress. The very first module should be starred.</p> <p>@p [\bar{P}, \bar{T}] The Pascal part of an unnamed module begins with @p (or @P). This causes TANGLE to append the following Pascal code to the initial program text T_0 as explained above. The WEAVE processor does not cause a '@p' to appear explicitly in the TEX output, so if you</p> | <p>are creating a WEB file based on a TEX-printed WEB documentation you have to remember to insert @p in the appropriate places of the unnamed modules.</p> <p>@< [\bar{P}, \bar{T}] A module name begins with @< followed by TEX text followed by @>; the TEX text should not contain any WEB control sequences except @@, unless these control sequences appear in Pascal text that is delimited by The module name may be abbreviated, after its first appearance in a WEB file, by giving any unique prefix followed by ..., where the three dots immediately precede the closing @>. No module name should be a prefix of another. Module names may not appear in Pascal text that is enclosed in ... , nor may they appear in the definition part of a module (since the appearance of a module name ends the definition part and begins the Pascal part).</p> <p>@- [<i>P, T</i>] The "control text" that follows, up to the next '@>', will be entered into the index together with the identifiers of the Pascal program; this text will appear in roman type. For example, to put the phrase "system dependencies" into the index, you can type</p> |
|--|---|

- '@system dependencies@>' in each module that you want to index as system dependent. A control text, like a string, must end on the same line of the WEB file as it began. Furthermore, no WEB control sequences are allowed in a control text, not even @@. (If you need an @ sign you can get around this restriction by typing '\AT!'.)
- ⓪. [P,T] The "control text" that follows will be entered into the index in typewriter type; see the rules for '@-', which is analogous.
 - ⓪: [P,T] The "control text" that follows will be entered into the index in a format controlled by the TeX macro '\@', which the user should define as desired; see the rules for '@-', which is analogous.
 - ⓪! [P,T] The module number in an index entry will be underlined if '⓪!' immediately precedes the identifier or control text being indexed. This convention is used to distinguish the modules where an identifier is defined, or where it is explained in some special way, from the modules where it is used. A reserved word or an identifier of length one will not be indexed except for underlined entries. An '⓪!' is implicitly inserted by WEAVE just after the reserved words `function`, `procedure`, `program`, and `var`, and just after `⓪d` and `⓪f`. But you should insert your own '⓪!' before the definitions of types, constants, variables, parameters, and components of records and enumerated types that are not covered by this implicit convention, if you want to improve the quality of the index that you get.
 - ⓪? [P,T] This cancels an implicit (or explicit) '⓪!', so that the next index entry will not be underlined.
 - ⓪, [P] This control code inserts a thin space in WEAVE's output; it is ignored by TANGLE. Sometimes you need this extra space if you are using macros in an unusual way, e.g., if two identifiers are adjacent.
 - ⓪/ [P] This control code causes a line break to occur within a Pascal program formatted by WEAVE; it is ignored by TANGLE. Line breaks are chosen automatically by TeX according to a scheme that works 99% of the time, but sometimes you will prefer to force a line break so that the program is segmented according to logical rather than visual criteria. Caution: '@/' should be used only after statements or clauses, not in the middle of an expression; use ⓪! in the middle of expressions, in order to keep WEAVE's parser happy.
 - ⓪! [P] This control code specifies an optional line break in the midst of an expression. For example, if you have a long condition between `if` and `then`, or a long expression on the right-hand side of an assignment statement, you can use '⓪!' to specify breakpoints more logical than the ones that TeX might choose on visual grounds.
 - ⓪# [P] This control code forces a line break, like ⓪/ does, and it also causes a little extra white space to appear between the lines at this break. You might use it, for example, between procedure definitions or between groups of macro definitions that are logically separate but within the same module.
 - ⓪+ [P] This control code cancels a line break that might otherwise be inserted by WEAVE, e.g., before the word 'else', if you want to put a short if-then-else construction on a single line. It is ignored by TANGLE.
 - ⓪; [P] This control code is treated like a semicolon, for formatting purposes, except that it is invisible. You can use it, for example, after a module name when the Pascal text represented by that module name ends with a semicolon.

The last six control codes (namely '@-', '@/', '@!', '@#', '@+', and '@;') have no effect on the Pascal program output by TANGLE; they merely help to improve the readability of the TeX-formatted Pascal that is output by WEAVE, in unusual circumstances. WEAVE's built-in formatting method is fairly good, but it is incapable of handling all possible cases, because it must deal with fragments of text involving macros and module names; these fragments do not necessarily obey Pascal's syntax. Although WEB allows you to override the automatic formatting, your best strategy is not to worry about such things until you have seen what WEAVE produces automatically, since you will probably need to make only a few corrections when you are touching up your

documentation.

Because of the rules by which every module is broken into three parts, the control codes '`@d`', '`@r`', and '`@p`' are not allowed to occur once the Pascal part of a module has begun.

Additional features and caveats.

1. The character pairs '`(*`', '`*`', '`(.`', and '`.`' are converted automatically in Pascal text as though they were '`@{`', '`@}`', '`[`', and '`]`', respectively, except of course in strings. Furthermore in certain installations of WEB that have an extended character set, the characters '`#`', '`≤`', '`≥`', '`+`', '`≠`', '`∧`', '`∨`', '`¬`', and '`ε`' can be used as abbreviations for '`<>`', '`<=`', '`>=`', '`:=`', '`==`', '`and`', '`or`', '`not`', and '`in`', respectively. However, the latter abbreviations are not used in the standard versions of WEAVE.WEB and TANGLE.WEB that are distributed to people who are installing WEB on other computers, and the programs are designed to produce only standard ASCII characters as output if the input consists entirely of ASCII characters.

2. If you have an extended character set, all of the characters listed in Appendix C of *The T_EXbook* can be used in strings. But you should stick to standard ASCII characters if you want to write programs that will be useful to the all the poor souls out there who don't have extended character sets.

3. The T_EX file output by WEAVE is broken into lines having at most 80 characters each. The algorithm that does this line breaking is unaware of T_EX's convention about comments following '%' signs on a line. When T_EX text is being copied, the existing line breaks are copied as well, so there is no problem with '%' signs unless the original WEB file contains a line more than eighty characters long or a line with Pascal text in `|...|` that expands to more than eighty characters long. Such lines should not have '%' signs.

4. Pascal text is translated by a "bottom up" procedure that identifies each token as a "part of speech" and combines parts of speech into larger and larger phrases as much as possible according to a special grammar that is explained in the documentation of WEAVE. It is easy to learn the translation scheme for simple constructions like single identifiers and short expressions, just by looking at a few examples of what WEAVE does, but the general mechanism is somewhat complex because it must handle much more than Pascal itself. Furthermore the output contains embedded codes that cause T_EX to indent and break lines as necessary, depending on the fonts used and the desired page width. For best results it is wise to adhere to the following restrictions:

- a) Comments in Pascal text should appear only after statements or clauses; i.e., after semicolons, after reserved words like `then` and `do`, or before reserved words like `end` and `else`. Otherwise WEAVE's parsing method may well get mixed up.
- b) Don't enclose long Pascal texts in `|...|`, since the indentation and line breaking codes are omitted when the `|...|` text is translated from Pascal to T_EX. Stick to simple expressions or statements.

5. Comments and module names are not permitted in `|...|` text. After a '`|`' signals the change from T_EX text to Pascal text, the next '`|`' that is not part of a string or control text ends the Pascal text.

6. A comment must have properly nested occurrences of left and right braces, otherwise WEAVE and TANGLE will not know where the comment ends. However, the character pairs '`\{`' and '`\}`' do not count as left and right braces in comments, and the character pair '`\\`' does not count as a delimiter that begins Pascal text. (The actual rule is that a character after '`\`' is ignored; hence in '`\\{`' the left brace does count.) At present, TANGLE and WEAVE treat comments in slightly different ways, and it is necessary to satisfy both conventions: TANGLE ignores '`|`' characters entirely, while WEAVE uses them to switch between T_EX text and Pascal text. Therefore, a comment that includes a brace in a string in `|...|`—e.g., '`{ look at this |"{| }`'—will be handled correctly by WEAVE, but TANGLE will think there is an unmatched left brace. In order to satisfy both processors, one can write '`{ look at this \leftbrace }`', after setting up '`\def\leftbrace{|"{|}`'.

7. Reserved words of Pascal must appear entirely in lowercase letters in the WEB file; otherwise their special nature will not be recognized by WEAVE. You could, for example, have a macro named `END` and it would not be confused with Pascal's `end`.


```

%%
%   WEB SYSTEM      : web
%   PROGRAM         : quad_eq.web
%   AUTHOR          : Peter J. Nuernberg [pnuern@@photon]
%   CREATION DATE   : Mon Sep  6 08:51:49 1993
%%

%
%   LIMBO MATERIAL  Last edited by Bart Childs on May 22, 1992.
%

\input limbo.sty
\def\title{{\tt Quadratic Equation}}

% begin Bottom of Contents Page macro
\def\botofcontents{\vskip 0pt plus 1fil minus 1.5in\rm
{\bigskip\parskip6pt plus2pt \parindent20pt

% begin abstract
\vskip0.5in
\noindent{\bf Abstract. }it
% The abstract is put right here!
The quadratic equation

$$ax^2 + bx + c = 0$$

has two roots (notice the  $a$ ):

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

}% end abstract

\vfil
\rightline{based on a program by:}
\rightline{Bart S. Childs}
\rightline{subsequently translated into \PASCAL{} by:}
\rightline{D. Dunn}
\rightline{\today}% today.tex should be preloaded, input it if not
\rightline{\miltme}% time.tex should be preloaded, input it if not
}%
end of Bottom of Contents Page macro

% This ends the limbo material and begins the WEB
%

@* The quadratic equation. This is one of the great little
steps in learning some of the fine points of mathematics.
The quadratic equation is probably most commonly written as

$$ax^2 + bx + c = 0$$

and is well known to have two roots (notice the  $a$ ):

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$


@ The quantity  $b^2 - 4ac$  is called the discriminant.
If it is negative, then we introduce the unsuspecting student
to the world of {\it imaginary} and {\it complex} numbers.

@^discriminant@

@* The program for solving the quadratic equation.
This is a rather straightforward program.
@p
program quadratic;
  @<Type declarations@>
  @<Variable declarations@>
begin
  @<Input parameters@>
  @<Calculate discriminant and solutions@>
  @<Output the solutions@>
end.

@ Since we are potentially handling complex numbers, we should declare

```

an appropriate type.

```
@<Type declarations@>=
  type
    complex = record
      real_part: real;
      imaginary_part: real;
    end; (record)
```

@ The three obvious variables will now be declared. We will use the simple declaration of `|real|` because it is logical.

```
@<Variable declarations@>=
  var
    a, b, c : real;
```

@ The input of the three parameters is easily done using the `\PASCAL() |readln|` statement. However, good programming practice should require that a prompt be issued first.

```
@<Input parameters@>=
  writeln('Enter the values of a, b, and c. ');
  readln(a, b, c);
```

@ The calculation is small, but worthwhile. This is a paper for pedagogical reasons and so we will be a little more detailed.

```
@<Calculate discriminant and solutions@>=
  discriminant := b*b - 4.0*a*c;
  real_part := -b/(2.0*a);
  maybe_part := sqrt(abs(discriminant))/(2.0*a);
```

@ It is rather obvious that we need to declare these variables. The additional two variables representing the two parts of the solution are given somewhat descriptive names.

```
@<Variable declarations@>=
  discriminant, real_part, maybe_part : real;
```

@ We will write assign the solutions to two variables. The discriminant must be checked for sign in order to correctly assign the solutions.

```
@<Calculate discriminant and solutions@>=
  if (discriminant > 0.0) then begin
    x1.real_part := real_part + maybe_part;
    x1.imaginary_part := 0.0;
    x2.real_part := real_part - maybe_part;
    x2.imaginary_part := 0.0;
  end (if)
  else begin
    x1.real_part := real_part;
    x1.imaginary_part := maybe_part;
    x2.real_part := real_part;
    x2.imaginary_part := -maybe_part;
  end; (else)
```

@ Once again, we need to declare these variables.

```
@<Variable declarations@>=
  x1, x2: complex;
```

@ We write the solutions, making sure to include the imaginary part only if it is non-zero.

```
@<Output the solutions@>=
  writeln ('The solutions are:');
  write ('    x1 = ', x1.real_part:5:2);
  if (x1.imaginary_part <> 0.0) then
    write (' + ', x1.imaginary_part:5:2, 'i');
  writeln;
  write ('    x2 = ', x2.real_part:5:2);
  if (x2.imaginary_part <> 0.0) then
    write (' + ', x2.imaginary_part:5:2, 'i');
  writeln;

@* Index.
```

```

(3:)program quadratic;(4:)type complex=record realpart:real;
imaginarypart:real;end;(5){5:)var a,b,c:real;(5){8:)
discriminant,realpart,maybepart:real;(8){10:)x1,x2:complex;(10)
begin(6:)writeln('Enter the values of a, b, and c.');
```

$$\text{discriminant} = b^2 - 4.0 * a * c;$$

```

(7:)discriminant:=b*b-4.0*a*c;realpart:=-b/(2.0*a);
maybepart:=sqrt(abs(discriminant))/(2.0*a);(7){9:)
if(discriminant>0.0)then begin x1.realpart:=realpart+maybepart;
x1.imaginarypart:=0.0;x2.realpart:=realpart-maybepart;
x2.imaginarypart:=0.0;end else begin x1.realpart:=realpart;
x1.imaginarypart:=maybepart;x2.realpart:=realpart;
x2.imaginarypart:=-maybepart;end;(9){11:)writeln('The solutions are:');
write('  x1 = ',x1.realpart:5:2);
if(x1.imaginarypart<>0.0)then write(' + ',x1.imaginarypart:5:2,'i');
writeln;write('  x2 = ',x2.realpart:5:2);
if(x2.imaginarypart<>0.0)then write(' + ',x2.imaginarypart:5:2,'i');
writeln;(11)end.(3)

```

Quadratic Equation

November 7, 1994

	Section	Page
The quadratic equation	1	1
The program for solving the quadratic equation	3	2
Index	12	4

Abstract. *The quadratic equation*

$$ax^2 + bx + c = 0$$

has two roots (notice the \pm):

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

based on a program by:
 Bart S. Childs
 subsequently translated into Pascal by:
 D. Dunn
 November 7, 1994
 16:21

§1 Quadratic Equation

THE QUADRATIC EQUATION 1

1. **The quadratic equation.** This is one of the great little steps in learning some of the fine points of mathematics. The quadratic equation is probably most commonly written as

$$ax^2 + bx + c = 0$$

and is well known to have two roots (notice the \pm):

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2. The quantity $b^2 - 4ac$ is called the discriminant. If it is negative, then we introduce the unsuspecting student to the world of *imaginary* and *complex* numbers.

2 THE PROGRAM FOR SOLVING THE QUADRATIC EQUATION

Quadratic Equation §3

3. The program for solving the quadratic equation. This is a rather straightforward program.

```
program quadratic; (Type declarations 4)(Variable declarations 5)
  begin (Input parameters 6)(Calculate discriminant and solutions 7)(Output the solutions 11)
  end.
```

4. Since we are potentially handling complex numbers, we should declare an appropriate type.

```
(Type declarations 4) ≡
type complex = record real_part: real;
  imaginary_part: real;
end; {record}
```

This code is used in section 3.

5. The three obvious variables will now be declared. We will use the simple declaration of *real* because it is logical.

```
(Variable declarations 5) ≡
var a, b, c: real;
```

See also sections 8 and 10.

This code is used in section 3.

6. The input of the three parameters is easily done using the Pascal *readln* statement. However, good programming practice should require that a prompt be issued first.

```
(Input parameters 6) ≡
writeln('Enter the values of a, b, and c. '); readln(a, b, c);
```

This code is used in section 3.

7. The calculation is small, but worthwhile. This is a paper for pedagogical reasons and so we will be a little more detailed.

```
(Calculate discriminant and solutions 7) ≡
discriminant ← b * b - 4.0 * a * c; real_part ← -b / (2.0 * a);
maybe_part ← sqrt(abs(discriminant)) / (2.0 * a);
```

See also section 9.

This code is used in section 3.

8. It is rather obvious that we need to declare these variables. The additional two variables representing the two parts of the solution are given somewhat descriptive names.

```
(Variable declarations 5) +≡
discriminant, real_part, maybe_part: real;
```

9. We will write assign the solutions to two variables. The discriminant must be checked for sign in order to correctly assign the solutions.

```
(Calculate discriminant and solutions 7) +≡
if (discriminant > 0.0) then
  begin z1.real_part ← real_part + maybe_part; z1.imaginary_part ← 0.0;
    z2.real_part ← real_part - maybe_part; z2.imaginary_part ← 0.0;
  end {if}
else begin z1.real_part ← real_part; z1.imaginary_part ← maybe_part; z2.real_part ← real_part;
  z2.imaginary_part ← -maybe_part;
end; {else}
```

§10 Quadratic Equation

THE PROGRAM FOR SOLVING THE QUADRATIC EQUATION 3

10. Once again, we need to declare these variables.

(Variable declarations 5) \equiv
z1, z2: complex;

11. We write the solutions, making sure to include the imaginary part only if it is non-zero.

(Output the solutions 11) \equiv
writeln('The solutions are:'); write('x1=', z1.real.part : 5 : 2);
if (z1.imaginary-part \neq 0.0) then write(' +', z1.imaginary-part : 5 : 2, 'i');
writeln; write('x2=', z2.real.part : 5 : 2);
if (z2.imaginary-part \neq 0.0) then write(' +', z2.imaginary-part : 5 : 2, 'i');
writeln;

This code is used in section 3.

4 INDEX

Quadratic Equation §12

12. Index.*a*: 5.*abs*: 7.*complex*: 4, 10.*discriminant*: 2.*discriminant*: 7, 8, 9.*imaginary.part*: 4, 9, 11.*maybe.part*: 7, 8, 9.*quadratic*: 3.*readln*: 6.*real*: 4, 5, 8.*real.part*: 4, 7, 8, 9, 11.*sqrt*: 7.*write*: 11.*writeln*: 6, 11.*x1*: 9, 10, 11.*x2*: 9, 10, 11.

§12 Quadratic Equation

NAMES OF THE SECTIONS 5

- (Calculate discriminant and solutions 7, 9) Used in section 3.
- (Input parameters 6) Used in section 3.
- (Output the solutions 11) Used in section 3.
- (Type declarations 4) Used in section 3.
- (Variable declarations 5, 8, 10) Used in section 3.

```

%%
%   WEB SYSTEM   : fweb
%   PROGRAM     : coolstuff.web
%   AUTHOR      : Peter Nuernberg [pnuern@photon]
%   CREATION DATE : Tue Oct 26 09:02:20 1993
%%

%
%   LIMBO MATERIAL Last edited by Bart Childs on May 22, 1992.
%
\input limbo.sty
\def\ItemLevelOne{\parindent=20pt
\par
\hangindent \parindent \textindent}
\def\ItemLevelTwo{\parindent=20pt
\par\indent
\hangindent2\parindent \textindent}
\def\ItemLevelThree{\parindent=20pt
\par\indent\indent
\hangindent3\parindent \textindent}
\def\ItemLevelFour{\parindent=20pt
\par\indent\indent\indent
\hangindent4\parindent \textindent}
\def\title{\tt CoolStuff}
% web-mode edits the previous line when creating a new web.
% Make the previous a comment and edit the next if you don't use web-mode.
\def\title{\tt ?? I need a Title ??}
% begin Bottom of Contents Page macro
\def\botofcontents{\vskip 0pt plus 1fil minus 1.5in
{\bigskip\parskip6pt plus2pt \parindent20pt
% begin abstract
\vskip0.5in
\noindent{\bf Abstract. } \it
% The abstract is put right here!
}% end abstract
% BC often puts this in as a comment about pre-release versions ...
\vskip0.5in
%{\vfill\it % comments on anything else ???}
%} % end of comments on anything else
\vfil
\rightline{Pete}
\rightline{\today }
\rightline{\miltme }
}% end of Bottom of Contents Page macro

% This ends the limbo material and begins the WEB
%
% In fweb's you want an AT-c. AT-c++, AT-n, AT-n9, or AT-Lx at this point
% and be sure to replace 'AT-' with the obvious character!!!!

@* Test new macros.

\ItemLevelOne(1.)
Havelock's portrait of Plato's attack upon the poets and Socrates'
tone of voice in carrying on the discussion both lead us to surmise
that he expected the popular reaction to his attack to be hostile.

\ItemLevelOne(2.)
As Havelock writes, 'He thus exhorts us to fight the good fight
against the powers of darkness.'

\ItemLevelTwo(2.1)
There are indeed, indications that the rhapsodes and poets
were highly popular, as we shall see in chapter three.

\ItemLevelThree(2.1.1)
The paradox of this popularity is that Plato's attack
upon the written word, on the other hand, was also a reflection of
popular feeling about the new technology.

```

\ItemLevelFour(2.1.1.1)

First, we find that Aristotle and others apparently accept Plato's understanding of the written word as removed from knowledge.

\ItemLevelTwo(2.2)

Of particular importance in our study is the special relationship between the maker and his work of art as Aristotle conceives it.

@* Index.

```

%%
%   WEB SYSTEM   : fweb
%   PROGRAM     : sample.web
%   AUTHOR      : Peter Nuernberg [pnuern@photon]
%   CREATION DATE : Wed Sep 15 07:45:55 1993
%%

%
%   LIMBO MATERIAL Last edited by Bart Childs on May 22, 1992.
%

#####
%
% The following code will be added automatically to your "limbo
% material" starting tomorrow. For now, if you want to produce a list
% of consecutively numbered items like the ones that appear in
% sections 4 and 6 of this file, type in the following lines:
%
\newcount\ItemCount

\def\BeginItems(
  \bgroup\global\ItemCount=0
  \parindent35pt \parskip1pt plus1pt
  \ifvmode \else\par \fi
)% end definition of BeginItems

\def\EndItems(
  \ifvmode \else\par \fi \egroup
)% end definition of EndItems

\def\numItem(
  \global \advance \ItemCount by 1
  \item{\the\ItemCount .})

%
% OK. Everything else is back to normal. If you don't want to have
% lists, just blow off the above section.
%
#####
\input limbo.sty
\def\title({\tt Sample Web})

% begin Bottom of Contents Page macro
\def\botofcontents{\vskip 0pt plus 1fil minus 1.5in
(\bigskip\parskip6pt plus2pt \parindent20pt

% begin abstract
\vskip0.5in
\noindent{\bf Abstract. } \it
This program uses the formula:

$$\sum_{\text{temp}=2}^z (z-1) \bmod \text{temp} = 0$$

to generate prime numbers.
The notation  $\{expr\}$  is taken from (Concrete Mathematics) by
Graham, Knuth, and Patashnik.
The convention therein established is that  $\{expr\} = 1$  if  $\$expr$  is a
true statement.
Otherwise,  $\{expr\} = 0$ .
)% end abstract

\vfil
\rightline{Peter J. Nuernberg}
\rightline{\today } % today.tex should be preloaded, input it if not
\rightline{\miltime } % time.tex should be preloaded, input it if not

```

}% end of Bottom of Contents Page macro

@*Program Design.

@ Problem Description.

This program will, given a positive integer, output the five smallest prime numbers which are greater than or equal to the given integer.

@^prime number@>

@ Program Inputs.

This program only requires 1 input - a positive integer.

Call this integer \$x\$.

The only condition on \$x\$ is that \$x > 0\$.

@ Algorithm.

The basic algorithm has the following steps.

```
\BeginItems
\numItem()
Get $x$ from the user.
\numItem()
If $x \le 0$, print an error message and exit.
\numItem()
Set $y = x$.
\numItem()
Set $iteration = 1$.
\numItem()
Find $p$, the smallest prime number greater than or equal to $y$.
\numItem()
Output $p$.
\numItem()
Increment $iteration$.
\numItem()
If $iteration > 5$, quit.
\numItem()
Set $y = p + 1$.
\numItem()
Goto step 5.
\EndItems
```

@ Program Outputs.

This program generates 1 output - a prime number.

However, it generates this output 5 times (see above algorithm.)

@^prime number@>

@ Calculations.

Given some number, say \$z\$, the following can be used to determine if \$z\$ is prime:

$$z \text{ is prime: } z \Big| \sum_{temp=2}^{z-1} (z \bmod temp) = 0$$

\indent\quad This can be done by using the following steps:

```
\BeginItems
\numItem()
If $z < 2$, quit and report that $z$ is not prime.
\numItem()
Set $temp = 2$.
\numItem()
If $temp \ge z$, quit and report that $z$ is prime.
\numItem()
```

```

If $z \bmod temp = 0$, quit and report that $z$ is not prime.
\numItem{}
Increment $temp$.
\numItem{}
Goto step 3.
\EndItems

```

```

@^prime number@>
@.mode>

```

@ Testing.

There is only one input to this program, so the testing is straightforward.

The first set of cases will involve "normal" input.

The first normal case will test if the input \$1\$ will produce expected output of \$3\$, \$5\$, \$7\$, \$11\$, and \$13\$.

The second normal case will test if the input \$10\$ will produce the expected output of \$11\$, \$13\$, \$17\$, \$19\$, and \$23\$.

The second set of cases will involve "exceptional" input.

The first exceptional case will test if the input \$-1\$ will produce the expected error message.

The second exceptional case will test if the input "A" will produce a run time error. It is expected that the program will (\bf not) be able to handle non-numeric input.

(\it These exceptional cases point out that the user must be informed that non-positive integer input will cause an error message to be printed and non-numeric input will cause a run-time error.)

```

@^error@>

```

```

@* Index.

```

Sample Web

November 7, 1994

	Section	Page
Program Design	1	1
Index	8	2

Abstract. *This program uses the formula:*

$$\left[\sum_{temp=2}^{z-1} \{ (z \bmod temp) = 0 \} = 0 \right]$$

to generate prime numbers. The notation [expr] is taken from Concrete Mathematics by Graham, Knuth, and Patashnik. The convention therein established is that [expr] = 1 if expr is a true statement. Otherwise, [expr] = 0.

Peter J. Nuernberg
November 7, 1994
16:35

§1 Sample Web

PROGRAM DESIGN 1

1. **Program Design.**
2. **Problem Description.** This program will, given a positive integer, output the five smallest prime numbers which are greater than or equal to the given integer.
3. **Program Inputs.** This program only requires 1 input - a positive integer. Call this integer x . The only condition on x is that $x > 0$.
4. **Algorithm.** The basic algorithm has the following steps.
 1. Get x from the user.
 2. If $x \leq 0$, print an error message and exit.
 3. Set $y = x$.
 4. Set *iteration* = 1.
 5. Find p , the smallest prime number greater than or equal to y .
 6. Output p .
 7. Increment *iteration*.
 8. If *iteration* > 5, quit.
 9. Set $y = p + 1$.
 10. Goto step 5.
5. **Program Outputs.** This program generates 1 output - a prime number. However, it generates this output 5 times (see above algorithm.)
6. **Calculations.** Given some number, say x , the following can be used to determine if x is prime:

$$\left[\sum_{temp=2}^{x-1} [(x \bmod temp) = 0] = 0 \right]$$

This can be done by using the following steps:

1. If $x < 2$, quit and report that x is not prime.
 2. Set $temp = 2$.
 3. If $temp \geq x$, quit and report that x is prime.
 4. If $x \bmod temp = 0$, quit and report that x is not prime.
 5. Increment $temp$.
 6. Goto step 3.
7. **Testing.** There is only one input to this program, so the testing is straightforward. The first set of cases will involve "normal" input. The first normal case will test if the input 1 will produce expected output of 3, 5, 7, 11, and 13. The second normal case will test if the input 10 will produce the expected output of 11, 13, 17, 19, and 23. The second set of cases will involve "exceptional" input. The first exceptional case will test if the input -1 will produce the expected error message. The second exceptional case will test if the input "A" will produce a run time error. It is expected that the program will not be able to handle non-numeric input. These exceptional cases point out that the user must be informed that non-positive integer input will cause an error message to be printed and non-numeric input will cause a run-time error.

2 INDEX

Sample Web §8

8. Index.

error: 7.

mod: 6.

prime number: 2, 5, 6.

```
$$ amt = \cases {  
1000.00 $ if $ GallonsUsed \leq 400000;\cr  
2000.00 $ if $ 400000 \leq GallonsUsed \leq 1000000.\cr  
}  
$$  
  
\settabs 6 \columns  
\+ & \hfill Acct \# & \hfill Code & \hfill Gallons & \hfill Amount Due & \cr  
\+ & \hfill 1234 & \hfill H & \hfill 200.00 & \hfill 5.10 & \cr  
\+ & \hfill 1234 & \hfill H & \hfill 50.00 & \hfill 5.10 & \cr  
\+ & \hfill 1234 & \hfill H & \hfill 1200.00 & \hfill 5.10 & \cr  
  
\bye
```

CPSC 110H
Fall 1993

Design: Due Thursday, 9/16
Program: Due Tuesday, 9/28

PROBLEM:

The manager of the Crowell Carpet Store has asked you to write a program to print customers' bills. The manager has given you the following information:

- a. The store expresses the length and width of a room in terms of feet and tenths of a foot. For example, the length might be reported as 16.7 feet.
- b. The amount of carpet purchased is expressed as square yards.
- c. The store does not sell a fraction of a square yard.
- d. The cost for carpet is expressed as the cost per square yard.
- e. All customers are sold a carpet pad at \$2.25 per square yard.
- f. Sales tax equal to 4 percent is applied to the cost of the carpet and the carpet pad.
- g. The labor cost is \$2.40 per square yard.
- h. Large volume customers may be given a discount. The discount may apply only to the carpet cost (before sales tax), only to the pad cost (before sales tax), only to the labor cost, or to any combination of the three charges.
- i. Each customer is identified by a five-digit number and that number should appear on the bill.

The sample output follows:

Croswell Carpet Store
Invoice

Customer number:	26817
Carpet :	574.20
Pad :	81.00
Labor :	86.40
Subtotal :	741.60
Less discount :	65.52
Subtotal :	676.08
Plus tax :	23.59
Total :	699.67

Write the program and test it for the following three customers.

- a. Mr. Wilson (customer 81429) ordered carpet for his family room, which measures 25 feet long and 18 feet wide. The carpet sells for \$12.95 per square yard and the manager agreed to give him a discount of 8 percent on the carpet and 6 percent on the labor.
- b. Mr. and Mrs. Adams (customer 04246) ordered carpet for their bedroom, which measures 16.5 feet by 15.4 feet. The carpet sells for \$18.90 per square yard and the manager granted a discount of 12 percent of everything.
- c. Ms. Logan (customer 39050) ordered carpet that cost \$8.95 per square yard for her daughter's bedroom. The room measures 13.1 by 12.5 feet. No discounts were given.

Lab 3 - College Station Utilities Billing

CPSC 110

Fall 1993

Design: Due Tuesday, 10/5
 Program: Due Tuesday, 10/12 - additional 5 points
 Due Thursday, 10/14

Note: Extra points for turning an assignment in early will be given
 ONLY if the ENTIRE program works correctly!!

PROBLEM:

You've been hired by College Station Utilities to develop a program they can use to calculate and print bills for water utilities.

ANALYSIS:

The water rates vary depending on whether the bill is for home use, commercial use, or industrial use. A code of H means home use, C means commercial use, and I means industrial use. Any other code should be treated as an error.

For each customer, read the following information from an input file:

Account Number (4-digit integer): columns 1-4
 Code (character) : column 6
 Gallons of water (real) : columns 8-?

For this particular program, you know that the file will contain 15 customers. Therefore, a FOR loop may be used. The water rates are computed as follows:

Code H: \$5.00 plus \$0.0005 per gallon used
 Code C: \$1,000.00 for the first 4 million gallons used and \$0.00025 for each additional gallon
 Code I: \$1,000.00 if usage does not exceed 4 million gallons; \$2,000.00 if usage is more than 4 million gallons but does not exceed 10 million gallons; and \$3,000.00 if usage exceeds 10 million gallons

You should produce a report that looks like the following:

College Station Utilities - Billing

Account Number	Code	Gallons Used	Amount Due
1234	H	200.0	5.10
5678	C	3,000,000.0	1000.00
9012	C	4,500,000.0	1125.00
3847	I	3,500,000.0	1000.00
9832	I	5,000,000.0	2000.00
3892	I	12,000,000.0	3000.00

BILLING REQUIREMENTS:

1. If a code is in error, a message should be displayed and the amount due set to \$0.00. However, you should still print the input information (in the output procedure).
2. Use a CASE statement for distinguishing the code.
3. Use IF-THEN-ELSE to calculate the amount due based on usage.

Lab 4 - Caswell Catering and Convention Service
 CPSC 110
 Fall 1993

Design: Due Thursday, 10/21
 Mr. Caswell has agreed to meet with each of the lab sections on Tuesday, 10/19 to answer any questions you might have. Please come to class prepared to obtain any necessary information.

Program: Due Thursday, 10/28 - additional 5 points
 Due Tuesday, 11/2

Note: Extra points for turning an assignment in early will be given ONLY if the ENTIRE program works correctly!!

PROBLEM:

You've been hired by The Caswell Catering and Convention Service to develop a program they can use to calculate and print customer bills.

ANALYSIS:

The catering rates vary depending on number/type of meals, type of banquet hall used (if any), day on which catering is done, and discount (if any).

- a. The adults may be served Deluxe or Standard meals, dessert included.
- b. Children's meals are priced as a fixed percent of adult meals.
- c. Everyone within a given party must be served the same meal type.
- d. There are five banquet halls. The Caswells are considering increasing the room fees in about six months and this should be taken into account.
- e. A surcharge is added to the total bill for catering done on certain days.
- f. All customers will be charged the same rate for tip and tax.
- g. To induce customers to pay promptly, a discount is offered if payment is made within ten days. This discount depends on the amount of the total bill.
- h. Bills are printed by party's last name.
- i. You should produce a report that itemizes the appropriate information.

BILLING REQUIREMENTS:

1. The customer information will be read from a file.
2. Use a separate procedure for each of the following:
 - a. compute meal cost;
 - b. compute room rate;
 - c. compute surcharge;
 - d. compute discount;
 - e. print a statement.
3. Use functions to compute the tax and tip.
4. You must pass parameters for this program. No procedure may access a global variable.

Lab 5 - The College Station Corner Grocery
COSC 110
Fall 1993

Design: Due Tuesday, 11/9

Program: Due Friday, 11/19 - additional 5 points
Due Tuesday, 11/23

Note: Extra points for turning an assignment in early will be given
ONLY if the ENTIRE program works correctly!!

PROBLEM:

Many supermarkets use computer equipment that allows the checkout clerk to drag an item across a sensor that reads the bar code on the product container. After the computer reads the bar code, the store inventory data base is examined, the item's price and product description are located, inventory is adjusted, and a receipt is printed. Your task is to write a program that simulates this process.

ANALYSIS:

Your program will need to read (and print) the starting inventory information from the data file on disk (GROC1.DAT) into an array of records. The data in the inventory file is written one item per line, beginning with a 2-digit product code, followed by a 30-character product description, its price, and the quantity of that item in stock. Your program will need to copy (and print) the revised version of the inventory to a new data file (GROC.OUT) after all purchases are processed.

Processing customers' orders involves reading a series of product codes representing each person's purchases from a second data file (GROC2.DAT). A zero product code is used to mark the end of each customer order. For each product purchased, the product price and description are printed on the receipt. At the bottom of the receipt, you are to print the total for the goods purchased by the customer.

REQUIREMENTS:

1. The inventory and customer information will be read from a file. The revised inventory must be written to a file.
2. You must pass parameters for this program. No procedure may access a global variable.

Lab 6 - The College Station Corner Grocery
CPSC 110
Fall 1993

Program: Due Tuesday, 12/7

PROBLEM:

This is the same problem as that of Lab 5. You've been hired by College Station Corner Grocery to develop a program they can use to maintain their store inventory.

ANALYSIS:

Same as that of Lab 5.

REQUIREMENTS: You must meet the following requirements:

1. Your main program should consist of a minimum of three procedure calls: initialize (for files), input, and processing. Remember, however, the main program should only have procedure calls. So, NO CODE OTHER THAN CALLS IN THE MAIN PROGRAM.
2. You are required to use the EOF function for reading in the file.
3. You ARE required to pass parameters for this program.

DIFFERENCES: Lab 6 differs from Lab 5 in the following ways:

1. Instead of using arrays, you should use a linked list of information. You should create a record type which contains the information in the input file, as well as a pointer to the next record.
2. You may add information to the list in any manner (i.e., add records to the front of the list, the middle of the list, or the end of the list).

The input file is the same as that of Lab 5. Just use LAB5.DAT.

Save the above program (name the file POINT.PAS) on a 3.5" disk. Turn in the disk AND printout of the program on the due date specified above. The code you turn in should adhere to all applicable style standards.

CPSC 110H
Exam 1
September 24, 1993

Name _____

Indicate whether the following statements are True or False (2 points each).

- ___ 1. The control unit is a part of the main I/O device.
- ___ 2. Because of the difficulty of producing programs as compared with producing equipment, programs are called the hardware and equipment is called the software.
- ___ 3. After a Pascal program is compiled successfully, the source code can be executed directly.
- ___ 4. Files can contain either the data for a program, or the program statements themselves.
- ___ 5. A syntax error in a program is an error that causes the program to produce incorrect output.
- ___ 6. A Pascal standard identifier (such as Real and WriteLn) has a special meaning and should not be redefined.
- ___ 7. Constants are used to hold numeric values that may or may not change during program execution.

Multiple Choice (2 points each):

- ___ 8. Which of the following is NOT a high-level computer language?
 - A) assembler
 - B) COBOL
 - C) BASIC
 - D) Pascal
 - E) all of the above are high-level languages
- ___ 9. Writing a string literal with a format specification causes the string to be
 - A) left-justified in the field, and truncated on the right if the field is too small.
 - B) right-justified in the field, and truncated on the left if the field is too small.
 - C) left-justified in the field, and truncated on the left if the field is too small.
 - D) right-justified in the field, and truncated on the right if the field is too small.
 - E) right-justified in the field, and the field is enlarged if it is too small.
- ___ 10. The value of $3 * 4 \text{ div } (10 \text{ mod } 4) - 18$ is
 - A) undefined
 - B) -12
 - C) -14
 - D) -6
 - E) none of the above

11. (5 points) A WEB file is built from units called sections. Each section can have a definition part. Name (or describe) the other two parts that may be contained in a section.
- a. _____
- b. _____
12. (15 points) Give the GNU emacs key sequence necessary to accomplish each of the following strokes (i.e., don't give the PC compatible keystrokes):
- a. Move to the end of the current line _____
- b. Delete a character forward (under cursor) _____
- c. Save a file on disk (without leaving emacs) _____
- d. Scroll to previous screen _____
- e. Go to the end of your buffer (or file) _____
13. (10 points) Given a WEB file named PAINT.WEB, list and describe the steps necessary to be able to execute the Pascal program PAINT.PAS (i.e., include a narrative which explains the purpose of each step).
14. (5 points) Distinguish between the WEB control codes @* and @ (space). Explain the purpose of each code. Why would you use one or the other?

15. (10 points) What is the output from this program? Put your answers on the lines below with one character on each dash, being sure to include any spaces that would appear in the output.

```
input:      155 13.68 UWYZ 123 ABC
           -9 XXX 12

program Fun;
var
  A : integer; B : real; C : char;
begin
  Read (A, B, C, C, C);
  ReadLn (C, B);
  ReadLn (A);
  B := B + (Abs(Sqr(A)));
  A := 25 div 9 mod 2 + 3 mod 2;
  C := Pred(Pred(C));
  Write ('The value of A is ', A : 5);
  WriteLn ('and B is ', B :5:2);
  WriteLn ('The value of C is ', C);
  Write ('A + B is ', A + B :5:2);
  Write ('and A * B is ', A * B :5:2)
end.
```


16. (20 points) Use your problem-solving skills to state the steps necessary to solve this problem. Your answer should be in paragraph form. No Pascal code!

Ima Aggy is quite a traveller and likes to keep track of statistics regarding her trips. She has hired you to write a program to compute her average speed, miles per gallon, and the average cost per mile for a given trip. You may assume Ima can give you the necessary information to make the calculation. You may also assume Ima fills her car immediately before and after a trip. (Note: the average speed will be low because it can include rests.)

17. (15 points) Write the PASCAL code to solve the problem described in question #16. NOTE: For test taking purposes, it is not necessary to document your programs.

CPSC 110
Exam 2
October 22, 1993

Name _____

(2 pts each) Indicate whether the following statements are True or False.

- ___ 1. In counter-controlled loops, the loop-control variable must be initialized to zero before the loop begins to execute.
- ___ 2. The simplest way to avoid side effects is to use all variables globally, since when all the declarations are centrally located, it is easier to see where individual variables are modified.
- ___ 3. A variable defined in a block can always be referenced in its block and any nested within its block.
- ___ 4. The sentinel value is always the last value added to a sum being accumulated in a sentinel-controlled loop.
- ___ 5. When a program starts, but before any Read statements are executed, EOLN could be true.
- ___ 6. A variable name defined in a block is hidden from being referenced outside the block in which it is defined.
- ___ 7. Give the value of the Boolean expression, assuming that A = True, B = False, and C = False
C or (A and (B or not C))

(2 pts each) Multiple Choice - select the BEST answer.

- ___ 8. Assuming that X is 15 and Y is 25, the value of the expression $X = (Y + X - Y)$ is
 - A. 15
 - B. 25
 - C. True
 - D. False
 - E. The expression is invalid, since the colon in front of the equal sign is missing.
- ___ 9. If a variable declared locally in a procedure has that same name as a global variable, then the compiler will
 - A. issue an error message indicating that a duplicate identifier has been declared.
 - B. issue an error message indicating that the name has multiple meanings.
 - C. interpret occurrences of the name in the procedure as referencing the locally declared variable.
 - D. interpret occurrences of the name in the procedure as referencing the program variable.
 - E. none of the above.

- ___ 10. Consider the following program:

```

program What;
var
  R, X, Y, Z, W : Char;
begin
  ReadLn(X, Y, Z, W);
  if X < Y then R := X
  else R := Y;
  if R > Z then R := Z;
  if R > W then R := W;
  WriteLn(R)
end.

```

What is the program output if the user types `runt` followed by `RETURN` when the program is run?

- A. r
B. u
C. n
- D. t
E. none of the above
- ___ 11. What does this program segment do?

```

X := (N mod 2) = 0;
S := 0;
for i := N downto 1 do
begin
  if X then S := S + i;
  X := not X
end;

```

- A. Add all numbers from 1 to N.
B. Add all the numbers from 1 to N-1.
C. Add the even numbers from 1 to N.
D. Add the odd numbers from 1 to N.
E. None of the above.

Consider the following program segment. Assume that all variables are of type integer.

```

t := 0; p := 0; n := 0; s := 0;
ReadLn(x);
while x <> s do
begin
  if x > 0 then p := p + 1
  else n := n + 1;
  t := t + 1;
  ReadLn(x)
end;

```

- ___ 12. The final contents of variable `t` can best be described as the
- A. count of the number of data items read.
B. count of the number of positive data items read.
C. count of the number of negative data items read.
D. sum of all negative data items read.
E. sum of all data items read.

For the next 3 questions, consider the following program skeleton:

```
program Main;
var
  X, Y, Z : Integer;
procedure Proc1 (X1, Y1 : Integer);
var
  Z1 : Integer;
procedure Proc2 (Y2 : integer);
var
  Z2 : Integer;
begin ... end;
begin ... end;
procedure Proc3 (X3 : Integer);
var
  Z3 : Integer;
begin ... end;
begin ... end.
```

- ___ 13. Proc2 could be called in Proc1 with the parameter Y1.
- ___ 14. Z3 can be accessed by all parts of the program.
- ___ 15. Proc1 could be called in Proc3 with the parameters X and Y.

Short answer:

16. (15 points) Write a function that computes the amount of money you owe for a specified number of parking tickets received at a university. Assume the charge per ticket is \$15 for up to 4 tickets, and a flat fee of \$75 is charged for 5 to 8 tickets. An additional \$15 is charged for each ticket received over 8. Use a CASE statement as the decision statement.

17. (15 points) The real estate tax on resident homes is to be computed as follows:

Assessed Value -----	Computed Tax -----
\$30,000 or less	\$800
$\$30,000 < \text{assessed value} \leq \$50,000$	$\$800 + 1\%$ of assessed value over \$30,000
$\$50,000 < \text{assessed value} \leq \$80,000$	$\$800 + 1.2\%$ of assessed value over \$30,000
$\$80,000 < \text{assessed value} \leq \$120,000$	$\$800 + 1.4\%$ of assessed value over \$30,000
assessed value > \$120,000	$\$800 + 1.5\%$ of assessed value over \$30,000

Write the statement(s) necessary to calculate the real estate tax. Assume that Value and Tax have been declared as Real variables. Use an IF statement and do not make any unnecessary tests. Any value less than \$0 should be flagged as an error.

18. (15 points) What is the output from this program?

```
program Strange;
var
  Who, Where : Integer;
procedure Stranger (var Who : Integer; What : Integer);
begin
  Who := 3 * What;
  What := 2 - Who;
  WriteLn (Who:4, What:4)
end;
procedure EvenStranger (What : Integer; var Where : Integer);
begin
  What := What + 5;
  Stranger (Where, What);
  WriteLn (What:4, Where:4)
end;
begin
  Who := 2;  Where := 4;
  WriteLn (Who:4, Where:4);
  EvenStranger (Who, Where);
  WriteLn (Who:4, Where:4);
  Stranger (Where, Who);
  WriteLn (Who:4, Where:4)
end.
```

19. (25 points) Use your problem-solving skills to state the steps necessary to solve this problem. Your answer should be in "problem-solving" form. No Pascal code!

Whatsamata Mining has hired you to write their payroll program. They give you the following information:

- * employees are paid hourly and will receive overtime for over 40 hours,
- * federal income tax is based on gross salary; however, there is a fixed dollar amount deduction per dependent before the tax is calculated,
- * social security is based on gross,
- * if an employee works in the city office, there is a city tax which is a fixed percentage of gross,
- * if an employee is a union member, dues are a fixed percentage of gross.

The president of the company would like to see information for each employee, as well as totals for the company. She is particularly interested in the number of hours of overtime and the amount of payroll money which is spent on overtime versus regular time.

CPSC 110
Exam 3 - Part I
November 19, 1993

Name _____

(25 points) Your computer science instructor wants you to write a program which will grade the final exam for the course and calculate the final grade for each student.

The final exam consists of 80 true-false and multiple choice questions. The results of the exam have been coded for input to the program. Your instructor would like to see the following for the exam:

- * each student's score and grade;
- * the number of students taking the exam;
- * exam statistics, including low score, high score, median score, and average.

In addition to the final exam, the instructor will provide the necessary scores for each student in order for you to calculate the final grades for the course. The final course grade is made up of 3 tests (at 15% each), a homework grade (which is 35% of the grade), and the final exam. The final grades will be calculated as follows:

- * A - the grade is at least 1 standard deviation above the average;
- * B - the grade is at least 1/2 standard deviation above the average;
- * C - the grade is at most 1/2 standard deviation below the average;
- * D - the grade is at most 1 standard deviation below the average;
- * F - otherwise.

Use your problem-solving skills to state the steps necessary to solve this problem. Your answer should be in "problem-solving" form. NO Pascal code!

Input:

- 1) exam key
- 2) student ID or name
- 3) students answers to final exam
- 4) test scores (3)
- 5) homework grade

Output:

- | | |
|---|--|
| <ol style="list-style-type: none"> 1) Final exam - <ol style="list-style-type: none"> a) student ID or name b) score on final exam c) grade on final exam d) number of students taking exam e) low score on the exam f) high score on the exam g) median score on the exam h) average score on the exam | <ol style="list-style-type: none"> 2) Final grades - <ol style="list-style-type: none"> a) student ID or name b) final grade in course |
|---|--|

Algorithm development:

- 1) Read and store the exam key.
- 2) For each student, determine the final exam grade and collect statistics -
 - a) read student answers;
 - b) compare answers to key and determine the number correct;
 - c) calculate exam score = number correct / number of questions;
 - d) calculate exam grade = if score \geq 90, then 'A'
 if $80 \leq$ score $<$ 90, then 'B'
 if $70 \leq$ score $<$ 80, then 'C'
 if $60 \leq$ score $<$ 70, then 'D'
 if score $<$ 60, then 'F';
 - e) print the student's ID, score, and grade;
 - f) count the student;
 - g) if score $<$ low score, then low score;
 - h) if score $>$ high score, then high score;
 - i) accumulate the score (for average);
 - j) store the score (for median).
- 3) Calculate the exam average = total of scores / number of students.
- 4) Determine the median score by arranging the scores in order and selecting the middle score.
- 5) Print the number of students taking exam, low score, high score, median score, and average.
- 6) For each student, determine the final grade in the course -
 - a) read student test scores and homework grade;
 - b) calculate test score = (test1 + test2 + test3) / 3 * 0.45;
 - c) calculate homework score = homework grade * 0.35;
 - d) calculate final exam score = final exam * 0.20;
 - e) calculate final score = test score + homework score + final exam score;
 - f) count the student;
 - g) accumulate the score (for average).
- 7) Calculate the class average = total of scores / number of students.
- 8) Calculate the standard deviation = formula given in class.
- 9) Calculate final grade = if score \geq 1 * standard deviation + average, then 'A' else
 if score \geq 0.5 * standard deviation + average, then 'B' else
 if score \geq average - 0.5 * standard deviation, then 'C' else
 if score \geq average - 1 * standard deviation, then 'D' else
 if score $<$ average - 1 * standard deviation, then 'F';
- 10) Print the student's ID, score, and grade.

CPSC 110
Exam 3 - Part II
November 22, 1993

Name _____

True/False (2 points each):

Consider the following declarations as you answer questions 1 - 4 and determine whether each assignment statement is legal (true) or not (false).

```

type
  Line = array [1..50] of Char;
  Last = array['A'..'Z'] of Integer;
  Name = string[10];
var
  AList, BList : Line;
  CList : array [1..50] of Char;
  LastName : Last;
  MyName : Name;
  AChar : Char;
  I : Integer;

```

- ___ 1. AList[30] := CList[50];
- ___ 2. AList := CList;
- ___ 3. WriteLn(Name);
- ___ 4. LastName['Q'] := AChar;
- ___ 5. The following double use of the identifier A is legal.

```

type R = record
  A, B : real
end;
var A : integer;

```

- ___ 6. If A is of type array [1..5, 1..10] of Boolean then the expression A[4,2] refers to element A row 2 and column 4.
- ___ 7. A recursive procedure must have only one stopping case, and all other cases must reduce to that stopping case in a finite number of steps to avoid infinite recursion.
- ___ 8. The elements of an array must be accessed one at a time, from the beginning to the end.
- ___ 9. The largest possible dimension of a multidimensional array is three.
- ___ 10. If the expression A[i].B is legal then A must be a one-dimensional array of records.

Multiple Choice (2 points each):

- ___ 11. Problems which lend themselves to recursive solution
 - A) have one or more simple cases that can be used to terminate repetition.
 - B) can be reduced to one or more simpler cases of the same problem.
 - C) can be reduced to one of the simplest non-recursive cases in a finite number of steps.
 - D) all of the above.
 - E) none of the above.

- ___ 12. Which of the following is not a correct use of a field selector, given the declarations below?

```

type
  Disc = record
    title, artist : string[20];
    year : 1900..2000;
    RPM : 16..78
  end;
var
  PhonoRecord : Disc;
  Character : char;

```

- A) PhonoRecord.RPM := 33;
 B) Character := PhonoRecord[4].Artist;
 C) PhonoRecord.Year := 1958;
 D) PhonoRecord.Title[1] := 'Z';
 E) All of the above are legal.

For the next two questions assume the following declarations:

```

type
  Range = 1..Max;
  ArrayType = array {Range} of Integer;
var
  A : ArrayType;
  I, J, Temp : Integer;

```

- ___ 13. What is the effect of the following program segment?

```

Temp := 0;
for I := 2 to Max do
  if A[I] > A[1] then
    Temp := Temp + 1;

```

- A) Reverses the numbers stored in the array.
 B) Puts the largest value in the last array position.
 C) Counts the number of elements of A greater than its first element.
 D) Arranges the elements of the array in increasing order.
 E) None of the above

- ___ 14. What is the effect of the following program segment?

```

for I := 1 to Max - 1 do
  if A[I] > A[I + 1] then
    begin
      Temp := A[I];
      A[I] := A[I + 1];
      A[I + 1] := Temp;
    end;

```

- A) Reverses the numbers stored in the array.
 B) Puts the largest value in the last array position.
 C) Counts the number of elements of A greater than its first element.
 D) Arranges the elements of the array in increasing order.
 E) None of the above.

- ___ 15. Which of the following is syntactically identical to the declaration
 type A = array [1..4, 'a'..'z'] of Integer?
 A) array [1..4] ['a'..'z'] of Integer;
 B) array ['a'..'z', 1..4] of Integer;
 C) array [1..4] of array ['a'..'z'] of Integer;
 D) array ['a'..'z'] of array [1..4] of Integer;
 E) none of the above.

```
const Top = 10;
type namestring = packed array [1..10] of char;
   index = 1..Top;
   recentery = record
     name : namestring;
     quantity : integer;
     price : real
   end;
   entrylist = array [index] of recentery;
var A : entrylist;      i, j : integer;      max : real;
```

- ___ 16. Given the above declarations, Which section of code below will print
 only the name of the item with the highest price in the inventory?

A) for i := 1 to 10 do
 if A[i].price >= A[i+1].price then
 Write(A[i].name);

B) max := A[1].price;
 j := 1;
 for i := 2 to 10 do
 if A[i].price > max then
 max := A[i].price;
 j := i;
 Write(A[j].name);

C) max := A[1].price;
 j := 1;
 for i := 2 to 10 do
 if A[i].price > max then begin
 max := A[i].price;
 j := i
 end;
 Write(A[j].name);

D) i := 1;
 max := A[i].price;
 j := i;
 while i < 10 do begin
 if A[i].price > max then begin
 max := A[i].price;
 j := i
 end
 end;
 Write(A[j].name);

- E) two of the above sections of code will perform correctly.

___ 17. What is written by the following program?

```

program num;
var x : integer;
function ampersand (n : integer) : integer;
begin
  if n = 0 then
    ampersand := 0
  else
    ampersand := (n mod 3) + ampersand (n - 1)
  end;
begin
  x := 8;
  WriteLn(ampersand(x))
end.

```

- A) 9
 B) 2
 C) 7
 D) 8
 E) none of the above

___ 18. Assuming that type Flavor = (Chocolate, Vanilla, ButterBrickle, Spumoni) and that the value of variable F (type Flavor) is Vanilla, what is the value of Ord(Succ(Succ(F)))?

- A. ButterBrickle
 B. Spumoni
 C. 3
 D. 4
 E. undefined

___ 19. Given that X, Y, and Z are records of different types with fields of different names, which expression has the same effect as the one shown below?

```

with X do
  with Y do
    with Z do

```

- A. with [X..Z] do
 B. with X, Y, Z do
 C. with X.Y.Z do
 D. with X or Y or Z do
 E. none of the above

___ 20. Consider the following code

```

F := 1;
for I := 2 to N do
  if A[I] >= A[F] then
    F := I;

```

Which item best describes the operation being performed:

- A. Rearrange the first N components of the array A in descending order.
 B. Rearrange the first N components of the array A in ascending order.
 C. Place the largest component of the array A in position N.
 D. Compute the value of largest component in array A.
 E. Compute the subscript of the last occurrence of the largest of the first N components of the array A.

Short answer:

21. (6 points) Given the following declarations, write the Pascal code necessary to initialize the elements of the array in column-major order to the value ' '. Declare any additional variables you may need.

```
type
  Color = (Red, Orange, Yellow, Green, Blue, Violet);
  Texture = (Satin, Velvet, Coarse, Rough);
  ArrayType = array [Color, Texture] of Char;
var
  SeeFeel : ArrayType;
```

22. (10 points) Write a Pascal procedure with 3 parameters, a 2-dimensional array of reals and two integers. The procedure is to return, via the two integer parameters, the row and column number of the largest real number in the array. You may assume that there are no duplicate values in the array. The following declaration appears in the main program:

```
type MatrixType = array[1..50, 1..20] of Real;
procedure FindBiggest (
```

23. (4 points) Study the following type and variable declarations and then write statements which modify the variable Applicant as described below.

```

type
  string20 = string[20];
  TwoChars = string[2];
  Relation = (Mother, Father, Son, Daughter, Sister, Brother);
  NameRecord = record
    First : string20;
    Middle : char;
    Last : string20
  end;
  AddressRecord = record
    Street, City : string20;
    State : TwoChars;
    Zip : string[5]
  end;
  PersonType = record
    Name : NameRecord;
    Address : AddressRecord
  end;
  DependentRecord = record
    Who : PersonType;
    Age : 1..99;
    Rel : Relation
  end;
  Dossier = record
    Person, Spouse : PersonType;
    NumDependents : 0..10;
    Dependent : array [1..10] of DependentRecord
  end;
var
  Applicant : Dossier;

```

- a) Set the applicant's last name to Smith.
- b) Set the spouse's state to TX.
- c) Set the third dependent's first name to Patrick.
- d) Add one to the second dependent's age.

24. (5 points) Declare a data type which could be used to hold information about a student in a class. The information that your record must hold is student name (15 characters), social security number, three test scores (integers), six lab scores, the final exam score, the average of all the scores, and the overall course grade (A, B, C, D, or F).
25. (5 points) Now declare a data type which contains the above information for the entire class. It must hold class number (an integer), the instructor's name (15 characters), information about 100 students, the overall class average, the highest average in the class, the lowest average in the class, and the median average in the class.
26. (5 points) Declare a type which contains information about all of the classes in the department. It should hold department name, number of faculty, and the above information about all of the classes. Assume that there are 50 classes.

CPSC 1104 - Fall 1993
FINAL EXAM

Name _____

The final exam is worth 200 points. Each of the true/false and multiple choice questions will be weighted 2 points, giving a total of 160 points. The short answer questions will be worth the remaining 40 points.

Indicate whether the following statements are true or false.

- ___ 1. When a program begins to execute, the contents of the memory cells it uses are initially empty.
- ___ 2. Semicolons must be inserted after every program statement occurring between the begin and end statements of the program body.
- ___ 3. Before a new value can be stored in a memory cell, a program must execute a statement to erase its former contents.
- ___ 4. More than one Pascal statement can be placed on a single line.
- ___ 5. Constants can be declared in procedures, but variables must be declared in the main program.
- ___ 6. If A and B are the names of procedures declared in a Pascal program, then the statement sequence `begin A; B; A end` is legal in the program body.
- ___ 7. A nested if statement occurs when the true or false statement of an if statement is itself an if statement.
- ___ 8. If A and B are arrays of the same data type, then the statement `A := B;` copies each element of B to the corresponding element of A.
- ___ 9. After the last statement of a procedure executes, control is transferred to the next declared procedure.
- ___ 10. The `Reset` procedure resets EOF and EOLN to false.
- ___ 11. Files should never be closed explicitly in Turbo Pascal.
- ___ 12. The string `*ABCE+D+*` is a legal postfix expression.
- ___ 13. A variable name defined in a block is hidden from being referenced outside the block in which it is defined.
- ___ 14. A recursive solution to a problem of size N, is always reducible to a problem of size N - 1.
- ___ 15. Blaise Pascal developed the Pascal language.
- ___ 16. When a variable of type `^Real` is created by the `New` procedure, it can thereafter hold a Real number.
- ___ 17. All pointers to a node that is returned to the heap (disposed) are automatically reset to nil.
- ___ 18. If the `Head` pointer to a linked list is passed as a value parameter to a procedure, then a copy of the list is made in the procedure's local data area.
- ___ 19. The condition in the `while` statement is tested at the end of each pass.

- ___ 20. If P, Q, and R are pointer variables, then the statements below interchange the contents of the nodes pointed to by P and Q.
 $P^{\wedge} := Q^{\wedge}; Q^{\wedge} := P^{\wedge};$
- ___ 21. While loops that iterate zero times indicate improper variable initialization or improper formulation of the while condition.
-
- Give the value of the Boolean expression, assuming that A = True, B = False, and C = False.
- ___ 22. not (B and C) or A
- ___ 23. C or A and B and C
- ___ 24. not B or (C and not A)
-
- ___ 25. If X = 3, then the Boolean condition X > 2 and Y < 4 is syntactically correct.
- ___ 26. If you define a variable X to be of type 1..8, and if the number 9 was typed in response to the statement Read (X), then the computer will prompt the user to enter a new value for X.
- ___ 27. The expression True < False is false.
- ___ 28. Each of the types Integer, Real, Char, and Boolean is an ordinal type, since each has a numerical representation in the computer's memory.
- ___ 29. Enumerated type variables can not be read or written directly.
- ___ 30. The ordinal value of the third value listed in the declaration of an enumerated type is 3.
- ___ 31. If X is a variable of the enumerated type (Apples, Bananas, Oranges) and has the value Bananas, then the expression X < Oranges has value _____.
- ___ 32. A variable declared of enumerated type (0,1,2,3,4,5,6,7,8,9) may be used to store an Integer in the range 0 to 9.
- ___ 33. The Read procedure, with numeric variables, skips all non-numeric characters until it comes to a '+', '-', digit, or <eof>.
- ___ 34. Procedures can be called several times from several different places in the program, since the program keeps track of where control is to return after a procedure finishes its last step.
- ___ 35. The field selector in a statement consists of the record variable name, followed by a period, followed by a field identifier from that record type.
- ___ 36. This assignment statement is valid if all variables are defined as type integer:
 $A = B + C.$
- ___ 37. The sentinel value is always the last value added to a sum being accumulated in a sentinel-controlled loop.
- ___ 38. An enumerated type variable can be a loop control variable for a counter-controlled loop.
- ___ 39. The Ord function may be applied to enumerated type variables.
- ___ 40. The condition in the while statement is tested at the end of each pass.
- ___ 41. The New statement must be executed before a pointer variable can be used.

- ___ 42. Statements in a high-level language are converted to statements in machine language by a loader.
- ___ 43. A syntax error in a program is an error that causes the program to produce incorrect output.
- ___ 44. If the value of X is 735, the statement WriteLn (X :2) will not cause the number to be displayed incorrectly.
- ___ 45. The insertion of comments in a program neither causes the program to run more slowly, nor causes the object code to take up more space.
- ___ 46. In an arithmetic expression, without any parentheses, the computer always performs the leftmost operation first.

Multiple Choice - Select the Best Answer.

- ___ 47. Which of the following is not an advantage of a high-level language?
 - a. It is easier to use than machine language.
 - b. Its statements resemble English.
 - c. It is portable.
 - d. Memory can be referenced symbolically.
 - e. It is easy for the machine to understand.
- ___ 48. Which of the following statements calls procedure XYZ?
 - a. Call XYZ;
 - b. procedure XYZ;
 - c. XYZ;
 - d. program ABC (XYZ);
 - e. none of these
- ___ 49. If a computer's collating sequence places upper-case letters in consecutive ordinal positions, then Ord('F') - Ord(Succ('A')) =
 - a. Not defined
 - b. 5
 - c. 4
 - d. 'D'
 - e. 'B'
- ___ 50. Which does not represent a Pascal reserved word?
 - a. WriteLn
 - b. program
 - c. var
 - d. begin
 - e. All are reserved words.
- ___ 51. The effect of the following program segment can best be described as


```
if X > Y then Z := X;
if X = Y then Z := 0;
if X < Y then Z := Y;
```

 - a. The smaller of X and Y is stored in Z.
 - b. The larger of X and Y is stored in Z.
 - c. The larger of X and Y is stored in Z, unless X and Y are equal, in which case Z is assigned zero.
 - d. The larger of X and Y is stored in Z, unless X and Y are not equal, in which case Z is assigned zero.
 - e. None of the above.
- ___ 52. A unique value that can be used to terminate a loop containing a ReadLn statement is called a
 - a. terminal value.
 - b. sentinel value.
 - c. loop control variable.
 - d. input value.
 - e. loop termination value.

- ___ 53. Which of the following types cannot be the type of a counter variable in a for loop?
- Integer
 - Real
 - Char
 - Enumerated
 - Boolean
- ___ 54. If N is an Integer variable and $N \geq 10$, then the expression whose value is N's tens digit (for example, 3 if $N = 436$) is
- $N \text{ div } 10 \text{ mod } 10$
 - $N - 10$
 - $N - 9$
 - $N \text{ mod } 100$
 - $N \text{ mod } 10 \text{ mod } 10$
- ___ 55. What does this program do?
- ```
S := 0; I := 1;
repeat
 S := S + I;
 I := I + 1
until I >= N;
```
- Add all numbers from 1 to N.
  - Add all the numbers from 1 to N-1.
  - Add the even numbers from 1 to N.
  - Add the odd numbers from 1 to N.
  - None of the above.
- \_\_\_ 56. Which of the following variable names are invalid?
- Write
  - abcD3
  - var
  - John's
  - 5count
  - abc#e
  - crazy8s
- i, ii, iii, v, vi
  - i, ii, iv, v, vi
  - iii, v
  - i, iii, iv, v
  - iii, iv, v, vi
- \_\_\_ 57. The if statement
- ```
if 13 < 12 then WriteLn ('never')
else WriteLn ('always')
```
- Writes 'never'.
 - Writes 'always'.
 - Won't compile since 13 is not less than 12.
 - Causes a run-time error since 13 is not less than 12.
 - Prints nothing since 13 is not less than 12.
- ___ 58. Which of the following types cannot be the element type of an array?
- Integer
 - Real
 - Enumerated
 - Boolean
 - None of the above

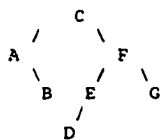
59. What would be printed by the following program? (The symbol '#' stands for one blank character.)

```

program Formats:
var
  A, B :Real ;
begin
  A := 37.56
  B := 101.117;
  Write ('Is it', A :6:1, B :9:4);
  WriteLn ('?')
end

```

- a. Is#it##37.6#101.1170?
 b. Is#it:###37.6##101.1170?
 c. Is#it##37.5#101.1170?
 d. Is#it##37.6#101.117?
 e. None of the above
60. Which of the types listed below can be returned as the value of a user-defined function?
 a. Integer
 b. Real
 c. Char
 d. Enumerated
 e. All of these
61. Given the binary tree below, what is the order in which the nodes would be visited during a preorder traversal?



- a. A B C D E F G
 b. B A D E G F C
 c. C A B F E G D
 d. C A B F E D G
 e. None of the above.
62. Which of the following can not be an element type of a two-dimensional array?
 a. another two-dimensional array
 b. a hierarchical record
 c. a Real
 d. an enumerated type
 e. None of the above

```

Consider the following program:
program HowAboutThis (input,output);
type
  X = (A, B, C, D, Z);
var
  R,S : X;
  T   : Integer;
begin
  T := 0;   R := A;   S := C;
  while R <> S do
    begin
      R := Succ(R);
      T := Pred(T)
    end; {while}
  WriteLn (T);      ( question 63 )
  WriteLn (Ord(S)) ( question 64 )
end.

```

- ___ 63. According to the standard rules for Pascal, what value will be printed by WriteLn (T) ?
- | | |
|-------|------|
| a. 0 | d. 3 |
| b. 5 | e. 2 |
| c. -2 | |
- ___ 64. According to the standard rules for Pascal, what value will be printed by WriteLn (Ord(S)) ?
- | | |
|------|------|
| a. 2 | d. 0 |
| b. A | e. 3 |
| c. 1 | |

Assume the following declarations:

```

type
  Range = 1..Max;
  ArrayType = array [Range] of Integer;
var
  A : ArrayType;
  I, J, Temp : Integer;

```

- ___ 65. What is the effect of the following program segment?

```

Temp := 0;
for I := 2 to Max do
  if A[I] > A[1] then
    Temp := Temp + 1;

```

- | | |
|----|--|
| a. | Reverses the numbers stored in the array. |
| b. | Puts the largest value in the last array position. |
| c. | Counts the number of elements of A greater than its first element. |
| d. | Arranges the elements of the array in increasing order. |
| e. | None of the above |

___ 66. What is the value of the assignment `A := F(3,3,4)` ?

```
function F (A,B,C : Integer) : Integer;
var
  I, J, K, L : Integer;
begin (F)
  L := 0;
  for I := 1 to A do
    begin
      for J := B downto 4 do
        L := L + J;
        for K := 3 to C do
          L := L + k
        end; ( for I )
      end;
    end;
  F := L
end; ( F )
```

a. 28
b. 30
c. 21

d. 33
e. None of these

___ 67. The emacs key binding for scroll to next screen is:
a. C-n
b. M-n
c. C-v
d. C-p
e. none of these

___ 68. The emacs key binding for search forward is:
a. M-s
b. C-g
c. C-r
d. C-s
e. none of these

Consider the following declarations:

```
type
  Fruit = (Apple, Orange, Kiwi, Banana);

var
  X : array [1..5, Fruit] of integer;
```

___ 69. The statement `X[4, Orange] := 12` causes
a. the value 12 to be placed in the second column of the fourth row of X.
b. the value of 12 to be placed in the second row of the fourth column of X.
c. a compilation error.
d. run-time error, after compiling correctly.
e. none of the above.

___ 70. The person who is known as the first programmer is:
a. Charles Babbage
b. Herman Hollerith
c. Ada Augusta Lovelace
d. Blaise Pascal
e. Niklaus Wirth

Use the function below:

```
begin (Wow)
  if M < 10 then
    if N < 10 then Wow := M + N
    else Wow := Wow(M, N - 2) + N
  else Wow := Wow(M - 1, N) + M
end; (Wow)
```

___ 71. What is the value of `Wow(12,15)`?
a. 84
b. 90
c. 75
d. 18
e. none of these

- ___ 72. The terminating condition is:
 a. M and N equal to 10
 b. M and N less than 10
 c. M or N less than 10
 d. M less than 10
 e. N less than 10
- ___ 73. Which of the following types cannot be the subscript type of an array?
 a. Integer
 b. Real
 c. Enumerated
 d. Boolean
 e. None of the above
- ___ 74. The person who was responsible for the Difference Engine and the Analytical Engine is:
 a. Charles Babbage
 b. Herman Hollerith
 c. Ada Augusta Lovelace
 d. Blaise Pascal
 e. Niklaus Wirth
- ___ 75. Which statement is true about recursion?
 a. Recursion is more efficient than iteration.
 b. Recursion requires less overhead than iteration.
 c. Recursion can specify more natural solutions for some problems than iteration.
 d. Recursive solutions are more complex than iterative solutions.
 e. None of the above.
- ___ 76. For what exact range of values of variable X does the following code segment print 'C'?
- ```

if X <= 200 then
 if X < 100 then
 if X <= 0 then WriteLn ('A')
 else WriteLn ('B')
 else WriteLn ('C')
else WriteLn ('D')
```
- a.  $0 < X < 100$   
 b.  $X \leq 0$   
 c.  $100 \leq X \leq 200$   
 d.  $X > 200$   
 e.  $100 < X \leq 200$
- \_\_\_ 77. The function below can best be described as
- ```

function What (Head : Ptr; X : Integer) : Ptr;
var
  Temp : Ptr;
begin (What)
  What := nil;
  while Head <> nil do
  begin
    if Head^.Data = X then
      What := Head;
      Head := Head^.Link
    end (while)
  end; (What)
```
- a. returning all of the addresses in the list where X was found.
 b. returning a pointer to the first occurrence of X in the list, and nil if X does not occur.
 c. returning a pointer to the last occurrence of X in the list, and nil if X does not occur.
 d. counting the number of occurrences of X in the list.
 e. none of the above.

78. Which of the following types cannot be the element type of a one-dimensional array?
- | | |
|---------------|----------------------|
| a. Integer | d. Boolean |
| b. Real | e. None of the above |
| c. Enumerated | |

79. The correct statements to insert a node containing 3 at the front of the linked list Head is:

- a. `New (Temp);`
`Temp^.Data := 3;`
`Temp^.Link := Head;`
`Head := Temp;`
- b. `New (Temp);`
`Temp^.Data := 3;`
`Head := Temp;`
`Temp^.Link := Head;`
- c. `New (Temp);`
`Temp^.Data := 3;`
`Temp^.Link := Head^.Link;`
`Head := Temp;`
- d. `New (Temp);`
`Temp^.Data := 3;`
`Temp^.Link := Head^.Link;`
`Head^.Link := Temp;`
- e. none of the above

Given the declaration

```

type
  Date = record
    Month : 1..12;
    Day : 1..31;
  end; (Date)
  Address = record
    Street, City, : string[30];
  end; (Address)
  EmpRec = record
    StartDate : Date;
    Home : Address;
    Salary : Real;
  end; (EmpRec)
var
  Employee : EmpRec;

```

80. Which of the following is not equivalent to the others?
- a. `WriteLn (Employee.Home.Street, Employee.StartDate.Month);`
- b. `with Employee do`
`WriteLn (Home.Street, StartDate.Month);`
- c. `with Employee, Home, StartDate do`
`WriteLn (Street, Month);`
- d. `with Employee do`
`with StartDate do`
`with Home do`
`WriteLn (Street, Address);`
- e. All of the above are equivalent.

81. (12 points) The administration at a college keeps a master file, on disk, of all current and former students. There is a record for each individual, but different information is kept depending on whether the person is current or former. The information for a current student is name, social security number, school address, home address, GPA, and number of library books currently checked out. The information for a former student is name, social security number, address, and total amount of money contributed to the college. After each graduation, the administration of the college wants to update the master file with a transaction file, also kept on disk, that contains a record for each student that just graduated. Assume that both files are kept sorted by social security number. Use your problem-solving skills to design a program to perform the necessary update.

82. (4 points) Consider this program:

```
1. program test;
2.   type
3.     intarray = array[1..10] of integer;
4.   var
5.     A : intarray;
6.     i : integer;
7.   procedure change (...);
8.   begin
9.     i := 5;
10.    A[i] := 20
11.  end;
12. begin
13.   for i := 1 to 10 do A[i] := i;
14.   i := 3;
15.   change(A,i);
16.   write(A[i])
17. end.
```

What is the output of this program when line 7 is equal to each of the following:

- a) `procedure change(var A : intarray; var i : integer);`
- b) `procedure change(var A : intarray; i : integer);`
- c) `procedure change(A : intarray; var i : integer);`
- d) `procedure change(A : intarray; i : integer);`

83. (4 points) Consider this program:

```

1. program test;
2.   type
3.     ptr = ^node;
4.     node = record
5.       data : integer;
6.       next : ptr
7.     end;
8.   var
9.     p : ptr;
10.    n : node;
11.  procedure change(...);
12.  begin
13.    p^.data := 3;
14.    new(p);
15.    n.next := p
16.  end;
17. begin
18.   n.data := 1;
19.   new(p);
20.   p^.data := 2;
21.   p^.next := nil;
22.   n.next := p.
23.   change(n,p)
24. end.
```

This problem asks you to draw diagrams indicating space allocation. Here are the rules: Draw a box for each location that is currently allocated to the program and label it with its name or names. Put the current value of the location inside the box, using "?" for uninitialized and arrows for pointers.

Draw a space allocation diagram for the point in the program immediately after line 23, for each of the following choices for line 11. It will probably be helpful if you start by drawing the diagram for the point just before line 23. You may draw the diagrams in the blank space above, to the right of the program. (Be sure to label them.)

- a) procedure change(var n : node; var p : ptr);
- b) procedure change(var n : node; p : ptr);
- c) procedure change(n : node; var p : ptr);
- d) procedure change(n : node; p : ptr);

84. (5 points) Give the `web-mode` key bindings for the following commands.
- a) `goto section #`
 - b) `which section`
 - c) `goto next section`
 - d) `count sections`
 - e) `kill emacs from web-mode`
85. (15 points) For this part of the test, use the attached copy of `primes.web` to answer the following questions.
- a) Give the name of section 8.
 - b) In what section(s) is "Other constants of the program" defined? In what sections is it used?
 - c) In what section does the chapter "Generating the primes" begin?
 - d) Chapter 5 begins in section _____ and goes through section _____.
 - e) In what modules is the variable "page-offset" used, and what type is it?
 - f) What sections are used in section 14?
 - g) How many sections does `primes.web` have?
 - h) How many chapters does `primes.web` have?
 - i) In which section(s) does the programmer discuss the format of the output?
 - j) What section(s) need(s) to be modified if you want to print the first 500 prime numbers?

PRINES**(March 31, 1986)**

	Section	Page
Printing primes: An example of WEB	1	1
Plan of the program	3	2
The output phase	5	3
Generating the primes	11	5
The inner loop	22	7
Index	27	8

§1 PRIMES

PRINTING PRIMES: AN EXAMPLE OF WEB 1

1. Printing primes: An example of WEB. The following program is essentially the same as Edsger Dijkstra's "first example of step-wise program composition," found on pages 26-39 of his *Notes on Structured Programming*,² but it has been translated into the WEB language.

[Double brackets will be used in what follows to enclose comments relating to WEB itself, because the chief purpose of this program is to introduce the reader to the WEB style of documentation. WEB programs are always broken into small sections, each of which has a serial number; the present section is number 1.]

Dijkstra's program prints a table of the first thousand prime numbers. We shall begin as he did, by reducing the entire program to its top-level description. [Every section in a WEB program begins with optional *commentary* about that section, and ends with optional *program text* for the section. For example, you are now reading part of the commentary in §1, and the program text for §1 immediately follows the present paragraph. Program texts are specifications of PASCAL programs; they either use PASCAL language directly, or they use angle brackets to represent PASCAL code that appears in other sections. For example, the angle-bracket notation '(Program to print ... numbers 2)' is WEB's way of saying the following: "The PASCAL text to be inserted here is called 'Program to print ... numbers', and you can find out all about it by looking at section 2." One of the main characteristics of WEB is that different parts of the program are usually abbreviated, by giving them such an informal top-level description.]

(Program to print the first thousand prime numbers 2)

2. This program has no input, because we want to keep it rather simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the *output* file.

Since there is no input, we declare the value $m = 1000$ as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m , as long as the computer's finite limitations are not exceeded.

[The program text below specifies the "expanded meaning" of '(Program to print ... numbers 2)'; notice that it involves the top-level descriptions of three other sections. When those top-level descriptions are replaced by their expanded meanings, a syntactically correct PASCAL program will be obtained.]

```
(Program to print the first thousand prime numbers 2) ≡
program print_primes(output);
  const m = 1000; (Other constants of the program 5)
  var (Variables of the program 4)
  begin (Print the first m prime numbers 3);
  end.
```

This code is used in section 1.

2 PLAN OF THE PROGRAM

PRINES §3

3. **Plan of the program.** We shall proceed to fill out the rest of the program by making whatever decisions seem easiest at each step; the idea will be to strive for simplicity first and efficiency later, in order to see where this leads us. The final program may not be optimum, but we want it to be reliable, well motivated, and reasonably fast.

Let us decide at this point to maintain a table that includes all of the prime numbers that will be generated, and to separate the generation problem from the printing problem.

[The WEB description you are reading once again follows a pattern that will soon be familiar: A typical section begins with comments and ends with program text. The comments motivate and explain noteworthy features of the program text.]

```
(Print the first  $m$  prime numbers 3) ≡
  (Fill table  $p$  with the first  $m$  prime numbers 11);
  (Print table  $p$  8)
```

This code is used in section 2.

4. How should table p be represented? Two possibilities suggest themselves: We could construct a sufficiently large array of boolean values in which the k th entry is *true* if and only if the number k is prime; or we could build an array of integers in which the k th entry is the k th prime number. Let us choose the latter alternative, by introducing an integer array called $p[1..m]$.

In the documentation below, the notation ' $p[k]$ ' will refer to the k th element of array p , while ' p_k ' will refer to the k th prime number. If the program is correct, $p[k]$ will either be equal to p_k or it will not yet have been assigned any value.

[Incidentally, our program will eventually make use of several more variables as we refine the data structures. All of the sections where variables are declared will be called '(Variables of the program 4)'; the number '4' in this name refers to the present section, which is the first section to specify the expanded meaning of '(Variables of the program)'. The note 'See also ...' refers to all of the other sections that have the same top-level description. The expanded meaning of '(Variables of the program 4)' consists of all the program texts for this name, not just the text found in §4.]

```
(Variables of the program 4) ≡
 $p$ : array [1.. $m$ ] of integer; { the first  $m$  prime numbers, in increasing order }
```

See also sections 7, 12, 15, 17, 23, and 24.

This code is used in section 2.

5. **The output phase.** Let's work on the second part of the program first. It's not as interesting as the problem of computing prime numbers; but the job of printing must be done sooner or later, and we might as well do it sooner, since it will be good to have it done. [And it is easier to learn WEB when reading a program that has comparatively few distracting complications.]

Since p is simply an array of integers, there is little difficulty in printing the output, except that we need to decide upon a suitable output format. Let us print the table on separate pages, with rr rows and cc columns per page, where every column is ww character positions wide. In this case we shall choose $rr = 50$, $cc = 4$, and $ww = 10$, so that the first 1000 primes will appear on five pages. The program will not assume that m is an exact multiple of $rr \cdot cc$.

(Other constants of the program s) \equiv

```
rr = 50; { this many rows will be on each page in the output }
cc = 4;  { this many columns will be on each page in the output }
ww = 10; { this many character positions will be used in each column }
```

See also section 19.

This code is used in section 2.

6. In order to keep this program reasonably free of notations that are uniquely PASCALesque, [and in order to illustrate more of the facilities of WEB,] a few macro definitions for low-level output instructions are introduced here. All of the output-oriented commands in the remainder of the program will be stated in terms of five simple primitives called *print.string*, *print.integer*, *print.entry*, *new.line*, and *new.page*.

[Sections of a WEB program are allowed to contain *macro definitions* between the opening comments and the closing program text. The general format for each section is actually tripartite: commentary, then definitions, then program. Any of the three parts may be absent; for example, the present section contains no program text.]

[Simple macros simply substitute a bit of PASCAL code for an identifier. Parametric macros are similar, but they also substitute an argument wherever '#' occurs in the macro definition. The first three macro definitions here are parametric; the other two are simple.]

```
define print_string(#)  $\equiv$  write(#) { put a given string into the output file }
define print_integer(#)  $\equiv$  write(# : 1) { put a given integer into the output file, in decimal
notation, using only as many digit positions as necessary }
define print_entry(#)  $\equiv$  write(# : ww)
{ like print_integer, but ww character positions are filled, inserting blanks at the left }
define new_line  $\equiv$  write_ln { advance to a new line in the output file }
define new_page  $\equiv$  page { advance to a new page in the output file }
```

7. Several variables are needed to govern the output process. When we begin to print a new page, the variable *page.number* will be the ordinal number of that page, and *page.offset* will be such that $p[\text{page.offset}]$ is the first prime to be printed. Similarly, $p[\text{row.offset}]$ will be the first prime in a given row.

[Notice the notation '+ \equiv ' below; this indicates that the present section has the same name as a previous section, so the program text will be appended to some text that was previously specified.]

(Variables of the program s) + \equiv

```
page_number: integer; { one more than the number of pages printed so far }
page_offset: integer; { index into p for the first entry on the current page }
row_offset: integer; { index into p for the first entry in the current row }
c: 0 .. cc; { runs through the columns in a row }
```

4 THE OUTPUT PHASE

PRIMES §8

8. Now that appropriate auxiliary variables have been introduced, the process of outputting table p almost writes itself.

```
(Print table  $p$ ) ≡
begin page.number ← 1; page.offset ← 1;
while page.offset ≤  $m$  do
begin (Output a page of answers 9);
page.number ← page.number + 1; page.offset ← page.offset +  $rr * cc$ ;
end;
end
```

This code is used in section 3.

9. A simple heading is printed at the top of each page.

```
(Output a page of answers 9) ≡
begin print.string("The First"); print.integer( $m$ );
print.string(" Prime Numbers -- Page "); print.integer(page.number); new.line; new.line;
{ there's a blank line after the heading }
for row.offset ← page.offset to page.offset +  $rr - 1$  do (Output a line of answers 10);
new.page;
end
```

This code is used in section 8.

10. The first row will contain

$$p[1], p[1 + rr], p[1 + 2 * rr], \dots;$$

a similar pattern holds for each value of the row_offset .

```
(Output a line of answers 10) ≡
begin for  $c \leftarrow 0$  to  $cc - 1$  do
if  $row\_offset + c * rr \leq m$  then print.entry( $p[row\_offset + c * rr]$ );
new.line;
end
```

This code is used in section 9.

§11 PRIMES

GENERATING THE PRIMES 5

11. **Generating the primes.** The remaining task is to fill table p with the correct numbers. Let us do this by generating its entries one at a time: Assuming that we have computed all primes that are j or less, we will advance j to the next suitable value, and continue doing this until the table is completely full.

The program includes a provision to initialize the variables in certain data structures that will be introduced later.

```
(Fill table  $p$  with the first  $m$  prime numbers 11)  $\equiv$ 
  (Initialize the data structures 16);
  while  $k < m$  do
    begin (Increase  $j$  until it is the next prime number 14);
       $k \leftarrow k + 1$ ;  $p[k] \leftarrow j$ ;
    end
```

This code is used in section 3.

12. We need to declare the two variables j and k that were just introduced.

```
(Variables of the program 4)  $\equiv$ 
 $j$ : integer; { all primes  $\leq j$  are in table  $p$  }
 $k$ : 0.. $m$ ; { this many primes are in table  $p$  }
```

13. So far we haven't needed to confront the issue of what a prime number is. But everything else has been taken care of, so we must delve into a bit of number theory now.

By definition, a number is called prime if it is an integer greater than 1 that is not evenly divisible by any smaller prime number. Stating this another way, the integer $j > 1$ is not prime if and only if there exists a prime number $p_n < j$ such that j is a multiple of p_n .

Therefore the section of the program that is called '(Increase j until it is the next prime number)' could be coded very simply: 'repeat $j \leftarrow j + 1$; (Give to j .prime the meaning: j is a prime number); until j .prime'. And to compute the boolean value j .prime, the following would suffice: ' j .prime \leftarrow true; for $n \leftarrow 1$ to k do (If $p[n]$ divides j , set j .prime \leftarrow false)'

14. However, it is possible to obtain a much more efficient algorithm by using more facts of number theory. In the first place, we can speed things up a bit by recognizing that $p_1 = 2$ and that all subsequent primes are odd; therefore we can let j run through odd values only. Our program now takes the following form:

```
(Increase  $j$  until it is the next prime number 14)  $\equiv$ 
  repeat  $j \leftarrow j + 2$ ; (Update variables that depend on  $j$  20);
    (Give to  $j$ .prime the meaning:  $j$  is a prime number 22);
  until  $j$ .prime
```

This code is used in section 11.

15. The repeat loop in the previous section introduces a boolean variable j .prime, so that it will not be necessary to resort to a goto statement. (We are following Dijkstra,² not Knuth.³)

```
(Variables of the program 4)  $\equiv$ 
 $j$ .prime: boolean; { is  $j$  a prime number? }
```

16. In order to make the odd-even trick work, we must of course initialize the variables j , k , and $p[1]$ as follows.

```
(Initialize the data structures 16)  $\equiv$ 
   $j \leftarrow 1$ ;  $k \leftarrow 1$ ;  $p[1] \leftarrow 2$ ;
```

See also section 18.

This code is used in section 11.

17. Now we can apply more number theory in order to obtain further economies. If j is not prime, its smallest prime factor p_n will be \sqrt{j} or less. Thus if we know a number ord such that

$$p[ord]^2 > j,$$

and if j is odd, we need only test for divisors in the set $\{p[2], \dots, p[ord-1]\}$. This is much faster than testing divisibility by $\{p[2], \dots, p[k]\}$, since ord tends to be much smaller than k . (Indeed, when k is large, the celebrated "prime number theorem" implies that the value of ord will be approximately $2\sqrt{k/\ln k}$.)

Let us therefore introduce ord into the data structure. A moment's thought makes it clear that ord changes in a simple way when j increases, and that another variable $square$ facilitates the updating process.

(Variables of the program 4) \equiv

ord : 2 .. ord_{max} ; { the smallest index ≥ 2 such that $p_{ord}^2 > j$ }

$square$: integer; { $square = p_{ord}^2$ }

18. (Initialize the data structures 16) \equiv

$ord \leftarrow 2$; $square \leftarrow 9$;

19. The value of ord will never get larger than a certain value ord_{max} , which must be chosen sufficiently large. It turns out that ord never exceeds 30 when $m = 1000$.

(Other constants of the program 5) \equiv

$ord_{max} = 30$; { $p_{ord_{max}}^2$ must exceed p_m }

20. When j has been increased by 2, we must increase ord by unity when $j = p_{ord}^2$, i.e., when $j = square$.

(Update variables that depend on j 20) \equiv

if $j = square$ then

begin $ord \leftarrow ord + 1$; (Update variables that depend on ord 21);

end

This code is used in section 14.

21. At this point in the program, ord has just been increased by unity, and we want to set $square := p_{ord}^2$. A surprisingly subtle point arises here: How do we know that p_{ord} has already been computed, i.e., that $ord \leq k$? If there were a gap in the sequence of prime numbers, such that $p_{k+1} > p_k^2$ for some k , then this part of the program would refer to the yet-uncomputed value $p[k+1]$ unless some special test were made.

Fortunately, there are no such gaps. But no simple proof of this fact is known. For example, Euclid's famous demonstration that there are infinitely many prime numbers is strong enough to prove only that $p_{k+1} \leq p_1 \dots p_k + 1$. Advanced books on number theory come to our rescue by showing that much more is true; for example, "Bertrand's postulate" states that $p_{k+1} < 2p_k$ for all k .

(Update variables that depend on ord 21) \equiv

$square \leftarrow p[ord] * p[ord]$; { at this point $ord \leq k$ }

See also section 25.

This code is used in section 20.

§22 PRIMES

THE INNER LOOP 7

22. The inner loop. Our remaining task is to determine whether or not a given integer j is prime. The general outline of this part of the program is quite simple, using the value of ord as described above.

```
(Give to j_prime the meaning: j is a prime number 22) ≡
  n ← 2; j_prime ← true;
  while (n < ord) ∧ j_prime do
    begin (If p[n] is a factor of j, set j_prime ← false 26);
    n ← n + 1;
  end
```

This code is used in section 14.

23. (Variables of the program 4) +≡
 n: 2 .. *ord_max*; { runs from 2 to *ord* when testing divisibility }

24. Let's suppose that division is very slow or nonexistent on our machine. We want to detect nonprime odd numbers, which are odd multiples of the set of primes $\{p_2, \dots, p_{ord}\}$.

Since *ord_max* is small, it is reasonable to maintain an auxiliary table of the smallest odd multiples that haven't already been used to show that some j is nonprime. In other words, our goal is to "knock out" all of the odd multiples of each p_n in the set $\{p_2, \dots, p_{ord}\}$, and one way to do this is to introduce an auxiliary table that serves as a control structure for a set of knock-out procedures that are being simulated in parallel. (The so-called "sieve of Eratosthenes" generates primes by a similar method, but it knocks out the multiples of each prime serially.)

The auxiliary table suggested by these considerations is a *mult* array that satisfies the following invariant condition: For $2 \leq n < ord$, *mult*[n] is an odd multiple of p_n such that $mult[n] < j + 2p_n$.

(Variables of the program 4) +≡
mult: array [2 .. *ord_max*] of integer; { runs through multiples of primes }

25. When *ord* has been increased, we need to initialize a new element of the *mult* array. At this point $j = p[ord - 1]^2$, so there is no need for an elaborate computation.

```
(Update variables that depend on ord 21) +≡
  mult[ord - 1] ← j;
```

26. The remaining task is straightforward, given the data structures already prepared. Let us recapitulate the current situation: The goal is to test whether or not j is divisible by p_n , without actually performing a division. We know that j is odd, and that *mult*[n] is an odd multiple of p_n such that $mult[n] < j + 2p_n$. If $mult[n] < j$, we can increase *mult*[n] by $2p_n$ and the same conditions will hold. On the other hand if $mult[n] \geq j$, the conditions imply that j is divisible by p_n if and only if $j = mult[n]$.

```
(If p[n] is a factor of j, set j_prime ← false 26) ≡
  while mult[n] < j do mult[n] ← mult[n] + p[n] + p[n];
  if mult[n] = j then j_prime ← false
```

This code is used in section 22.

27. Index. Every identifier used in this program is shown here together with a list of the section numbers where that identifier appears. The section number is underlined if the identifier was defined in that section. However, one-letter identifiers are indexed only at their point of definition, since such identifiers tend to appear almost everywhere. [An index like this is prepared automatically by the WEB software, and it is appended to the final section of the program. However, underlining of section numbers is not automatic; the user is supposed to mark identifiers at their point of definition in the WEB source file.]

This index also refers to some of the places where key elements of the program are treated. For example, the entries for 'Output format' and 'Page headings' indicate where details of the output format are discussed. Several other topics that appear in the documentation (e.g., 'Bertrand's postulate') have also been indexed. [Special instructions within a WEB source file can be used to insert essentially anything into the index.]

Bertrand, Joseph, postulate: 21.
boolean: 15.
c: 1.
cc: 5, 7, 8, 10.
Dijkstra, Edsger: 1, 15.
Eratosthenes, sieve of: 24.
false: 13, 26.
integer: 4, 7, 12, 17, 24.
j: 12.
j_prime: 13, 14, 15, 22, 26.
k: 12.
Knuth, Donald E.: 15.
m: 2.
mult: 24, 25, 26.
n: 23.
new_line: 6, 9, 10.
new_page: 6, 9.
ord: 17, 18, 19, 20, 21, 22, 23, 24, 25.
ord_max: 17, 19, 23, 24.
output: 2, 6.
output format: 5, 9.
p: 4.
page: 6.
page headings: 9.
page_number: 7, 8, 9.
page_offset: 7, 8, 9.
prime number, definition of: 13.
print_entry: 6, 10.
print_integer: 6, 9.
print_primes: 2.
print_string: 6, 9.
row_offset: 7, 9, 10.
rr: 5, 8, 9, 10.
square: 17, 18, 20, 21.
true: 4, 13, 22.
WEB: 1.
write: 6.
write_in: 6.
ww: 5, 6.

§27 PRIMES

NAMES OF THE SECTIONS 9

- (Fill table *p* with the first *m* prime numbers 11) Used in section 3.
- (Give to *j.prime* the meaning: *j* is a prime number 22) Used in section 14.
- (If *p[n]* is a factor of *j*, set *j.prime* ← *false* 26) Used in section 22.
- (Increase *j* until it is the next prime number 14) Used in section 11.
- (Initialize the data structures 16, 18) Used in section 11.
- (Other constants of the program 5, 19) Used in section 2.
- (Output a line of answers 10) Used in section 9.
- (Output a page of answers 9) Used in section 8.
- (Print table *p* 8) Used in section 3.
- (Print the first *m* prime numbers 3) Used in section 2.
- (Program to print the first thousand prime numbers 2) Used in section 1.
- (Update variables that depend on *j* 20) Used in section 14.
- (Update variables that depend on *ord* 21, 23) Used in section 20.
- (Variables of the program 4, 7, 12, 15, 17, 23, 24) Used in section 2.

APPENDIX B

OVERALL COURSE STATISTICS

This appendix contains the actual numbers for the distribution tables which were presented in the text.

Table 25 is a summary of the student classification distribution for the CS/1 course for the subject and comparison classes.

Table 25. Student Distribution by Classification (Actual)

Semester	U1	U2	U3	U4	Total
Fall 90-H	28	6	1	1	36
Fall 92-H	29	7	6	0	42
Fall 93-H	26	11	0	1	38

Table 26 is a summary of the student major distribution for the CS/1 course for the subject and comparison classes.

Table 26. Student Distribution by Major (Actual)

Semester	CPSC/CSEN	Other	Total
Fall 90-H	20	16	36
Fall 92-H	25	17	42
Fall 93-H	29	9	38

Table 27 is a summary of the overall grade distribution for the CS/1 course for the subject and comparison classes.

Table 28 is a summary of the computer science major grade distribution for the CS/1 course for the subject and comparison classes.

Table 27. Overall Grade Distribution (Actual)

Semester	A	B	C	D	F	Other	Total
Fall 90-H	7	17	5	2	3	2	36
Fall 92-H	20	8	8	1	2	3	42
Fall 93-H	9	15	8	2	3	1	38

Table 28. Grade Distribution for CPSC/CSEN Majors (Actual)

Semester	A	B	C	D	F	Other	Total
Fall 90-H	4	11	2	1	2	0	20
Fall 92-H	12	4	5	1	2	1	25
Fall 93-H	8	9	7	2	2	1	29

Table 29 is a summary of the non-computer science major grade distribution for the CS/1 course for the subject and comparison classes.

Table 29. Grade Distribution for Other Majors (Actual)

Semester	A	B	C	D	F	Other	Total
Fall 90-H	3	6	3	1	1	2	16
Fall 92-H	8	4	3	0	0	2	17
Fall 93-H	1	6	1	0	1	0	9

Table 30 is a summary of the overall grade distribution for the subsequent CS/2 course for those students in the subject and comparison classes.

Table 31 is a summary of the overall grade distribution for the Data Structures course for those students in the subject and comparison classes.

Table 30. Overall CS/2 Grade Distribution (Actual)

Semester	A	B	C	D	F	Other	Total
Fall 90-H	17	7	1	0	0	0	25
Fall 92-H	19	5	2	0	0	1	27
Fall 93-H	13	10	1	0	1	0	25

Table 31. Overall Data Structures Grade Distribution (Actual)

Semester	A	B	C	D	F	Total
Fall 90-H	4	12	3	0	0	19
Fall 92-H	11	3	5	2	1	22
Fall 93-H	9	6	2	0	0	17

APPENDIX C

INDIVIDUAL COURSE STATISTICS

This appendix consists of the individual statistics for the CS/1 classes upon which much of the validation is based.

The first 14 pages are the information for the students enrolled in the comparison classes. The first set of statistics (8 pages) is for the Fall 1990 honors class. This is followed by the information for the students enrolled in the Fall 1992 honors class (6 pages).

The remaining 8 pages are the information for the subject class which is made up of those students enrolled in the Fall 1993 honors class.

Computer Science 110H - Fall 1990

Class	Major	110 Total Exams Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (69)	Design Problem (0)	Parm. Passing (0)	Linked Lists (0)	web mode (0)	WEB Program (0)
390H-1	U1	CSEL	239.0	88	76	75	73	48				
390H-2	U1	PSYC	203.0	67	76	60	52	42				
390H-3	U4	ENGL	0.0	0.0								
390H-4	U1	CPSL	233.0	81	80	72	72	50				
390H-5	U1	CSEL	221.0	81	71	69	79	53				
390H-6	U1	GEST	238.0	83	81	74	76	56				
390H-7	U1	CPSL	258.0	94	87	77	89	60				
390H-8	U2	GEST	233.0	83	88	62	83	57				
390H-9	U1	CPSL	225.0	78	85	62	81	57				
390H-10	U2	CSEL	237.0	92	78	67	80	52				
390H-11	U2	PETE	263.0	87	86	90	89	62				
390H-12	U2	CHEN	212.0	76	69	67	66	46				
390H-13	U2	APMS	250.0	93	77	80	82	57				
390H-14	U1	BICH	243.0	87	79	77	81	53				
390H-15	U1	GEST	278.0	93	98	87	87	59				
390H-16	U1	MATH	130.0	59	37	34	44	35				
390H-17	U1	CPSL	254.0	93	88	73	79	55				
390H-18	U1	CSEL	260.0	89	89	82	85	55				
390H-19	U1	CPSL	185.0	62	72	51	59	48				
390H-20	U1	PETL	271.0	97	89	85	75	49				
390H-21	U2	CSEL	261.0	98	87	76	78	53				
390H-22	U1	APMS	213.0	79	67	67	57	41				
390H-23	U1	GEST	0.0	0.0								
390H-24	U1	CSEN	243.0	91	81	71	81	55				
390H-25	U1	CPSL	236.0	90	81	65	77	52				
390H-26	U1	MATH	235.0	79	78	78	65	44				
390H-27	U1	CPSL	208.0	84	64	60	64	45				
390H-28	U1	CPSL	210.0	84	73	53	66	46				
390H-29	U1	PETL	237.0	81	85	71	83	58				
390H-30	U1	CPSL	98.0	50	48	48	63	44				
390H-31	U1	CSEL	193.0	66	58	59	65	44				
390H-32	U3	APMS	234.0	86	79	69	82	55				
390H-33	U1	CPSL	221.0	79	75	67	83	57				
390H-34	U1	CPSL	269.0	88	96	85	77	49				
390H-35	U1	CSEL	257.0	91	87	79	82	58				
390H-36	U1	CPSL	256.0	95	76	85	82	58				

Overall Average: 83.06 77.68 70.58 74.39 51.364

Computer Science 110H - Fall 1990

	Class	Major	110 Total Exams	110 Exam Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (69)	Design Problem (0)	Parm. Passing (0)	Linked Lists (0)	web mode (0)	WEB Program (0)
Overall Standard Deviation:					11.06	12.26	11.53	10.81	6.2852					
Comp.Sci. Average:					83.7	77.6	69.89	76.32	52.211					
Comp.Sci. Standard Deviation:					11.82	11.13	9.673	7.901	4.4789					
Non-Major Average:					82.14	77.79	71.5	71.79	50.214					
Non-Major Standard Deviation:					9.797	13.71	13.61	13.37	7.9748					

Computer Science 110H - Fall 1990

	110 Total Programs	110 Program Average	110 Lab 1	110 Design 2	110 Lab 2	110 Design 3	110 Lab 3	110 Design 4	110 Lab 4	110 Design 5	110 Lab 5	110 Design 6	110 Lab 6	110 Design 7
390H-1	661.0	94.4	100	90	98	90	90	94	90	94	90	99	99	
390H-2	470.0	67.1	100	100	95	75	80	80	80	80	80	20	20	
390H-3	0.0	0.0												
390H-4	661.0	94.4	100	100	95	88	90	90	90	90	93	95	95	
390H-5	665.0	95.0	100	100	95	95	95	92	91	92	91	92	92	
390H-6	608.0	86.9	100	100	90	78	94	94	66	80	66	80	80	
390H-7	684.0	97.7	100	100	95	97	97	98	94	100	94	100	100	
390H-8	613.0	87.6	90	96	95	90	90	80	97	65	80	97	65	
390H-9	640.0	91.4	85	100	98	93	93	93	80	91	93	80	91	
390H-10	677.0	96.7	100	100	95	95	95	94	96	97	94	96	97	
390H-11	684.0	97.7	100	100	98	97	97	98	94	97	94	94	97	
390H-12	680.0	97.1	100	100	98	98	97	97	95	92	97	95	92	
390H-13	652.0	93.1	100	100	95	88	95	95	78	96	95	78	96	
390H-14	676.0	96.6	100	100	95	95	95	95	92	99	95	92	99	
390H-15	677.0	96.7	100	100	95	98	95	97	90	97	97	90	97	
390H-16	620.0	88.6	100	98	95	93	93	91	73	70	91	73	70	
390H-17	681.0	97.3	100	100	100	100	100	96	96	89	96	96	89	
390H-18	682.0	97.4	100	100	95	96	96	96	96	99	96	96	99	
390H-19	343.0	49.0	100	98	90	55	90	90	55	90	90	55	90	
390H-20	680.0	97.1	100	100	100	100	100	97	90	97	90	97	97	
390H-21	666.0	95.1	100	100	90	92	92	97	94	93	97	94	93	
390H-22	662.0	94.6	100	100	93	92	92	94	88	95	94	88	95	
390H-23	0.0	0.0												
390H-24	659.0	94.1	100	96	98	98	98	83	95	89	83	95	89	
390H-25	679.0	97.0	100	100	95	95	95	97	97	95	97	97	95	
390H-26	646.0	92.3	100	100	90	87	87	94	88	87	94	88	87	
390H-27	661.0	94.4	100	95	95	90	90	94	91	96	94	91	96	
390H-28	656.0	93.7	95	100	88	93	93	96	96	88	96	96	88	
390H-29	672.0	96.0	95	100	95	98	98	95	92	97	95	92	97	
390H-30	348.0	49.7	100	100	60	88	88	60	88	88	60	88	88	
390H-31	630.0	90.0	100	100	93	92	92	92	92	92	96	53	96	
390H-32	682.0	97.4	100	100	95	94	94	99	99	98	99	96	98	
390H-33	692.0	98.9	100	100	100	100	100	100	100	98	97	97	98	
390H-34	692.0	98.9	100	100	100	100	100	100	97	100	98	97	100	
390H-35	680.0	97.1	100	100	95	95	95	97	95	98	97	95	98	
390H-36	683.0	97.6	100	100	97	99	99	98	97	92	98	97	92	
Overall Average:	98.97	99.21	94.15	91.97	94.13	89.9	90.53							

Overall Average:

Computer Science 110H - Fall 1990

	110 Total Programs	110 Program Average	110 Lab 1	110 Design 2	110 Lab 2	110 Design 3	110 Lab 3	110 Design 4	110 Lab 4	110 Design 5	110 Lab 5	110 Lab 6	110 Lab 7
Overall Standard Deviation:			3.157		2.055		6.656		8.435		4.722	9.923	14.91
Comp.Sci. Average:			99		98.95		93.6		92.4		94.78	91.56	94.83
Comp.Sci. Standard Deviation:			3.391		2.479		8.351		9.281		3.583	10.16	3.833
Non-Major Average:			98.93		99.57		94.93		91.36		93.29	81.36	85
Non-Major Standard Deviation:			2.789		1.116		2.631		7.006		5.762	24.21	20.86

Computer Science 110H - Fall 1990

	Previous 110 Course Grade(s)	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Course Points	120 Semester	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
390H-1	B	3.00		A	4.00	91A-201		0.167	0.212	0.212	1.000
390H-2	D	1.00									
390H-3	Q										
390H-4	B	3.00		B	3.00	91A-202		-0.733	-0.788	-0.788	0.000
390H-5	B	3.00		A	4.00	91A-201		0.167	0.212	0.212	1.000
390H-6	B	3.00		B	3.00	91C-502		0.067	-0.229	-0.229	0.000
390H-7	A	4.00		A	4.00	91A-202		0.267	0.212	0.212	0.000
390H-8	B	3.00		B	3.00	91A-201		-0.833	-0.788	-0.788	0.000
390H-9	B	3.00		A	4.00	91A-202		0.267	0.212	0.212	1.000
390H-10	B	3.00		B	3.00	91A-202		-0.733	-0.788	-0.788	0.000
390H-11	A	4.00		A	4.00	91A-201		0.167	0.212	0.212	0.000
390H-12	C	2.00									
390H-13	B	3.00		A	4.00	91A-202		0.267	0.212	0.212	1.000
390H-14	B	3.00									
390H-15	A	4.00									
390H-16	F	0.00									
390H-17	B	3.00		A	4.00	91A-201		0.167	0.212	0.212	1.000
390H-18	A	4.00		A	4.00	91A-202		0.267	0.212	0.212	0.000
390H-19	F	0.00									
390H-20	A	4.00		A	4.00	93A-515		0.571	1.126	1.126	0.000
390H-21	B	3.00		A	4.00	91A-202		0.267	0.212	0.212	1.000
390H-22	C	2.00		B	3.00	91A-507		0.417	0.07	-0.013	1.000
390H-23	Q										
390H-24	B	3.00		C	2.00	91A-505		-0.941	-0.93	-1.013	-1.000
390H-25	B	3.00		A	4.00	91A-202		0.267	0.212	0.212	1.000
390H-26	C	2.00									
390H-27	C	2.00		A	4.00	91A-201		0.167	0.212	0.212	2.000
390H-28	C	2.00		A	4.00	91A-202		0.267	0.212	0.212	2.000
390H-29	B	3.00		A	4.00	91A-201		0.167	0.212	0.212	1.000
390H-30	F	0.00									
390H-31	D	1.00	A-91C	B	3.00	92A-510		0.154	0.071	0.071	2.000
390H-32	B	3.00									
390H-33	B	3.00		B	3.00	91A-201		-0.833	-0.788	-0.788	0.000
390H-34	A	4.00		A	4.00	91A-202		0.267	0.212	0.212	0.000
390H-35	B	3.00		A	4.00	91A-201		0.167	0.212	0.212	1.000
390H-36	A	4.00		A	4.00	91A-201		0.167	0.212	0.212	0.000
Overall Average:		2.676			3.64			0.02432	0.01392	0.00728	0.6

Computer Science 110H - Fall 1990

	Previous 110 Grade(s)	110 Course Grade	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Course Points	120 Semester	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
Overall Standard Deviation:			1.13			0.557			0.431491	0.464465	0.471307	0.748331
Comp.Sci. Average:			2.7			3.667			-0.01194	-0.02594	-0.03056	0.666667
Comp.Sci. Standard Deviation:			1.145			0.577			0.430855	0.428494	0.438527	0.816497
Non-Major Average:			2.643			2.5			0.0823	0.0815	0.0732	0.3
Non-Major Standard Deviation:			1.109			1.688			0.354632	0.44884	0.449742	0.458258

Computer Science 110H - Fall 1990

	210 Course Grade	210 Course Points	Increase In Grade	GPA After 210
390H-1	B	3.00	0.000	3.000
390H-2				
390H-3				
390H-4	C	2.00	-1.000	2.214
390H-5	B	3.00	0.000	2.543
390H-6				
390H-7	A	4.00	0.000	3.404
390H-8	B	3.00	0.000	2.676
390H-9	A	4.00	1.000	3.125
390H-10	A	4.00	1.000	3.438
390H-11	B	3.00	-1.000	3.674
390H-12				
390H-13				
390H-14				
390H-15				
390H-16				
390H-17	B	3.00	0.000	3.417
390H-18	C	2.00	-2.000	3.163
390H-19				
390H-20				
390H-21	B	3.00	0.000	3.079
390H-22				
390H-23				
390H-24				
390H-25	B	3.00	0.000	3.545
390H-26				
390H-27	B	3.00	1.000	2.905
390H-28	B	3.00	1.000	3.094
390H-29	B	3.00	0.000	3.596
390H-30				
390H-31	C	2.00	1.000	2.639
390H-32				
390H-33	B	3.00	0.000	2.067
390H-34	A	4.00	0.000	3.500
390H-35				
390H-36	B	3.00	-1.000	3.800
Overall Average:		3.052632	0	3.099

Computer Science 110H - Fall 1990

	210 Course Grade	210 Course Points	Increase In Grade	GPA After 210
Overall Standard Deviation:		0.604691	0.794719	0.478
Comp.Sci. Average:		3.0625	0.0625	3.058
Comp.Sci. Standard Deviation:		0.658478	0.826797	0.471
Non-Major Average:		1	-0.11111	1.105
Non-Major Standard Deviation:		1.414214	0.31427	1.585

Computer Science 110H - Fall 1992

	Class	Major	110 Total Exams Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (0)	Design Problem (32)	Param. Passing (16)	Linked Lists (16)	web mode (0)	WEB Program (0)
392H-1	U1	PSYC					127		22	16	11		
392H-2	U1	CPEL					144		32	16	9		
392H-3	U1	CPEL					77		16	16	0		
392H-4	U2	CPEL					156		30	16	16		
392H-5	U3	CPSL					103		28	16	7		
392H-6	U2	ELEL					167		16	16	10		
392H-7	U1	CPEL					152		32	16	14		
392H-8	U1	POLL					134		27	16	9		
392H-9	U1	CPEL					169		28	16	12		
392H-10	U3	CHEM					146		27	16	13		
392H-11	U3	CVEN					155		31	12	9		
392H-12	U1	CPEL											
392H-13	U1	GEST											
392H-14	U1	GEST											
392H-15	U3	CPEL											
392H-16	U3	APMS											
392H-17	U2	PHYS					183		30	12	13		
392H-18	U1	CPSL					146		24	16	11		
392H-19	U1	CPSL					179		30	16	16		
392H-20	U1	CPEL					181		32	16	15		
392H-21	U1	GEST					111		20	12	7		
392H-22	U1	CPSL					184		28	16	16		
392H-23	U3	INST					177		32	16	13		
392H-24	U1	CPSL					101		22	16	14		
392H-25	U1	CPEL					142		22	16	2		
392H-26	U1	CPSL					123		26	12	14		
392H-27	U1	CPEL					61		20	16	0		
392H-28	U1	CPSL					93		24	12	10		
392H-29	U1	CPSL					156		27	16	11		
392H-30	U1	CPSL					125		22	16	7		
392H-31	U1	GEST					150		32	16	5		
392H-32	U1	CPSL					154		22	16	16		
392H-33	U1	CPSL					148		32	16	11		
392H-34	U1	MEEL					181		22	16	16		
392H-35	U2	MATH					165		32	16	13		
392H-36	U1	CPSL					182		22	16	13		
392H-37	U2	GEST					126		22	16	9		
392H-38	U2	MATH					171		26	16	15		

Computer Science 110H - Fall 1992

	Class	Major	110 Total Exams	110 Exam Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (0)	Design Problem (32)	Parm. Passing (16)	Linked Lists (16)	web mode (0)	WEB Program (0)
392H-39	U1	GEST						150		28	16	8		
392H-40	U1	CPSL						176		29	16	14		
392H-41	U2	MEEL						193		32	16	15		
392H-42	U1	CPSL						152		27	16	15		
Overall Average:								147		26.703	15.459	10.972		
Overall Standard Deviation:								31.14		4.2609	1.3675	4.3172		
Comp.Sci. Average:								138.9		25.636	15.636	10.333		
Comp.Sci. Standard Deviation:								34.14		4.1181	1.1499	4.8041		
Non-Major Average:								159		28.267	15.2	11.867		
Non-Major Standard Deviation:								21		3.9744	1.6	3.324		

Computer Science 110H - Fall 1992

	Previous 110 Course Grade(s)	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Course Points	120 Semester	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
392H-1	Q										
392H-2	B	3.00		B	3.00	93A-505		0.2	0.126	0.126	0.000
392H-3	F	0.00									
392H-4	A	4.00		A	4.00	93A-203		0.25	0.205	0.205	0.000
392H-5	D	1.00	A-94A	C	2.00	92C-503		-1	-1.123	-1.123	1.000
392H-6	C	2.00									
392H-7	C	2.00		A	4.00	93A-203		0.25	0.205	0.205	2.000
392H-8	B	3.00		B	3.00	93A-201		-0.813	-0.795	-0.795	0.000
392H-9	B	3.00									
392H-10	A	4.00		A	4.00	93C-508		1.056	0.879	0.879	0.000
392H-11	A	4.00									
392H-12	A	4.00									
392H-13	WP										
392H-14	B	3.00		Q		93A-506					
392H-15	A	4.00		B	3.00	93A-202		-0.818	-0.795	-0.795	-1.000
392H-16	Q										
392H-17	A	4.00		A	4.00	93A-203		0.25	0.205	0.205	0.000
392H-18	B	3.00		A	4.00	93A-202		0.182	0.205	0.205	1.000
392H-19	A	4.00		A	4.00	93A-201		0.187	0.205	0.205	0.000
392H-20	A	4.00		A	4.00	93A-201		0.187	0.205	0.205	0.000
392H-21	C	2.00		C	2.00	93A-501		-0.167	-0.874	-0.874	0.000
392H-22	A	4.00		A	4.00	93A-203		0.25	0.205	0.205	0.000
392H-23	A	4.00									
392H-24	C	2.00		B	3.00	93A-202		-0.818	-0.795	-0.795	1.000
392H-25	A	4.00		A	4.00	93A-202		0.182	0.205	0.205	0.000
392H-26	B	3.00		A	4.00	93A-201		0.187	0.205	0.205	1.000
392H-27	F	0.00									
392H-28	C	2.00		A	4.00	93C-512		0.429	0.879	0.879	2.000
392H-29	A	4.00		B	3.00	93A-203		-0.75	-0.795	-0.795	-1.000
392H-30	C	2.00									
392H-31	C	2.00									
392H-32	C	2.00		A	4.00	93A-201		0.187	0.205	0.205	2.000
392H-33	A	4.00		A	4.00	93A-202		0.182	0.205	0.205	0.000
392H-34	A	4.00		A	4.00	93A-203		0.25	0.205	0.205	0.000
392H-35	A	4.00		A	4.00	93A-202		0.182	0.205	0.205	0.000
392H-36	A	4.00									
392H-37	B	3.00									
392H-38	A	4.00		A	4.00	93A-202		0.182	0.205	0.205	0.000

Computer Science 110H - Fall 1992

	Previous 110 Grade(s)	110 Course Grade	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Course Points	120 Semester	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
392H-39		B	3.00									
392H-40		A	4.00		A	4.00	93A-203		0.25	0.205	0.205	0.000
392H-41		A	4.00		A	4.00	93A-201		0.187	0.205	0.205	0.000
392H-42		A	4.00		A	4.00	93A-202		0.182	0.205	0.205	0.000
Overall Average:			3.103			3.654			0.032538	0.007385	0.007385	0.307692
Overall Standard Deviation:			1.128			0.617			0.466156	0.512433	0.512433	0.773067
Comp.Sci. Average:			3.045			3.5			0.039	0.0418	0.0418	0.45
Comp.Sci. Standard Deviation:			1.147			0.975			0.48599	0.509299	0.509299	0.804674
Non-Major Average:			3.333			2			0.080364	0.013727	0.013727	0
Non-Major Standard Deviation:			0.789			1.907			0.129005	0.297249	0.297249	0

Computer Science 110H - Fall 1992

	210 Course Grade	210 Course Points	Increase In Grade	GPA After 210
392H-1				
392H-2	A	4.00	1.000	2.644
392H-3				
392H-4	B	3.00	-1.000	3.355
392H-5	C	2.00	1.000	1.375
392H-6				
392H-7	A	4.00	2.000	3.000
392H-8	B	3.00	0.000	2.289
392H-9				
392H-10	A	4.00	0.000	3.541
392H-11				
392H-12	A	4.00	0.000	3.780
392H-13				
392H-14				
392H-15	D	1.00	-3.000	3.120
392H-16				
392H-17				
392H-18	A	4.00	1.000	2.384
392H-19	C	2.00	-2.000	2.259
392H-20	B	3.00	-1.000	2.923
392H-21	F	0.00	-2.000	2.000
392H-22	A	4.00	0.000	3.508
392H-23				
392H-24				
392H-25	C	2.00	-2.000	2.600
392H-26	A	4.00	1.000	3.488
392H-27				
392H-28	D	1.00	-1.000	2.276
392H-29	C	2.00	-2.000	2.517
392H-30				
392H-31				
392H-32	C	2.00	0.000	1.697
392H-33	A	4.00	0.000	3.316
392H-34	A	4.00	0.000	3.409
392H-35				
392H-36				
392H-37				
392H-38				

Computer Science 110H - Fall 1992

	210 Course Grade	210 Course Points	Increase in Grade	GPA After 210
392H-39				
392H-40				
392H-41	A	4.00	0.000	3.652
392H-42	A	4.00	0.000	3.875
Overall Average:		2.954545	-0.36364	2.863
Overall Standard Deviation:		1.223901	1.226431	0.689
Comp.Sci. Average:		2.736842	-0.15789	2.475
Comp.Sci. Standard Deviation:		1.331485	1.088851	1.061
Non-Major Average:		1.333333	-0.22222	1.427
Non-Major Standard Deviation:		1.885618	0.628539	1.664

Computer Science 110H - Fall 1993

	Class	Major	110 Total Exams Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (80)	Design Problem (12)	Parm. Passing (4)	Linked Lists (4)	web mode (5)	WEB Program (15)
393H-1	U1	CPSL	184.0	68	50	66	124	50	10	3	0	0	10
393H-2	U1	CPSL	247.0	85	76	86	131	73	12	4	2	2	13
393H-3	U1	GEST	232.0	69	82	81	120	69	9	4	2	4	13
393H-4	U2	CPSL	250.0	85	81	84	167	65	12	4	4	3	13
393H-5	U1	CPSL	248.0	92	73	83	149	62	7	4	0	1	13
393H-6	U2	CPSL	233.0	88	82	63	136	56	9	4	1	0	10
393H-7	U1	CPEL	106.0	45	33	28	70	29	2	1	0	0	9
393H-8	U2	CPSL	60.0	60									
393H-9	U1	CPSL	250.0	91	83	76	154	63	7	4	0	4	13
393H-10	U1	CPSL	246.0	87	84	75	161	64	12	4	3	3	11
393H-11	U1	CPSL	238.0	79	75	84	188	76	12	4	3	4	13
393H-12	U1	CPSL	194.0	79	66	49	128	51	8	4	0	4	10
393H-13	U2	ELEL	228.0	86	73	69	168	69	12	4	1	4	9
393H-14	U1	CPSL	245.0	85	84	76	133	56	2	4	3	0	12
393H-15	U1	CPSL	236.0	83	76	77	143	55	12	4	10	4	12
393H-16	U1	CPSL	226.0	72	72	82	166	69	9	4	4	4	11
393H-17	U1	CPSL	263.0	85	89	89	183	74	12	4	2	5	12
393H-18	U2	CPSL	259.0	85	83	91	183	73	12	4	4	5	12
393H-19	U2	CPSL	242.0	79	81	82	163	66	9	4	2	4	12
393H-20	U2	CPSL	257.0	90	78	89	154	62	7	4	3	5	11
393H-21	U1	CPSL	241.0	88	79	74	141	55	12	4	0	2	13
393H-22	U1	CPSL	168.0	63	61	44	128	53	7	4	0	3	12
393H-23	U1	CPSL	260.0	79	95	86	182	73	12	4	3	4	13
393H-24	U2	BIMS	220.0	77	65	78	148	60	12	4	4	4	8
393H-25	U1	CPSL	210.0	62	68	80	158	62	8	4	2	5	13
393H-26	U1	CPSL	269.0	92	88	89	182	75	8	4	4	5	11
393H-27	U4	ENDS	221.0	67	84	70	144	57	12	4	1	4	9
393H-28	U2	MATH	238.0	79	83	76	156	63	8	4	2	4	12
393H-29	U1	CPSL	221.0	76	82	63	172	71	7	4	4	3	12
393H-30	U2	GEST	187.0	70	59	58	119	51	2	0	0	3	10
393H-31	U1	BUAD	216.0	75	70	71	113	44	9	3	0	4	9
393H-32	U2	BICH	235.0	70	83	82	182	74	10	4	4	4	12
393H-33	U1	CPSL	240.0	82	74	84	147	61	8	3	0	1	13
393H-34	U1	CPSL	223.0	88	52	83	135	55	9	4	0	3	9
393H-35	U1	GEST	221.0	74	72	75	146	57	10	4	2	4	12
393H-36	U1	CPSL	255.0	81	84	90	157	63	12	4	3	4	12
393H-37	U1	CPSL	259.0	90	84	85	181	73	12	3	3	3	13
393H-38	U1	CPSL	209.0	80	58	71	141	58	7	4	4	5	8

Computer Science 110H - Fall 1993

Class	Major	110 Total Exams	110 Exam Average	110 Exam 1	110 Exam 2	110 Exam 3	110 Final Exam	Multiple Choice (80)	Design Problem (12)	Par. Passing (4)	Linked Lists (4)	web mode (5)	WEB Program (15)
Overall Average:				78.58	74.65	75.38	150	61.811	9.1892	3.5946	1.9444	3.3243	11.27
Overall Standard Deviation				10.23	12.31	13.35	24.42	9.8168	2.8649	1.0256	2.0131	1.4342	1.6216
Comp.Sci. Average:				79.97	74.68	76.04	152	62.25	9.1429	3.6429	2.1481	3.1429	11.536
Comp.Sci. Standard Deviation:				10.91	13.31	14.78	24.85	10.041	2.8374	0.934	2.1723	1.5972	1.4996
Non-Major Average:				74.11	74.56	73.33	144	60.444	9.3333	3.4444	1.3333	3.8889	10.444
Non-Major Standard Deviation:				5.626	8.5	6.96	21.97	8.9457	2.9439	1.2571	1.2472	0.3143	1.7069

Computer Science 110H - Fall 1993

	110 Lab 1		110 Design 2		110 Lab 2		110 Design 3		110 Lab 3		110 Design 4		110 Lab 4		110 Design 5		110 Lab 5		110 Lab 6		110 Lab 7	
	Total Programs	Program Average	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design	Lab	Design
393H-1	580.0	72.5	100	42	98	39	81	45	81	39	45	75	40	40	40	40	40	40	40	20	20	
393H-2	786.0	98.3	100	45	95	44	105	50	105	44	50	98	46	98	46	98	98	98	98	105	105	
393H-3	792.0	99.0	100	50	100	50	105	50	105	50	49	85	50	93	50	93	93	93	110	110	110	
393H-4	723.0	90.4	100	42	100	49	94	45	94	45	45	55	47	92	47	92	92	92	99	99	99	
393H-5	562.0	70.3	100	45	100	47	85	45	85	45	45	70	30	30	30	30	30	30	10	10	10	
393H-6	726.0	90.8	100	37	98	40	90	40	90	40	41	96	30	95	30	95	95	95	99	99	99	
393H-7	305.0	38.1	70	35	0	5	5	5	5	5	30	15	45	55	45	55	55	45	45	45	45	
393H-8	335.0	41.9	85	30	40	46	90	44	90	44	44	80	44	101	44	101	65	65	65	65	65	
393H-9	682.0	85.3	100	45	96	46	60	45	60	45	45	45	45	0	45	45	0	10	10	10	10	
393H-10	498.0	62.3	100	35	85	44	95	39	95	39	39	104	49	98	49	98	110	110	110	110	110	
393H-11	804.0	100.5	100	47	97	97	105	49	105	49	42	40	35	20	20	20	20	20	5	5	5	
393H-12	372.0	46.5	90	25	88	35	80	35	80	35	44	85	45	86	45	86	70	70	70	70	70	
393H-13	695.0	86.9	75	42	98	45	105	44	105	44	44	85	45	86	45	86	70	70	70	70	70	
393H-14	666.0	83.3	70	42	88	45	90	44	90	44	44	93	40	60	40	60	94	94	94	94	94	
393H-15	655.0	81.9	95	42	88	35	95	35	95	35	43	44	43	85	43	85	85	85	85	85	85	
393H-16	501.0	62.6	70	35	35	35	35	35	35	35	93	48	48	91	48	91	94	94	94	94	94	
393H-17	797.0	99.6	100	45	100	44	101	49	101	49	49	98	50	100	50	100	110	110	110	110	110	
393H-18	798.0	99.8	90	48	98	48	105	48	105	48	50	99	50	100	50	100	110	110	110	110	110	
393H-19	780.0	97.5	90	45	98	48	98	48	98	48	48	99	49	105	49	105	100	100	100	100	100	
393H-20	556.0	69.5	75	43	60	41	81	30	81	30	30	96	45	0	45	0	85	85	85	85	85	
393H-21	417.0	52.1	95	47	88	39	39	47	39	47	47	55	46	46	46	46	0	0	0	0	0	
393H-22	400.0	50.0	45	42	90	44	95	39	95	39	39	55	35	35	35	35	10	10	10	10	10	
393H-23	749.0	93.6	85	45	98	49	105	45	105	45	45	98	49	100	49	100	75	75	75	75	75	
393H-24	660.0	82.5	100	46	93	39	93	39	93	39	49	55	50	60	50	60	75	75	75	75	75	
393H-25	563.0	70.4	40	40	83	39	60	44	60	44	44	59	45	90	45	90	63	63	63	63	63	
393H-26	791.0	98.9	100	46	98	48	105	48	105	48	48	98	50	96	50	96	102	102	102	102	102	
393H-27	797.0	99.6	100	45	97	49	100	49	100	49	49	99	45	105	45	105	108	108	108	108	108	
393H-28	748.0	93.5	100	45	93	45	103	45	103	45	45	96	48	98	48	98	75	75	75	75	75	
393H-29	672.0	84.0	95	0	85	44	89	44	89	44	50	96	43	90	43	90	80	80	80	80	80	
393H-30	269.0	33.6	94	45	40	45	10	45	10	45	35	35	47	0	47	0	0	0	0	0	0	
393H-31	477.0	59.6	85	45	80	45	80	45	80	45	40	43	47	98	45	98	110	110	110	110	110	
393H-32	780.0	97.5	90	45	98	46	103	46	103	46	47	98	45	101	45	101	93	93	93	93	93	
393H-33	767.0	95.9	99	45	100	47	100	47	100	47	44	89	49	101	49	101	93	93	93	93	93	
393H-34	72.0	9.0	42	42	42	42	30	42	30	42	30	30	49	88	49	88	65	65	65	65	65	
393H-35	671.0	83.9	100	48	95	40	60	43	60	43	43	83	49	88	49	88	65	65	65	65	65	
393H-36	786.0	98.3	95	45	95	46	103	46	103	46	49	97	50	101	50	101	105	105	105	105	105	
393H-37	698.0	87.3	100	42	88	34	90	34	90	34	49	91	49	60	49	60	95	95	95	95	95	
393H-38	683.0	85.4	90	46	95	20	90	20	90	20	41	79	40	90	40	90	92	92	92	92	92	

Computer Science 110H - Fall 1993

	110 Total Programs	110 Program Average	110 Lab 1	110 Design 2	110 Lab 2	110 Design 3	110 Lab 3	110 Design 4	110 Lab 4	110 Design 5	110 Lab 5	110 Lab 6	110 Lab 7
Overall Average:			89.81	41.55	85.83	41.81	87.81	44.22	77.88	44.89	76.55	75.71	
Overall Standard Deviation			14.75	8.456	22.29	8.508	22.99	5.241	23.67	5.365	32.45	34.71	
Comp.Sci. Average:			88.54	40.28	85.04	40.93	88.52	43.75	78.76	44.15	75.92	75.42	
Comp.Sci. Standard Deviation:			16.06	9.243	23.51	9.269	20.12	5.711	23.79	5.804	32.48	34.91	
Non-Major Average:			93.78	45.67	89.22	44.88	85.67	45.67	75.44	47.38	78.5	76.63	
Non-Major Standard Deviation:			8.417	2.108	17.92	3.586	29.87	2.944	23.17	2.058	32.26	34.04	

Computer Science 110H - Fall 1993

	Previous 110 Course Grade(s)	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Semester Points	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
393H-1	C	2.00		A	4.00	94A-203	0.312	0.5	0.5	1.000
393H-2	B	3.00		A	4.00	94A-203	0.312	0.5	0.5	1.000
393H-3	B	3.00		A	4.00	94A-203	0.312	0.5	0.5	1.000
393H-4	B	3.00		F	0.00	94A-202	-3.235	-3.5	-3.5	-2.000
393H-5	C	2.00								
393H-6	C	2.00								
393H-7	F	0.00								
393H-8	Q	3.00		B	3.00	94A-505	0.375	-0.024	0.077	0.000
393H-9	B	3.00		B	3.00	94A-201	-0.6	-0.5	-0.5	1.000
393H-10	C	2.00		A	4.00	94A-201	0.4	0.5	0.5	0.000
393H-11	A	4.00								
393H-12	D	1.00	F-94A							
393H-13	B	3.00		B	3.00	94A-202	-0.235	-0.5	-0.5	0.000
393H-14	B	3.00		A	4.00	94A-203	0.312	0.5	0.5	1.000
393H-15	B	3.00		B	3.00	94A-509	0.4	-0.024	0.077	1.000
393H-16	C	2.00		A	4.00	94A-201	0.4	0.5	0.5	0.000
393H-17	A	4.00		A	4.00	94A-203	0.312	0.5	0.5	0.000
393H-18	A	4.00		B	3.00	94A-203	-0.688	-0.5	-0.5	-1.000
393H-19	A	4.00								
393H-20	B	3.00		C	2.00	94A-203	-1.688	-1.5	-1.5	0.000
393H-21	C	2.00								
393H-22	D	1.00	A-94A							
393H-23	A	4.00		B	3.00	94A-201	-0.6	-0.5	-0.5	-1.000
393H-24	B	3.00								
393H-25	C	2.00		A	4.00	94A-506	0.667	0.976	1.077	2.000
393H-26	C	2.00		A	4.00	94A-504	1.357	0.976	1.077	0.000
393H-27	A	4.00		B	3.00	94A-509	0.4	-0.024	0.077	0.000
393H-28	B	3.00								
393H-29	B	3.00		B	3.00	94A-202	-0.235	-0.5	-0.5	0.000
393H-30	F	0.00								
393H-31	C	2.00								
393H-32	A	4.00		A	4.00	94A-501	0.611	0.976	1.077	0.000
393H-33	B	3.00		B	3.00	94A-202	-0.235	-0.5	-0.5	0.000
393H-34	F	0.00								
393H-35	B	3.00		A	4.00	94A-503	0.529	0.976	1.077	1.000
393H-36	A	4.00		A	4.00	94A-506	0.667	0.976	1.077	0.000
393H-37	A	4.00		B	3.00	94A-505	0.375	-0.024	0.077	-1.000
393H-38	B	3.00		A	4.00	94A-505	1.375	0.976	1.077	1.000

Computer Science 110H - Fall 1993

	Previous 110 Grade(s)	110 Course Grade	110 Course Points	Subsequent 110 Grade(s)	120 Course Grade	120 Course Points	120 Semester	Additional Semesters	Difference In Section	Difference In Instructor	Difference In Semester	Increase In Grade
Overall Average:			2.676			3.36			0.064	0.0504	0.0908	0.2
Overall Standard Deviation			1.14			0.889			0.922045	0.967559	0.990498	0.848528
Comp.Sci. Average:			2.679			3.286			-0.012	-0.05562	-0.02195	0.142857
Comp.Sci. Standard Deviation:			1.167			0.933			0.98665	1.005865	1.026955	0.888322
Non-Major Average:			2.667			2.5			0.308667	0.404667	0.455167	0.333333
Non-Major Standard Deviation:			1.054			1.803			0.237647	0.442191	0.471297	0.471405

Computer Science 110H - Fall 1993

	210 Course Grade	210 Course Points	Increase In Grade	GPA After 210	Problem Solving Pre-Test(20)	Problem Solving Exam 1(20)	Problem Solving Exam 2(25)	Problem Solving Exam 3(25)	Problem Solving Final (12)
393H-1					12	14	6	20	10
393H-2	B	3.00	0.000	3.609	16	14	19	20	12
393H-3	A	4.00	1.000	3.063	12	16	17	22	9
393H-4	A	4.00	1.000	3.400	20	14	15	22	12
393H-5					18	18	18	25	7
393H-6					18	14	13	18	9
393H-7					2	16	5	7	2
393H-8					3	10			
393H-9	C	2.00	-1.000	2.387	15	16	21	21	7
393H-10	A	4.00	2.000	2.935	17	18	17	21	12
393H-11	B	3.00	-1.000	3.767	18	14	13	22	12
393H-12					19	16	16	12	8
393H-13					18	16	18	18	12
393H-14					14	20	22	24	2
393H-15					13	18	17	23	12
393H-16					0	18	14	18	9
393H-17	A	4.00	0.000	3.805	20	14	19	22	12
393H-18	B	3.00	-1.000	3.600	21	14	21	23	12
393H-19					15	16	15	22	9
393H-20					20	20	24	24	7
393H-21					15	16	21	22	12
393H-22					17	14	14	16	7
393H-23	B	3.00	-1.000	2.838	10	16	24	20	12
393H-24					16	14	16	25	12
393H-25	B	3.00	1.000	2.571	11	16	12	15	8
393H-26	A	4.00	0.000	3.932	15	18	21	25	8
393H-27					16	12	24	22	12
393H-28					13	16	16	20	8
393H-29	A	4.00	1.000	2.974	19	16	16	19	7
393H-30					9	20	15	18	2
393H-31					9	16	18	24	9
393H-32	A	4.00	0.000	3.634	16	13	22	20	10
393H-33	C	2.00	-1.000	2.400	14	18	9	19	8
393H-34					14	14	14	18	9
393H-35	A	4.00	1.000	3.000	14	14	15	17	10
393H-36	A	4.00	0.000	2.951	18	12	23	25	12
393H-37					21	20	20	18	12
393H-38	B	3.00	0.000	3.108	12	18	11	21	7

Computer Science 110H - Fall 1993

	210 Course Grade	210 Course Points	Increase in Grade	GPA After 210	Problem Solving Pre-Test(20)	Problem Solving Exam 1(20)	Problem Solving Exam 2(25)	Problem Solving Exam 3(25)	Problem Solving Final (12)
Overall Average:	3.411765	0.117647	3.175	14.52632	15.76316	16.78378	20.21622	9.189189	
Overall Standard Deviation	0.69102	0.899827	0.476	4.956319	2.355667	4.603965	3.691738	2.864865	
Comp.Sci. Average:	3.285714	0	3.163	14.7931	15.93103	16.42857	20.07143	9.142857	
Comp.Sci. Standard Deviation:	0.699854	0.92582	0.506	5.390792	2.37706	4.96724	3.963481	2.837432	
Non-Major Average:	2.4	0.4	1.939	13.66667	15.22222	17.88889	20.66667	9.333333	
Non-Major Standard Deviation:	1.959592	0.489898	1.599	3.018462	2.199888	2.960647	2.624669	2.94392	

APPENDIX D**INDIVIDUAL PROBLEM SOLVING TEST STATISTICS**

This appendix consists of the individual statistics for the problem solving portion of each exam for the test study CS/1 class. The total points possible for the Pre-Test, Exam 1, Exam 2, Exam 3, and the Final Exam were 20, 20, 25, 25, and 12, respectively.

.

Table 32. Problem Solving Statistics (Actual)

Student	Pre-Test	Exam 1	Exam 2	Exam 3	Final Exam
393-1	12	14	6	20	10
393-2	16	14	19	20	12
393-3	12	16	17	22	9
393-4	20	14	15	22	12
393-5	20	18	18	25	7
393-6	18	14	13	18	9
393-7	2	16	5	7	2
393-8	3	10			
393-9	15	16	21	21	7
393-10	17	18	17	21	12
393-11	18	14	13	22	12
393-12	19	16	16	12	8
393-13	18	16	18	18	12
393-14	14	20	22	24	2
393-15	13	18	17	23	12
393-16	0	18	14	18	9
393-17	20	14	19	22	12
393-18	21	14	21	23	12
393-19	15	16	15	22	9
393-20	20	20	24	24	7
393-21	15	16	21	22	12
393-22	17	14	14	16	7
393-23	10	16	24	20	12
393-24	16	14	16	25	12
393-25	11	16	12	14	8
393-26	15	18	21	25	8
393-27	16	12	24	22	12
393-28	13	16	16	20	8
393-29	19	16	16	19	7
393-30	9	20	15	18	2
393-31	9	16	18	24	9
393-32	16	13	22	20	10
393-33	14	18	9	19	8
393-34	14	14	14	18	9
393-35	14	14	15	17	10
393-36	18	12	23	25	12
393-37	21	20	20	18	12
393-38	12	18	11	21	7

APPENDIX E**SUMMARY CS/2 COURSE STATISTICS**

Tables 33, 34, 35, and 36 are summaries of the statistics for the CS/2 classes upon which some of the validation is based.

Information for each CS/2 course from the Spring 1991 semester through the Spring 1994 semester is included.

Table 33. CS/2 Course Statistics - 1991

Semester	Section	Instructor	Semester GPA	Section GPA	Instructor GPA
Spring 91	201	I-1	3.788	3.833	3.788
Spring 91	202	I-1	3.788	3.733	3.788
Spring 91	501	I-1	3.013	3.278	2.930
Spring 91	502	I-1	3.013	3.118	2.930
Spring 91	503	I-1	3.013	3.063	2.930
Spring 91	504	I-1	3.013	2.765	2.930
Spring 91	505	I-1	3.013	2.941	2.930
Spring 91	506	I-1	3.013	2.571	2.930
Spring 91	507	I-1	3.013	2.583	2.930
Spring 91	508	I-1	3.013	2.875	2.930
Spring 91	509	I-1	3.013	3.111	2.930
Spring 91	510	I-1	3.013	2.769	2.930
Spring 91	511	I-2	3.013	3.375	3.203
Spring 91	512	I-2	3.013	3.125	3.203
Spring 91	513	I-2	3.013	3.357	3.203
Spring 91	514	I-2	3.013	3.222	3.203
Spring 91	515	I-2	3.013	3.000	3.203
Summer 91	302	I-1	3.347	2.769	3.347
Summer 91	303	I-1	3.347	3.417	3.347
Summer 91	305	I-1	3.347	3.538	3.347
Summer 91	306	I-1	3.347	3.727	3.347
Fall 91	501	I-3	3.229	3.500	3.229
Fall 91	502	I-3	3.229	2.933	3.229
Fall 91	503	I-3	3.229	3.063	3.229
Fall 91	504	I-3	3.229	3.250	3.229
Fall 91	505	I-3	3.229	3.300	3.229
Fall 91	506	I-3	3.229	3.412	3.229
Fall 91	507	I-3	3.229	3.313	3.229
Fall 91	508	I-3	3.229	3.235	3.229
Fall 91	509	I-3	3.229	3.294	3.229
Fall 91	510	I-3	3.229	3.588	3.229
Fall 91	511	I-3	3.229	3.118	3.229
Fall 91	512	I-3	3.229	3.000	3.229
Fall 91	514	I-3	3.229	3.091	3.229
Fall 91	515	I-3	3.229	3.111	3.229

Table 34. CS/2 Course Statistics - 1992

Semester	Section	Instructor	Semester GPA	Section GPA	Instructor GPA
Spring 92	201	I-1	3.372	3.294	3.372
Spring 92	202	I-1	3.372	3.353	3.372
Spring 92	203	I-1	3.372	3.556	3.372
Spring 92	501	I-1	2.929	2.867	2.929
Spring 92	502	I-1	2.929	2.333	2.929
Spring 92	503	I-1	2.929	2.647	2.929
Spring 92	504	I-1	2.929	2.600	2.929
Spring 92	505	I-1	2.929	3.067	2.929
Spring 92	506	I-1	2.929	2.818	2.929
Spring 92	507	I-1	2.929	3.100	2.929
Spring 92	508	I-1	2.929	3.118	2.929
Spring 92	509	I-1	2.929	2.692	2.929
Spring 92	510	I-1	2.929	2.846	2.929
Spring 92	511	I-1	2.929	3.167	2.929
Spring 92	512	I-1	2.929	3.353	2.929
Spring 92	513	I-1	2.929	2.769	2.929
Spring 92	514	I-1	2.929	3.385	2.929
Spring 92	515	I-1	2.929	3.143	2.929
Spring 92	516	I-1	2.929	3.182	2.929
Summer 92	302	I-1	2.923	3.000	2.923
Summer 92	303	I-1	2.923	3.000	2.923
Summer 92	304	I-1	2.923	2.700	2.923
Fall 92	502	I-1	3.123	3.375	3.123
Fall 92	503	I-1	3.123	3.000	3.123
Fall 92	504	I-1	3.123	2.588	3.123
Fall 92	505	I-1	3.123	2.889	3.123
Fall 92	506	I-1	3.123	3.563	3.123
Fall 92	508	I-1	3.123	3.438	3.123
Fall 92	509	I-1	3.123	2.944	3.123
Fall 92	510	I-1	3.123	3.118	3.123
Fall 92	511	I-1	3.123	3.188	3.123
Fall 92	513	I-1	3.123	3.353	3.123
Fall 92	514	I-1	3.123	3.263	3.123
Fall 92	515	I-1	3.123	2.833	3.123

Table 35. CS/2 Course Statistics - 1993

Semester	Section	Instructor	Semester GPA	Section GPA	Instructor GPA
Spring 93	201	I-1	3.795	3.813	3.795
Spring 93	202	I-1	3.795	3.818	3.795
Spring 93	203	I-1	3.795	3.750	3.795
Spring 93	501	I-1	2.874	2.167	2.874
Spring 93	502	I-1	2.874	3.400	2.874
Spring 93	503	I-1	2.874	3.250	2.874
Spring 93	505	I-1	2.874	2.800	2.874
Spring 93	506	I-1	2.874	2.833	2.874
Spring 93	507	I-1	2.874	2.235	2.874
Spring 93	509	I-1	2.874	3.111	2.874
Spring 93	510	I-1	2.874	2.900	2.874
Spring 93	511	I-1	2.874	2.882	2.874
Spring 93	512	I-1	2.874	3.000	2.874
Spring 93	513	I-1	2.874	3.167	2.874
Spring 93	514	I-1	2.874	2.714	2.874
Spring 93	515	I-1	2.874	3.429	2.874
Spring 93	516	I-1	2.874	2.769	2.874
Summer 93	301	I-2	3.233	2.933	2.933
Summer 93	303	I-4	3.233	3.533	3.533
Fall 93	501	I-1	3.121	2.692	3.121
Fall 93	502	I-1	3.121	3.214	3.121
Fall 93	503	I-1	3.121	3.267	3.121
Fall 93	504	I-1	3.121	3.000	3.121
Fall 93	505	I-1	3.121	3.000	3.121
Fall 93	507	I-1	3.121	3.267	3.121
Fall 93	508	I-1	3.121	2.944	3.121
Fall 93	509	I-1	3.121	3.000	3.121
Fall 93	510	I-1	3.121	3.250	3.121
Fall 93	511	I-1	3.121	2.313	3.121
Fall 93	512	I-1	3.121	3.571	3.121
Fall 93	513	I-1	3.121	3.765	3.121
Fall 93	514	I-1	3.121	3.188	3.121

Table 36. CS/2 Course Statistics - 1994

Semester	Section	Instructor	Semester GPA	Section GPA	Instructor GPA
Spring 94	201	I-1	3.500	3.600	3.500
Spring 94	202	I-1	3.500	3.235	3.500
Spring 94	203	I-1	3.500	3.688	3.500
Spring 94	501	I-1	2.923	3.389	3.024
Spring 94	502	I-1	2.923	2.824	3.024
Spring 94	503	I-1	2.923	3.471	3.024
Spring 94	504	I-1	2.923	2.643	3.024
Spring 94	505	I-1	2.923	2.625	3.024
Spring 94	506	I-1	2.923	3.333	3.024
Spring 94	507	I-1	2.923	3.000	3.024
Spring 94	509	I-1	2.923	2.600	3.024
Spring 94	510	I-5	2.923	2.706	2.765
Spring 94	511	I-5	2.923	2.938	2.765
Spring 94	513	I-5	2.923	2.895	2.765
Spring 94	514	I-5	2.923	2.529	2.765
Spring 94	515	I-5	2.923	2.750	2.765

APPENDIX F

STUDENT REACTIONS TO WEB

The following are reproductions of reports submitted by the Fall 1993 test study group at the end of the semester. The students were asked to state what they expected from the class and why, their initial reaction to web programming, and and their current reaction/feeling to web programming. The reports were reproduced as written, with no corrections to spelling or grammar.

Evaluation of Fall 1993 CPSC 110H Students

Evaluator: _____

Circle the appropriate response to each of the questions below.
Please answer each of the questions to the best of your ability based on the student's written evaluation. If there is nothing in the evaluation to give you feedback on a particular item, select the 'Not Discussed' option.

1. What was the student's original reaction to being told they were going to learn something called WEB?

0	1	2	3	4	5
Not Discussed	Upset	Unhappy	Worth A Try	Looking Forward to it	Enthusiastic

2. What do you believe the student's expectation of the class was coming into the class?

0	1	2	3	4	5
Not Discussed	Beginning CS Course		Turbo Pascal		Problem Solving and Programming

3. What was the student's reaction to the emacs editor?

0	1	2	3	4	5
Not Discussed	Poor	Fair	Average	Good	Excellent

4. What was the student's reaction to TeX?

0	1	2	3	4	5
Not Discussed	Poor	Fair	Average	Good	Excellent

5. What was the student's reaction to WEB programming?

0	1	2	3	4	5
Not Discussed	Poor	Fair	Average	Good	Excellent

6. How well did the student understand the overall WEB process/concepts?

0	1	2	3	4	5
Not Discussed	Poor	Fair	Average	Good	Excellent

My Expectations and Reactions to Web Programming

My original expectations for this class weren't too high. Before class started, I felt that this was just a hindrance to getting on to real, useful programming in C and that Pascal was just a necessary evil. I already have some programming experience in C so Pascal seemed to be a step down for me and I was anxious to get on with bigger and better things. I've since re-evaluated my thinking as I've found Pascal helpful for strengthening some of my C knowledge and an interesting language itself that has some very interesting big brothers (Modula-2 and Oberon-2). I've also found this class interesting due to the literate programming we are doing.

Initially, I was very excited about literate programming as it was something new and different. I was neat to know that we were privileged enough to be the first undergraduate class to get to try out this method of programming. It was also interesting to find out that large companies employ this method of programming; something of great value to me as I want to learn things that I can readily apply outside of college. The format for programming in web mode, initially, seemed helpful for focusing on the problem and I could definitely see the advantages it had for maintaining code.

Currently I have a few problems with web programming. First, it seems that I'm doing a lot of redundant, unnecessary writing/programming. I find myself listing out something for the T_EX document only to be typing out a similar list for the Pascal code. Second, I strongly dislike the implementation of emacs we are using. It's entirely antiquated and its user interface (or lack thereof) is very poor. I feel like I'm back using Word Perfect v1.0 when I know Word Perfect v6.0 or whatever is out there. Finally, when working with complicated programs, like lab 4, the web style seems to get in the way a lot and I find myself getting tangled (no pun intended) when trying to change things or when trying to get things to work.

Great Expectations?

1. What did you expect from this class and why?

I expected, from Computer Programming I Honors (CPSC 110H), a challenging PASCAL class with group analyses and discussions, group projects, and a large main project for the end of the semester. I believed this because that is what I have been associated with in all of my Honors programs before. I expected the learning of PASCAL to be quick and in depth so that by the end of the semester we could do many interesting and complicated things with the computers.

2. What was your reaction/acceptance to web programming?

My initial reaction to web programming was total disgust. I felt that I had been tricked into a course which was just off the chalkboard. We were told that we were guinea pigs, and I felt like it. If I did not enjoy PASCAL, computer programming, and a new challenge so much, I would have dropped the course and changed into the regular 110 class which is all PASCAL. Therefore, my initial reaction to the web was surprise, but I accepted it because it was a new horizon that I knew nothing about.

3. What are your current reactions/acceptances/feelings/. to web programming? How? and Why?

Although I was disappointed with the class structure in the beginning, I have begun to grow attached to web programming. Just as in all programs, there are things that are special that I like better than other programs, but there are also the things that I find annoying.

I like the way webs produce such nice documentation. It is very understandable, and it looks very organized and professional. It makes me feel good when I finish a program and it comes out with such a nice output along with a program.

On the other hand, there are many things that I do not care for in web programming. For the keys are not always what they seem, like the backspace key for example. Those things are hard to get used to. Another thing is the continuous auto-save function. It is really distracting and it can break your concentration easily having to wait on the computer. Finally, the lack of knowledge that we were given on the webbing was also a disappointment. The program designs could have been even better and more exciting if we would have had a little more information.

My Thoughts On TeX and WEB In The Comp Sci 110H Atmosphere

I. What did you expect from this class and why?

Unlike the usual computer science 110-H student, I am neither a freshman or a computer science major. I took the class because I liked learning about the computer, and I wanted to be able to use a variety of high level languages proficiently. I took a regular programming class my freshman year, and I was bored to tears. I took an honors class in the hopes of learning useful applications of pascal and having mentally stimulating programs that would challenge my intellect, not just my ability to enter code.

II. What was your reaction to WEB initially?

At first, I felt confused. Like most young adults, I am very eager to learn, and when knowledge is kept from me, I get very frustrated and confused. I wanted to learn about the WEB and TeX immediately. I felt that the information was not given out in an organized manner, and thus I really didn't view WEB as important or necessarily useful. Maybe if it had been taught in a more organized manner, I would have placed more importance on it. However, I was impressed by what services the WEB could offer a programmer. I could definitely understand its usefulness to a programmer who had the responsibility of a project like X-Windows, and I also understood that it was necessary for me to learn how to manipulate it with relatively simple programs first. Overall, despite being sceptical, I was perfectly willing to give WEB a chance.

III. What are your reactions to WEB now?

Now that I know much more about WEB and TeX, I can utilize more of its features and can appreciate its value more. However, I feel that a way of documentation that requires less work by the programmer will evolve and that WEB is just a phase in the development of such a tool. I am glad that I am learning it, and I do feel that it has a proper place in the comp sci 110-H class, but I also think that future classes will be using other methods of documentation and organization for their programs.

Assignment #1

1. What did I expect from this class and why?

I expected a challenging programming course in which we would learn vast amounts of PASCAL quickly. Through our knowledge of PASCAL, I expected to write complex programs that went above and beyond the programs I had previously written in high school. Through these programs, I expected to acquire complex problem solving skills as well.

2. What was your reaction/acceptance to Web programming initially and why?

Initially I didn't like Web programming. I thought it pretty senseless, had no meaning except to take up more of our time. I thought that there wouldn't be practical benefits from knowing Web. I have never really liked documenting my programs much, so naturally a system designed around documentation wouldn't appeal to me.

3. What are your current reactions / acceptance / feelings to Web now? and why?

Now that I know Emacs well enough to get around, it is not too bad. I still don't enjoy programming in it much. It seems to make it harder than it actually is. I do see the advantages of documentation such as organization and the ability to emphasize portions of the program. I understand where documentation is important to me as a programmer as well as other programmers who read my program. The Web system, though, just seems too complex for something that is elementary, at least to the extent that we use Emacs.

8

1. What did you expect from this class and why?

When I enrolled in this class, I was under the impression that it would be strictly a course in PASCAL programming. I was told during my summer registration conference that Computer Science 110 was a computer programming course that focused on the use of PASCAL. When we were told on the first day of class that the emphasis in this course would lie mainly on problem solving I was fairly surprised, but I did not think I would find it as difficult as it has turned out to be. Because I have had absolutely no prior experience in PASCAL or WEB programming, I have had some difficulty adjusting to this class. Although this course has turned out to be nothing like what I expected, I still find it very interesting and I am learning more than I probably ever wanted to know about computers, PASCAL, and this wonderful new thing called WEB programming.

2. What was your initial reaction/acceptance to WEB programming?

After our first exposure to WEB programming in this class, I decided that I hated it. I had absolutely no idea what those funny commands meant or how they created such neat output, and all the talk about EMACS, WEAVE, TANGLE, and TEX made things even more confusing for me. I couldn't understand how this new type of programming was supposed to make programs easier to read and more understandable, because it only made things even more confusing for me.

3. What are your current reactions/acceptance/feelings to WEB now? Why?

After working with WEB programming since the beginning of this semester, I am happy to say that I think I almost understand it. I still have no idea why the codes I enter produce such professional output, but at least now I can finally produce output. However, I still do not think that WEB programming makes programs more understandable. I think I would much rather have a pure PASCAL program in front of me than spend hours flipping through chapters, sections, and small pieces of code trying to find the information I need. Although I think I still pretty much hate it, I am trying my best to adapt to this new method of programming, and I hope that someday (preferably pretty soon) I will discover the benefits of WEB programming.

Web HomeWork

1) What did you expect from the class?

I thought that the class would deal with the normal introductory aspects to Pascal. On the first day of class I heard the teacher stress that the class was not "Intro to Programming" at all, we were to think of it as problem solving. This did not bother me, I just thought that it meant this class was legitimate. And as far as Pascal, the course has been everything I had expected because I managed to take a lot of this in high school and have my own computer.

2) What was your reaction to web?

At first, I did not take it seriously until I realized how much more there was to every assigned problem than just an hour of Pascal. Actually, for the first couple of weeks following the first project I was pretty upset with web and thought it must be a crutch for people unfamiliar to programming. The real reason I did not like web was that with it hurt my grade in the class.

3) What do I currently feel about web?

By now, I am comfortable with Web and have no problem with it. Since I have had some experience with Pascal beforehand, I'm sure I would have found it simpler if we learned on that solely but I can see why we use Web now.

Homework Assignment CPSC 110H

1.) My expectations for this class consisted of obtaining a more thorough knowledge of Pascal. I had already learned a little bit about Pascal in my high school computer science class, but I was hoping to gain a more extensive knowledge of Pascal in this class. Even though my class last year was a whole year and this class is only one semester, I expected we would cover more in this class just because it is college and more independent work can be given.

2.) I had mixed reactions when I heard we would be using WEB programming. I was kind of apprehensive since that would mean I would have to learn another language just to program in Pascal. However, since WEB programming is used in the real world, I was glad I was getting a chance to learn it. Thus, overall, I was happy we were using WEB programming.

3.) After learning the basics of WEB programming, I have come to like it. Although it took a while to learn some of the basic commands, it is now just as easy to use WEB programming as it is to use the plain Pascal Editor. It also allows for easier and more complete documentation. Furthermore, it is also beneficial to me since now I am familiar with one more programming tool and may have a need for it in later life.

1. Before I started this class, I thought it would teach me more than the average class about PASCAL. I thought that it would go in greater detail about the intricacies of PASCAL, I never thought that it would be more on problem solving. All the honors classes I have been in just speeded up the learning process, trying to fit as much information into the time available.
2. When we first started this class, I was wary of how good I could do in it without having any previous programming experience. When I found out that we were programming in an environment new to everyone I knew that I would have an easier time in the class, getting a good grade, etc.. I had never heard of WEB programming, I didn't know what to think. I had no problem accepting it because I had no idea what it entailed.
3. After programming with WEB, I don't understand the need for it. I understand that it helps with the programs readability, but a good programmer should be able to make a very readable program by using comments. It might make it easier to write modular programs, but it also makes a beginning programmer lazy in his programming practices. I also found that it confused me because it was hard to remember what I had and hadn't done until I compiled it. Not only that, but in order to compile it, you have to go through many steps that seems to make the debugging process much more difficult than with a PASCAL specific editor. I can understand, and appreciate the Emacs editor in a UNIX environment, but cannot understand the reason for using it as an introductory course when you should worry more about the PASCAL of the program.

1) What did you expect this class to be, and why?

I expected this class to be a little tougher. I expected that we would have covered the material much more rapidly. I thought that in an honors class, everyone would have known Pascal already, although now I see that this isn't the case, and I'm quite glad for that fact. I was glad to see that in this honors class, there was a certain unity that isn't in any of my other non-honor classes. I was quite disappointed, however, that we weren't taught any audio or visual concepts.

2) What did you think of Web, in the beginning?

I've had mixed feelings about T_EX and Web. At first when I had heard about it, but hadn't used it, I thought that it was a good programming idea, which I still do. However, once I began using it, I didn't care for the demacs editor. I also disliked all of the T_EX commands we needed to learn. I understand that T_EX is the primary editor for writing technical documentation. I also know that it completely supports Web mode, which is a quite handy, yet I feel that there must be a better editor that we could use, that would also support Web.

3) What do you think of Web now?

Now I think that Web mode, in theory, is an excellent idea. I also think that in practice it works quite well. The only hesitations I have is with the Demacs program, and some of the limits of the Weave program. I really have grown to hate the auto saving, and the capitalization problem. In my opinion, no programming language should be case sensitive, including Weave.

1) Upon registering for this class, my first feeling was elation. I had the very last, I do mean last, registration. Needless to say, my schedule wasn't exactly what I had wanted. I filled the last available seat in this class. I took two years of advanced placement pascal in high school. Of course that sounds a lot better than it actually is. My teacher moved very slowly. In two years we covered pascal up to records, files and arrays. This class met every day and we only had four people in it. Despite her fallacies, dear Mrs. Gano was thorough, so I did feel confident with what little I did know. I expected this class would broaden and enhance my pascal knowledge. Mrs. Gano did not believe in turbo, so I did expect (hope) to learn some turbo pascal. I did not expect to be known by name but it's nice. I did not expect to spend my college life in the computer lab, but I've adjusted. I certainly did not expect web mode programming, which leads to the next question.

2) The only word I can think of to best describe my reaction to web mode programming is confusion. Not a blurry confusion, or even a puzzled confusion, I was completely and utterly cluelessly confused. The second choice word would be why? I tried to be open minded even though I could not see any 'method to the madness'. Even after the completion of our first lab I had no idea how to use web mode. Mrs. Gano, bless her heart, was a firm

believer in structured programming. Every detail, capitalization, spacing etc had to be flawless, orderly and organized. I did not know how to make the transformation and I was annoyed that most of my peers did. When first introduced to web mode, it is fair to say that I did not like it at all nor did I understand it.

3) Wiser and more schooled in the world of web, I have come to accept web mode and even see it's good points. It requires a different approach to writing a program that isn't all bad. It would be very easy for a non-programmer to read a program written in web and fully understand the aim of the program. It's nice to know how to do something other than just strict pascal. And there is a certain joy in weaving and texing and not getting any errors that could not be attained any other way. I would not use web to write a program unless it was required, but I might if I hadn't already known pascal, but if I was required to it would be okay.

Assignment: Response to WEB Programming**Questions****1. What did you expect from this class and why?**

To begin, I was expecting CPSC 110 H to be a "review" class from my point of view. Having learned Pascal in high school, I felt that I had mastered most of the concepts that dealt with the language. Although I understand it, I find problems dealing with refreshing my mind to specific syntax. (Examples: where and where not to use quotes with characters, accessing specific fields within records, etc.)

2. What was your reaction/acceptance to WEB programming initially?

After encountering WEB style programming for the first time I thought "What bored person came up with this system?" I could not imagine why someone would want to get into my program and read page after page of documentation in order to see how my program functioned. I was always accustomed to worrying about whether or not the program works and not writing a "paper" on how it would work. Compared to using Think Pascal, the editor I had previously used, DEMACS was a nightmare. The user interface was terrible and having to memorize five thousand different keystrokes was not fun. I was also bogged by all the Tex commands that were being thrown in my face by Pete. Tex commands were a pain to learn and the time consumed learning these commands really got me frustrated. For further elaboration on my initial feelings concerning WEB please see the illustration provided.

3. What are your current reactions acceptance/feelings to WEB and why?

At this time I understand why WEB programming can be useful when dealing with the design and maintenance of a program. I suppose I never realized that these two concepts were the most important aspects of program design. In addition, it is easy to pick up where I left off with out having to review what I did previously. As for DEMACS, I still hate it; WEB keystrokes are too

numerous and are a hassle. A pull down menu or mouse interface would be a lot more convenient. Tex can be confusing at times but once I start using certain commands regularly I think that the time consumed will be reduced. Also, I find that going to and from DEMACS in order to compile, weave, tangle, tex, and edit is very inconvenient when it comes time to debug a program. I probably spend twenty percent of my time in the lab waiting for the machine to re-tangle, re-compile, and reload WEB every time I find the smallest bug. Then again, I may just be one of the few people who is that picky. Please see my second illustration to give you a better idea on my current feelings towards WEB.

Before WEB



Illustration 1: Initial reaction towards WEB

After WEB



Illustration 2: Current Feelings towards WEB



Expectations of Computer Science 110 from Students

1) What did you expect from this class and why?

Being completely new to the science of computers, I was rather unsure as to what to expect from this class. I had declared my major as Computer Science because I enjoyed working with computers using applications software such as WordPerfect, Display Write, Quicken, and, of course, games. I was also fairly knowledgeable with commands and functions of DOS. However, I had no idea how similar programming was in comparison to simply using software. I suppose it was this extreme lack of knowledge in reference to programming and computers overall that enticed me to take the Computer Science 110 Honors. I hoped that since the class was much smaller, I would be able to grasp the programming concepts and applications more rapidly than if I had taken the regular 110 course. I also hoped to discover whether or not I truly wanted to make Computer Science my major and my chosen profession. I was quite relieved and a little excited when I discovered that I enjoyed programming a great deal, but, more importantly, I enjoyed the design process the most. In essence, I was hoping to discover through this course whether or not I would enjoy spending the rest of my life designing programs. As of right now, I have been very content with what I have learned in this class, and my expectations have been met in full.

2) What was your reaction/acceptance within the first couple of weeks to WEB programming? Why?

Since I have never programmed before this class, my acceptance to WEB programming was almost immediate and unquestioning. After all, I was unfamiliar with all other types of editors and did not have a clue as to the advantages and disadvantages of using EMACS in comparison to other editors such as Turbo Pascal. Putting aside this apparent lack of experience, my first initial reaction to WEB programming was one akin to surprised amazement. After all, I was fascinated with the fact that EMACS not only read our program code amid the multitudes of text, but also ripped out all of our documentation (through the TeX function) and made an extremely nice, professional guide to our programs. To put it simply, I was very impressed with WEB programming and its many advantages.

3) What are your current reactions/acceptances to WEB programming? How? Why?

To be quite honest, I believe that I am one of the only students who actually still believes that WEB programming is worth all of the extra work and documentation it requires. Since I am new to programming, I believe I see the importance of careful and accurate documentation to help guide individuals

through the program. I have finally had an opportunity now to work with the Turbo Pascal editor, and I am wholly convinced that WEB programming is quite a bit more advanced and advantageous than the Turbo Pascal editor. WEB programming allows so much more opportunities for good documentation than other editors. It is quite easy to get lost in the code of a complex program, and, if it wasn't for the neat documentation that EMACS allows a programmer to place with the code, it would be very difficult to follow the program through its entirety. I am an avid supporter of computer designs (probably because I would get lost in MY own code if it wasn't for the design to help guide me through the program), and WEB programming is the most efficient editor in aiding documentation and design that I have encountered. As a result, I have accepted WEB programming as the most effective means for writing code AND for writing neat, professional documentation.

1. What did you expect from this class and why?
I expected to be in a boring class and get an easy A because I have had 2 years experience in PASCAL . However this has not been the case. I feel that I have learned better programming style, etc. from this class.
2. What was your reaction/acceptance/??? to WEB programming initially and why?
At first I thought that WEB programming would would make it easier to write programs since you don't really have to worry about order quite as much as in conventional programming.
3. What are your current reactions/acceptance/feelings/??? to WEB now and why?
Now that I have more experience with WEB I don't like it as much. I have found that the small amount of ease in programming doesn't compare to the headaches caused by using WEB. Overall, the concept is great but the actions leave much to be desired.

Opinions of CPSC 110

1. What did you expect from this class and why?

As a freshman, my expectations for computer science 110 were only a part of my expectations for college as a whole. My first semester of classes brought with them several individual expectations.

However, my expectations for the computer science class were considerably higher than for my other classes. One reason for this is simply because my major is computer science. I would naturally be anxious about any computer related classes.

Another reason for my high expectations is the fact that I enrolled in an honors section. I was prepared to face a rather small class with a more personal feel to it. By comparison, I am just a face in an extremely large crowd in my other classes. I received the impression during registration that honors courses are more difficult than the other courses. Therefore, I came into the class expecting a more difficult work load than the normal computer science sections. I must admit that at times, writing the program designs and program code seem like an enormous amount of work. However, I generally enjoy programming, and I am usually pleased to see the results of my work.

The one thing that I expected most from my computer science course was learning the Pascal programming language. I had read the course description in the undergraduate catalog, so I knew that Pascal was the language used in this particular class. Even before I read the student catalog, though, I knew that I would be programming in Pascal. I had talked to several people who had taken the course, and I was told by my high school computer science teacher that generally Pascal was the initial

language taught at universities.

I expected my computer science course to contain a degree of problem solving. However, I never expected it to be stressed as much as it has been. I have always enjoyed programming, not because of the language itself, but because of the opportunity it presents to solve problems. The one thing that I enjoy most about programming is being able to come up with a solution to any given problem, and more importantly, to come up with a solution that works. Obviously then, I was prepared to encounter problem solving, but I expected the course to focus more on the Pascal language.

2. What was your reaction/acceptance to web programming initially and why?

Before this class, I had never heard of web-mode. I had never heard of Emacs or TeX. Even though I previously had used computers extensively, most of my experience with them had involved either a Macintosh or windows based applications. Therefore, when discussion turned from Turbo Pascal to web-mode, I became completely lost. It was all foreign to me. I immediately became apprehensive and worried about the class. Because I knew nothing about web-mode at that time, I mistakenly assumed that it was something obscure that I would never be able to comprehend.

Thankfully, the class eased into the web-mode material relatively gently. My initial shock gradually wore off, though I still had several doubts. I feared that I would have to spend long hours mastering the information necessary just to write a Pascal program, something I had previously had quite a bit of experience doing. The experience made me realize that there was more to programming than just knowing a high-level language, and that I did not begin to know as much as I thought I did.

3. What are your current reactions/acceptance to web-mode and why?

Despite my initial reaction to web-mode programming, I have started to really like it. It has proved to be much easier to master and use than I originally thought.

Web-mode adds a rather strange twist to programming. It is certainly a more preferable means of documentation. This method provides a neat, good looking report. When compared to trying to sort through the documentation as it appears in the listing of a Pascal program, it is a wonder that anyone could not prefer web-mode. The web-mode documents logically structured and easy to read. I love the fact that it automatically creates a table of contents and an index.

Web-mode provides a wonderful way to create a modular program. I find it much easier to work out a problem step-by-step when using web-mode. I like breaking down the problem into every little detail and then finding a solution for each of them. I don't feel as restricted when I program; I don't have to stick to strict, standard problem solving methods. Instead, I can simply solve each problem in an order that makes sense to me, and therefore, hopefully also makes sense to anyone reading the document.

1. Before I had registered for CPSC 110 I had talked to Ms. Rierson who is my academic advisor from the MEP (Minority Engineering Program) office about what this course would be about in general. Basically, what she told me is that it was a programming class that implemented Pascal. I had taken a Computer Science class using Pascal in high school during my junior year where I was taught programming. So, naturally I thought that this class would also be a class that focused strongly on the language of Pascal and coding techniques. I expected to learn how to program in Pascal and become well versed in its syntax and special features. These were areas that my Computer Science class focused on in high school and since I used the same book then as we are using now I thought the class would be basically the same.

2. Prior to the first day of CPSC 110 I had never heard of WEB programming and its style and syntax were quite a change from what I was used to. At first, I was completely overwhelmed, because I thought we were going to have to learn several brand new computer languages (WEB, T_ex, TANGLE, and WEAVE). I thought that these were new *programming* languages and that I would be completely lost.
However, when I finally understood what WEB was used for, my anxiety gradually subsided, because I realized that learning WEB was supposed to help coordinate my thought process and organize my design so that the flow and readability of my program would be optimal. I was still a little worried afterwards, because I was not sure if I would be able to handle Pascal *and* Demacs.

3. My opinion of WEB right now is slightly different. After I realized that it was supposed to enhance my program I was really interested in learning it, but as deadlines came around I realized that, although WEB was supposed to improve my program it, succeeded more in delaying its completion. However, I must admit that it takes out a lot of the pain that accompanies actual internal documentation of the program itself.

As a matter of fact, if had not been for the documentation that I had done in Demacs there would be some problems that would have been more difficult to solve. In summary, my opinion of WEB is this: If I manage my time working on the lab assignment, then WEB enhances my program a hundred-fold, but if time is short then WEB, becomes a great hindrance to the completion of the program itself.

Computer Science Expectations and Fulfillments

Since Computer Science 110 is required for my major, I didn't read the course description in great detail. The only expectations I had for the course material were to learn another programming language and to write programs for a grade. I chose honors because I knew the class would be smaller, the programming would be more complex, and because I could register early. I also wanted a honors class because it is easier to get individual help in a small class environment.

When I first used *web* to write a program, I didn't like it because it was more time consuming than simply writing code. I've always used descriptive variable names, but I've never documented my programs before. I was against *web* because I felt it was a waste of time to describe every step in detail. I can see the logic behind the whole design process, but it's still a pain to manually calculate every test case, and I still don't understand the purpose of having an abstract when I say the same thing in my introduction and problem description.

Overall, however, I see more pros than cons. The process of *weave*, *tex*, *tangle*, and *lpc* makes debugging easier. The autosave feature of *web* is convenient for me, because it allows me to concentrate on what I'm typing instead of worrying about saving my file in case the system crashes. Even though it can be annoying when I'm in high gear and in the middle of a complicated thought or edit, I know autosave will save my sanity one day when I think I've erased my entire file.

I think that this class has taught me more than if I had just taken the non-honors class. I now know the benefits of using an editor, and the necessity of the design process. While the *web* system has some quirks that take some getting used to (like using `del` instead of the backspace key), it seems to be a better way to program.

1. *What did you expect from this class and why?*

To be honest, I really didn't know what to expect from this class. I know I initially worried about being the only person in the entire class that didn't know Pascal (which I wasn't). I also worried about taking this class in an Honors section with no previous computer programming experience, except in high school.

I wasn't too worried about learning the code itself, because my boyfriend, Clint, assured me that I would pick it up quickly. So, far I have struggled with some of the concepts with the code, but I feel that is probably due to the speed at which we cover the material. Overall, I feel that I am learning the main concepts of Pascal, and I am beginning to feel confident in my programming ability.

2. *What was your reaction/acceptance /feelings to WEB programming initially? Why?*

I was overwhelmed by the thought of not only having to learn Pascal in an Honor's class, but to also learn an editor. After the first couple times on the computer, I began to feel more comfortable with WEB. On the shorter assignments, I was pleased with the results and enjoyed making my initial designs look good. It was a challenge to not only write a good design, but also make the appearance of this design look desirable too.

3. What are your current reactions/acceptance/feelings to WEB programming now? Why?

After writing numerous labs on the Web editor, I have decided this type of documentation is for the birds. I realize the importance of documenting one's work, but the amount of agony and grief spent on these programs was not worth the output. The more complicated a design became, the more time I spent on the web documentation. In the last couple of program designs (after the initial design is due), I have found myself spending more time "weaving", "tangling", "texing", and auto saving than I care to mention. If these processes did not take so much time, I would have little objection to using such an editor.

It would also be helpful to design an editor that could be used with a mouse. The process of cut and paste is quite evil with WEB, and by using a notebook on my PC or the edit mode in DOS, I can cut my editing time in half.

I also feel that it is totally unfair to test student over Emacs commands. This memorization is not only a total waste of brain power, but also a waste of time, especially since you provided the class with emacs "cheat sheets".

Overall, I have really enjoyed this class. I don't want you to think because I dislike WEB that I haven't enjoyed the work I've done in this class. I feel that the time spent on WEB, might have been put to better use somewhere else. But, then again, I could be wrong. I have no idea how important documentation is, or how effective this editing program can be. I am just learning. Maybe with enough exposure, I would learn to like WEB or maybe some other editor like L^AT_EX. But again, I have enjoyed the problem solving aspect of this class. It is a challenge that I have enjoyed struggling with.

1) What did you expect from this class? and Why?

I expected to learn a program language. I was not looking for problem solving skills. I am a Math major and this is the only Comp.Sci. I have to take. I took the Honors section because I wanted to register early.

2) What was your reaction / acceptance / ... to web programing initially? and why?

I was OK. I have nothing to base my reaction on, because I have no previous experience. It is helpful for short programs and It help me because I didn't know how to program at all.

3) What was your current reaction / acceptance / feeling ... to web now? and why?

I hate it. It Clutters the file. I wish I could just write a program. I can see why YOU would use it but for me it is a pain. It takes to long to open the demacs' file it takes to long to tangle and tex and weave.

What did you expect from this class and why?

When I signed up for this class, I did not know what to expect. I signed up for the class because I am interested in taking a AI course (CPSC 320). In order to do that I either had to take ENGR 109 or CPSC 110. Being a Biochemistry Major, ENGR 109 has absolutely no bearing on what I want to do in life. I took CPSC 110H because I figured the class would provide me with the background necessary to prepare me for CPSC 120 and eventually CPSC 320.

What was your reaction to WEB programming initially and why?

Initially, I looked at WEB programming as interesting and was looking forward to doing it. I have had very little experience with Pascal in the past and saw the need for the documentation. I have looked at programs in the past and not understood anything about what they were trying to accomplish. This causes great problems when trying to modify the program.

What is your current reaction to WEB now and why?

I still enjoy the WEB programming. The problem I have is the efficiency of the interface. The time it takes to launch emacs and then the time it takes to tangle, weave, and TEX the WEB program is excessive. Emacs is a tedious way to debug a program because every little change requires changing the WEB source code and then tangling and compiling the code. This takes a large amount of time that could be more efficiently used in editing and debugging the program in a Turbo Pascal editing environment, but in doing this the programmer must go back to the WEB code and change the Pascal code later.

When I decided to take Computer Science I was expecting to learn to program in Pascal using TurboPascal. I did not know of any other Pascal editors, nor did I know of Emacs, WEB mode, T_EX, or weaving and tangling. I thought that the honors part would involve only more in-depth programming and maybe an accelerated pace. This is what I was expecting from what I had experienced and from what I had heard.

When I found out we were going to use Emacs, it wasn't a big deal, it was just a matter of learning a few command keys. However, WEB mode programming was something totally alien to me and the thought processes involved seemed totally backward to me, but I thought it had possibilities. I saw the benefits of WEB and was prepared to learn it. This acceptance did not endure long though.

After being repeatedly foiled in my attempts to add Pascal in to the T_EX material I developed for my first lab, I began to really dislike the backward way in which we were programming. Additionally, the lack of a practical method of drilling myself in the Pascal I was learning from the book and the lecture soon left me drowning in a whirlpool of incomplete labs and assignments. I am still a little shaky on Pascal itself, and have no clue about plugging the little Pascal that I do know into a WEB file in such a manner that it functions properly. I like the documentation produced by the WEBs but would be much happier using straight programming in TurboPascal with internal documentation in the program, not a program within the documentation.

What I Expected From This Class

After reading the course description of Computer Science 110, I was unsure of what to expect. By the wording in this description, I expected the class to be more lecture oriented and less programming. I thought this class would just be an introduction to the process of programming. I did not expect us to jump into programming as fast as we did. I figured that the first semester we would learn the practices and the second semester we would put them to use.

Also, I thought that I would have no problems with this course. Such is not the case. I must say that I have not reached my goals in this course. I guess I thought that since all the other courses in computers that I have taken have been easy for me, that this one would be too. That was a bad assumption.

The final thing that I expected from this course was the closeness that accompanies having small classes. I thought that since this was an honors course the class would be closer as a whole than in my other classes. For some reason I expected this class to be more similar to my honors classes from high school. My honors classes were very close and the students had a very close relationship with the teacher as well as each other. I guess comparing high school classes to college classes was naive on my part, but that is what I expected. I am sure that students in this class will cross paths again.

My Initial Reaction To WEB Programming

My initial reaction to WEB programming will probably differ greatly from other students due to my lack of previous experience with programming. My first reaction to WEB programming was one of utter confusion. As far as including the definitions and explanations within the program body, this was something new to me. I had no idea what the "limbo" material did. To tell you the truth, I thought that was part of the PASCAL program. For the first program I had completely no idea what was going on. I was trying to figure out what the individual statements in the "limbo" material did. As you can tell, those first two or three weeks were very tense for me. Later on, I formed different opinions about WEB programming.

By the second program I felt more comfortable using EMACS. During this phase, I felt that doing the design and then "filling in" the code was a pretty good idea. I also liked the fact that longer descriptions can be used and still be understandable. This phase lasted up until about the fourth and fifth programs.

Current Feelings About WEB Programming

Recently, I have formed new opinions about Programming using WEB mode. Up until the last few programs, I had a pretty high opinion about using it. Now, I am not really sure how I feel. WEB programming has its advantages and disadvantages.

On one hand, I like how the output is formatted. I also like how the text and code is integrated using WEB mode better than it would be otherwise. The thing I like most about programming using WEB mode is being able to declare variables or anything in any order and call them in the main program. I would like WEB programming even more if I knew how to do more of the specialized procedures. More handouts with examples would help.

More recently I have discovered some of the disadvantages that are associated with using WEB programming. When working with longer programs, the compilation and similar processes are much more tedious. If you do not make programming errors, I guess you do not have to worry about correcting them. But if you do make mistakes, then correcting errors with WEB programming is also very tedious. It would not be so bad if you did not have to close your EMACS file every time, but you do.

My opinion about WEB programming changes nearly every time I use it. Right now I can not decide whether EMACS is more of a help or hinderance. It is slow, but the printed product looks so much more elegant. Which would you rather have?

Evaluation of CPSC 110

1. When I enrolled in Computer Programming 110 I was expecting a class that would teach me the basics of programming in Pascal. In taking a honors I was expecting to be in a class that was taught with a greater degree of knowledge and to a class with higher degree of intelligence. I was also expecting the class to be taught by a professor who would teach the Pascal language to a greater degree than a regular class professor would. Not knowing how to program in Pascal was one of the draw backs of signing up for this class but I figured that I would learn more in the class if it was an honors class than a regular class.

2. As I stated in the previous paragraph I have not programmed in Pascal before so I do know the differences between web programming and other types of systems. In my untutored opinion the web program is very helpful in putting together a program that is very easy to understand and also very easy to correct if there is a change in the program. I really cannot give any comparisons between this style of programming because of my lack of knowledge in this subject. I have however tried to program in turbo pascal and found that web style of programming was easier for me however, I am not sure if it is just that I am used to the web programming so much that everything else is just foreign to me.

3. Right now I feel that the web mode is a very useful tool for programming in Pascal. If it wasn't for the web programming system I do not feel that I would be able to complete any of my programs to the degree of accuracy that so far all my programs have been. The web programming has helped me to understand how to program in Pascal better than any other program probably would. The only thing that I noticed about the turbo Pascal that I wish was on the web programming was to be able to show you exactly where your errors are while the program is still up on the screen.

Question 1: What did you expect from this class and why?

—Before I started this class in the fall, I expected several things from it. Being an honors course, I expected the class to move at a quicker pace and to cover more than a regular course. I expected hands on experience with current hardware and software because of the large amount of resources available at Texas A&M. I expected to learn a little about the campus computer network because, as a Computer Science major, I will need to know how to use the computer facilities. I also expected to gain some knowledge about the current computer industry, realizing that the purpose of college is preparation for real world involvement. Last, I expected an instructor who is enthusiastic about teaching.

Question 2: What was your reaction to Web programming initially? Why?

After the first few lectures and before working extensively with Web programming, I was confused about how it worked and didn't see any dramatic differences between it and the Turbo Pascal editor. I was unsure of Web simply because I didn't have any experience working with it. It was just an abstract idea in my mind and its uses were vague to me. However, I *did* accept the program and was eager to learn more about it. Now that I reflect on it, I do see that my current impression of Web was formed by being open minded and by willingness to accept the program beyond its initial presentation.

Question 3: What are your current reactions to Web? Why?

Now that I have been able to work extensively with Web, I see that it is a very useful tool, and I enjoy doing labs with it. I believe it is useful because of the way it forces organization and the emphasis it places on planning. I am beginning to understand the importance of program documentation, and I realize that Web is very valuable in that regard. I enjoy working with Web because I enjoy writing, I like the style of documenting text that Web uses, and I like the control the user has over the output. My overall impression of Web now is positive. I do see the advantages it has over the Turbo Pascal editor, and I am glad I have the chance to use it.

Programming I ?

I knew this class was a programming one because of the title and description in the 116th Undergraduate Catalog. Plus, I had a good idea that the language would not be in BASIC; that one's too simple. The notion of programming really did not hit me until I actually got into class on the first day. To tell you the truth, I didn't know what to expect from this class; I took it because it's the first course any Computer Science major must take, and I took the honors because I felt it would be very beneficial to have an honors class that was in my major. As to the contents of the class, I had no idea what was to be expected, nor did I really care, since I had to take it anyway. All I knew was that I would find out more about the class when I got there.

When I first got into class and realized it would be programming in Pascal, I thought, "That's cool." But, when you started mentioning the WEB program/file and weaving and tangling, I began to think, "Am I in the wrong class? This sounds like basket weaving. What does basket weaving have to do with computers?" I am not kidding you; I really thought that. The first day I came to fully understand what you were talking about when you said "Tangle" and "Weave" was the day when you gave us the diagram of the WEB world, in which the WEB file goes down the two paths via Tangle and Weave.

When we actually started working with Emacs and our WEB files, I was a little leery about it because of all the strange, new

commands. I didn't let it get to me because I'm the type of person that just goes with the flow. When we got a chance to see the final output of the DVing, I was amazed and awed that the computer could do that. I thought it was really interesting; yet, I still had a few little doubts here and there about a variety of items. At the time, however, I was still trying to get settled down into college life, so I didn't worry over it too much.

Now that we have done several labs using Emacs and WEB, I have a better feeling about it. I like it. WEB allows me to write down my thoughts at the same time and place where my code is written. It's very helpful in keeping your thoughts straight in your head. It allows you and others to know what you did at a specific point in the program and why you did it that way. Also, breaking up the code into different sections makes it a lot easier to write code, for two reasons. The first is because this allows parts of code to be written and placed in the area where the documentation for that code is found. The second is that by breaking down the code into little parts it is easier to work with and figure out the problem and the bits of code to go with that problem. This stepwise refinement really helps in turning a monster task into simple, easy assignment statements. WEB and all its attributes are very useful to programmers and very beneficial to everyone involved with the program.

CPSC 110

My initial expectations for the Computer Science 110 honors class were numerous. My primary expectation of the class was the size. For fourteen years I attended a small private school where the classes never exceeded 20 and by enrolling in the CPSC 110 honors I hoped that the size would be similar to that of my high school. I felt that this was not an unreasonable expectation because the way in which A & M presents the honors program it can only be deduced that the size of any honors course would be significantly smaller than the regular equivalent.

My second expectation of CPSC 110 honors was that it would be taught by someone who was not only interested in the field but also energetic about the class. This expectation was also reasonable in my mind simply because, that is what the honors program is all about, teachers and students who care. This brings me to the third expectation and only expectation that has not been adequately fulfilled. I had hoped that the class would be made up of students who were not taking the course just to fulfill their honors requirements but rather students who had the desire to learn everything that they could, Students who came to class because they wanted to be there, and students who respect those around them and the teacher. Needless to say, that is not the case.

My final expectation of CPSC 110 honors was that it would solidify my base in problem solving and reinforce my knowledge of Paschal. The course has thus far done an outstanding job of both solidifying my base and reinforcing my knowledge of Paschal.

My initial reaction to the web style of programming was one of complete confusion. We were thrown into the lab and left to understand the concept by trial and error. Not to say that there was absolutely no instruction about the nature and purpose of a WEB file, but, for myself, someone who has previously programmed in paschal, there was not enough explanation as to the advantage, the proper implementation, or the integration

of code into a WEB file. The first lab was intensely annoying. We had no idea what was happening, what we were typing in, and most importantly, why we got all the errors and how were we going to correct them. The errors in the T_EX were the most frustrating because we had absolutely no idea what the T_EX commands were doing and no way of finding out if we were working in the lab at night or some other time that help was not available.

However, I have been converted. I now believe that the WEB style of programming is much better than that of any other I have thus far encountered. The reason for this one hundred and eighty degree turn around was that when I attempted to go back and edit a program that I had previously written in my high school career it took me several hours to even determine the general area that I needed to concentrate on. The program was an address book that used arrays, records, color, windows, and everything else that is now nothing more than code with no rhyme or reason as to why. I can now not only see the advantage to the WEB style of programming but I can appreciate it.

However much I appreciate the WEB style of programming it is still to this day very frustrating at times. The only thing that I could think of to alleviate the confusion would be to teach more of what the T_EX is all about and the commands. But perhaps the best way to learn is trial by fire.

Reactions to Class

1. *What did you expect from this class and why?* I didn't really know what to expect. I had heard that it was just basically PASCAL, so I expected a more in-depth version of the computer science class that I took in high school. I assumed that the Turbo editing environment would be used, simply because I didn't know about any more sophisticated or pragmatic editors. Since the course is honors, I guessed that the class size would be smaller and the assignments would involve more thought than the regular sections' problems.

2. *What was your reaction/acceptance/whatever to WEB programming? (In the first couple of weeks of class) Why?* I found EMACS to be a cryptic, bulky editor and WEB to be a slow, inefficient way of programming. I found myself spending the same amount of time planning the program and developing pseudocode, and taking longer to type the program into the machine. Though I recognized then the value of WEB programming for maintenance and readability, it annoyed me.

3. *What are your current reactions/acceptances/feelings/etc. to WEB? Why?* I recognize the importance of WEB, but I still do not like it. I am glad that I am learning it, but I wish it was never invented. I would probably feel better about WEB if I had not already learned to program; it would be more successful with first time programmers. Since I already think in terms of "what should happen next in this program" the quality of WEB that allows the program to be assembled from bits scattered throughout the document is of little use to me. I tried to write like that, but I had so many compilation errors (variables

declared too many times, etc.) that I abandoned this style. I now write the comments, then write the whole program, in the order that I would enter it into a Turbo editor. I still think that the EMACS editor is very inefficient and quite obnoxious, but there have to be *some* programs for computer scientists of my generation to improve. Another frustration I have with WEB is that I have to program in the lab; I do not know of a way to use my machine at home. This would be less of a problem if I could somehow access EMACS over my modem, but I do not know that this is available. I realize that, as a C.S. major, I need to get used to cryptic, awkward systems, but WEB programming is still a challenge.

§10 Class Evaluation

QUESTION AND ANSWER SESSION 1

1. Question and Answer Session.

2.

3. CPSC 110-H

4.

5. What did you expect from this class and why?

6. Actually, I didn't know what to expect from this class. Because it was the first semester of my freshman year, I couldn't make any reasonable assumption as to what was going to happen. The whole arena of college life and academics was a complete mystery to me. In spite of older friends talking to me about college, something was always lost in the translation. It seemed more sensible to enter this new world without any preconceived notions. Throw in the fact that this was an honors course, and expectations lose even more of their significance. Who can say what twists and turns an honors course will take? Because of all this uncertainty, I decided to not develop expectations where they would only serve to confuse the issue.

7. What was your reaction to Web programming initially and why?

8. At first I couldn't understand why we needed to know Web programming. It seemed unimportant. I didn't realize the significance of program documentation. I thought that as long as I wrote a program code that worked, there wasn't much else to know. Chalk it up to ignorance on my part. But we were told it would definitely help us in the long run, and that was enough to make me follow through. In the beginning, the ideas behind Web programming were very confusing to me. It took me a long time to realize that it was a combination of documentation and code. Up to this point, I'd had extremely little experience in documentation. The entire concept of writing down in prose form what a program was supposed to accomplish was relatively new to me. Because of that, I was probably a little intimidated. And again, I couldn't do anything but trust in the instructors and try to understand what was going on. That's typically what a student does to learn anything.

9. What are your current reactions to Web now and why?

10. I see now how important Web, or any documentation programming, can be. I also understand the importance of clear design tactics to facilitate maintenance. The actual code seems to be of somewhat lesser priority. I see that the whole idea of design and documentation is developed to make code formation less of a hassle. Of course, I'm still at least a little intimidated by Web. It seems that the more I find out about it, the more there is that I don't know. It gets frustrating at times, but I'm starting to see that pattern in everything I'm learning. So, I guess it's just a natural progression of learning. Finally, Web programming is definitely a useful tool, and I'm glad I had the opportunity to be introduced to it before going further in my computer science career.

**PARTICIPATION ASSIGNMENT I
CLASS EVALUATION**

1) WHAT DID YOU EXPECT FROM THIS CLASS AND WHY?

I expected this class to be much more PASCAL intensive, mainly due to the description of the course that was presented during Freshman orientation. The title of the course is 'Programming I,' thus leading to the conclusion that the primary thrust of the course would be programming.

2) WHAT WAS YOUR REACTION AND ACCEPTANCE TO WEB PROGRAMMING AND WHY?

My initial reaction was disgust. The course seemed to be going in a different direction than I had hoped, and I honestly did not like it. WEB style programming seemed to be a time consuming and wasteful process that was nothing more than programming documentation overkill.

3) WHAT ARE YOUR CURRENT REACTIONS AND ACCEPTANCE TO WEB NOW AND WHY?

I am now comfortable with the WEB process. After working (many hours!) with the programming style, I feel that I can be fairly efficient with WEB. The TEX experience has its obvious advantages, and the emacs editor experience will be useful in the future. However, the style in which the programs are documented still has a wordy and

cumbersome feel. I was taught to document a program with remark statements in a way that is concise and descriptive. The WEB style programming is a departure from this in-program style that is still uncomfortable for me.

1. *What did you expect from this class and why?* Actually, I expected computer science to be my worst class this semester. The first day of class scared me half to death. Since I knew nothing about Pascal or programming (and assumed everyone else did), I was somewhat paranoid in general. When we started talking about WEB and T_EX, literate programming and editing environments, etc. I began to ponder dropping the class and losing my honors credit.

2. *What was your reaction/acceptance to WEB programming initially?* As was stated in the last response, my very first reaction to WEB was to run as far away from the computer science department as possible. After the first lab, I started feeling a little bit better because no one else in the class understood WEB either. One day in the middle of the second lab, something sort of hit me and it all kind of fit together--the way modules work and what the different commands do, etc. I understood why WEB was used to maintain programs mainly because when I looked at a turbo written Pascal program, it just made absolutely no sense to me.

3. *What is your current reaction/acceptance to WEB?* Now, I'm pretty much enjoying the class. It seems easier to me to work with a WEB--the design process work out the programming problem before you code, and the actual WEB file helps me keep my code segments separated and organized. The only thing I really like better about regular Pascal is declaring all the variables, constants, and types together (which is fine because it can still be done in a WEB). I might feel differently if I had known Pascal before taking this class, but I'm fairly comfortable with the WEB format and simple T_EX and emacs commands at the moment.

1. Because it is called an 'Honors' class, I expected the process of teaching Turbo Pascal to be accelerated. I also expected it to be about 90% lab, because that is where you actually learn how to write a program.

2. When I was first introduced to emacs, I felt curious, because it was something new, and also wonderful, because I thought we were to learn programming in Turbo Pascal, and not editing a textfile using emacs.

3. Now, since I have become a little familiar with emacs, I believe that it might be a good idea to acquaint students to emacs, if there is a good chance of them needing it as they progress. I believe that having us edit our Turbo Pascal programs and documents through emacs is inappropriate for this class, since much of our time to actually 'write a program' is lost on emacs, weave, tex, tangle etc.

VITA

Deborah Lynn Byrum Dunn was born on July 31, 1961, in Washington, D. C., to Burton and Aura Byrum. She received a B. B. A. in Business Data Processing in 1979 from Stephen F. Austin State University. Upon completing her bachelors degree, she was employed by Mobil Oil Corporation. She received an M. S. in Computer Science at Stephen F. Austin State University in 1989. She began work on her Ph. D. at Texas A&M University in June of 1989. Since that time she has worked as a graduate assistant teaching and is currently a lecturer in the Computer Science Department. She may be reached at the Department of Computer Science, Texas A&M University, College Station, Texas, 77843-3112.