

Portland State University

PDXScholar

---

Dissertations and Theses

Dissertations and Theses

---

3-5-1993

# Comprehension of Literate Programs by Novice and Intermediate Programmers

Christopher Forrest Bertholf  
*Portland State University*

Follow this and additional works at: [https://pdxscholar.library.pdx.edu/open\\_access\\_etds](https://pdxscholar.library.pdx.edu/open_access_etds)



Part of the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

---

## Recommended Citation

Bertholf, Christopher Forrest, "Comprehension of Literate Programs by Novice and Intermediate Programmers" (1993). *Dissertations and Theses*. Paper 4572.

<https://doi.org/10.15760/etd.6456>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: [pdxscholar@pdx.edu](mailto:pdxscholar@pdx.edu).

AN ABSTRACT OF THE THESIS OF Christopher Forrest Bertholf for the Master of Science in Computer Science presented March 5, 1993.

Title: Comprehension of Literate Programs by Novice and Intermediate Programmers.

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

[REDACTED]

Jeanne Scholtz, Chair

[REDACTED]

Maria Balogh

[REDACTED]

Leonard Shapiro

[REDACTED]

Beatrice Oshika

The studies reported herein compare comprehension of Lit style literate programs to that of traditional modular programs documented by embedded comments. Novice and intermediate programmers participated in three experiments designed to determine the comprehensibility of literate programs

written using a language-independent system for abstraction-oriented literate programming compared with programs written using traditional modular programming techniques (traditional modular programs). Programs were written in either the C or FORTRAN programming language. Half of the subjects in each group received a literate program, while the other half received a traditional modular program with embedded documentation. Subjects received a problem specification, input and output specifications, and a language reference for use in the study. Subjects were asked to perform a program maintenance task (complete an incomplete program). The maintenance task was used as a measure of comprehension; it simulates an actual task in the software engineering industry that requires program comprehension in order to be completed. The elapsed time to effect a solution was recorded. The completed programs were judged as correct, functionally correct with syntax errors, or incorrect; several reconstructive program comprehension measures were also collected and analyzed. The clear overall result was that subjects using the literate programs found a solution (correct or functionally correct with syntax errors) more often than did subjects using the traditional modular programs with embedded comments. In fact, none of the subjects in this study who modified the traditional programs were able to effect a solution that was totally correct, nor even one that was functionally correct with syntax errors.

**COMPREHENSION OF LITERATE PROGRAMS**  
**BY**  
**NOVICE AND INTERMEDIATE PROGRAMMERS**

**by**  
**CHRISTOPHER FORREST BERTHOLF**

**A thesis submitted in partial fulfillment of the  
requirements for the degree of**

**MASTER OF SCIENCE**  
**in**  
**COMPUTER SCIENCE**

**Portland State University**  
**1993**

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Christopher  
Forrest Bertholf presented March 5, 1993.



Jeanne Scholtz, Chair



Maria Balogh




Leonard Shapiro



Beatrice Oshika

APPROVED:



Leonard Shapiro, Chair, Department of Computer Science



Roy W. Koch, Vice Provost for Graduate Studies and Research

## ACKNOWLEDGEMENTS

The following people were instrumental, in many different ways, in the completion of this study.

My thanks to Maria Balogh, for substantial support and encouragement. Many thanks to Jean Scholtz for inspiration, for being such an excellent research resource, and for her patience with my passion for perfection. My appreciation is also extended to Donald Knuth, without whom I would have had no programming paradigm to refine and test. I would also like to thank Mike Kephardt, who introduced me to literate programming and gave me a reference implementation of CWEB, a literate programming preprocessor in the style of WEAVE and TANGLE that he designed and implemented in 1985. Andy McKnight, my co-worker and friend, spent countless hours developing a WordPerfect macro set that allowed Lit to be used with WordPerfect version 5.1, and he also helped me with my pilot studies; I can't thank Andy enough for all of his help. I would like to thank all of the students in my 'Introduction to FORTRAN' class, and all of the students at Portland State University who beta-tested Lit, without whom many of the revisions to the Lit system would never have been suggested or implemented. I would also like to thank Wes Brenner for all of the help he gave me with statistical theory; it helped shape the way the studies were designed and analyzed. My thanks also go to Beatrice Oshika;

her comments in the early stage of writing my thesis helped focus and structure my thoughts and the organization of the thesis. Many thanks to Leonard Shapiro for serving on the Thesis Committee and for his insightful and probing questions about the implications of the research reported in my thesis. My deepest gratitude is extended to Leslie Hammer, whose careful contemplation and thoughtful discussions of my studies, help with post-hoc measures, and inordinate patience were all instrumental in my being able to complete this thesis. Finally, I would like to thank Heidi Bertholf, my mom, for all of the support she has provided throughout my college education.

Thank you all. I couldn't have done it without you!

## TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS .....	iii
LIST OF TABLES .....	viii
LIST OF FIGURES .....	x
CHAPTER	
I INTRODUCTION .....	1
The Problem .....	3
Requirements Analysis .....	5
Program Design .....	6
Program Coding .....	7
Program Maintenance .....	8
Modularity .....	11
The Role of Program Documentation .....	16
Expert Programming Knowledge .....	19
Program Comprehension .....	22
Program Testing .....	29
II LITERATE PROGRAMMING .....	34
The Lit System .....	36
Differences Between Traditional Programs and Literate Programs .....	41
How Literate Programs Enhance Program Comprehension .....	42
How the Lit System Works .....	46
III EMPIRICAL STUDIES OF PROGRAM COMPREHENSION .....	54
Experimental Context .....	54



	General Methodology .....	55
	Subject Selection .....	55
	Materials .....	56
	Procedure .....	59
	Establishing Time Constraints for the Studies .....	61
	Measures .....	61
	Research Hypothesis .....	64
IV	EXPERIMENT ONE .....	66
	Subjects .....	66
	Materials .....	67
	Procedure .....	67
	Results .....	68
	Analysis of Subject's Subjective Data .....	70
	Discussion .....	75
V	EXPERIMENT TWO .....	79
	Subjects .....	79
	Materials .....	80
	Procedure .....	80
	Results .....	81
	Analysis of Subject's Subjective Data .....	83
	Discussion .....	89
VI	EXPERIMENT THREE .....	92
	Subjects .....	92
	Materials .....	93

	vii
Procedure .....	93
Results .....	94
Analysis of Subject's Subjective Data .....	97
Discussion .....	103
VII CONCLUSIONS .....	107
General Principles for Assisting Program Comprehension	107
VIII DISCUSSION .....	110
IX FUTURE DIRECTIONS .....	115
REFERENCES .....	117
APPENDICES	
A EXPERIMENT CONSENT FORM .....	122
B INTRODUCTION TO THE EXPERIMENT .....	125
C POST-EXPERIMENT QUESTIONNAIRE .....	127
D EXAMPLE LITERATE PROGRAM (FROM EXPERIMENT 1) ...	131
E EXAMPLE TRADITIONAL PROGRAM (FROM EXPERIMENT 1)	144
F EXPERIMENT 1 SPECIFICATIONS .....	149
G EXPERIMENT 3 SPECIFICATIONS .....	153
H EXPERIMENT 2 SPECIFICATIONS .....	156
I EXAMPLE PROGRAM SOLUTION .....	159
J EXAMPLE LITERATE PROGRAM DOCUMENT (FROM EXPERIMENT ONE) .....	161
K WHAT THE LIT SYSTEM DOES FOR THE USER .....	169
L SPECIFIC FORMATTING RULES USED BY LIT .....	181

## LIST OF TABLES

TABLE	PAGE
I      Group Performance Percentages for Experiment One . . . . .	69
II     Experiment One: Perceptions of Subjects Given Literate Programs . . . . .	71
III    Experiment One: Perceptions of Subjects Given Literate Programs Who Found a Solution . . . . .	72
IV    Experiment One: Perceptions of Subjects Given Literate Programs Who Did Not Find a Solution . . . . .	73
V     Experiment One: Perceptions of Subjects Given Standard Programs . . . . .	74
VI    Experiment One: Subjects' Evaluation of Experimental Materials . . . . .	74
VII   Group Performance Percentages for Experiment Two . . . . .	82
VIII   Experiment Two: Perceptions of Subjects Given Literate Programs . . . . .	84
IX    Experiment Two: Perceptions of Subjects Given Literate Programs Who Found a Solution . . . . .	85
X     Experiment Two: Perceptions of Subjects Given Literate Programs Who Did Not Find a Solution . . . . .	86
XI    Experiment Two: Perceptions of Subjects Given Standard Programs . . . . .	87
XII   Experiment Two: Subjects' Evaluation of Experimental Materials . . . . .	88
XIII   Group Performance Percentages for Experiment Three . . . . .	95

XIV	Experiment Three: Perceptions of Subjects Given Literate Programs . . . . .	97
XV	Experiment Three: Perceptions of Subjects Given Literate Programs Who Found a Solution . . . . .	98
XVI	Experiment Three: Perceptions of Subjects Given Literate Programs Who Did Not Find a Solution . . . . .	99
XVII	Experiment Three: Perceptions of Subjects Given Standard Programs . . . . .	101
XVIII	Experiment Three: Subjects' Evaluation of Experimental Materials . . . . .	102

## LIST OF FIGURES

FIGURE	PAGE
1. Traditional Program Fragment (Written by a Professional Software Programmer) . . . . .	45
2. Traditional Program Fragment . . . . .	46
3. Literate Program Fragment . . . . .	47
4. How the Lit System Works . . . . .	48
5. The Lit System Interface: The Main Menu . . . . .	52
6. Invoking Lit . . . . .	172
7. The <i>Edit</i> Option . . . . .	173
8. The <i>Compile</i> Option . . . . .	174
9. The <i>Format</i> Option . . . . .	175
10. The <i>View</i> Option . . . . .	176
11. The <i>Print</i> Option . . . . .	177
12. The <i>Debug</i> Option . . . . .	178
13. The <i>Run</i> Option . . . . .	178
14. The <i>Goto</i> Option . . . . .	179
15. The <i>Exit</i> Option . . . . .	180

## CHAPTER I

### INTRODUCTION

This study compares comprehension of literate programs with that of traditional modular programs. Literate programming (Knuth, 1984) enhances a computer program by incorporating program text into a comprehensive design document.

Virtually no research into the efficacy of literate programming as an alternative programming paradigm has been done since Knuth introduced the WEB system in 1984. In this respect, the present study is completely new work. The present goal of this researcher is to identify the elements of the software engineering process which substantially enhance the comprehensibility of computer programs. It is hypothesized that by enhancing program comprehensibility, there are resultant gains in the productivity of computer programmers, and most importantly, resultant gains in the maintainability of computer programs.

The approach the researcher has taken with respect to enhancing the comprehension of computer programs is to emphasize the use of elements in the design and maintenance processes which have been shown to assist the programmer with program comprehension. One idea that has been overlooked for many years is Knuth's literate programming. Knuth's concept has great

intuitive appeal, fits in well with a multi-disciplinary approach to automating portions of the software engineering process, and can be adapted easily to the incorporation of empirically derived principles of program comprehension. It is interesting that no conclusive comprehension studies have been done in the area of literate programming since Knuth introduced it over 8 years ago.

Ultimately, the research focus is a multi-disciplinary approach to software engineering that utilizes basic and applied research in psychology, software engineering, and empirical studies of computer programmers to provide a unified system for Computer Assisted Software Engineering (CASE). The desired result of the research is to develop a tool-integration framework and an integrated set of program development and maintenance tools that are platform, operating system, programming language, and text formatter independent. The difference between this approach to CASE and the traditional approach to CASE is the emphasis on using principles that have been shown (empirically) to assist in program design, coding, testing, debugging, implementation, and maintenance. This study is the first in a series of studies that are designed to address each of these areas. In a larger sense, it is the objective of this study to contribute, through empirical investigation, to the understanding of one issue (enhancing program comprehension) that affects programmer productivity. It is my hope that this study can be used to help provide a basis for doing further work in the areas of software engineering that are critical to programmer productivity. It is also my hope that the results

of these studies will encourage others to take a first or second look at the benefits offered by the literate programming paradigm.

Outlined below is an introduction to some of the problems of software engineering, computer programming in general, and a description of a system called Lit, based on empirical principles, designed to address these problems. The Lit system was used to create the programs that are the subject of this study. Three experiments that evaluate the efficacy of the literate programming paradigm, as it relates to program comprehension, are presented in detail and discussed. Comprehension is evaluated using several measures: traditional measures; modified CLOZE tests (Entin, 1984; Taylor, 1953), and constructive measures that are more indicative of the actual comprehension required of a programmer to modify a computer program. Subjective measures gathered from a post-test questionnaire are also reported and analyzed. Finally, the implications of the three experiments are discussed, and future directions for related research are proposed.

## THE PROBLEM

Software engineering is an extremely complex task. The basic components of software engineering are *analysis, design, coding, testing, documentation, and maintenance*. The phases of design, coding, testing, maintenance, and documentation, take up the largest percentage of the time spent in the process. Each of these phases is very complex and time



consuming, and requires great attention to detail. In the design phase, the designer must be able to create an abstract design, often with minimal attention to the computer language or languages that will be used to implement the task. At the same time, pragmatic concerns dictate that the design cannot be too difficult to implement given the constraints of the hardware and software that are available. Thus, to a certain degree, the designer must take into account the language, or at least the type of language that will be used to implement the software. Similarly, the coding phase requires that the programmer(s) be able to understand both the computer-related concepts and the task domain-related concepts in order to form a global model of program design which will be used to implement a programmatic solution. Testing requires that the problem be specified in such a way that the program can be evaluated for correctness. Testing is especially frustrating because no matter how well a program is tested, it is generally impossible to prove the correctness of a complex program. No matter how much testing is done, the most that can be hoped for is confidence in the software; in general, testing does not prove correctness but it does give anecdotal evidence of fitness for a particular purpose. Testing (by the programmer) does give the programmer an indication of how well the implemented solution meets the requirements of the software specification, and assists the programmer in solving algorithmic problems. The documentation, although time consuming to produce, maintain, and read, is the only link a new maintenance programmer has with the original design. Without

proper and thorough documentation, the design must be inferred from the source code and any other documents about the software (which may be outdated or unavailable).

There is very little assistance available for the processes of analysis, design, programming, testing, and maintenance of computer software. It is hypothesized that the entire process can be significantly aided if each of these processes can be assisted mechanically, and all the information required to specify, code, and test computer software is included in the program document, . The idea is to aid the programmer by methodically researching the processes that underlie the complex task of programming, and to design tools that enable such processes to occur efficiently, effectively, and economically. To understand how this can be done, a deeper look at some of the processes involved in software engineering is warranted.

### Requirements Analysis

The requirements analysis is an intensive process. There are two phases of the requirements analysis: user requirements analysis and resource requirements analysis. User requirements analysis is the portion of the software engineering cycle where the user driven software specification is designed. The resource requirements analysis is performed by the software engineers based on the user requirements analysis. Basically, the resources that are available to implement a software application must be determined including people, time, machinery, software, computers, and funding. Project standards and

conventions must be identified and/or developed. A development schedule must be implemented, including a software development plan and a quality assurance plan. A configuration management plan must be put in place to assure administrative control of the design and implementation process. A requirements document must be drafted, and a functional specification for the software must be developed. Data flow, data structures, and allocation of functions in the functional specification to processes in the software is the final step in the analysis process.

Generally, none of this documentation is included in the source code of a computer program or system of programs. These documents tend to be external documentation. Often, as a program evolves, these documents no longer reflect the actual state of the program. The information is out-of-date, and often multiple addenda or errata, in yet another external document, describe the actual state of the software.

### Program Design

Another process involved in computer programming is program design, which begins where the functional description leaves off. There is a small overlap in the analysis and design phases of development, where the data flow, data structures, and allocation of functions to processes in the software need to take into consideration certain pragmatic concerns such as the programming language to be used, the hardware constraints, and algorithmic complexity constraints. Eventually, a detailed design of the program is developed, usually

in conjunction with a plan for testing the completed programs. There are often many hierarchical designs, data-flow and control-flow diagrams, data-structure diagrams and functionality pseudo-code that are created during this phase of development. None of this information tends to reside in the program source code; it is usually part of documentation external to the program. Often, this information is also out-of-date with an evolving program; the information is up-to-date for the initial implementation but gets out-of-date as the programmer(s) spend more and more time in the maintenance cycle performing adaptive and perfective maintenance.

### Program Coding

The process of program coding is not too difficult if the programmer is also the analyst and designer. With large software systems, this is usually not the case; often programmers who were not involved in the software design perform the coding. In the best case, there are few flaws in design methodology, and coding fairly accurately reflects the intended design. In the average case, there are many changes to design methodology, data-structures, data-flow, and even control-flow. Many of these changes are made by the programmer, and do not appear in the design document, although they may appear in an erratum or addendum to the document. As problems are encountered, they are solved systematically, but very little of the knowledge used to solve the problems (underlying structures, assumptions, reasons that a particular coding was chosen from a set of acceptable alternatives, etc.) is

included in the program document. Usually the only reliable description of the program's functionality and method of implementation is the program source code; other design documents are incomplete, out-of-date, or simply do not contain the correct information.

### Program Maintenance

After an application has been implemented, the largest portion of the software engineering cycle is program maintenance (e.g., fixing errors, adding functionality, optimizing, etc.). It is estimated that between 40 to 75 percent of the development cycle is devoted to performing maintenance tasks (Zehr, 1992). Larger and more complex software application programs take more time than smaller, less complex software applications. Although pinning down the exact percentage of time spent performing program maintenance is difficult, most experts agree that the largest portion of the development process is, in fact, maintenance, and that the percentage of time spent doing maintenance is very high. Traditionally, maintenance is broken down into three types: corrective, perfective, and adaptive (Bendifallah & Scacchi, 1987).

Unfortunately, it is not well understood how programmers' comprehension strategies adapt to the changes in maintenance requirements, or how much of each type of maintenance is performed in the software development cycle.

What is known, however, is that many strategies and techniques are used in all three types of maintenance activities and that program comprehension is one of the most time consuming portions of the maintenance task. In fact, the major

difficulty cited by maintenance programmers is understanding the intent and style of another programmer's source code (Fjeldstad & Hamlen, 1983). It is estimated that maintenance programmers spend between 47 and 62 percent of their time trying to comprehend code (Parikh & Zvegintzov, 1983). A simple calculation shows the range of time spent by maintenance programmers attempting to comprehend program code is somewhere between 19 and 47 percent of the software development cycle. Obviously, if this time could be significantly reduced, the cost of the development cycle would be reduced as well.

Most large software engineering projects suggest that a program maintenance manual be developed (Softky, 1983). The problem is that the document is rarely produced or, if it is produced, it is inadequate for solving many of the maintenance problems that arise. Often this is due to maintenance changes in the software over time that do not get added to the documentation in the program maintenance manual. The program maintenance manual is usually not revised after product delivery; as the product evolves, the manual tends to get out of date with the software, and eventually is near useless in assisting the maintenance programmer with the finding and fixing of software bugs. This tends to make the maintenance programmer disregard the manual altogether. If the program maintenance manual were included as a part of the program source code, it would be easier to keep the manual up to date, easier

to use the manual, and would be more likely to be trusted as an aid in problem diagnosis and repair.

In many instances, the only reliable description of a program is the source code itself. Thus much of the effort devoted to making programs more understandable has been in the area of typographic style changes to program source code. Until recently, empirical studies on the contribution of typographic style to program understandability have been inconclusive (Love, 1977; Miara, Musselman, Navarro, & Shneiderman, 1983; Shneiderman & McKay, 1980). The disagreements about the importance of typographic style prompted Sheil (1981) to note that the existence of both negative and positive results suggested searching for a set of principles indicative of how and when formatting techniques could be used to improve program comprehension. Several researchers have recently explored effects of style in program formatting. Baecker (1988) developed a framework for "program visualization", based on a set of principles drawn from graphics design, for use with high resolution bitmapped displays. Oman and Cook (1990) identified several principles of typographic style that are consistent and compatible with the results of program comprehension studies; they show how a book-style program format significantly aids program comprehension and reduces software maintenance effort.

## Modularity

Another important consideration in the design and implementation of computer programs is modularity. Over the past 15 years many changes have taken place in how computer programming is taught. The computer has become more powerful; address space is larger, the number of instructions that can be executed per second has increased greatly, and direct-access mass storage use has increased as the price per unit of storage for such devices have dropped. Thus, the emphasis on machine efficiency has shifted to human efficiency. Cryptic, efficient, 'spaghetti code' is no longer the norm; it has been replaced by modular programs with some (albeit small) attention to human readability. Unstructured non-modular approaches to programming have been replaced by modular highly-structured approaches.

Gauthier and Ponto described the philosophy of modular programming as:

A well defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching. (Gauthier & Ponto, 1970; p. 180)

Many claims have been made for highly-structured techniques, including: shortened program development time, ease of modification and maintenance,



lower incidence of 'bugs', ease of testing, and higher reliability. Most of these claims are in dispute; many have not been evaluated empirically, or the results of empirical investigation has been inconclusive. Most of the claims are supported by substantial anecdotal evidence, case histories, and offer a favorable intuitive appeal. It makes sense, psychologically, to theorize that limiting the amount of information (cognitive load) the programmer must consider simultaneously while developing or maintaining a program should yield improvement in these areas. Because programmers use modularity to try to limit the amount of information that must be considered simultaneously, modularity should assist in obtaining these benefits. If all of the task domain and implementation-specific details are provided explicitly in the program document, this should increase the benefits obtained by the programmer. Because inputs and outputs of each module are well defined, inclusion of the input-output specifications in the program documentation can be used to help the programmer debug program modules.

Theoretical support for the above ideas comes from complexity theory; complexity theory says that the chance of survival for a complex system is increased if the system is composed of a hierarchy of subsystems which are loosely coupled, but only if each subsystem is internally cohesive. The simpler the subsystems and the smaller the interactions between them, the easier it is to understand the system as a whole, and the better its chances are for longevity and reliability.

Application support for the above ideas is embodied in current high-level languages, macro assemblers, and separate compilation tools. Languages that allow modules to be developed independently of each other, and provide for separate recompilation or reassembly of a module without recompilation or reassembly of the whole system, are thought to be extremely valuable aids to program developers and maintenance programmers.

Parnas (1972) suggested that modular program design would be most effective when it was used to implement information hiding. The suggestion was an intuitive suggestion, based on experience with computer programming, and was not based on any empirical investigation into the effectiveness of modular program design. Empirical support for the utility of using modular, structured program design can be found in a study by Korson and Vaishnavi (1986). They found that modular programs are faster to modify than non-modular, but otherwise equivalent, versions of the same program. The difference is detectable only when one of several conditions hold: (1) modularity has been used to implement information hiding (as suggested by Parnas (1972)); (2) existing modules in the program perform generic operations which can be used to implement modifications; or (3) when a significant understanding of the existing code is required to make a modification and the modification to be made is substantial.

Abstraction Capabilities and Program Modularity. Abstraction is the process of separating program components such that they can be considered

independently. Programmers tend to use abstraction as a tool to help focus the development process of a computer program. It is the process by which good, clean, program modularity is achieved. The programmer tends to define an overall algorithm for solving a problem; a good algorithm has many component parts which can be considered separately from the rest of the system, as long as the interface with the other program components is well understood. The interface usually takes the form of well-defined input and output for the module that allows the internal operations and structure of the module's local data to be treated separately from the rest of the program modules' structure and data. If the amount of data passed through the interface (interface width) is small and the interactions with other modules are well defined, debugging, testing, and maintenance are thought to be significantly improved.

Abstraction capabilities in a programming system also allow the programmer to develop the program algorithms and associated documentation in any order, free of the constraints of the underlying programming language. This allows the programmer to program in a more natural order, considering only the details the programmer wishes to concentrate on, and leaving other details to be expanded and finalized later. Breaking the detailed expression of the program up in such a way reduces the cognitive load on the programmer.

Take for example a programmer who wishes to write a simple language compiler. The programmer might wish to begin with an algorithm that looks something like:

Algorithm Compiler

- Perform lexical analysis and report syntax errors.
- Perform parsing and quadruple generation.
- Perform code generation.
- Perform optimization.

End Algorithm Compiler

The algorithm accurately describes the process, but not the details of the different operations. Later, each of the operations can be expanded in detail. Often such expansions will result in more abstractions, each of which the programmer may wish to treat separately. The programmer is able to create freely the basic structures and operations required to perform a task, without worrying excessively about language and/or implementation-dependent details. When the author is ready to expand a section, it is defined, the code is written, and it is inserted into the program. Having an automated program design and maintenance tool to assist with the abstraction process may assist the programmer substantially. If the tool also enforces a presentation paradigm and assists the programmer in documenting and testing the code, it could also be an invaluable aid for debugging, testing and maintenance.

Abstraction capabilities are a large part of the newest programming paradigm, object-oriented programming. Object-oriented programming allows

the programmer to abstract functionally independent operations and data structures into what is known as an object. Data encapsulation, and a well-defined interface between the objects (message passing), are thought to assist in the design, testing, maintenance, and reusability of the objects. Unfortunately, in practice, the object-oriented programming paradigm has not been as useful as its proponents have suggested it should be. Empirical research in the area of object-oriented programming (Kim, J. & Lerch, F., 1992; Rosson, M. B. & Alpert, S. R., 1990; Rosson, M.B. & Gold, E., 1989) is only now beginning to uncover the shortcomings and actual benefits of object-oriented programming. What is known is that there are no programs that can be written in an object oriented programming language that cannot be implemented as efficiently and securely in a traditional high-level programming language. (Early versions of CFRONT, a language translator for the object-oriented language C++, produced standard, procedural, C code as output.) The Lit system used to produce the programs that are the subject of this study can be used with object-oriented programming languages (such as C++), but I have chosen to concentrate on standard procedural languages because they are still the most widely used of all languages.

### The Role of Program Documentation

The purpose of program documentation is to explain to a human reader the way in which a program works, so that it can be successfully adapted after it goes into service, either to meet the changing requirements of its users, to improve it in the light of increased knowledge, or just to remove latent errors and

oversights. The view that documentation is something that is added to a program after it has been commissioned seems to be wrong in principle and counterproductive in practice. Instead, documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear self-documenting code, and even perhaps develop and display a pleasant style of writing. *The readability of programs is immeasurably more important than their writability.* [Emphasis added] (Hoare, 1973; p. 4)

Hoare (1973) accurately describes what documentation is, and how it should be incorporated into program development. Unfortunately, there are no programming languages (to date) that possess the qualities that promote the use of good program documentation. Program documentation has traditionally been a neglected portion of the design process. Hoare (1973) writes: "The objective of readability by human beings has sometimes been denied in favor of readability by a machine; and sometimes even been denied in favor of abbreviation of writing, achieved by a wealth of default conventions and implicit assumptions" (Hoare, 1973; p. 11). In practice, documentation for programs may be inaccurate, out-of-date, or may not be present. One of the reasons for this is that, in the past, many programmers subscribed to the idea that "... it is very unlikely that the output of a computer [language compiler] will ever be more readable than its input, except in such trivial but important aspects as improved indentation" (Hoare, 1973; p. 11). I believe that the output of a language compiler, or programming system, can be measurably more readable than its input.

Currently, there is a trend to provide improved program documentation as an integral part of any complete programming methodology. Traditional structured programming methodologies have de-emphasized the role of program documentation, and emphasized the role of modular style programming. The emphasis on modular programming rests on the idea that, if modules are small enough, their meaning and usage can be easily gleaned by reading the source code.

Traditional modular programming is readable by a compiler (by definition) but is not required to be comprehensible to the human reader. Traditional programs tend to be written in as compressed a form as possible, often without embedded comments of any kind. Often, the emphasis is on optimized program code, to the extent that the solution, as implemented, requires in-depth knowledge of the language, the computer and/or operating system characteristics, and the task domain of the application to even begin to understand the solution that is present in the source code. In fact, some languages lend themselves to cryptic expression so well that contests for the most functional and cryptic programs are held annually (e.g. the annual Obfuscated C contest). Although the power of expression is important in a language, it should not become the cornerstone of a language that uses cryptic syntax. Readability and understandability, the human components of computer programming, should be emphasized and the use of the cryptic features of the language should be de-emphasized, except where such usage can be

adequately documented; if a cryptic, difficult to understand, advanced concept or bizarre language feature is used to implement a function in a design, it should be documented extensively. This not only helps maintainers of the program, but anyone wishing to port the program to another, possibly incompatible, operating environment.

### Expert Programming Knowledge

Soloway and Ehrlich (1984) showed that expert programmers use two types of programming knowledge: 1) Programming plans which are generic program fragments that represent stereotypic action sequences in programming, and 2) rules of programming discourse that capture the conventions in programming and govern the composition of the plans into programs.

This finding is consistent with findings in other domains of experts' ability to organize and structure knowledge. For example, Chase and Simon (1973), building on the work of de Groot (1965), showed that Master chess players are able to remember the board positions of chess pieces better than non-Masters when the chess board is organized in some meaningful configuration. They also showed that when the pieces were placed at random on the board in what amounted to a non-meaningful configuration, the Masters had no statistically significant advantage over the non-Masters in recalling board positions of chess pieces. The authors attributed this result to the Masters' higher level of knowledge about chess. Similar findings in the domain of electronic circuitry



(Egan & Schwartz, 1979) are also consistent with psychological theory; people develop chunks that represent functional units in their respective domains.

These chunks are used to classify and decompose the new problems.

Apparently, experts have and use specific and elaborate plans that novices can not use because they have not been developed fully. This is consistent with the notion of schemas as generic knowledge structures that guide the interpretation, expectations, and inferences that are made in the comprehension process. Because it is thought that schemas are developed through experience, it makes sense that novices would not have the same underlying schemas as experts in most domains.

Shneiderman (1976), Adelson (1981), and McKeithen, Reitman, Rueter, and Hirtle (1981) have replicated the Chase and Simon (1973) experiments in the domain of computer programming. All of the experiments have shown that expert programmers can remember programs better than novices when the programs have some meaningful structure; but the experts do no better than the novices when the programs are made up of random lines of code. Again, the theory is that the expert programmers are better able to use their higher level knowledge to encode the presented programs into meaningful chunks for easier recall.

In this researcher's experience, expert programmers tend to be the programmers who are assigned to new development and intermediate programmers (e.g., Bachelor of Science in Computer Science) tend to be

assigned to maintenance tasks. Because the maintenance programmer usually is not an expert, program comprehension assistance needs to be provided. The maintenance programmer does not know the original design or why certain decisions were made in the design, but it is his/her job to alter in some way the program's functionality.

Novice and intermediate solutions are usually data-driven or goal-driven strategies that yield problem decompositions which tend to elaborate to a solution that is inferior to an expert solution for the same problem (Adelson, B., Littman, D., Ehrlich, K., Black, J. & Soloway, E. 1985; Ehrlich, K. & Soloway, E., 1984;). It is important to minimize the effects of any factor or factors that promote the usage of the inferior strategies. One can conjecture that to do so would actually help assure that such practices do not become entrenched in a programmer's design knowledge, leading to regular use of inferior problem solving strategies by the maintenance programmer.

How can the expert pass on some of the implicit knowledge from the original design to the maintenance programmer, such that the maintenance programmer can see it from an 'expert' point of view? One possible answer is to teach programmers structured program design, with most of the attention being given to the development of abstraction skills (Ratcliffe & Siddiqi, 1985). It is suggested here that the system used for program development, debugging, and maintenance purposes, should support abstraction oriented programming. If such systems were utilized in education and industry, it is

possible that expert programming knowledge could be transferred to novices much more easily, and the resultant productivity increase would make up for any of the up-front costs such as increased development time and additional educational support. Soloway, Bonar, and Ehrlich (1983) suggest that insight in this area could be drawn from looking at the cognitive load placed on the programmer by syntax and semantic demands of programming languages. I further suggest that the cognitive load placed on the programmer by having to hypothesize about the program's functionality in the absence of proper documentation is a confounding problem. Programming is an extremely demanding skill and the comprehension process is only complicated by not removing as many cognitive hurdles as possible.

### Program Comprehension

Obviously, when documentation is not viewed as critically important, the comprehensibility of most resulting computer programs is not high. In fact, there have been many experiments that attempt to analyze out how computer programmers comprehend computer programs (Adelson, 1981; Basili & Mills, 1982; Brooks, 1983; Ehrlich & Soloway, 1984; Entin, 1984; Konneman & Robertson, 1991; Littman, et.al., 1986; Pennington, 1987; Ratcliffe & Siddiqi, 1985; Soloway, et.al. 1983; Soloway & Ehrlich 1984). Program maintenance tasks involving large and/or complex programs are not simple, even for an expert. Many of the principles of cognitive psychology, human factors, typography, and the results of empirical studies of programmers have been

successfully applied to several aspects of understanding computer programmer comprehension strategies. Yet, computer programming remains a highly difficult, and sometimes daunting process. Many researchers have suggested the difficulty of programming is due mainly to the inherent problem solving nature of the task, and to the complexity of the task. Programming styles and methodologies, programming environments, and the programming languages used also vary from programmer to programmer. In addition, the amount of documentation for a program, both in-line and external, as well as the completeness and style of the documentation, vary from program to program.

One method for improving program comprehension strategies is to change the programming paradigm. Several alternative programming paradigms have been suggested (Cunningham & Beck, 1987; Knuth, 1984; Oman & Cook, 1990a). Unfortunately, the research evaluating most of these suggestions has not been forthcoming. The few studies that deal specifically with literate programming systems are: The Literate Program Browser, (Beck & Cunningham, 1987) and An Interactive Tool for Literate Programming, (Brown & Childs, 1989). The Brown and Childs study evaluated the efficacy of a literate programming tool. Although a focus of the study was to determine if literate programs were more comprehensible than traditional programs in a maintenance task, the study did not directly address the components of literate programming which can be emphasized to enhance program comprehension. The Brown and Childs study was inconclusive with respect to determining the

comprehensibility of literate versus non-literate programs. The study did find that the programming environment itself was highly preferred by the subjects in the study. Efforts such as the WEB system (Knuth, 1984) (and many WEB variants such as CWEB (Levy, 1987; Thimbleby, 1986) and 'the WEB system for Modula-2' (Sewell, 1987)) have been attempts to change the programming paradigm, although the efficacy of these alternate paradigms with regard to program comprehension and enhanced programmer productivity has not been evaluated empirically.

Recent studies by Oman and Cook (1990b) have suggested organizing programs using the book paradigm. In addition, Oman and Cook (1990a) reported on a study dealing with typographic style as an aid to program comprehension. The suggested programming paradigms all differ, but in general, the paradigms tend to agree that computer programmers, and maintainers of these programs, need a method of formatting and documenting programs that is consistent with programmer comprehension strategies and maintenance activities. Most of the research in this area has also recognized the importance that plans (Adelson, 1981; Soloway & Ehrlich, 1984), program beacons (Brooks, 1983; Pennington, 1987; Wiedenbeck & Scholtz, 1989), and chunks (Adelson, 1981) play in the process of reading and understanding program source code.

There are several models of programmer comprehension strategies to date. Probably the most well known are the models of Shneiderman and Mayer

(1979), Basili and Mills (1982), and Brooks (1983). Both the Shneiderman and Mayer and the Basili and Mills models are similar in that they focus on bottom-up processes and reject the idea that a program is understood on a line by line basis. Both models are driven by the program text and they are basically inductive.

A model of programmer comprehension strategies proposed by Koenemann and Robertson (1991) suggests that program comprehension is understood as a goal-oriented, hypothesis-driven problem-solving process. Programmers follow a pragmatic as-needed strategy and restrict their understanding to portions of a program that are judged relevant for accomplishing a given task, with bottom-up comprehension used only for directly relevant code and in cases of missing, insufficient, or failing hypotheses. These comprehension strategies may have been developed because of the difficulty of understanding a program due to the lack of crucial documentation. Koenemann and Robertson suggest that both anticipatory and design history documentation should be included in programs to facilitate program comprehension by revealing portions of the original design process that cannot be easily reconstructed from the code itself.

A study by Littman, Pinto, Letovsky, and Soloway (1986) found that both as-needed and systematic strategies were used in program comprehension. The systematic strategy identified was employed by programmers using extensive symbolic execution of the data and control flow between subroutines

to gain detailed understanding prior to modifying any code to accomplish a new task or modify an existing task. The as-needed strategy was first put forth by Brooks (1893) in his model of "Beacons" that guide comprehension.

Brooks' (1983) theory of program comprehension assigns a large portion of the task to top-down processes. Brooks' model is basically an iterative process of hypothesizing, verification, and hypothesis modification, that relies heavily on programmer expectations. The programmer begins by making an overall hypothesis about the functionality of the program from the program's name and/or a brief description of the program. The general model the programmer has formed leads the experienced programmer to expect that certain structures and operations will appear in the program based on the programmer's knowledge of the task domain and of computer programming concepts. These expectations form another more specific hypothesis about the program's function and implementation.

The programmer attempts to verify these hypotheses by effecting a search of the program text for the expected key features (beacons) which are indicative of certain operations or structures. An example of a beacon is the 'swap' where two values are swapped, which is commonly found in several sorts. A beacon is associated with a task with a high probability, and if it is found, this strengthens the current hypothesis of the program's function. Otherwise, if the beacon is not found, this tells the programmer that the code must be looked at more carefully, possibly using other techniques and

knowledge of alternate algorithms. If this deeper search still fails to confirm the presence of the expected structures and/or operations, the programmer revises or rejects the current hypothesis and begins the process again.

Related research by Pennington (1987), Wiedenbeck (1986), and Wiedenbeck and Scholtz (1989) in the area of program beacons has led to the hypothesis that there are key features in a program which play an important role in understanding. Each line of a program does not have equal importance; experienced programmers make use of well known patterns to help in understanding the program. Obviously, non-expert programmers do not have the rich set of expectations that expert programmers do; thus, the theory of Shneiderman and Mayer (1979) may be more accurate with respect to non-expert programmers, as it does not rely on programmer expectations.

A model of text comprehension by van Dijk and Kintsch (1983) suggests that a reader makes two distinct representations of the text; a textbase and a situation model. The textbase includes a hierarchy of representations made up of a surface memory of the text, a microstructure of the interrelationships among the text prepositions, and a macrostructure that organizes the text representation. The situation model is a mental model of what the text is about referentially (*i.e.*, the task domain). The model has been extended into the domain of program comprehension by Pennington (1987). The textbase is a mental representation that focuses on the procedural program relations in terms of the programming language. The situation model is a mental



representation based on the functional relations between the program objects that is expressed in terms of the language of the domain objects. The textbase is referred to as the program model, and the situation model is referred to as the domain model. The textbase (program model) and the situation model (domain model) must be cross referenced in some meaningful way that relates the program parts to the domain functions. Pennington (1987) suggests that the program model is constructed prior to the domain model, and that the construction of the domain model, especially one connected to the program model, is essential to good program comprehension.

A study by Oman and Cook (1990b) identified that typographical style in programs is an aid to programmer comprehension. Several macro-typographic and micro-typographic principles which made the components and organization of the program more comprehensible were identified, including: identify the use and purpose of program components; make the execution control and information flow apparent; make the program readable and easy to browse using a variety of access paths into the code (e.g., bottom-up, top-down, browsing, and focused); make the sections and organization of the modules obvious; identify the use and purpose of each section; and use spatial cues to indicate statement groupings and separation.

Additionally, research has shown that it is easier to remember a picture than it is to remember textual information (Anderson, 1980). In a related finding by Santa (1977) it was reported that objects such as geometric figures tend to

be stored and remembered according to the spatial position in which they were presented, while words tend to be stored linearly. This suggests that 'stereotypical problem solutions' might be better remembered if presented graphically and backed up with textual information.

Traditional structured programs do not have the ability to present graphical information, and thus may be lacking in this crucial area of comprehension. Cuniff and Taylor (1987) reported that for short program segments, graphical representation of programs improves novices' comprehension by two specific measures: time and accuracy. Thus a comprehensive programming system should allow for a variety of graphical representations to be imbedded in the text of the program document, including graphs, diagrams, charts, equations, and pictures. A comprehensive programming methodology should dramatically enhance the textbase by logically sectioning it, consistently formatting it, and could assist in linking the textbase with the situation model through thorough documentation. Graphical representations are not required, although it is hypothesized that they would further enhance programmer comprehension.

### Program Testing

In addition to comprehending program source code, maintenance programmers and designers need to test programs as they are implemented and modified. There are several schools of thought relating to software testing. Although not the focus of this paper, one method for testing is discussed, as it

relates to a complete programming methodology such as that proposed for the Lit system.

Software testing is another one of the time consuming tasks in the software development cycle. Good tests are difficult to develop and time consuming to verify. For example, assume there is a module, call it  $M$ , that computes a function,  $F$ , with domain  $D$ . The correctness of  $M$  can be determined by testing  $M$  with each element of  $D$ . But, in most cases,  $D$  is infinite; thus the approach is effectively infeasible. The approach of the tester is to find some set  $S$ , such that  $S$  is contained in  $D$ . The assumption is that if  $M$  produces correct results for all elements in  $S$  it will do so for all elements in  $D$  also. Although this assumption may not be true (and in most cases is not true), it gives the programmer confidence in the design and the programmatic implementation of the design. The idea then is to find a test set  $S$  such that our confidence in the module is increased if it passes all of the tests specified in  $S$ , even if these tests fail to certify the module as correct.

Testing program modifications requires very good comprehension of the program; appropriate tests must be designed to exercise the areas of modification, as well as exercising areas that have not been changed, to insure that the program modification has not introduced an error in an area of the program that used to work correctly. Small changes can affect the entire program, especially with programs that are not modular, or when the modularity is not based on functional independence and data encapsulation.

Many researchers have suggested methods to automate portions of the software testing process. Hamlet (1977) described a system that assisted in program testing with the aid of a compiler modified to allow additions of input-output specifications to the program. The system added a notation to the syntax of the language that allowed the programmer to specify input-output pairs in the program code, independent of program details. The notation is easy to use and was suggested as a method of assisting in the development of programs that are resistant to the introduction of errors in the maintenance process. Hamlet also suggested the following as desirable goals of any scheme which would be used to assist in the derivation of input-output specifications to be used in a system for program testing: 1) the specification should be independent of program details; 2) the specifications should be substantially easier to produce and use than the programming language; and 3) human effort at verifying the specifications should be minimal and should be automated such that computer time is not prohibitive to perform the checks. If the specification system does not take into account all three goals, it is surmised that the specifications: 1) may end up describing the code and can not be used as an independent certification of the code, 2) may not be used if they increase the cognitive load on the programmer, and 3) may not be easily used to verify later modifications to the program. If the specification system does take all three goals into account, it could be useful not only in testing programs, but in debugging and maintaining them.

Rapps and Weyuker (1985) defined a family of program test data selection criteria derived from data flow analysis techniques. The proposed criteria associates each point at which a variable is defined, each point at which the variable is used. Furthermore, the number of paths selected for testing is always finite, and is chosen in an intelligent and systematic fashion in order to assist in finding program errors. The fulfillment of the selection criteria can be automated; given a program, a test set, and selection criteria, it can be determined programmatically whether or not the paths that would be traversed by the test set satisfy the criteria.

If the method outlined by Hamlet is combined with the method outlined by Rapps and Weyuker, an extremely powerful software testing tool that may assist in program debugging and maintenance could be the result. The idea is that a finite collection of tests based on such criteria, automated within the programming system, may be very useful in testing and debugging, even though it fails to certify the program as correct. As is noted by Hamlet (1977) this idea is supported from two divergent directions: (1) Maintenance programmers tend to test modifications to a program by trying to find data that will invoke the portions of code that have been changed and test it for correctness, while other portions of test data are used to verify that unchanged sections of code continue to work correctly, and (2) computability theory says that, because a program is finite, a finite number of tests will invoke each portion of the program that can be invoked; the problem is finding a finite test

set which does in fact exercise the program in the specified way. Using criteria such as that suggested by Rapps and Weyuker helps us to find such a set, and automating the testing process should assist the programmer in testing the suitability of the program for the designated purpose. Whether incorporating this scheme into a programming paradigm would assist in program comprehension is unknown. However, it may still assist in debugging and maintenance, even if it cannot be shown to assist in program comprehension.

## CHAPTER II

### LITERATE PROGRAMMING

Donald Knuth (1984) proposed a programming methodology that called for significantly improved documentation of programs. What he proposed was that computer programs should be viewed as "works of literature"; that computer programs should be written with "human consumption" in mind instead of "computer consumption" in mind. Knuth coined the phrase "Literate Programming" to describe this methodology.

Simply put, Literate Programming provides significantly better documentation of programs (as compared to traditional modular programs) by embedding the code of the program into the text of a technical design document. Instead of having separate documentation, design specifications, maintenance guides, and the coded program including embedded comments, we could write a single document which contains all of the information necessary to write the program and the program itself. This document would include an introduction to the problem, possibly some background material, the developed algorithm in pseudo code, and the program modules, main program, and subprograms with comments about future modifications. The advantage of such a program development method should be obvious; all of the information about the program is included in one document.

Basically, Knuth believed that literate programmers could be regarded as essayists, whose main concern should be exposition and excellence of style. As such, computer programmers would carefully choose variable names, and would write the program in a manner that was comprehensible to the reader. The concepts would be introduced in an order and manner that is best suited for human understanding, using a mixture of formal and informal methods that are natural reinforcements of each other.

Knuth prototyped and released a programming system called WEB, for the Pascal language (1984). WEB relies on a tool called TeX to perform formatting of the source code into sections and subsections. WEB supports forward referencing macros, and forces a presentation style of the output document on the user that is consistent from program to program. WEB also automatically generates a table of contents, and can be coerced into providing an index as well.

Although Knuth's (1984) WEB system was a wonderful advancement, it was difficult for the novice user (who did not understand the TeX text processing language) to use. It also worked only with the programming language Pascal, and was not designed to present the program based on any empirically derived principles for fostering program comprehension. The difficulty of using WEB, the lack of empirically derived design principles, and the limited manner in which it addressed the full spectrum of problems associated with computer programming were the major motivations for designing and



implementing the Lit system. It was hypothesized that a comprehensive programming tool that addressed each problem related to programming could substantially assist computer programmers and maintainers of computer programs.

## THE LIT SYSTEM

This section describes the development of Lit, a system designed to support the design, coding, testing, debugging, documentation and maintenance of literate computer programs. The hypothesis underlying Lit was that an altered programming paradigm, rich in textual and task domain information, could be an effective aid in improving program comprehension. The first implementation of Lit was written in FORTRAN as an undergraduate programming project by this researcher in 1987. The system was a simple and basic implementation inspired by Knuth's (1984) WEB system. The presentation paradigm was similar to the format of a technical paper, had very few features, and was not very extensible. The implementation was extended to cover the C programming language in 1988. To make the system a more generalized tool, it was redesigned to be language independent, and reimplemented in the C programming language. In late 1989, the current Lit system was implemented as a language-independent abstraction-oriented system for literate programming.

The 1989 implementation of Lit was designed to be independent of programming language and text formatter, and a menu driven interface was

added to simplify its use. The system was designed to be used by novice, intermediate, and advanced programmers, and did not require them to have an underlying knowledge of the text formatting system in use (unlike Knuth's (1984) WEB system and most WEB variants). A book style presentation paradigm was adopted, and additional customizable features were added to the system. As the system became used more often by students of the Computer Science Department at Portland State University, the suggestions of users were incorporated to make the interface more intuitive and simpler to use. Over time, the system evolved to its current state, and has been modified to use principles that have been identified or put forth as aids in computer program comprehension (Fjeldstad & Hamlen, 1983; Kernighan & Plaughter, 1978; Ledgard & Tauer, 1987; Miara et.al., 1983; Oman and Cook, 1990). With the help of a colleague (Andrew J. McKnight), a version for use with WordPerfect 5.1 was designed and implemented in 1991. The Lit system has been used to teach an introductory computer programming course and has been used in several undergraduate programming classes at Portland State University.

The goal of the Lit system is to give program designers and maintenance programmers a development and maintenance environment with the following characteristics (italicized features have not been fully implemented yet).

- 1) an easily recognized information transfer paradigm that:

- a) provides explicit high level organizational clues about the program source code
  - b) provides low level organizational chunks
  - c) provides multiple access paths to the source code using the table of contents and index
  - d) table of contents for chapters, sections, and subsections
  - e) *variable cross referencing*
  - f) *module cross referencing*
  - g) *abstraction cross referencing*
  - h) provides formatting and organization that is consistent with programmer comprehension strategies and textual comprehension studies
  - i) can have embedded graphical information in the text of the program document
  - j) provides task domain information which is explicitly linked with the programming constructs used to implement the functions from the task domain
  - k) encourages the inclusion of anticipatory documentation
  - l) encourages the inclusion of design history documentation
- 2) provides revision control information and capabilities
- 3) provides abstraction capabilities that allow programming in an order independent of that required by the language in use

- 4) programming language independence
- 5) text formatter independence
- 6) *provides automated testing*
  - a) *module testing (local testing)*
  - b) *program testing (global testing)*
- 7) provides automated debugging facilities
- 8) *provides reverse engineering capabilities for non-literate programs that assists in conversion to a literate-style program*
- 9) *provides data-flow diagrams*
- 10) *provides control-flow diagrams*
- 11) provides a flexible, easy-to-use code and documentation browser
- 12) provides an integrated system through a simple, consistent, and customizable user interface

The Lit system defines a simple "language" or "command set" that allows the programmer to write very modularized programs, and then produces two documents from the original document: One for human consumption, and one for computer consumption.

In the Lit system, programs are divided into chapters, sections, and subsections. Each of these sections may contain abstraction definitions or references and/or embedded source code. What results is a single document containing all of the information necessary to understand and specify a computer program, to both the computer and the human reader.

One difference between the type of document Lit produces for computer consumption and the type of document WEB produces for computer consumption is the readability of the document. Because it might be desirable (although it should not be necessary) to view the document produced for "computer consumption" (e.g., a compiler), Lit produces a document for computer consumption that is not only easily readable, but also has a one-to-one correspondence with the lines of embedded source code in the original document file. This is not a consideration of WEB, which produces files for computer consumption that are in as compressed a form as is possible, with some simple markers that point the user back to the general area of the original file from which a statement was generated. This is an important consideration when a program is under development, since most compilers generate error messages based on the line number of the offending code in the source code file.

Another major difference between Lit and WEB is that WEB was designed to support a single language (Pascal) and a single text formatter (TeX). Lit, on the other hand, is language independent and text formatter independent. Currently, Lit supports 22 languages including C, FORTRAN, Pascal, and COBOL. Lit is also designed to support multiple text formatters, although the current UNIX implementation has only the support routines for nroff and troff. Future plans call for the support of at least TeX, LaTeX,

WordPerfect (a version of which has been prototyped by the researcher and a colleague at Portland State University), and Waterloo Script.

### Differences Between Traditional Programs and Literate Programs

Literate programming is by definition 'very readable'. It incorporates the design, limitations, future modification possibilities, and the code of the current implementation in one document. With a little practice, a literate program can be made to read like a book instead of a program. As for maintainability, the literate program not only contains the current implementation but also contains ideas for future modifications, the history of the problem, the algorithm currently in use, and the motivations of the author of the implementation. In the best case a description of the known bugs and/or limitations of the algorithm are also included. Each of these pieces of documentation are invaluable debugging and maintenance aids which are not usually found in traditional programs.

It is often very difficult to maintain traditional structured programs, especially when the program is large and not well documented. Often just figuring out the intent of the original author and the algorithms used to express that intent can take several hours or days. Variable names may be meaningless to a maintenance programmer without a documentation reference on how the variable is used. As the program increases in complexity and size, variable names and documentation become more important. Lacking documentation as most programs do, programmers may use other comprehension clues to assist

in determining the program's methodology for solving a particular problem. If an adaptive maintenance task is required, the programmer must understand the methodology well enough to modify it. This is obviously not a simple requirement if the methodology is very complex.

### How Literate Programs Enhance Program Comprehension

Using the models of program comprehension reviewed earlier, a description of how literate programs might actually enhance the comprehension process, and thus improve program modifiability, is outlined below.

In a literate program, the purpose of the program is explicitly stated in the introductory section. Furthermore, so is the history of the problem and the motivations for writing a program to solve the problem. Sections of critical code are documented with anticipatory documentation, often including stubs that are null in anticipation of a future modification. The algorithms in use are documented explicitly in pseudo-code. The program is sectioned like a book, with meaningful chapter headings, section headings, and subsection headings that define the logical organization of the program. Spatial cues, point size changes, and highlighting are used to further aid in program comprehension. Explicit documentation of execution control and information flow are contained in the document, as well as a table of contents for the program.

The initial hypothesis stage (determine program function from program name and/or brief description of the program) should be greatly enhanced by literate programming methodology. The programmer does not have to

hypothesize about the functioning of the program, it is spelled out. Because perfective, corrective, and adaptive maintenance changes are anticipated and documented, it facilitates searching the program for the most appropriate place to make the required modifications. Each logical division of the program has a separate chapter, section, or subsection used to separate different program components and to group related program components. Each of these divisions has a title indicative of its function and content; thus the understanding of portions of the working hypothesis that are related to the program subcomponents may also be facilitated by the literate program.

The introductory section serves a purpose not apparent at first: for the programmer who is unfamiliar with the task domain it may offer some insight into the task being performed and how it is performed. This would be a definite advantage over a non-literate program because the programmer can become somewhat familiar with the task-related concepts and the computer-related concepts that apply to the problem. This may help the programmer not only in comprehending the problem but also in remembering specifics about the implemented solution.

Additionally, structures and operations that can be used to confirm the working hypothesis about program functionality are directly documented and immediately available. The hypothesis testing process may be positively altered in a dramatic way; if the programmer decides to verify the working hypothesis,



the search for the expected structures and operations should be made simpler by the sectioning of the literate program.

In terms of the model of text comprehension put forth by van Dijk and Kintsch (1983), a literate program offers the programmer unfamiliar with the task domain a method of becoming informed about the task domain so that a domain model can be constructed and linked with the program model, forming a global model of program design. In terms of mental schemas, the literate program also provides a way for the programmer to chunk the information related to a particular portion of the task domain into a simple concept (e.g., the section name of the portion of the literate program that does the task).

Finally, even if the programmer does not read the documentation, the literate program might still be more comprehensible. The indentation would follow a rule, the keywords could be highlighted, and the program would be logically sectioned, which would enrich the textbase and should make beacons much more visible than in a traditional modular program.

In summary, a combination of the elements identified in the studies previously described was used to refine the design of the Lit system. A well written literate program should assist the maintenance programmer in developing both the program model and the domain model; the textbase is significantly enhanced with textual cues that help the programmer chunk the code, identify beacons, and develop a mental plan. Information about the task domain and how to relate the task domain to the program model are spelled

out in the program documentation, which should assist the programmer in developing a global model of the program.

The best way to describe the differences between a traditional program and a literate program is by example. Figures 1, 2, and 3, are examples of program fragments that all perform the same task. Figure 1 is an actual code fragment written by a professional C programmer working on a UNIX platform.

```

/* is-a-file(): return true if argument is a readable file */
int is_a_file(path)
char *path;
{
    return ( (stat(path, &stat_rec) == -1) ? 0 :
              (((stat_rec.st_mode & S_IFMT) != S_IFDIR)
               && (access(path,R_OK) == 0)) );
}

```

Figure 1. Traditional program fragment (written by a professional software programmer).

Figure 2 is the routine as it would be re-written for use in a traditional modular implementation for this study. Figure 3 was re-written as a routine for a literate program from a functional description for the program from which Figure 1 was taken.

Figure 1 is actual code, taken from a non-proprietary piece of software, written by a professional C programmer. Notice the complete lack of documentation and the compressed cryptic syntax of the program module as compared to Figure 2.

No "expert-style" code was used in the experiments so there would be no differences in the program source code of the literate programs and the

```

BOOLEAN is_a_file(path)
char *path;
{
    /* Structure of file status record from <sys/stat.h> */
    struct stat stat_rec;

    /* Check to see if it is a file */
    if (stat(path, &stat_rec) == -1)
        return ( FALSE );
    else
        /* Make sure it is not a directory, and it is readable */
        if ( ((stat_rec.st_mode & S_IFMT) != S_IFDIR)
            && (access(path,R_OK) == 0) )
            return ( TRUE );
        else
            return ( FALSE );
}

```

Figure 2. Traditional program fragment.

traditional programs. Clarity of expression required rewriting the code for the literate program, and thus the traditional program as well; this is done to eliminate any contaminating effect from source code incompatibilities that might enhance or hinder comprehension.

Examine the completeness of the documentation in Figure 3, the literate program fragment; all of the information required to understand the fragment are documented, including where to look for further information that is not contained in the literate program.

### How the Lit System Works

The Lit system is made up of a user interface and several application programs that do most of the underlying work. The basis of the system is a preprocessor that separates a literate programming file into its component parts: a source code file for computer consumption, and a formattable text file

### [8.3.1] `is_a_file()`: Return TRUE if argument is a file

The `is_a_file()` function accepts as an argument a null terminated string that is the absolute or relative pathname to a file. `is_a_file()` returns TRUE if the argument is a readable file, FALSE otherwise.

For our purposes, a FIFO, a character special file, a block special file, a symbolic link, or a regular file all qualify as files; a readable directory is not considered a file.

This routine makes use of the `stat()` and `access()` routine from the C system call library. More information about these routines can be found in section 2 of the UNIX programmers manual.

#### General Algorithm

Check to see that the string passed is a valid filename.

If the string is a filename, it can't be a directory.

If the file is not a directory make sure it is readable.

If the all of the above criteria are met, return TRUE.

If any of the above criteria are not met, return FALSE.

#### Declaration

BOOLEAN `is_a_file(path)`

char \*path;

#### Arguments

path                      string containing a relative or absolute pathname

**Calls:**                    `stat()` and `access()`

**Called By:**              `open_input_file()`

**Returns:**                *BOOLEAN* values TRUE or FALSE

```

BOOLEAN is_a_file(path)
char *path;
{
    /* Structure of file status record from <sys/stat.h> */
    struct stat stat_rec;

    /* Check to see if it is a file */
    if (stat(path, &stat_rec) == -1)
        return ( FALSE );
    else
        /* Make sure it is not a directory, and it is readable */
        if ( ((stat_rec.st_mode & S_IFMT) != S_IFDIR)
            && (access(path,R_OK) == 0) )
            return ( TRUE );
        else
            return ( FALSE );
}

```

**Figure 3.** Literate program fragment.

for human consumption (see Figure 4). The Lit system allows most compilers to generate error messages that have a one-to-one correspondence with the literate programming file. When forward referencing macros are allowed, the error messages generated by the compiler (with the exception of the C compiler) usually cannot be made to have a one-to-one correspondence with the literate programming file. For this reason, beginning programmers are discouraged from using forward referencing abstractions, unless the C programming language is being used.

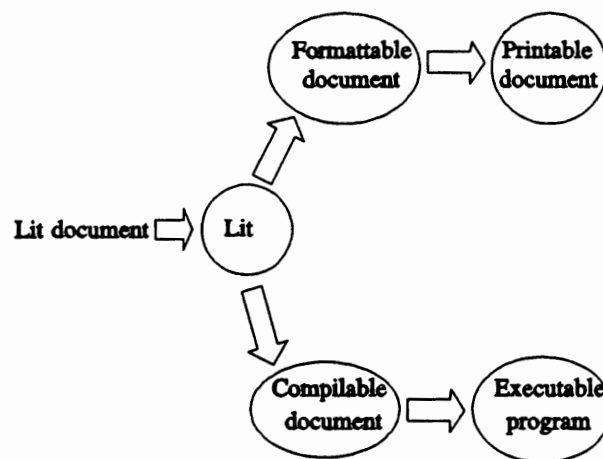


Figure 4. How the Lit system works

From the input document, the Lit system produces two output files:

project-name.src (compilable source code file)

project-name.doc (formattable document file)

where the ".src" suffix is either ".src" or the filename suffix required by the compiler for the language selected.

The Lit system currently has provisions for handling the following languages:

BAL, bash, BASIC, C, C++, COBOL, csh, Dbase, efl, FORTRAN, HyperTalk, ksh, LISP, MASM, MODULA-2, MUMPS, Paradox, Pascal, PostScript, ratfor, ReXX, SAS, sh, and SmallTalk.

and the following text formatters (or word processors):

troff, TeX, Waterloo Script, and WordPerfect

All commands must be preceded by the literate escape character to be interpreted as Lit language elements. For example, the command to start a chapter is { Chapter name } but the characters would not be interpreted as a chapter command unless they were preceded by the literate escape character, (i.e., @{ Chapter name }).

The Lit system understands the following commands:

{ Chapter name }	- Start a chapter
[ Section name ]	- Start a section
[[ Subsection name ]]	- Start a subsection
< abstraction > =	- Define an abstraction
< abstraction >	- Reference an abstraction
(	- Start a code section
<i>Code section: source code goes here</i>	
)	- End a code section

The Lit system also accepts some special formatting commands:

A name	- Author's name
B name	- Author's institution
D description	- One line terse description of program
F string	- Page footer
H string	- Page header
I	- Introduction
P name	- Program name

R string	- Revision number
T	- Date and time
U	- User defined
\$ string	- Comment

The default commands shown above can add to the cognitive load of the programmer, as they introduce yet another notation that must be remembered.

The following alternate selection of English-like commands is also understood by the Lit system:

chapter: chapter name.	- Start a chapter
section: section name.	- Start a section
subsection: subsection name.	- Start a subsection
abstraction: name.	- Define an abstraction
codebegin:	- Code section begin
<i>Code section: source code goes here</i>	
codeend:	- Code section end

and the equivalent special formatting commands:

author: name	- Authors name
business: name	- Authors institution
description: ...	- One line description of program
footer: string	- Page footer
header: string	- Page header
intro:	- Introduction
program: name	- Program name
revision: string	- Revision number
date:	- Date and time
comment: string	- Comment
\$:	- User defined

The reference implementation of the Lit system currently runs on the UNIX operating system. A version has been ported to the VM environment, to MS-DOS, and one version has been written in WordPerfect®'s macro language. The UNIX version of the system is designed to port directly to any POSIX

compliant operating system, but there are very few systems with strict POSIX compliance, even in the UNIX world. The Lit system interface is currently written to work from the C shell (csh(1)) and a version that is completely POSIX compliant is currently being developed.

The standard interface to the Lit system basically presents the user with a main menu of choices: Edit, Compile, Format, View, Print, Run, Debug, Goto new project, and Quit (Figure 5). From program design through program maintenance, the programmer can use the Lit system to produce, execute, debug, view, modify, and print literate programs. Lit allows the user to specify the editor, compiler, debugger, or other tools to be accessed by setting environment variables. If the user does not set the environment variables Lit will use defaults if possible, and will prompt for any other information that is needed to set up the user's programming environment. For a complete description of what the Lit system does for the user at each step, see Appendix K.

The Lit system isolates the user from the name and number of programs required to effectively use the system (see Appendix K). It was designed to be used with many already existing programs, so the user could have access to the tools with which the user is most comfortable. Lit keeps track of the file names and the required suffixes for the user, as well as launching the appropriate applications, in the appropriate order, when a simple request like "Compile" is entered by the user.



```
Lit:  Compiler = cc                               Date: Mon Dec 14 09:51:24 PST 1992
      Directory = /user/chrisb/projects/CASE

      Current project:      'CASE-tool'

      The following options are defined:

          1.  Edit
          2.  Compile
          3.  Format for printing
          4.  Print
          5.  View
          6.  Run
          7.  Debug
          8.  Goto another project
          9.  Quit

      To execute any other command, type the command
      name with all options, and press <Return>.
      Use <Control-Z> to suspend Lit.

      Choose an option by name or number and press <Return>:
```

Figure 5. The Lit system interface: the main menu.

Lit allows the user to enter an option from the menu in several ways. The number preceding the option can be entered, the name of the option, as presented in the menu, can be entered, or the upper-case or lower-case equivalent of the name or the first letter of the first word in a menu selection, can be entered.

Lit also allows the user to suspend the Lit system by pressing <Control-Z>, and allows the user to restart it, provided it was launched from a POSIX compliant shell or csh(1) using the command 'fg' (for foreground).

The user can also execute any other command from Lit's main menu prompt by simply typing the command with all relevant parameters and options and pressing <Return>. In addition, the user may define a file of aliases (shorthand notations that will invoke a long and/or complex command) for Lit, which **MUST** be stored in the file \$HOME/.LitAlias.

In any case, Lit isolates the user from dealing directly with the underlying application programs and their unwieldy parameters and file naming conventions. The system is customizable, operates as a menu-driven shell with all of the capabilities and the interaction possibilities of a shell, and minimizes the addition of any cognitive load on the programmer. The system was designed in this way for four reasons: 1) it would have taken too long to reinvent all of the applications, most of which are adequate for performing portions of the work that the system must do; 2) as each of the applications is re-engineered to assist programmer comprehension strategies it can be replaced; 3) expert programmers are not usually willing to give up their favorite tools; and 4) it would have been difficult to enable the programmer by adding to their cognitive load as much knowledge as is required to manage all of the underlying tools that comprise the Lit system.

For an example of literate program output, see Appendix D. For an example of a literate program, in the raw state (the actual input file containing the program source code embedded in the design document) see Appendix J.

## CHAPTER III

### EMPIRICAL STUDIES OF LITERATE PROGRAM COMPREHENSION

In order to determine the effectiveness of the literate programming paradigm, three empirical experiments were performed. Two studies were performed with novice programmers, and one experiment was performed with intermediate programmers. In the novice experiments, programming expertise level was held constant, and familiarity with task domain concepts was varied. In the intermediate experiment, only performance in the familiar task domain was investigated. Each of the experiments compared the ability of subjects to modify an existing program. There were two groups in each experiment: one group worked with a literate program, the other group worked with a traditional, but otherwise equivalent, version of the same program. This section outlines the general methodology used in the three studies, and subsequent sections look at each of the studies in detail and the overall implications of the findings of all three studies.

### EXPERIMENTAL CONTEXT

Experimental investigation into programmer comprehension strategies is still somewhat new, although many associated areas have already been investigated. Most of the experimentation in this area has been "reconstructive";

typically, subjects are asked to memorize and then recall lines of code, or to modify existing code while thinking aloud. In contrast, the approach employed here is essentially "constructive"; Subjects are asked to modify an existing program, but the modification consists of creating some missing piece of functionality, and inserting the usage of that functionality into the existing program. It is constructive in the sense that performance is analyzed in terms of entirely original program material generated as a result of goal-oriented hypothesis-driven problem solving processes. This type of measure of program comprehension was selected because of its relevance to the actual comprehension that is required of a professional programmer. Recall measures were also deemed necessary in order to establish a baseline of comprehension that would be consistent with standard reconstructive measures; in the event none of the subjects could effect a solution, the standard comprehension measures could be used exclusively.

## GENERAL METHODOLOGY

The general methodology was held constant across all three experiments.

### Subject Selection

Subjects were recruited from a sample of students with backgrounds appropriate to the classification levels of "novice programmer" or "intermediate programmer": Novice programmers were classified as having had less than

three computer programming courses and under one year of experience with computer programming. Intermediate programmers were classified as having between two and five years of computer programming courses, and under three years of full-time work experience in a job with the title "programmer", "programmer/analyst", or some similar job title. Subjects were paid \$5.00 for participating in the study.

All subjects were recruited from undergraduate level computer science courses at Portland State University.

In each of the experiments, the subjects were randomly divided into two groups of equal size; one group received the literate program to modify, the other group received the traditional, modular (but otherwise equivalent) version of the program to modify.

### Materials

Subjects in each of the studies received several documents (see Appendices D, F, G, and H): 1) a program specification, 2) an input/output specification, 3) a programming language reference, and 4) either a literate program or a standard modular program.

The program source code was identical for both the standard modular program and the literate programs, including all in-line source code comments.

The Literate Programs. In the literate programs, the purpose of the program was explicitly stated in the introductory section. Also stated were the history of the problem and the motivations for writing a program to solve the

problem. Sections of critical code were documented with anticipatory documentation, including stubs that were null in anticipation of a future modification. The algorithms in use were documented explicitly in pseudo-code. The program was written like a book, with meaningful chapter headings, section headings, and subsection headings that defined the logical organization of the program. Functional sections of the code were broken out into separate chapters, sections, and subsections, as dictated by functional independence.

Both programming implementation details and task domain information were documented extensively. Each chapter and section always started on a new page. Embedded code was never split over a page boundary unless it exceeded one page in length.

Routines were documented fully: The general algorithm was specified; All assumptions made were specified; Parameters passed and their uses, locally declared variables and their uses, and global variables used were specified; Calling procedures and procedures called by the routine were also specified.

A consistent style of indentation and program formatting was used for all literate programs.

Figure 3 illustrates the type and style of information included in a literate program. The subsection in the example might be contained in a chapter entitled "Support Functions" in a section entitled "File status utilities". See

Appendix D for an actual literate program as used in this study (the example in Appendix D was used in Experiment One).

The Traditional Modular Programs. The source code for the traditional modular programs was identical to the literate programs. All in-line comments in the traditional program were also included in the literate programs. The in-line comments tersely described the steps being taken to effect a solution. The traditional modular programs contained a header describing the name and function of the program. Each routine had terse style comment that described its purpose. White space was used to denote functional groupings and separation based on functional independence.

The traditional programs contained the identical in-line documentation as the literate programs. In order to minimize any effect of using different source code in each study, it was determined that it would be best to have the actual program source code be identical in both versions of the programs. Thus the program structure, the presentation order of the routines, and the indentation and coding style were identical for both the traditional programs and the literate programs. The only differences were the additional documentation and the programming paradigm specifics.

Page breaks in the listing were made such that a routine was never split over a page boundary, unless it was too long to fit on a single page.

Figure 2 illustrates the type and style of information included in the non-literate programs used for this study. Note that the format, indentation,

and in-line documentation are identical to that of the literate program. However, this version is informationally deficient in comparison with Figure 3. Appendix E contains the traditional modular program used in Experiment One.

### Procedure

The experiments were all controlled studies. Subjects were introduced to the study and informed how the study would be conducted. Subjects were randomly divided into two groups, one which received the literate program to modify and the other which received the standard modular program to modify. Each subject was given a sheet of instructions (see Appendix B) and verbal instructions. The subjects were instructed to use any of the reference materials provided, if needed. Subjects were given a time limit to complete the required modifications to the program. The time limit to complete the modifications had been established in a prior study. Subjects were notified when only 10 minutes were remaining in which to complete the experiment. After the subject felt the program was completed, or when time had run out, a follow-up questionnaire was administered (see Appendix C).

The questionnaire was used to measure whether the subjects had (1) understood the instructions and (2) understood the purpose and function of the program. Some additional subjective measures were also collected; subjects were asked to: (1) indicate if they felt they had identified the problem with the program, (2) indicate how many subroutines did they think were missing from the program, (3) describe the function of the missing subroutines, (4) identify



which elements of the program were most helpful in solving the problem, (5) indicate if they felt the solution that was found (if one was found) was accurate, (6) identify which elements of the program did not contribute to solving the problem, (7) state the overall function of the program, (8) rate the difficulty of the problem on a Lichert scale, and (9) rate the accuracy of their solution, (a) ignoring the possibility of syntax errors, and (b) including the possibility of having made syntax error(s). When the questionnaire was completed, the subjects were given the solution to the problem (see Appendix I for an example solution), given thanks for participating in the study, and asked if they had any questions regarding the study.

The subjects did not have the use of a compiler or any other program development tool. Because of the variation in programming tools and programmers' familiarity with different tools, it was determined that the most unbiased test would be to have all subjects work with only a printed program listing, specifications, and a language reference. All program modifications were made on paper.

Program modifications were designed to simulate common maintenance programming activities. The maintenance task was essentially completing a program that was not finished by a previous programmer; the task had been specified in the original program specification, but had not been completed.

### Establishing Time Constraints for the Studies

Prior to running the experiments, two expert programmers completed the required modifications to the standard modular programs used in the studies. The two experts, were a Systems Analyst with ten years of experience, and a Programmer/Analyst with three years of experience. For each of the programs, the time it took the expert programmers was averaged, rounded off to the nearest 5 minutes, and then doubled for use as a time constraint for the experiment.

A pilot study was conducted using 12 computer science graduate students. The performance of the graduate students and their comments on the questionnaire were used to refine the methodology and materials.

### Measures

The researcher analyzed all modifications to the programs for correctness. Several 'correctness' criteria were used: 1) completely correct and identical to the original solution (with the exception of variable names and choice of flow control statements), 2) functionally correct alternative solution, 3) any functionally correct solution with syntax errors, and 4) a functionally incorrect modification.

Several other recall criteria were used to identify comprehension: 5) Did the subject find where the missing calls to the missing subroutine(s) should be placed? 6) When the position for the call was found, was the inserted call correct for the subject's modification? 7) Was there an attempt to modify the

wrong code? 8) Could the subject describe the intended functionality of the program? 9) Could the subject describe the intended functionality of the missing routine?

Criterion 1 was the litmus test for comprehension. If the program was well comprehended and the motivations and style of the original author were well understood, the solution of the subject should be close to or identical to the solution of the original author.

Criterion 2 was an expected outcome; no two people program exactly in the same style, and multiple solutions are a natural outcome for any hypothesis-driven problem-solving task.

Criterion 3 was used to identify problems that would have been found at compile time because of a syntax error (or errors), flagged by a compiler, and when corrected would have resulted in a correct solution. This is a natural occurrence when programming. Because subjects did not have the ability to correct these problems due to the paper and pencil orientation of the task, it was judged that a correct solution could contain syntax errors. Semantic errors that would not be found by the compiler, and that would result in an executable program which did not operate correctly, were judged as incorrect (criterion 4).

All programs not meeting criteria 1, 2, or 3 were judged to fit in category 4-functionally incorrect modification.

Criterion 5 was judged important for the subjects with functionally incorrect solutions to determine a level of comprehension. If the subject found

and inserted the missing calls, but the subroutine created by the subject was incorrect, this was judged to be a better outcome than if the subroutine was incorrect or missing, or the calls to the subroutine were incorrect in their placement and/or usage, or no missing subroutine calls were found or inserted.

Criterion 6 was used as a measure of how well the code was understood. Just finding the position of the missing call is not as important as finding the missing call and inserting a call that passes the parameters that must be used and modified to affect a solution.

Criterion 7 was judged important because if the program was well understood, the subject should never have modified a section of code that could not help in effecting a solution.

Criteria 8, 9, and questionnaire measure 2 are standard recall measures commonly used to evaluate comprehension of computer programs. Recall measures tend to be weaker measures, but were included in the event that the more discriminating measures were too discriminating and could not be used to identify comprehension. It was determined that, although there is much evidence to support reconstructive measures of comprehension, such measures be inadequately measuring comprehension that is indicative of that required to actually perform correct modifications to a computer program. With this in mind, measures 1 through 7 were developed to measure program comprehension. Our results do in fact show that the significance of our

findings would have been lessened had we not developed the more stringent comprehension measures.

Subjects' opinions of which elements of the program were most helpful in solving the problem are used to identify areas for further study and for confirmation of the researchers hypothesis about which elements are most beneficial to the programmer for comprehension.

### RESEARCH HYPOTHESIS

Does altering the programming paradigm to contain typographic cues, task domain information, and a book-style presentation paradigm increase program comprehension. Specifically, will Lit style literate programs be more comprehensible than traditional modular programs by novice and/or intermediate programmers. In order to measure program comprehension, the maintenance code generated by subjects was analyzed. Increased comprehension of the program should result in a higher percentage of correct solutions by one group. Subjects given literate programs were compared with subjects given traditional modular programs on their ability to:

1. Correctly complete the modifications to the program.
2. Produce more functionally correct programs with syntax errors.
3. Find which routines are missing.
4. Describe the function of the missing subroutine(s).
5. Find the correct place to insert any missing calls to the missing routines.

6. Insert correct calls to the missing subroutines or functions.
7. Modify only sections of code that can be used to solve the problem.
8. Explain the purpose and function of the program.

## CHAPTER IV

### EXPERIMENT ONE

#### SUBJECTS

For Experiment One, 20 novice subjects were recruited from an undergraduate course in FORTRAN programming for non-computer science majors. Many of the subjects had no prior experience with computers, and only one had prior experience with computer programming before completing the introductory FORTRAN programming course. The subjects were all familiar with the FORTRAN programming language, standard modular programming, and had been instructed and allowed to use both standard UNIX programming tools and the Lit system for eight weeks prior to the experiments. Subjects were familiarized with both traditional printed listings and Lit style literate program listings.

The subjects were randomly divided into two groups of equal size; one group received the literate program, the other group received the traditional modular program.

Experiment One involved programming in a task domain that none of the subjects were familiar with (economic forecasting using Leontief modeling);

## MATERIALS

The program the subjects worked with in Experiment One (unfamiliar task domain) was designed and written by the researcher and involved Leontief modeling. The portion of the program that was missing was a subroutine that created a matrix (the technology matrix) from the initial input matrix by subtracting the input matrix from its identity matrix. The call to the routine that created the technology matrix was also missing from the program. (See Appendix D for a xero-reduced copy of the actual program used for this experiment).

## PROCEDURE

Experiment One was a controlled study. Each subject was given a sheet of instructions (see Appendix F) and the following verbal instructions.

You have been given the task of maintaining a computer program. The original author completed the analysis and design of the program, but did not have time to complete the coding. Your job is to determine what functional units of code have been left out and to create them and indicate where in the program they would be inserted. The code that is missing is one or more subroutines or functions, and the calls to those routines or functions. You must also insert the calls to the routines you create in the appropriate place or places in the program for the solution to be considered correct.

The subjects were given either the literate or traditional modular program to modify and were instructed to use any of the reference materials provided, if needed. A time limit of 50 minutes to complete the modifications had been



established in a previous pilot study. Subjects were notified when only 10 minutes were remaining in which to complete the experiment. After completing the program modifications or running out of time, subjects filled out a questionnaire (see Appendix C).

## RESULTS

Results were analyzed using nonparametric one-way analysis of variance. Analysis of variance of group performance in the unfamiliar task domain (Table I) showed that a significantly greater percentage of the subjects in the literate program group found a solution that was either completely correct or functionally correct with syntax errors; none of the traditional modular program group found a solution ( $F(1,19) = 13.50$ ,  $p < .0017$ ,  $\eta^2 = .43$ ). Of the subjects that found a solution, one third found a completely correct solution, and two thirds found a functionally correct solution with syntax errors. The latter finding was also significant ( $F(1,19) = 6.00$ ,  $p < .024$ ,  $\eta^2 = .25$ ). Table I also shows that all of the subjects in the traditional modular program group attempted to modify a section or sections of code that did not require a modification to solve the problem. None of the subjects in the literate program group modified incorrect code. This finding was significant ( $F(1,19) = 9999$ ,  $p < .0001$ ,  $\eta^2 = 1.0$ ). The differences between which groups found where to insert the missing calls to the missing subroutines were significant ( $F(1,19) = 9999$ ,  $p < .0001$ ,  $\eta^2 = 1.0$ ). Finally, there were significant differences in which

groups were able to insert the call correctly ( $F(1,19) = 13.5$ ,  $p < .0017$ ,  $\eta^2 = .43$ ).

TABLE I  
GROUP PERFORMANCE PERCENTAGES FOR EXPERIMENT ONE

Comprehension Criteria	Literate Program Group Performance	Traditional Program Group Performance
Completely Correct	20%	0%
Functionally Correct	40%	0%
Incorrect	40%	100%
Found Missing Call	100%	0%
Inserted Call Correctly	60%	0%
Did not Modify Wrong Code	100%	0%
Described Program's Intended Functionality Correctly	100%	0%
Number of Missing Subroutines Identified Correctly	80%	10%
Accurately Described Function of Missing Routines	60%	0%

Equally impressive were the results of the analysis of the reconstructive measurements. Table I also shows the groups' ability to accurately describe the program's function. This finding was significant ( $F(1,19) = 9999$ ,  $p < .0001$ ,  $\eta^2 = 1.0$ ). Additionally, the subjects in the literate program group significantly outperformed the subjects in the traditional modular program group in identifying the number of missing subroutines ( $F(1,16) = 56.47$ ,  $p < .0001$ ,  $\eta^2 = .79$ ) and in accurately describing their intended functionality ( $F(1,15) =$

47.25,  $p < .0001$ ,  $\eta^2 = .77$ ). Three subjects did not provide an answer to the question of how many subroutines were missing, and four did not provide an answer describing the function of the missing subroutine. The missing values were excluded from the analysis, as is reflected by the reported F values.

### Analysis of Subjects' Subjective Data

The questionnaire was analyzed in order to gauge the subjects' perception of which elements of the program were aids in solving the problem, and which elements of the program were caused difficulty in solving the problem. Overall, 70 percent of the subjects in the literate program group found that the program documentation helped with problem solving. This indicates that even in the traditional program group, the documentation was perceived as helpful. Since the traditional programs were written with much more documentation than would typically be in-line, this suggests that the additional documentation may have been helpful. All of the elements of the literate program (documentation, code style, table of contents, and program format) were perceived as helpful in problem solving by at least 20 percent of the subjects in the literate program group. It was determined that the perceptions of subjects who found a solution might be more indicative of which elements were most helpful. Conversely, it was also determined that the perceptions of subjects who could not find a solution might be indicative of which elements hindered problem solving. Or, it might give an indication of

which subjects were able to use the additional information, and which subjects were possibly confused by it, or just unable to utilize it.

**TABLE II**  
**EXPERIMENT ONE: PERCEPTIONS OF SUBJECTS GIVEN**  
**LITERATE PROGRAMS**

Program element	Helped with problem solving	Hindered problem solving
Documentation	70%	30%
Code Style	40%	0%
Table of Contents	30%	0%
Input Specifications	30%	0%
Problem Description	20%	50%
Indentation	20%	0%
Program Format	20%	0%
Output Specifications	20%	0%

Of the subjects who found a solution (N=6) and answered the questions pertaining to the factors that contributed most to solving the problem, Table III documents which elements they perceived as helpful in solving the problem. Overwhelmingly, the most helpful factors were program documentation and code style. Only one subject who found a solution indicated the table of contents was helpful. This may be because it was not needed to find the solution by the others, or the subjects who found a solution were unaware of how much it helped them because of their familiarity with such usage to find areas of interest in books, and its value was not perceived as important. It could also be that the table of contents was not seen as helpful in relation to

the help the documentation provided (although the table of contents is a portion of that documentation).

TABLE III

EXPERIMENT ONE: PERCEPTIONS OF SUBJECTS GIVEN LITERATE PROGRAMS WHO FOUND A SOLUTION

Program element	Helped with problem solving	Hindered problem solving
Documentation	83%	33%
Code Style	50%	0%
Table of Contents	16%	0%
Input Specifications	33%	0%
Problem Description	16%	50%
Indentation	16%	0%
Program Format	16%	0%
Output Specifications	16%	0%

Table IV describes the perceptions of the subjects who did not find a solution (N=4) that were given literate programs. Note that 50 percent of these subjects indicated that the table of contents was helpful to problem solving, and none indicated it hindered problem solving. Documentation was also perceived as helpful to 50 percent of the subjects (two subjects) in this group. One subject in this group indicated that documentation hindered problem solving.

For subjects given the traditional programs (see Table V), the factors perceived as a hinderance to problem solving were documentation (33 percent), input specifications (44 percent), and the problem description (66 percent). The difficulty with the problem description can most easily be

attributed to the subjects' unfamiliarity with the task domain and the language used in the problem description. This was an expected result. The problems

TABLE IV

EXPERIMENT ONE: PERCEPTIONS OF SUBJECTS GIVEN LITERATE PROGRAMS WHO DID NOT FIND A SOLUTION

Program element	Helped with problem solving	Hindered problem solving
Documentation	50%	25%
Code Style	25%	0%
Table of Contents	50%	0%
Input Specifications	25%	0%
Problem Description	25%	50%
Indentation	25%	0%
Program Format	25%	0%
Output Specifications	25%	0%

with the input specifications are difficult to analyze, since over 71 percent of the subjects in the experiment indicated that the input specifications were "easy to understand". Finally, the problem with documentation may be that there wasn't enough of it, or more accurately, it was informationally inadequate to assist the subjects in forming a global model of program design. Thus program modifications could not be made, and the documentation was perceived as a hinderance.

Overall perceptions (N=18) for both groups of experiment instructions, problem description, input specifications, and output specifications rated as "easy to understand" or "not easy to understand" are presented in Table VI.

**TABLE V**  
**EXPERIMENT ONE: PERCEPTIONS OF SUBJECTS GIVEN**  
**STANDARD PROGRAMS**

Program element	Helped with problem solving	Hindered problem solving
Documentation	11%	33%
Code Style	11%	22%
Table of Contents	N/A	N/A
Input Specifications	44%	44%
Problem Description	22%	66%
Indentation	11%	11%
Program Format	22%	22%
Output Specifications	33%	22%

**TABLE VI**  
**EXPERIMENT ONE: SUBJECTS' EVALUATIONS OF**  
**EXPERIMENTAL MATERIALS**

Experiment material	Easy to understand	Not easy to understand
Instructions	76%	24%
Problem Description	35%	65%
Input Specification	71%	29%
Output Specification *	69%	31%

*\* indicates N=16 for this variable*

Many (65 percent) of the subjects found the problem description difficult to understand. This is most likely due to the fact that subjects were not familiar with the task domain (economic modelling) and the terminology used to

describe the required processing was not familiar to the subjects. It was expected that for problems in unfamiliar task domains the problem description would be rated as difficult to understand, and that the perception of the level of difficulty of the problem would be high. The perceptions of the subjects in Experiment One indicate this hypothesis is accurate; the perceived level of difficulty (obtained by finding the mean of the difficulty scale ranging from 1 (very difficult) to 5 (very easy) for all subjects) was 2.05, indicating that subjects perceived the level of difficulty of the problem as more difficult than easy.

It was also expected that the perception of difficulty for the literate program group would be perceived as less difficult than the perceptions of the traditional program group. The literate program group rated the level of difficulty as much less difficult than did the traditional program group. Group means indicated that the perception difference was 1.78 levels of difficulty more difficult for the traditional program group (1.22) than it was for the literate program group (3.0). No tests for significance were performed.

## DISCUSSION

The results of this study indicate that the Lit style of formatting code and documenting code are superior to traditional methods in assisting with program comprehension. Results indicate that comprehension is improved by at least two measures: ability to effect a solution (indicative of high comprehension due to successful application of the learned concepts); and ability to correctly recall



and describe the purpose of the program, the missing portions of the program, and several specifics about the program as written (modified CLOZE (Entin, 1984; Taylor, 1953) measures of comprehension).

It is also interesting to note that this experiment measured performance in an unfamiliar task domain. As has been noted, the development of a domain model and the ability to link the domain model with the program model to form the global model of the program is essential to program comprehension. Apparently, the literate program allowed more subjects to form a global model and make the required modifications; the subjects with the non-literate programs apparently could not develop a global model and thus were unable to make the required modifications. This is impressive, in that both groups overwhelmingly rated the problem description as difficult to understand, and as a hinderance to problem solving. Yet, the literate program group was able to overcome these difficulties and 60 percent found a solution. This indicates that the literate program did in fact contain features which assisted the programmer in understanding both the domain model and the program model, and assisted in linking up these two models into a global model of program design. Even more impressive is that the global model formed by 60 percent of the subjects allowed them to make the required modifications to the program in a short time period.

The post-experiment questionnaire had some supporting anecdotal commentary. Subject one, who found a solution to the problem, wrote:

[The] existence of the general algorithm made it possible to write the code without having any idea of what the Leontief [modeling] program is trying to do here.

In response to the question "What contributed most to the difficulty of modifying the program", Subject five, who also found a solution, wrote: "Not being familiar with what we are trying to accomplish."

Subject one indicates that the presence of the algorithm made it possible to write the code without understanding the task domain concepts. This is consistent with the idea that the task domain concepts do not have to be fully understood to be programmed if there is some documentation that can link up the task domain concepts with computer related concepts. Apparently, the presence of the general algorithm in the documentation did exactly that for this particular subject. This may also be true for the other subjects who found a solution (such as subject five, whose comment appears above), although they may not have realized it or reported it on the post-experiment questionnaire.

Of the subjects given traditional programs who did not find a solution, this theme is also present in the post-experiment questionnaire comments.

Subject 13 wrote:

I had to read through the [documentation for the] model several times to figure out exactly what did what. [The] documentation was clear - to a degree - the algorithms to be used were unclear.

Subject 19 commented:

... I just don't understand the problem well enough. If you don't understand the problem, you need more clarification explanations.

And subject 14 made the comment:

I couldn't make sense of what the input variables were supposed to do in the missing subroutines. I was not sure of how the matrix operations were supposed to be performed in the missing subroutines.

The comments from some of the subjects with the traditional programs point out that those subjects realized the need for more informationally complete documentation. Specifically, these three subjects each requested one element that is present in the literate programs: documentation on which algorithm to use, documentation on what each variable was used for, and documentation that clarified the task with explanations (task domain information).

In summary, Experiment one supports the hypothesis that programs should be written in a different format. The Lit style programming format is one such possibility which has been shown to be significantly more comprehensible than the format of the traditional programs used in this study. In addition, the subjective evaluation of many of the subjects supports the ideas on which literate programming is based, and anecdotal commentary by the subjects points directly to some of the flaws of the traditional programs, and some of the strengths of the literate programs suggested by the research hypothesis.

## CHAPTER V

### EXPERIMENT TWO

#### SUBJECTS

For Experiment Two, 21 novice subjects were recruited from an undergraduate course in FORTRAN programming for non-computer science majors. Many of the subjects had no prior experience with computers, and only one had prior experience with computer programming before completing the introductory FORTRAN programming course. The subjects were all familiar with the FORTRAN programming language, standard modular programming, and had been instructed and allowed to use both standard UNIX programming tools and the Lit system for eight weeks prior to the experiments. Subjects were familiarized with both traditional printed listings and Lit style literate program listings.

The subjects were randomly divided into two groups of equal size; one group received the literate program, the other group received the traditional modular program.

The difference between Experiment One and Experiment Two was that Experiment Two involved programming in a task domain that all of the subjects

were familiar with (calculating letter grades from weighted test and assignment scores).

## MATERIALS

The program the subjects worked with in Experiment Two (familiar task domain) was designed and written by the researcher and involved the problem of preparing a grade report from a file of students' weighted test and assignment scores. Omitted from the program was a routine that computed the average grade and then called a routine that assigned the student a letter grade. Also omitted was the call to the missing routine. The routines could either be called from the mainline of the program, or one routine could be called from the mainline and then that routine could call the other missing routine.

## PROCEDURE

Experiment Two was a controlled study. Each subject was given a sheet of instructions (see Appendix G) and the following verbal instructions.

You have been given the task of maintaining a computer program. The original author completed the analysis and design of the program, but did not have time to complete the coding. Your job is to determine what functional units of code have been left out and to create them and indicate where in the program they would be inserted. The code that is missing is one or more subroutines or functions, and the calls to those routines or functions. You must also insert the calls to the

routines you create in the appropriate place or places in the program for the solution to be considered correct.

The subjects were given either the literate or traditional modular program to modify and were instructed to use any of the reference materials provided, if needed. A time limit of 50 minutes to complete the modifications had been established in a previous pilot study. Subjects were notified when only 10 minutes were remaining. After completing the program modifications or running out of time, subjects filled out a questionnaire (see Appendix C).

## RESULTS

Results were analyzed using one-way nonparametric analysis of variance. Analysis of variance of group performance in the familiar task domain (Table VII) showed that 64 percent of the literate program group found either a completely correct solution or a functionally correct solution with syntax errors and none of the traditional modular program group found a solution. This finding was significant ( $F(1,19) = 15.83$ ,  $p < .0008$ ,  $\eta^2 = .45$ ). A functionally correct solution (equal to the proposed solution of the experimenters) with syntax errors was found by 36 percent of the subjects in the literate program group. This finding was significant ( $F(1,20) = 5.17$ ,  $p < .035$ ,  $\eta^2 = .23$ ). Also significant was that 29 percent of the literate program group found a functionally correct alternative solution with syntax errors ( $F(1,20) = 7.54$ ,  $p < .013$ ,  $\eta^2 = .28$ ). A completely correct solution equal to the solution proposed

by the experimenters was found by 18 percent of the subjects in the literate program group, which was not significant. Results also showed that group

TABLE VII  
GROUP PERFORMANCE PERCENTAGES FOR EXPERIMENT TWO

Comprehension Criteria	Literate Program Group Performance	Traditional Program Group Performance
Completely Correct	18%	0%
Functionally Correct	46%	0%
Incorrect	34%	100%
Found Missing Call	100%	50%
Inserted Call Correctly	64%	0%
Did not Modify Wrong Code	91%	50%
Described Problem Correctly	100%	90%
Number of Missing Subroutines Identified Correctly	100%	63%
Accurately Described Function of Missing Routines	91%	40%

differences were significant with regard to attempts at modifying a section or sections of code that did not require a modification to solve the problem ( $F(1,20) = 4.89$ ,  $p < .04$ ,  $\eta^2 = .20$ ). Also significant were the group differences related to finding where to insert the missing calls to the missing subroutines ( $F(1,20) = 6.03$ ,  $p < .019$ ,  $\eta^2 = .26$ ). In addition, analysis showed that the ability to insert the call correctly (see Table VII) was

significantly different between the literate program group and the traditional program group ( $F(1,20) = 15.83, p < .0008, \eta^2 = .45$ ).

The reconstructive measures were not as dramatically different as those in Experiment One. There was no significant group difference in ability to describe the overall functionality of the program; all of the subjects in the literate program group accurately described the program, and 90 percent of the subjects in the traditional modular program group accurately described the program. This may be due to the subjects familiarity with the task domain. The subjects in the literate program group significantly outperformed the subjects in the traditional modular program group (see Table VII) in identifying the number of missing subroutines ( $F(1,18) = 5.91, p < .027, \eta^2 = .25$ ) and accurately describing their intended functionality ( $F(1,19) = 7.79, p < .012, \eta^2 = .77$ ).

#### Analysis of Subjects' Subjective Data

The questionnaire was analyzed in order to gauge the subjects' perception of which elements of the program were aids in solving the problem, and which elements of the program caused difficulty in solving the problem. The results are presented in Tables VIII through XII.

Of the subjects given the literate programs to modify, 82 percent found the documentation helpful, 64 percent found the problem description helpful, 18 percent found the code style and indentation helpful, and 27 percent found the table of contents helpful. The factors that hindered problem solving most were the input and output specifications and the program format (see Table VIII).



The differences between Experiment One and Experiment Two are most obvious in the perception of the problem description. As expected, the problem description was perceived as helpful, probably due to the fact that the subjects were familiar with the task domain. As can be seen from the data in Table VII, less than two subjects in the literate program group found any one element of the literate program hindered problem solving.

Of the subjects who found a solution ( $N=7$ ) and answered the questions pertaining to what contributed most to solving the problem, Table IX documents which elements were perceived as helpful in solving the problem. Notice that the only element that more than one subject had trouble with was the input specification, which was external to the program. Not more than one subject perceived any other program element as hindering problem solving.

TABLE VIII

EXPERIMENT TWO: PERCEPTIONS OF SUBJECTS GIVEN  
LITERATE PROGRAMS

Program element	Helped with problem solving	Hindered problem solving
Documentation	82%	9%
Code Style	18%	9%
Table of Contents	27%	0%
Input Specifications	27%	18%
Problem Description	64%	9%
Indentation	18%	0%
Program Format	27%	18%
Output Specifications	18%	18%

Table X describes the perceptions of the subjects given literate programs who did not find a solution (N=4) . Only one subject in this group indicated that the program format hindered problem solving, and only one subject in this group indicated that the output specifications hindered problem solving. All of the rest of the subjects in this group indicated that one or more elements were helpful, and none of the subjects in this group indicated that documentation, code style, the table of contents, the input specifications, problem description, code style, the table of contents, the input specifications, problem description,

TABLE IX

**EXPERIMENT TWO: PERCEPTIONS OF SUBJECTS GIVEN  
LITERATE PROGRAMS WHO FOUND SOLUTIONS**

Program element	Helped with problem solving	Hindered problem solving
Documentation	86%	14%
Code Style	14%	14%
Table of Contents	29%	0%
Input Specifications	0%	29%
Problem Description	43%	14%
Indentation	0%	0%
Program Format	14%	14%
Output Specifications	0%	14%

or indentation hindered problem solving. Indentation, program format, and output specifications were indicated as helpful by 50 percent of the subjects in this group. Documentation and output specifications were rated as helpful by 75 percent of the group. Code style was rated as helpful by 25 percent of the

subjects, and the problem description was rated as helpful by all of these subjects.

Table XI clearly shows that, even for the standard program group (none of whom found a solution), the only factor perceived as a hinderance by more subjects than found the same factor helpful was the output specification.

TABLE X

EXPERIMENT TWO: PERCEPTIONS OF SUBJECTS GIVEN LITERATE  
PROGRAMS WHO DID NOT FIND SOLUTIONS

Program element	Helped with problem solving	Hindered problem solving
Documentation	75%	0%
Code Style	25%	0%
Table of Contents	25%	0%
Input Specifications	75%	0%
Problem Description	100%	0%
Indentation	50%	0%
Program Format	50%	25%
Output Specifications	50%	25%

Program format was perceived both as helpful and as a hinderance by 11 percent of the subjects in this group. All other factors were perceived as helpful by at least twice as many subjects as perceived the same factor as a hinderance; the biggest difference being that documentation was perceived as helpful by 4 times as many subjects than perceived documentation as a hinderance. Overall, the subjects tend to indicate that the traditional program did not present any large barriers to problem solving. In fact, Table XI shows

that the program was perceived as having documentation that could assist in problem solving by 44 percent of the subjects in this group. This was not an expected result. Perhaps the program documentation in the traditional program was explicit enough to give the impression that it was helpful. However, since none of these subjects found a solution, it is not at all clear what was the contribution of the documentation.

TABLE XI  
EXPERIMENT TWO: PERCEPTIONS OF SUBJECTS GIVEN  
STANDARD PROGRAMS

Program element	Helped with problem solving	Hindered problem solving
Documentation	44%	11%
Code Style	33%	11%
Table of Contents	N/A	N/A
Input Specifications	22%	11%
Problem Description	44%	22%
Indentation	0%	0%
Program Format	11%	11%
Output Specifications	11%	22%

Overall perceptions of experiment instructions, problem description, input specifications, and output specifications rated as "easy to understand" or "not easy to understand" are presented in Table XII. Only 18 of the subjects answered the questions pertaining to their ability to understand the experimental materials.

TABLE XII  
EXPERIMENT TWO: SUBJECTS' EVALUATIONS OF  
EXPERIMENTAL MATERIALS

Experiment material	Easy to understand	Not easy to understand
Instructions	94%	6%
Problem Description	100%	0%
Input Specification	78%	22%
Output Specification	78%	22%

Unlike Experiment One, all of the subjects in Experiment Two who answered the questions pertaining to their abilities to understand the experimental materials found the problem description easy to understand. This is most likely due to the fact that subjects were familiar with the task domain and the terminology used to describe the required processing was familiar to the subjects. It was expected that for problems in familiar task domains the problem description would be rated as easy to understand, and that the perception of the level of difficulty of the problem would be low. The perceptions of the subjects in Experiment Two indicate this is true; the perceived level of difficulty (obtained by finding the mean of the difficulty scale which ranged from 1 (very difficult) to 5 (very easy) for all subjects) was 2.90, indicating that subjects perceived the level of difficulty of the problem as between difficult and easy.

It was also expected that the perception of difficulty for the literate program group would be perceived as less difficult than the perceptions of the

traditional program group. The literate program group rated the level of difficulty as much less difficult than did the traditional program group. Group means indicated that the perception difference was 1.63 levels of difficulty more difficult for the traditional program group (2.0) than it was for the literate program group (3.63). No tests for significance were performed.

## DISCUSSION

The results of Experiment Two indicate that the Lit style of formatting code and documenting code are superior to traditional methods in assisting with program comprehension. Results also indicate that program comprehension is improved by at least two measures: ability to effect a solution (indicative of high comprehension due to successful application of the learned concepts); and ability to correctly recall and describe the purpose of the program, the missing portions of the program, and several specifics about the program as written (modified CLOZE (Entin, 1984; Taylor, 1953) measures of comprehension).

As in Experiment One, comments from subjects tend to support the research hypothesis, and suggest that the features of the literate programs that are different (documentation, code style, program format, etc.) are in fact the ones that are perceived as helpful to problem solving when present, and as a hinderance to problem solving when not present or when what is present is informationally inadequate. Subject two commented:

The main help [in solving the problem] was in the documentation, especially the algorithm. This made it extremely easy to locate the missing [subroutine] call and [the missing] subroutine.

Subject five also indicated that the algorithm contributed most to solving the problem. Subject ten wrote:

It just takes time (a very short time) to get the use of the Lit program style. ... I think I learned a lot in a very short time. It all came together at once. Everything was very logical and understandable.

Subject nine commented (emphasis added):

Reading the general problem [description] and then [the] algorithms helps first. Then I look [to see] if all the code of the main [driver] seems to match the algorithm. Next I check calls to sub[routine]s. *I sure wouldn't want to try this interpreting code alone.* The first time through [the program] I didn't catch the [missing] subroutine, but I hit [the] index and caught it [the] second time through [the program].

Finally, a comment from subject 11, who worked with a traditional program: "I can't think of one thing that I found helpful [for modifying the program]."

Thus, as in Experiment One, subjective commentary by the subjects supports the research hypothesis and gives a strong indication of the elements that were perceived as helpful by the subjects.

Finally, another result deserves commentary. The percentage of subjects who found a solution in Experiment Two (64 percent) was roughly equivalent to the percentage of subjects who found a solution in Experiment One (60 percent). This was not an expected result. It was expected that the percentage of subjects finding a solution in Experiment Two would be much greater (although perhaps not significantly) than for Experiment One, due to the

subjects familiarity with the task domain. However, this was not the case. This suggests that the literate programming paradigm may be just as effective in assisting with program comprehension for programs in unfamiliar task domains as it is for programs in familiar task domains. Intuitively, one would assume that the performance of the subjects would be worse as their subjective evaluation of the problem's difficulty increased. Yet, the percentage of novice subjects finding a solution remained roughly the same in both experiments, even though the mean level of difficulty was perceived as much higher in Experiment One than it was in Experiment Two. This suggests that the Lit style of formatting and documenting code may boost the comprehensibility of programs in unfamiliar task domains to that of programs in familiar task domains. Although this is not a part of the major research hypothesis, an in-depth look in the literature at knowledge of task-content and knowledge of task-process as it relates to perceptions of task-complexity and ability to perform a task could shed some light on this counter-intuitive result.



## CHAPTER VI

### EXPERIMENT THREE

#### SUBJECTS

For Experiment Three, 36 intermediate subjects were recruited from an undergraduate computer science course for computer science majors. All of the subjects had extensive prior experience with computers and computer programming. All subjects had just completed a three month course on algorithmic languages and compiler design. Subjects were familiar with recursive descent parsing algorithms, the C programming language, and standard modular programming techniques. Subjects were not familiar with Lit style programs and were given no special instructions on how to read or understand them.

The subjects were randomly divided into two groups of equal size; one group received the literate program, the other group received the traditional modular program.

Experiment Three involved programming in a task domain all of the subjects were very familiar with (recursive descent parsing).

## MATERIALS

The program the subjects worked with in Experiment Three (familiar task domain) was designed and written by the researcher and involved recursive descent numeric expression evaluation. Omitted from the program was a routine that handled the unary minus operator. Also omitted was the call to the missing routine.

## PROCEDURE

Experiment Three was a controlled study. Each subject was given a sheet of instructions and the following verbal instructions.

You have been given the task of maintaining a computer program. The original author completed the analysis and design of the program, but did not have time to complete the coding. Your job is to determine what functional units of code have been left out and to create them and indicate where in the program they would be inserted. The code that is missing is one or more subroutines or functions, and the calls to those routines or functions. You must also insert the calls to the routines you create in the appropriate place or places in the program for the solution to be considered correct.

The subjects were given either the literate or traditional modular program to modify and were instructed to use any of the reference materials provided, if needed. A time limit of 60 minutes to complete the modifications had been established in a previous pilot study. Subjects were notified when only 10

minutes were remaining. After completing the program modifications or running out of time, subjects filled out a questionnaire.

## RESULTS

Results were analyzed using one-way nonparametric analysis of variance. Analysis of variance of group performance (Table XIII) showed that 39 percent of the literate program group found either a completely correct solution or a functionally correct solution with syntax errors and none of the traditional modular program group found a solution. The finding was significant ( $F(1,35) = 10.82$ ,  $p < .0023$ ,  $\eta^2 = .24$ ). Also significant was that 33 percent of the literate program group found a functionally correct alternative solution with syntax errors ( $F(1,35) = 8.50$ ,  $p < .0062$ ,  $\eta^2 = .20$ ). Results (see Table XIII) also showed that group differences were significant with regard to attempts at modifying a section or sections of code that did not require a modification to solve the problem ( $F(1,35) = 39.36$ ,  $p < .0001$ ,  $\eta^2 = .54$ ); only 6 percent of the subjects in the literate program group attempted to modify a section of code that did not require a modification, but 78 percent of the subjects in the traditional program group made such modifications. Also significant were the group differences related to finding where to insert the missing calls to the missing subroutines ( $F(1,35) = 39.36$ ,  $p < .0001$ ,  $\eta^2 = .54$ ). In addition, analysis showed that the ability to insert the call correctly (see Table XIII) was

significantly better for the literate program group than it was for the traditional program group ( $F(1,35) = 10.82$ ,  $p < .0023$ ,  $\eta^2 = .24$ ).

TABLE XIII

## GROUP PERFORMANCE PERCENTAGES FOR EXPERIMENT THREE

Comprehension Criteria	Literate Program Group Performance	Traditional Program Group Performance
Completely Correct	0%	0%
Functionally Correct	39%	0%
Incorrect	61%	100%
Found Missing Call	78%	6%
Inserted Call Correctly	39%	0%
Did not Modify Wrong Code	94%	22%
Described Problem Correctly	100%	88%
Number of Missing Subroutines Identified Correctly	88%	64%
Accurately Described Function of Missing Routines	73%	27%

The reconstructive measures were not as dramatically different as those in Experiment One. There was no significant group difference in ability to describe the overall functionality of the program; all of the subjects in the literate program group accurately described the program, and 88 percent of the subjects in the traditional modular program group accurately described the program. This finding is most likely due to the subjects' familiarity with the task domain. The subjects in the literate program group did not significantly

outperform the subjects in the traditional modular program group in identifying the number of missing subroutines. However, as shown in Table XIII, subjects in the literate program group significantly outperformed the subjects in the traditional program group in accurately describing the intended functionality of the missing subroutine ( $F(1,29) = 7.80, p < .0093, \eta^2 = .22$ ). Finally, subjects in the literate program group also outperformed subjects in the traditional program group in the mean time required to complete the modifications ( $F(1,35) = 5.39, p < .027, \eta^2 = .14$ ); the mean time for the literate program group was 45.83 minutes, while the mean time for the traditional program group was 54.28 minutes. Timing information would be more meaningful if subjects had been given an unlimited amount of time to solve the problem, and the mean time to find a solution was calculated. However, it would also have made it impossible to gather the accuracy statistics if all subjects were allowed to find a solution before terminating the experiment. In any case, the subjects in the literate program group did in fact perform better in the time dimension, and comprehension was measurable not only by accuracy, but also by time, for this experiment. Time was measured and calculated without any log transformation on the times, which may have skewed the result to show a significant difference existed when it did not. No efforts were made to check for this; time is not the measure of comprehension being used for this experiment.

### Analysis of Subjects' Subjective Data

The questionnaire was analyzed in order to gauge the subjects' perception of which elements of the program were aids in solving the problem, and which elements of the program caused difficulty in solving the problem. The results are presented in Tables XIV through XVIII. All factors were perceived as more of a help than a hinderance for the entire literate program group. The most helpful factors were documentation (82 percent), problem description (41 percent), input specifications (36 percent), code style, program format, and output specifications (24 percent), and the table of contents (18 percent). Only one subject indicated that the code style was a hinderance.

**TABLE XIV**

**EXPERIMENT THREE: PERCEPTIONS OF SUBJECTS GIVEN  
LITERATE PROGRAMS**

Program element	Helped with problem solving	Hindered problem solving
Documentation	82%	12%
Code Style	24%	5%
Table of Contents	18%	5%
Input Specifications	36%	23%
Problem Description	41%	11%
Indentation	5%	5%
Program Format	24%	5%
Output Specifications	24%	11%

This subject was unable to find a solution, perhaps due to the non-traditional format of the source code, as it differs distinctly from the Kernighan and Ritchie (1988) style of C program coding with which the subject was familiar.

Of the subjects who found a solution (N=7) and answered the questions pertaining to what contributed most to solving the problem, Table XV describes which elements were perceived as helpful in solving the problem or hindered problem solving. Documentation was perceived as helpful by all of the subjects who found a solution. Code style was also perceived as helpful by 42 percent of the subjects in this group, and was not perceived as a hinderance by any subjects in this group. The input specifications were perceived as a hinderance

TABLE XV

**EXPERIMENT THREE: PERCEPTIONS OF SUBJECTS GIVEN  
LITERATE PROGRAMS WHO FOUND A SOLUTION**

Program element	Helped with problem solving	Hindered problem solving
Documentation	100%	0%
Code Style	42%	0%
Table of Contents	0%	14%
Input Specifications	29%	57%
Problem Description	42%	14%
Indentation	14%	14%
Program Format	14%	14%
Output Specifications	29%	14%

by more subjects (57 percent) than perceived it as helpful (29 percent). The table of contents was not perceived as helpful, and one subject from this group found it to be a hinderance in problem solving.

Table XVI describes perceptions of the subjects given literate programs who did not find a solution (N = 11). One subject in this group did not respond to any of the subjective questions thus the N for this group decreased by one to N=10. Note that all of the factors were indicated as a help by as

TABLE XVI

EXPERIMENT THREE: PERCEPTIONS OF SUBJECTS GIVEN LITERATE PROGRAMS WHO DID NOT FIND A SOLUTION

Program element	Helped with problem solving	Hindered problem solving
Documentation	70%	20%
Code Style	10%	10%
Table of Contents	30%	0%
Input Specifications	40%	0%
Problem Description	40%	10%
Indentation	0%	0%
Program Format	30%	0%
Output Specifications	20%	10%

many subjects or more subjects than indicated the same factor was a hinderance. Over 70 percent of the subjects in this group indicated the documentation was helpful, 30 percent indicated the table of contents was helpful, and 30 percent indicated the program format was helpful. 20 percent



of the subjects in this group indicated that the documentation was a hinderance to problem solving.

Of the subjects given the traditional programs, 50 percent indicated that documentation was a hinderance to problem solving. Because they did not have the augmented documentation of the literate program, this result is not surprising. The Lit style program documentation was rated as helpful by all subjects that found a solution. This suggests that not only is the documentation helpful, but the format and presentation of the documentation plays an important role in its perceived usefulness. Because all in-line documentation was identical, the only difference between the Lit style programs and the traditional modular programs is the additional documentation; specifically, the content, organization, and format, and presentation paradigm. Table XVII also shows that the problem description and the output specification were also indicated as hindrances by more subjects than indicated that those factors were helpful. Code style was indicated as helpful by 39 percent of the subjects, and the program format indicated as helpful by 28 percent of the subjects. This result is not surprising considering that the subjects had to gain comprehension from the source code, and the program format and the consistent code style would be the two most important aids to comprehension that are in the traditional modular programs. Although code style was indicated as helpful, indentation was perceived as helpful by only 33 percent of these subjects. This result is unexplainable since indentation is a major portion of a

consistent code style. Subjects could have been thinking of some other element of coding style (naming conventions, use of white space, etc.).

TABLE XVII

EXPERIMENT THREE: PERCEPTIONS OF SUBJECTS GIVEN  
STANDARD PROGRAMS

Program element	Helped with problem solving	Hindered problem solving
Documentation	16%	50%
Code Style	39%	28%
Table of Contents	0%	0%
Input Specifications	22%	22%
Problem Description	22%	33%
Indentation	33%	0%
Program Format	28%	11%
Output Specifications	11%	17%

Overall perceptions (N=34) of experiment instructions, problem description, input specifications, and output specifications rated as "easy to understand" or "not easy to understand" are presented in Table XVIII.

As shown in Table XVIII, the input and output specification were rated as easy to understand by an overwhelming majority of the subjects, yet some subjects still found them as hindrances to problem solving. This result is unexplainable; it may be that the input and output specifications were easy to understand, but were perceived as incomplete, or difficult to implement, although the problem did not require the subjects to do anything with the input or output portions of the program. Unlike Experiment One, 85 percent of the

subjects in Experiment Three found the problem description easy to understand. This is most likely due to the fact that subjects were familiar with the task domain and the terminology used to describe the required processing was familiar to the subjects. It was expected that for problems in familiar task

TABLE XVIII  
EXPERIMENT THREE: SUBJECTS' EVALUATIONS OF  
EXPERIMENTAL MATERIALS

Experiment material	Easy to understand	Not easy to understand
Instructions	97%	3%
Problem Description	85%	15%
Input Specification	91%	9%
Output Specification*	88%	12%

\* indicates N=33 for this variable

domains the problem description would be rated as easy to understand, and that the perception of the level of difficulty of the problem would be low. The perceptions of the subjects in Experiment Three indicate this is true; the perceived level of difficulty (obtained by finding the mean of the difficulty scale ranging from 1 (very difficult) to 5 (very easy) for all subjects) was 2.80, indicating that subjects perceived the level of difficulty of the problem as between difficult and easy.

It was also expected that the perception of difficulty for the literate program group would be perceived as less difficult than the perceptions of the traditional program group. The literate program group rated the level of

difficulty as much less difficult than did the traditional program group. Group means indicated that the perception difference was 1.19 levels of difficulty more difficult for the traditional program group (2.22) than it was for the literate program group (3.41). No tests for significance were performed.

## DISCUSSION

The results of Experiment Three also indicate that the Lit style literate programs are a more natural form for formatting and documenting code which are superior to traditional methods in assisting with program comprehension. Results also indicate that program comprehension is improved by at least three measures: ability to effect a solution (indicative of high comprehension due to successful application of the learned concepts); ability to correctly recall and describe the purpose of the program, the missing portions of the program, and several specifics about the program as written (modified CLOZE (Entin, 1984; Taylor, 1953) measures of comprehension); and amount of time required to effect a solution.

The percentage of intermediate subjects finding a solution was much lower than it was in Experiments One and Two. This is probably due to the complexity of the material (recursive descent parsing is not a simple concept, per-se) and the small amount of time allotted for the experiment. Several subjects noted on the post-experiment questionnaire that there was not enough time to complete the experiment. Another possibility is that, at this point in their

familiarity with computer programming, the novel presentation paradigm, indentation, and augmented documentation were so much different from what the intermediate subjects tend to think of as a program, that it took time to adapt to the literate programs and to be able to utilize the information contained therein.

The results are encouraging. Intermediate subjects given literate programs also significantly outperformed intermediate subjects given standard programs. In addition, the comments of several subjects that were given the traditional programs underscore the need for an altered paradigm that can assist in program comprehension. The type of problem (recursive descent parsing) requires either explicit documentation of data flow and control flow, or the ability to do extensive symbolic computation and a time consuming code-walkthrough of the algorithm, in order to find the problem. The literate programs had the documentation, and the subjects with the traditional programs were forced to take the second, more time consuming, avenue of program maintenance. For example, subject 28 wrote:

The depth of the calling [sequence] where the missing procedure should have been [contributed most to the difficulty of modifying the program]. (I had to trace the program[s recursive calls] several levels deep.)

Subject 31 commented:

I am not sure that I finished doing the modifications or not because too many functions [had] to [be] chase[d] through, so it was hard to keep track.

And subject 34 suggested that the problem could not be solved unless the subject could run it and observe the run time behavior to determine the problem with the program.

In summary, Experiment Three supports the hypothesis that programs should be written in a different format. The Lit style programming format is one such possibility which has now been shown to be significantly more comprehensible than the format of the traditional programs used in this study. Because none of the subjects had prior experience with Lit style literate programs, the results of this study are very encouraging. In Experiments One and Two subjects had familiarity with both traditional modular programming and literate programming using the Lit system. In experiment three, subjects had no experience only with Lit style programs, yet a large percentage of them were able to effectively utilize the programs' comprehension aids for problem solving. Because no additional instruction in the use of literate programs was given to the subjects, this suggests that the Lit presentation paradigm is a more natural form for information transfer which is superior to that of traditional modular program listings. In addition, the subjective evaluation of many of the subjects supports the ideas on which literate programming is based, and anecdotal commentary by the subjects points directly to some of the flaws of the traditional programs, and some of the strengths of the literate programs suggested by the research hypothesis.

For future research with intermediate programmers, it would be interesting to see if the percentage of subjects finding a solution to a problem in an unfamiliar task domain would be about the same as the percentage that found a solution in this experiment. Such a finding would be consistent with the finding in the novice experiments (Experiment One and Experiment Two), and could suggest new research questions for exploration; in particular, can the inclusion of certain types of documentation ameliorate or extinguish the maintenance problems associated with lack of task domain familiarity (see the discussion section of Experiment Two for more suggestions).

## CHAPTER VII

### CONCLUSIONS

The results of all three experiments indicated that Lit style literate programs greatly enhance computer program comprehension. This study emphasized the use of typographic style, program organization, and documentation that have been empirically shown to assist in program comprehension, and demonstrates through empirical studies that application of these concepts in an automated system for program design and maintenance significantly impacts program comprehension in a positive manner.

### GENERAL PRINCIPLES FOR ASSISTING PROGRAM COMPREHENSION

The Lit style literate programming format shows that the use of the following principles, when incorporated into a program presentation paradigm, significantly aid computer program comprehension.

- (1) Macro typographic principles including:
  - a) Make obvious the components and organization of the program
  - b) Identifying the purpose and use of each program component
  - c) Make the program easy to browse and readable by using a familiar information-transfer paradigm (*i.e.*, a book)
  - d) Identify and document the control flow of the program



- e) Identify and document the data flow of the program
- f) Provide cues to enable non-linear code searches (*e.g.*, Table of Contents, Index, cross reference listings, etc.)

(2) Micro-typographic principles including:

- a) Make obvious the logical sections of program modules using highlighting.
- b) Use spatial cues and white space to indicate statement groupings and separation.
- c) Use point size changes, white space, and highlighting to make the control flow and information flow within and between modules obvious.
- d) Use point size changes, white space, and highlighting to indicate separations in program sections.
- e) Identify the use and purpose of each section.
- f) Use consistent indentation for language constructs.

(3) Documentation principles including:

- a) Explicitly document the usage of variables.
- b) Explicitly document module declaration and usage.
- c) Explicitly document all subroutine and function calls made by every routine.
- d) In each module, explicitly document which subroutines and functions call the module.

- e) Explicitly document the algorithms in use.
- f) Explicitly document control and information flow within and between modules.
- g) Include *design history* documentation.
- h) Include *anticipatory* documentation.
- i) Explicitly document any obscure language features that are being used to implement the program.
- j) Include ample task domain information, examples, and documentation that explicitly links the domain model to the program model, so that programmers with little or no familiarity with the task domain can perform program maintenance.
- k) Allow for inclusion of graphical documentation such as equations, pictures, tables, and charts; this type of information should be included where a written description can't fully convey the concepts, layout, usage, or relationships without excess verbiage.

For more specific information on the document formatting conventions used by Lit, see Appendix L.

## CHAPTER VIII

### DISCUSSION

The results indicate that program comprehension is improved based on at least two measures: ability to effect a solution (indicative of high comprehension due to successful application of the learned concepts); and ability to correctly recall and describe the purpose of the program, the missing portions of the program, and several specifics about the program as written (modified CLOZE (Entin, 1984; Taylor, 1953) measures of comprehension). Although not explicitly part of the research hypothesis, use of Lit style programs also reduced the time needed for program comprehension in Experiment 3. The most encouraging facet of these experiments is that significant results were obtained when the statistical power to detect such effects was quite low due to the sample sizes.

It is also interesting to note that the largest difference between the groups given the literate programs and the groups given standard modular programs was in the group working with an unfamiliar task domain. As has been noted, the development of a domain model and the ability to link the domain model with the program model to form the global model of the program is essential to program comprehension. Apparently, the literate program allowed more subjects to form a global model and make the required

modifications; the subjects with the non-literate programs apparently could not develop a global model and thus were unable to make the required modifications. This is impressive, in that both groups overwhelmingly rated the problem description as difficult to understand, and as a hinderance to problem solving. Yet, the literate program group was able to overcome these difficulties and 60 percent found a solution. This indicates that the literate program did, in fact, contain features which assisted the programmer in understanding both the domain model and the program model, and assisted in linking up these two models into a global model of program design. Even more impressive is that the global model formed by 60 percent of the subjects who were given literate programs allowed them to make the required modifications to the program in a very short time period.

Additionally, the subjective evaluation of many of the subjects supports the ideas on which literate programming is based; that understanding the task domain and the programming domain and the link between the two facilitates comprehension. Anecdotal commentary by the subjects points directly to some of the flaws of the traditional programs, and some of the strengths of the literate programs suggested by the research hypothesis.

The implications for the use of Lit style literate programming are wide-ranging: The time that is currently devoted to program maintenance activities may be substantially reduced; Program development and debugging activities would be assisted by the Lit programming paradigm; And,

programmers modifying programs that model unfamiliar task domains may be substantially enabled if the program being modified is written as a literate program. Companies could require less familiarity with the task domain on the part of their programmers. Educators could present students with more complex programs than the usual simple examples used for teaching. These examples could be more complex programmatically and algorithmically. The choice of the task domain would not have to be limited to the simple examples of scientific problems currently used in most curricula. Non-computer scientists could understand (and maybe even modify) applications for their own use, or for the purposes of verifying methodology and application for a particular purpose, or just to satisfy curiosity. Most importantly, the results of this study suggest that maintenance programmers can be significantly enabled by Lit style literate programs. This will have a direct impact on programmer productivity; it should increase significantly. Literate programs are easier to comprehend and thus easier to maintain. Since maintenance is the largest percentage of the software development cycle, reducing the time spent in the maintenance portion of the development cycle should significantly decrease the overall expense of the cycle, and thus improve the profit margin of software for software developers.

Improving the way programs are written using expository writing as the model for development of computer programs may drastically change the way programs are written and read; changes that will help remove some of the

mysticism that surrounds programmers and programming. Writing programs which are viewed as expository technical writing describing a solution to a problem is preferable to writing them such that only the original programmer can hope to make sense of the program, and then only if he/she has been working on it steadily.

On the down side, it does take more time to produce a literate program. Much of this may be due to the demanding housekeeping tasks required of the programmer, such as keeping the documentation and code synchronized. In addition, there are very few programming tools which are designed to facilitate program comprehension. I suggest that many comprehension problems could be overcome if tools, designed using empirically derived principles for facilitating program comprehension, were developed and integrated in the standard environments of computer programmers. For example, a language intelligent (not just sensitive) editor could automatically highlight and indent control structures consistently, create a table of contents, cross reference guides, and an index as the programmer types in the code. This could be integrated into a programming environment that would allow the programmer to program in a way which is best for them (e.g., allowing focused, browsing, top-down, and bottom-up searches of code within the editor, etc.). Many programming systems do address some of these issues, but I believe the main reason programming is still such a difficult task is the lack of adequate programmer productivity and support tools, and reliance on an outdated and

informationally deficient programming paradigm which does not assist programmer comprehension strategies. Systems like Lit can have a profound effect on program comprehension, and thus on programmer productivity. Automating the formatting and presentation of computer programs would allow programmers to concentrate on programming. Unlike past approaches to improving the presentation paradigm, the Lit approach would not add to the cognitive load of the programmer the language independent typographic style principles that must be used to produce program listings that assist with program comprehension.

## CHAPTER IX

### FUTURE DIRECTIONS

There are many questions related to program comprehension that are not addressed by the current studies. It is my hope to address the question of how to incorporate information delivery technologies with literate programming such that entire programming systems are aids in program comprehension. I envision the incorporation of Lit style literate programming into a CASE framework in which individual tools cooperate through an object messaging system to provide the programmer with a comprehensive programming environment that assists in the design, coding, testing, documenting, and maintenance of computer programs.

As additional empirically derived principles related to information-presentation and content are identified, it will become more important to address programming as a system of complex, interrelated activities all of which must be enabled through the use of technology.

For example, how should a flexible code browser be designed? Should the program document contain all of the textual, graphical, and other information for a program? Or, should programs be viewed as hypertext documents with links to graphical and other pieces of information that can be browsed on demand? Should literate programming systems use an object



database model to store and retrieve program fragments using browsers that allow the programmer to control the presentation of information on an as needed basis? These questions, and many other like them, must be addressed before programmer productivity can be significantly increased.

Finally, not all of the typographical style elements identified by Oman and Cook (1990b) are currently implemented in the Lit system; more will be added as it is determined which elements aid in program comprehension, and which elements or combinations of elements may detract from program comprehension of Lit style programs due to information overload.

All of the above questions present serious challenges to the experimenter. It is my hope to investigate each of the ideas in future studies, and to modify the Lit system to incorporate each of the elements that are found to enhance programmer comprehension strategies; hopefully, the end result will be a system that assists in most program development and maintenance activities.

## REFERENCES

- Adelson, B. (1981). Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory and Cognition*, 9, 422-433.
- Adelson, B., Littman, D., Ehrlich, K., Black, J., & Soloway, E. (1985). Novice-Expert Differences in Software Design. In *Human-Computer Interaction*, B. Shackel (Ed.). 473-478.
- Anderson, J. R. (1980). *Cognitive Psychology and its Implications*. New York; W. H. Freeman.
- Basili, V. and Mills, H. (1982). Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, 270-283.
- Beck, K and Cunningham, W. (1987). The Literate Program Browser. *Tektronix Technical Reports CR-86-52*.
- Bendifallah, S. & Scacchi, W. (1987). Understanding software maintenance work. *IEEE Transactions on Software Engineering*, March, 311-323.
- Bertholf, C. (1989). Lit: A Language Independent System for Abstraction Oriented Literate Programming. *Academic Computing Services Technical Documents*, Portland State University.
- Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18, 543-554.
- Chase, W. & Simon, H. (1973). Perception in chess. *Cognitive Psychology*, 4, 55-81.
- Cunningham, W. and Beck, K. (1987). Scroll Controller Explained. *Tektronix Technical Reports CR-86-51*.
- DeGroot, A. (1965). *Thought and Choice in Chess*. Mouten: Paris, France.
- Egan, D. & Schwartz, B. (1979). Chunking in recall of symbolic drawings. *Memory and Cognition*, 7, 149-158.

- Entin, E. B. (1984). Using the CLOZE Procedure to Assess Program Reading Comprehension. *Papers of the ACM SIGCSE Technical Symposium on Computer Science Education*, 15. Philadelphia: ACM Press. 44-50.
- Ehrlich, K. Soloway, S. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. In *Human Factors and Computer Systems*, J.C. Thomas and M.L. Schneider (Eds.) Norwood, NJ: Ablex. 113-133.
- Fjeldstad, R. K. and Hamlen, W. T. (1983). Applications program maintenance study: Report to our respondents. *Tutorial on Software Maintenance*, G. Parikh & N. Zvegintzov (Eds.), 13-27. IEEE/CS Press, Silver Spring, Md.
- Gauthier, R. & Ponto, S. (1970). *Designing Systems Programs*, Prentice-Hall, Englewood Cliffs, N.J.
- Hamlet, D. (1977). Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4). 279-290.
- Hoare, C. (1973) Hints on Programming Language Design. Invited address at *ACM SIGACT/SIGPLAN Symposium on Programming Language Design*.
- Kernighan, B. W. & Plauger, P.J. (1974). *The Elements of Programming Style*. McGraww-Hill, New York, NY.
- Kim, J. & Lerch, F. (1992). Toward a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition in Software Methodologies, in P. Bauersfeld, J. Bennet and G. Lynch (Eds.), *Proceedings of CHI '92: Human Factors in Computing Systems*. Monterey, CA. 489-498.
- Knuth, D. (1984). Literate Programming. *The Computer Journal*, 27(2), 97-112.
- Koenemann, J. and Robertson, S. P. (1991). Expert Problem Solving Strategies for Program Comprehension. *Proceeding of CHI '91: Human Factors in Computing Systems*. 125-130.
- Korson, T. & Vaishnavi, V. K. (1986). An empirical study of the effects of modularity on program modifiability. In E. Soloway & S. Lyengar (Eds.): *Empirical Studies of Programmers. First Workshop*. 168-186.
- Ledgard, H. & Tauer, J. (1987). *Professional software, Volume II, Programming Practice*. Addison-Wesley: Reading, Mass.

- Levi, S. (1987). WEB adapted to C, another approach. *TUGboat*, 8(1), 12-14.
- Littman, D., Pinto, J., Letovsky, S., and Soloway, E. (1986). Mental Models and Software Maintenance. In E. Soloway & S. Iyengar (Eds.): *Empirical Studies of Programmers. First Workshop*. 80-98.
- Love, T. (1977). An experimental investigation of the effect of program structure on program understanding. *ACM SIGPLAN Notice* 12(3). 105-113.
- McKeithen, K., Reitman, J., Rueter, H., and Hurtle, S. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13. 307-325.
- Miara, R. J., Musselman, J. A., Navarro, J. A. & Shneiderman, B. (1983). Program indentation and comprehension. *Communications of the ACM*, 26(11). 861-867.
- Oman, P and Cook, C. (1990a). The Book Paradigm for Improved Maintenance. *IEEE Software* 7(1). 39-45.
- Oman, P. and Cook, C. (1990b). Typographic Style is More than Cosmetic. *Communications of the ACM*, 33(5), 506-519.
- Parikh, G. & Zvegintzov, N. (1983). The world of software maintenance. In *Tutorial on Software Maintenance*, G. Parikh & N. Zvegintzov, Eds. 1-3. IEEE/CS Press, Silver Spring, Md.
- Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12). 1053-1058.
- Pennington, N. (1987). Comprehension Strategies in Programming. *Empirical Studies of Programmers. Second Workshop*. Norwood, NJ: Ablex. 100-113.
- Rapps, S. & Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4). 367-375.
- Ratcliffe, B. & Siddiqi, J. (1985). An Empirical Investigation Into Problem Decomposition Strategies Used in Program Design. *International Journal of Man-Machine Studies*, 22(1). 77-90.
- Rosson, M. B. & Alpert, S.R. (1990). The Cognitive Consequences of Object-Oriented Design. *Human Computer Interaction*, 345-380.

- Rosson, M. B. & Gold, E. (1989). Problem-Solution Mapping in Object-Oriented Design. In *Object-Oriented Programming: Systems, Languages and Applications: OOPSLA '89 Conference Proceedings*, N. Meyrowitz (Ed), 7-10.
- Santa, J. L. (1977). Spatial Transformations of Words and Pictures. *Journal of Experimental Psychology: Human Learning and Memory*, ???, 418-427.
- Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computing and Information Science*, 5(2). 123-143.
- Shneiderman, B. and Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences*, 8. 219-233.
- Shneiderman, B. & McKay, D. (1980). Experimental Investigations of Computer Program Debugging and Modification. *Software Psychology*. Winthrop Publishers, Cambridge Mass. 72-74.
- Sewell, E. W. (1987). How to MANGLE your software: the WEB System for Modula-2. *TUGboat*, 8(2). 118-122.
- Sheil, B. A. (1981). The psychological study of programming. *Computing Surveys*, 13(1). 101-120.
- Softky, S. (1983). *The ABC's of Developing Software*. Menlo Park, Ca: ABC Press.
- Soloway, E., Bonar, J., Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26(11). 853-860.
- Soloway, E. and Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions of Software Engineering*, 10(5), 595-609.
- Taylor, W. (1953) Cloze Procedure: A New Tool for Measuring Readability. *Journalism Quarterly*, 30, 415-433.
- Thimbleby, H. (1986). Experiences of 'Literate Programming' Using cweb. *Computing Journal*, 29(3), 201-211.
- Van Dijk, T. and Kintsch, W. (1983). *Strategies of Discourse Comprehension*. New York; Academic Press.

- Wiedenbeck, S. (1986). Processes in Computer Program Comprehension. In E. Soloway & S. Lyengar (Eds.): *Empirical Studies of Programmers. First Workshop*. 48-57. Norwood, NJ: Ablex.
- Wiedenbeck, S. and Scholtz, J. (1989). Beacons: A Knowledge Structure in Program Comprehension. In G. Salvendy & M. Smith (Eds.): *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*. Amsterdam: Elsevier. 82-87.
- Zehr, W. (1992). CASE Technology: The Shift Towards Immediate Gratification. Invited Presentation at OACIS '92: *Oregon Advanced Computing Conference*.

## APPENDIX A

### EXPERIMENT CONSENT FORM

## Consent to Participate in an Experimental Study

Title of Proposed Study: Program Comprehension of Literate Programs by Novice, Intermediate and Expert Programmers

Investigator: Christopher F. Bertholf

### Invitation to Participate:

You are invited to participate in this research study because you are enrolled in an undergraduate computer science course and you fit into one of the following categories:

1. You are a novice programmer in an introductory programming class
2. You are an intermediate or expert programmer with 2 or more years of computer programming experience.

### Purpose of the Study:

This research investigates program comprehension of Literate Programs as compared with comprehension of traditional structured programs.

### Explanation of Procedures:

You will be asked to read a program and determine what functions or subroutines are missing, and where the calls to those routines should go in the main program. You will also be asked to generate the missing function or subroutine, and insert the missing call(s) in the main driver of the program. Your name will not be associated in any way with the testing materials; it is completely anonymous. After completion of the test, data will be compiled from your and other tests, a statistical analysis will be performed, and the data will be used for a Masters Thesis in Computer Science.

You will be paid \$5.00 for your participation in the study. The study will not exceed one hour in length.

### Potential Risks and Discomforts:

The methods used in this experiment present no danger to you or any other persons.

### Potential Benefits:

You will receive \$5.00 for participating in this study. In addition, it is hoped that the results of this study will aid in providing programmers with a programming paradigm which results in more readable, more maintainable, and more understandable computer programs.

### Assurance of confidentiality:

There will be absolutely no data which connects you to the testing materials. The study is completely anonymous in this respect.



**Withdrawal from the Study:**

Participation is voluntary. Your decision whether or not to participate will not affect your present or future relationship with Portland State University. If you decide to participate, you are free to withdraw your consent and to discontinue participation at any time.

**Offer to Answer Questions:**

If you have any questions, please do not hesitate to ask. If you think of questions later, please feel free to contact the investigator below.

If you have any additional questions concerning the rights of research subjects, you may contact the Human Subjects Research Review Committee, Office of Grants and Contracts, 303 Cramer Hall, PSU. Telephone: (503) 725-3417.

YOU ARE VOLUNTARILY MAKING A DECISION WHETHER OR NOT TO PARTICIPATE. YOUR SIGNATURE INDICATES THAT YOU HAVE DECIDED TO PARTICIPATE HAVING READ THE INFORMATION PROVIDED ABOVE. YOU WILL BE GIVEN A COPY OF THIS CONSENT FORM TO KEEP.

\_\_\_\_\_  
Signature of Subject

\_\_\_\_\_  
Date

\_\_\_\_\_  
Signature of Investigator

\_\_\_\_\_  
Date

Investigator: Chris F. Bertholf (503) 725-4052 or (503) 725-3367

## **APPENDIX B**

### **INTRODUCTION TO THE EXPERIMENT**

### Introduction

The following experiment is designed to measure program comprehension of Literate programs versus programs written with a standard structured programming methodology.

Please follow the instructions below EXACTLY. You will be asked to fill out a report at the end of the experiment.

### Instructions

You have been handed a computer program. The program is not finished. One or more lines of code are missing from the main program. Additionally, one or more subroutines or functions are missing from the program. Your job is to complete the program. To do this, you must determine what subroutines or functions are missing, and which lines of code from the main program are missing. You must then write the missing subroutine(s) and/or function(s) and insert the routines and the missing calls to the routines in the appropriate place in the unfinished program.

With each program you have also been given a problem description that spells out what the program is supposed to do, the input required, and the output specifications.

A programming language reference is available, should you need it to complete this experiment. It is attached to this packet following the program.

There is a 60 minute time limit to complete the modifications to each program. If you have not completed the modifications when the time limit is up, do not worry, this is an expected result for some of the programs.

When you finish, the experimenter will record the elapsed time it took you to effect a solution. You will be asked to answer some questions about the program and the experiment. If you finish prior to the time limit, be sure to have the experimenter note the time it took you to complete the program modifications.

## **APPENDIX C**

### **POST-EXPERIMENT QUESTIONNAIRE**

# QUESTIONNAIRE

Questions about the program and the experiment:

Did you use the language reference? \_\_\_\_\_

Briefly describe what the program is supposed to do?

How many subroutine(s) or function(s) did you feel were missing? \_\_\_\_\_

Briefly describe the purpose of the subroutine(s) and/or functions(s) that were missing?

On a scale from 1 to 5, 1 being totally incorrect, 5 being totally correct, rate the correctness of the modifications you made to the main program. Do not ignore the possibility of syntax errors.

Totally incorrect					Totally correct
1	2	3	4	5	

On a scale from 1 to 5, 1 being totally incorrect, 5 being totally correct, rate the logical correctness of the subroutine(s) and/or functions you wrote. Ignore the possibility of syntax errors.

Totally incorrect					Totally correct
1	2	3	4	5	

In your opinion, how difficult was it to make the modifications?

very difficult   somewhat difficult   difficult   somewhat easy   very easy

Please circle the features of the program that contributed most to the difficulty of modifying the program.

Documentation	Problem description
Code style	Indentation
Index or Table of Contents	Program format
Input specifications	Output specifications

Other (indicate)

Please circle the features of the program that contributed most to the ease of modifying the program.

Documentation

Code style

Index or Table of Contents

Input specifications

Problem description

Indentation

Program format

Output specifications

Other (indicate)

Were the instructions clear and easy to understand?

(Yes/No) \_\_\_\_\_

If NO, How could the instructions have been improved?

Was the program problem description easy to understand?

(Yes/No) \_\_\_\_\_

If NO, How could the problem description have been improved?

Were the program input specifications easy to understand?

(Yes/No) \_\_\_\_\_

If NO, How could the input specifications have been improved?

Were the program output specifications easy to understand?

(Yes/No) \_\_\_\_\_

If NO, How could the output specifications have been improved?

If you DID NOT COMPLETE the modifications:

Explain why it was difficult to complete the modifications:

=====Filled out by the experimenter=====

Elapsed time to complete the experiment after reading the instructions:

\_\_\_\_\_.

Other notes:

## APPENDIX D

### EXAMPLE LITERATE PROGRAM (FROM EXPERIMENT 1)



*Chris F. Bertholf*

Portland State University

# **LeontiefModeling**

**Leontief Input/Output Analysis of Multiple Industry Model**

**Revision: 1.0**

**1 December 1991**

### Introduction

One interesting application of matrices is the Leontief Input-Output model, named for Wassily Leontief. The model Leontief developed is useful for predicting the effects to the economy of price changes or shifts in government spending.

Leontief's work divided the economy into 500 sectors, which was later reduced to a more manageable 42 departments of production. We can examine the working of the model with a very simplified view of the economy.

This program attempts to show a working three industry Leontief Input/Output model based on the mining, manufacturing, and energy industry. The model uses several subroutines from the LINPACK Scientific Subroutine Library for solving linear systems of equations.

-2-

### [1.0.0] Three Industry Leontief Model

Suppose we consider a simple economy as being based on three commodities: the mining industry, the manufacturing industry, and the energy industry. Suppose further that production of one dollars worth of mining requires \$0.40 units from mining, \$0.40 units from manufacturing, \$0.20 units from energy; Production of one dollars worth of manufacturing requires \$0.20 units from mining, \$0.40 units from manufacturing, and \$0.20 units from energy. Production of one dollars worth of energy requires \$0.10 units from mining, \$0.20 units from manufacturing, and \$0.40 units from energy. The following table summarizes this information:

Inputs:	Outputs		
	mining	energy	manufacturing
mining:	\$0.40	\$0.40	\$0.20
manufacturing:	\$0.10	\$0.20	\$0.40
energy:	\$0.20	\$0.40	\$0.20

Note that the sums of the columns need not add up to 1.00. This is because not all commodities or industries are represented in this model. In particular it is customary to omit labor from these models.

From the preceeding table we can form a matrix A called the technology matrix, (or the Leontief matrix):

$$A = \begin{bmatrix} 0.4 & 0.4 & 0.2 \\ 0.1 & 0.2 & 0.4 \\ 0.2 & 0.4 & 0.2 \end{bmatrix}$$

For this simplified model of the economy, not all information is contained in the Leontief matrix. In particular each industry has a gross production, the gross production can be represented as a column matrix X:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Where  $x_1$  is the gross production from mining,  $x_2$  is the gross production from manufacturing, and  $x_3$  is the gross production from energy. Those units of gross production not used by these industries are called surpluses, and may be considered as being available for consumers. If we place the surpluses in a column matrix D, then the surplus can be represented by the equation

$$x - Ax = D$$

which is equivalent to:

$$(I - A)x = D$$

where I is an identity matrix. This matrix equation is called the technology equation.

Note: An Identity matrix is a matrix in which every element is zero (0) except the elements on the diagonal, which have the value one (1).

If we call the matrix formed by  $(I - A)$  the Technology Matrix, and we represent this quantity with T. we can rewrite the equation as:

-3-

$$Tx = D$$

To find a solution to the system of equations there are several methods. The most straight-forward method is to do gaussian elimination to solve the equation:

$$Tx = D$$

Not only is this the most straight-forward solution, but compared to the other obvious solution (compute inverse of Technology matrix and multiply by D) it is far less expensive in terms of computational time.

Because the gaussian elimination problem has been solved by many programmers, we will use a library routine to do the factoring (decomposition) of the technology matrix (T), and another routine to solve the equation:

$$LUx = D$$

Where L is the lower triangular matrix and U is the upper triangular matrix found during decomposition of the Technology Matrix (T). Because the matrix may be singular, or very close to singular (to the working precision of the machine) we make sure that it is not before we solve the equation. This is done by checking the return value of the call to the routine that will do the decomposition on the technology matrix. If the value returned causes some wonder as to whether or not the matrix may be singular to the working precision, or if the return value indicates that there may be a divide by a zero pivot, we will ask the user if they would like us to check for singularity by estimating the condition number of the technology matrix. If the condition number is ok then we will go ahead and solve the above equation, if not we exit the program.

The subroutine we need are part of the LINPACK Subroutine Library for Genreal Matrices. The routines we will be using are SGECO (estimate the condition number of the matrix while decomposing it) and SGESL to solve a system of linear equations decomposed into an  $LUx = D$  format.

For a description of the subroutines themselves, the user is referred to chapter one the Linpack User manual: General Matrices.

-4-

## [2.0.0] The main driver

The main driver simply defines the variables required to generate and solve the model; The Leontief matrix is defined and initialized, the solution matrix (which contains the desired surplus production values) is defined and initialized, and the technology matrix is then formed from the Leontief matrix.

Once the technology matrix has been formed (by calling the TecMat routine) the Linpack subroutine SGECO is called to do the LU factorization of the technology matrix. If SGECO returns a non-zero value in the info variable, there is a possibility that the matrix is singular to the working precision of the machine, or that there is a possibility of a divide by zero (0) if SGESL is used to solve the system of equations. If the Info variable is not smaller than the working precision of the computer, the Linpack routine SGESL is called to solve the system of equations, and the results are printed on the terminal screen.

When Info is returned as non-zero, the user is asked if they wish to test for singularity. If the Matrix is singular to the working precision of the machine, the user is told and the program aborts. If the test for singularity fails (i.e., the matrix is not singular) then the program continues and the SGESL routine is called to solve the system of equations.

### General algorithm:

```

Initialize Leontief Matrix
Initialize Production matrix
Transform Leontief matrix into Technology matrix
Call SGECO to factor the Matrix
If Technology matrix may be singular
  Warn the user
  Test for singularity
  If the Technology matrix is singular
    Tell the user
    Abort the program
  Endif
Endif
Call SGESL to solve the system of equations
Print the resulting solution

```

### Calls:

```

TecMat - routine to form the technology matrix
ReadAR - routine to read an array
PrnWrm - routine that prints the singular matrix warning message
PrnSol - routine to print the solution

```

### Library routines used

#### FROM THE LINPACK LIBRARY

```

SGECO - Factor a matrix and estimate its condition number
SGESL - Solve a system of linear equations

```

### Called by:

Operating system

Variables:

Mat - The technology matrix  
 Prod - The solution matrix  
 Info - Holds estimate of singularity  
 IPvt - LINPACK uses this to store pivot information  
 Work - Work array for LINPACK  
 LDM - The leading dimension of Mat  
 Dim - The Dimension of Work, Prod, and IPVT

Program Leontief

```
C
C   This program tests several subroutines that were written to
C   solve variable sized Leontief Input/Output economy models.
C
C   Define the variables:
C
      Real Mat(3,3), Prod(3), IPVT(3), Work (3), Info
      Integer LDM, Dim
      Character Real

C   Initialize LDM and N to be 3. Also init REAL to be 'F'
      Data LDM /3/, Dim /3/, Real/'F'/

C   Read the Leontief matrix, product surplus array, and
C   form the technology matrix
      Call ReadAR(Mat,Dim,Dim)
      Call ReadAR(Prod,1,Dim)

C   Use LINPACK subroutine SGECO to do LU factorization of Mat
      call SGECO (Mat,LDM,Dim,Ipvt,Info)

C   Check for singularity and exit if singular
      If (Info.NE.0.) Call PmWm(Info)

C   Use LINPACK subroutine SGESL to compute [A]x = b
      Call SGESL (Mat,LDM,Dim,Ipvt,Prod,0)

C   Print the results
      Call PmRes(Prod,Dim)

      Stop
      End
```

-6-

### [3.0.0] Support Routines

The following routines are used to support the main driver. This chapter is divided into sections that are used to manipulate data, read data, or write results out to the user.

The support routines consist of:

- TecMat - routine to form the technology matrix
- ReadAR - routine to read an array
- PrnWrn - routine to print a warning message and exit  
if necessary
- PrnRes - routine to print the results

All other support routines are called from the LINPACK Scientific Subroutine Library.

-7-

### [3.1.0] Matrix manipulation routines

The following routine manipulates the Leontief matrix into a form that can be used to solve the system of equations.

#### [3.1.1] TecMat: Form a Technology Matrix from a Leontief Matrix

The Leontief matrix is subtracted from the Identity matrix, which results in the Technology matrix.

An Identity matrix is a matrix in which all elements of the matrix are zero (0) except the elements on the diagonal, which have the value one (1).

It would be inefficient to generate an identity matrix and then call a subroutine to do matrix subtraction. Instead, we can simulate the subtraction of a matrix from its identity matrix by realizing that the characteristics of an identity matrix can be simulated using two do loops. When the looping variables used for each loop are equal, the value of a corresponding element in an identity matrix indexed by those variables would be a one (1). When the looping variables are not equal, the values of a corresponding element in an identity matrix indexed by these variables would be zero (0). This suggests that, given the dimensions of any square matrix, the following algorithm would solve the problem of subtracting any it from its identity matrix.

General algorithm:

```

For Row index in [1 ... NDim] do
  For Column index in [1 ... NDim] do
    If (Row Index = Column index) (the diagonal elements)
      Matrix element = 1 - Matrix Element
    Else
      Matrix element = 0 - Matrix element
    Endif
  EndDo
EndDo

```

Calls: None

Called by: The Main Driver

Arguments:

LeoMat    - The Leontief Matrix to be subtracted from the identity matrix  
RCDim    - The row and column dimension of the Leontief matrix

Local Variables:

RowIdx    - Row index  
ColIdx    - Column index



-8-

### [3.2.0] Input routines

The following routines are used to read information from the user. Information is assumed to be entered from the terminal. On systems with input redirection (DOS, UNIX, Minix, OS/2, Xenix, etc.), the information can be stored in a file and redirected to the program as input.

#### [3.2.1] ReadAr: Read a two dimensional array of unknown size

This routine reads a two dimensional array with unknown Row and Column size. Reading is done using an implied do loop, which is based on the column size of the array. Unformatted input is used to give the user flexibility of input format. The only requirement is that data values for a row of data be consecutive and be separated by at least one space.

The information to be read is assumed to be REAL data.

##### General algorithm:

```
For Row index in [1 ... Row dimension] Do
  Read a row of the matrix
```

Calls: None

Called by: The Main Driver

##### Arguments:

InArray - Array variable to read information into

Rows - Number of rows in the array

Cols - Number of columns in the array

##### Local Variables:

RowIdx - Row index

ColIdx - Column index

```
Subroutine ReadAR(Array,Rows,Cols)
```

```
Integer Rows, Cols, RowIdx, ColIdx
```

```
Real Array(Rows,Cols)
```

```
C
```

```
C Given the numbers of rows and columns in any two dimensional
```

```
C array, read the array into the matrix row by row. Assume the
```

```
C input file is in no specific format.
```

```
C
```

```
Do 10 RowIdx = 1,Rows
```

```
10 Read (*,*) (Array(RowIdx,ColIdx), ColIdx = 1,Cols)
```

```
Return
```

```
End
```

-9-

### [3.3.0] Output Routines

The following routines are used to print warning messages or to print the results of the calculations performed by the program.

#### [3.3.1] PrnRes: Print the results of the calculations

PrnRes prints the resulting Product array when the solution has been found.

General algorithm:

For each element in the array  
Write the element number and its value

Calls: None

Called by: The Main Driver

Arguments:

Prod - The product array

Dim - The dimension of the product array

Local variables:

Index - The index into the array

Subroutine PrnRes(Prod,Dim)

Integer Dim, Index

Real Prod(Dim)

C

C Given the result array from solving the system of equations that  
C make up the Leontief model and its dimension, print the results  
C out for the user.

C

Do 10 Index = 1,Dim

10 Print 20, Index, Prod(I)

20 FORMAT (1x,'X(' ,J2,' )' ,3x,'=' ,3x,f10.4)

Return

End

-10-

**[3.3.2] PrnWrn: Warn User and Exit If Matrix is Singular**

This routine warns the user that the array might be singular, checks the condition number passed to the routine, and if it is smaller than machine accuracy (i.e., the condition number + 1 is indistinguishable from the condition number) the program is aborted.

General algorithm:

```

Print the warning message
If Check for singularity is true
    inform user of singularity
    exit program
Endif

```

Calls: None

Called by: The Main Driver

Arguments:

Info - Condition number estimate of the array (from SGECCO)

```

Subroutine PrnWrn(Info)
Real Info
C
C Print a warning message to the user indicating the system of
C equations may not be solvable. Then test to see if the
C decomposition routine returned a condition number that
C indicates the matrix may be singular to the working precision
C of the machine. If it is, tell the user and abort the program.
C
C Print warning message
C
C Print *, 'Matrix may be singular to working precision'
C Print *, 'or there is a possibility of a divide by zero'
C Print *, 'during the calculation of the result.'
C Print *, 'Checking for singularity ...'
C
C Check condition number estimate and exit if matrix is singular
C
C if (Info.EQ.Info+1) Then
C   Print *, 'Matrix is singular to working precision: aborting.'
C   Print *
C   Print *, 'Execution completed, no results generated.'
C   Stop
C Endif
C
C Return
C End

```

-11-

## Table of Contents

[1.0.0] Three Industry Leontief Model . . . . .	2
[2.0.0] The main driver . . . . .	4
[3.0.0] Support Routines . . . . .	6
[3.1.0] Matrix manipulation routines . . . . .	7
[3.1.1] TecMat: Form a Technology Matrix from a Leontief Matrix . . . . .	7
[3.2.0] Input routines . . . . .	8
[3.2.1] ReadAr: Read a two dimensional array of unknown size . . . . .	8
[3.3.0] Output Routines . . . . .	9
[3.3.1] PmRes: Print the results of the calculations . . . . .	9
[3.3.2] PmWm: Warn User and Exit If Matrix is Singular . . . . .	10

## APPENDIX E

### EXAMPLE TRADITIONAL PROGRAM (FROM EXPERIMENT 1)

## Leontief Modelling

Page: 1

```
      Program Leontief
C
C      This program tests several subroutines that were written to
C      solve variable sized Leontief Input/Output economy models.
C
C      Define the variables:
C
      Real Mat(3,3), Prod(3), IPVT(3), Work (3), Info
      Integer LDM, Dim
      Character Real
C
C      Initialize LDM and N to be 3. Also init REAL to be 'F'
      Data LDM /3/, Dim /3/, Real/'F'/
C
C      Read the Leontief matrix, product surplus array, and
C      form the technology matrix
      Call ReadAR(Mat,Dim,Dim)
      Call ReadAR(Prod,1,Dim)
C
C      Use LINPACK subroutine SGECON to do LU factorization of Mat
      Call SGECON (Mat,LDM,Dim,Ipvt,Info)
C
C      Check for singularity and exit if singular
      If (Info.NE.0.) Call PrnWrn(Info)
C
C      Use LINPACK subroutine SGESL to compute  $[A]x = D$ 
      Call SGESL (Mat,LDM,Dim,Ipvt,Prod,0)
C
C      Print the results
      Call PrnRes(Prod,Dim)
C
      Stop
      End
```

```
Subroutine ReadAR(Array,Rows,Cols)
Integer Rows, Cols, RowIdx, ColIdx
Real Array(Rows,Cols)
C
C   Given the numbers of rows and columns in any two dimensional
C   array, read the array into the matrix row by row. Assume the
C   input file is in no specific format.
C
Do 10 RowIdx = 1,Rows
10   Read (*,*) (Array(RowIdx,ColIdx), ColIdx = 1,Cols)

Return
End
```

```
Subroutine PrnRes(Prod,Dim)
Integer Dim, Index
Real Prod(Dim)
C
C   Given the result array from solving the system of equations that
C   make up the Leontief model and its dimension, print the results
C   out for the user.
C
Do 10 Index = 1,Dim
10   Print 20, Index, Prod(I)
20   FORMAT (1x,'X(',I2,' )',3x,'=',3x,f10.4)

Return
End
```



```
Subroutine PrnWrn(Info)
Real Info
C
C Print a warning message to the user indicating the system of
C equations may not be solvable. Then test to see if the
C decomposition routine returned a condition number that
C indicates the matrix may be singular to the working precision
C of the machine. If it is, tell the user and abort the program.
C
C
C Print warning message
C
C Print *, 'Matrix may be singular to working precision'
C Print *, 'or there is a possibility of a divide by zero'
C Print *, 'during the calculation of the result.'
C Print *, 'Checking for singularity ...'
C
C Check condition number estimate and exit if matrix is singular
C
C If (Info.EQ.Info+1) Then
C   Print *, 'Matrix is singular to working precision: aborting.'
C   Print *
C   Print *, 'Execution completed, no results generated.'
C   Stop
C Endif
C
C Return
C End
```

## **APPENDIX F**

### **EXPERIMENT 1 SPECIFICATIONS**

## Program Specifications

You are to write a program which solves a multiple industry Leontief Input/Output model. The program will be tested with a three industry model, and should have a main program that tests this capability.

Write the subroutines such that they will work for any size model. Write the main program in such a way that changing the model size requires changing values in a minimum of places.

The FULL Leontief model originally had the economy divided into over 500 sectors, but has since been reduced to a more manageable 42 departments of production.

The program subroutines should handle variable sized models up to 42 x 42.

If the system of equations is a poor model, the possibility exists that the system will be singular, and thus not solvable. Test for this possibility and abort the program if the system of equations is singular to machine precision.

The system of equations can be solved using the formula:

$$x - Ax = D \quad \text{or} \quad Tx = D.$$

Where  $x$  is the gross production array,  $A$  is the Leontief Matrix, and  $D$  is the desired surplus production, and  $T$  is the technology matrix of  $A$  (see below how to form the Technology matrix of  $A$ ).

The most straight forward method is to use the second equation above using the technology matrix and then use gaussian elimination to solve the system of equations.

Use the following equations to form the technology matrix from the Leontief matrix that is read in.

$$T = I - A$$

where  $I$  is the identity matrix of  $A$ .

Use the LINPACK subroutine library to solve the system of equations that make up the model. The Routines SGECO and SGESL should be used to factor and solve the system of equations (respectively).

## Required processing

1. Read the data values for the Leontief matrix.
2. Read the data values for the gross production array.
3. If possible, solve the resulting system of linear equations (as described in the program specification) and print the results. Otherwise, print a meaningful error message and exit.

## Input and Output Requirements

1. The program will read all data from the keyboard.
2. The program will write all data to the terminal.

Input consists of:

1. Data for a square matrix (the Leontief input/output model values).
2. A one row matrix with as many columns as the model has rows (the gross production array).

Make no assumptions about the format of the input data other than the assumption that the user will always supply the data values separated by at least one space, the values will be consecutive columns of a single row, and there will always be enough data to fill both arrays.

A graphic description of what the input might look like follows.

The Leontief matrix could look like:

```
value value value
value value value
value value value
```

or

```
value value value value value value value value value
```

The Gross Production matrix could look like:

```
value
value
value
```

or

```
value value value
```

For a system with 3 inputs and three outputs. ALL DATA IS REAL.

Output consists of either:

1. The solution to the model.

OR

2. An error message indicating that the technology matrix is singular to the working precision of the machine.

## **APPENDIX G**

### **EXPERIMENT 3 SPECIFICATIONS**

## INSTRUCTIONS

You are to write a program that acts like a limited desk calculator.

Use the C programming language. K&R style is expected. DO NOT USE ANSI-C.

The calculator will allow for 26 variables to be assigned values. The variable names are:

a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y, and z

For simplicity, you can assume the user always enters the variable name in lower case. Variables are used in expressions, and are the only thing allowed on the left hand side of the assignment operator.

The operations the calculator will perform are

OPERATOR	ACTION	EXAMPLE
=	assignment	$x = 3, y = (3 + 5^y - 4^{(x / 2)})$
+	add	$2 + 3, 3 + x$
-	subtract	$3 - 2, a - 3$
*	multiply	$3 * 5.3, x * -y$
^	exponentiate	$3^3, 4^{(x*y)}$
()	subexpression	$x = (y * (5 + x) / (z^{(r/a)}))$

Additionally, operands may be signed (e.g., -5, -X, -(x\*y))

The precedence of the operators is as follows:

assignment operator  
sign operator  
subexpression  
exponentiate operator  
multiply and divide operators  
add and subtract operators

Precedence classes with two operators (such as add and subtract) are evaluated from left to right (i.e., they have equal precedence, and the left to right rule is used as a secondary precedence rule in these cases).

Use a recursive descent parser/evaluator to implement the program. There should be one procedure for each of the operators in the precedence table above.

## INPUT:

The user will enter a mathematical expression to be evaluated. Numbers entered can be either integer or real. All integer numbers are immediately converted to their real equivalent for use in the calculations.

The user can enter as many expressions as they like, one per line. Each expression must be terminated with a \$ by the program after the user enters it. Use the dollar sign as the base case on which to de-recurse and form the solution.

When EOF is reached, terminate the program.

## OUTPUT:

The output to the user will always be the floating point approximation of the answer to the expression entered, or an error message indicating what was wrong with the expression.

When finished evaluating an expression, print the results of the calculation and then print out all variables whose values are not equal to 0.0 (to remind the users which variables have been set to a value other than 0.0).

If a calculation was performed, it may in fact be correct. Always print the results of the calculation, and when an error has occurred, print the message:

The Results MAY BE INCORRECT

on the same line as the line that printed the answer. The answer to a calculation should be printed as:

The answer is <answer (not including the angle brackets)>

Ex:

The answer is 125.76894

## ERROR HANDLING:

The program should be able to detect at least:

Unablanced parenthesis	[e.g., $x = (y^{(z-5)})$ ]
Syntax error	[e.g., $x = * 5$ ]

You may also want to check that an expression is present, and if not warn the user.



## **APPENDIX H**

### **EXPERIMENT 2 SPECIFICATIONS**

### Program Specifications

You are to write a program which does the end of term grading for a class. Each student has 7 test or assignment scores, which are weighted unevenly. The program should compute the final grade for the student based on the sum of the weighted test scores.

For each student compute or save the following data:

The numeric total grade (e.g. 96.1, or 78.6, or 85.0, etc.)

The letter grade:

A = 89.5 and above

B = 79.5 - 89.4

C = 69.5 - 79.4

D = 59.5 - 69.4

F = 59.4 and lower

The students highest grade

The students lowest grade

For the entire class compute or save the following data:

The lowest grade in the class

The highest grade in the class

The average grade

The weights of the test or assignment scores are as follows:

Assignment 1 weight = .05

Assignment 2 weight = .05

Assignment 3 weight = .10

Midterm weight = .30

Assignment 4 weight = .10

Assignment 5 weight = .10

Final weight = .30

### Input and Output Specifications

#### Input:

Input consists of one line per student formatted as follows:

Student name (First, Last)      FORMAT = A40  
 Grades 1 - 7                      FORMAT = 7(F5.1,1X) (decimal in data)

The format of the assignments and tests is as follows. The first three numbers on the input line are student assignments 1 through 3 (respectively). The fourth number is the midterm, followed by assignments 4 and 5, and finally the last number is the final examination score.

Graphically, an input line looks like:

Students name                      Asgn1 Asgn2 Asgn3 Mdtm Asgn4 Asgn5 Final

#### Output:

Headings which describe the entries in the columns below the heading

For each student (all information on one line):

Student name (First, Last)      FORMAT = A40,3X  
 Total numeric grade              FORMAT = F5.1,5X  
 Letter grade                      FORMAT = 'Grade: ',1A  
 Highest grade                    FORMAT = 'Highest grade: ',F5.1,5X  
 Lowest grade                    FORMAT = 'Lowest grade: ',F5.1

Summary report (one per line after all student information):

Lowest grade in the class      FORMAT = '//Lowest grade: ',F5.1  
 Highest grade in the class      FORMAT = 'Highest grade: ',F5.1  
 The class average                FORMAT = 'Class average: ',F5.1

## **APPENDIX I**

### **EXAMPLE PROGRAM SOLUTION**

### Program Solution

The solution to the problem required writing the following routine:

```

Subroutine TecMat(LeoMat,RCDim)
Integer RCDim, RowIdx, ColIdx
Real LeoMat(RCDim,RCDim)
C
C   Given the dimensions of a square two dimensional Leontief
C   matrix form a technology matrix by subtracting the Leontief
C   Matrix from its identity matrix.
C
C   Form the Technology matrix (I - A)
C
  Do 20 RowIdx = 1,RCDim
    Do 10 ColIdx = 1,RCDim
      If (RowIdx.EQ.ColIdx) Then
        LeoMat(RowIdx,ColIdx) = 1 - LeoMat(RowIdx,ColIdx)
      Else
        LeoMat(RowIdx,ColIdx) = 0 - LeoMat(RowIdx,ColIdx)
      Endif
    10 Continue
  20 Continue

  Return
End

```

The call to the routine should have been placed in the main driver of the program, directly following the two calls to the routine that read the input arrays (ReadAr). It should have been coded as:

```
Call TecMat(Mat,Dim)
```

## APPENDIX J

### EXAMPLE LITERATE PROGRAM DOCUMENT (FROM EXPERIMENT 1)

@A Chris F. Bertholf  
 @B Portland State University  
 @P LeontiefModeling  
 @D Leontief Input/Output Analysis of Multiple Industry Model  
 @R 1.0

@T  
 @I Introduction

.PP  
 One interesting application of matrices is the Leontief Input-Output model, named for Wassily Leontief. The model Leontief developed is useful for predicting the effects to the economy of price changes or shifts in government spending.

.PP  
 Leontief's work divided the economy into 500 sectors, which was later reduced to a more manageable 42 departments of production. We can examine the working of the model with a very simplified view of the economy.

.PP  
 This program attempts to show a working three industry Leontief Input/Output model based on the mining, manufacturing, and energy industry. The model uses several subroutines from the LINPACK Scientific Subroutine Library for solving linear systems of equations.

@{ Three Industry Leontief Model }  
 Suppose we consider a simple economy as being based on three commodities: the mining industry, the manufacturing industry, and the energy industry. Suppose further that production of one dollars worth of mining requires \$0.40 units from mining, \$0.40 units from manufacturing, \$0.20 units from energy; Production of one dollars worth of manufacturing requires \$0.20 units from mining, \$0.40 units from manufacturing, and \$0.20 units from energy. Production of one dollars worth of energy requires \$0.10 units from mining, \$0.20 units from manufacturing, and \$0.40 units from energy. The following table summarizes this information:  
 .nf

Inputs:	Outputs		
	mining	energy	manufacturing
mining:	\$0.40	\$0.40	\$0.20
manufacturing:	\$0.10	\$0.20	\$0.40
energy:	\$0.20	\$0.40	\$0.20

.PP  
 Note that the sums of the columns need not add up to 1.00. This is because not all commodities or industries are represented in this model. In particular it is customary to omit labor from these models.

.PP  
 From the preceding table we can form a matrix  $\backslash fBA \backslash fR$  called the technology matrix, (or the Leontief matrix):

.EQ  
 delim \$\$  
 .EN  
 .ce  

$$A = \begin{bmatrix} 0.4 & 0.1 & 0.2 \\ 0.4 & 0.2 & 0.4 \\ 0.2 & 0.4 & 0.2 \end{bmatrix}$$

.PP  
 For this simplified model of the economy, not all information is contained in the Leontief matrix. In particular each industry has a gross production, the gross production can be represented as a column matrix  $\backslash fBX \backslash fR$ :  
 .nf

.ce  

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

.PP  
 Where  $x_1$  is the gross production from mining,  $x_2$  is the gross production from manufacturing, and  $x_3$  is the gross production from

energy. Those units of gross production not used by these industries are called surpluses, and may be considered as being available for consumers. If we place the surpluses in a column matrix  $\text{\fBD\fr}$ , then the surplus can be represented by the equation

```
.ce
\fbx\fr - \fBAx\fr = \fBD\fr
```

```
.nf
which is equivalent to:
```

```
.ce
(\fBI\fr - \fBA\fr)\fbx\fr = \fBD\fr
```

```
.fi
where I is an identity matrix. This matrix equation is called the technology equation.
```

```
.PP
Note: An Identity matrix is a matrix in which every element is zero (0) except the elements on the diagonal, which have the value one (1).
```

```
.EQ
delim off
.EN
```

```
.PP
If we call the matrix formed by  $\text{\fB (I - A) \fr}$  the Technology Matrix, and we represent this quantity with  $\text{\fB T \fr}$ . we can rewrite the equation as:
```

```
.ce
\fbTx\fr = \fBD\fr
```

```
.PP
To find a solution to the system of equations there are several methods. The most straight-forward method is to do Gaussian elimination to solve the equation:
```

```
.ce
\fbTx\fr = \fBD\fr
```

```
.PP
Not only is this the most straight-forward solution, but compared to the other obvious solution (compute inverse of Technology matrix and multiply by  $\text{\fBD\fr}$ ) it is far less expensive in terms of computational time.
```

```
.PP
Because the gaussian elimination problem has been solved by many programmers, we will use a library routine to do the factoring (decomposition) of the technology matrix ( $\text{\fBT\fr}$ ), and another routine to solve the equation:
```

```
.ce
\fbLx\fr = \fBD\fr
```

```
.PP
Where  $\text{\fBL\fr}$  is the lower triangular matrix and  $\text{\fBU\fr}$  is the upper triangular matrix found during decomposition of the Technology Matrix ( $\text{\fBT\fr}$ ). Because the matrix may be singular, or very close to singular (to the working precision of the machine) we make sure that it is not before we solve the equation. This is done by checking the return value of the call to the routine that will do the decomposition on the technology matrix. If the value returned causes some wonder as to whether or not the matrix may be singular to the working precision, or if the return value indicates that there may be a divide by a zero pivot, we will ask the user if they would like us to check for singularity by estimating the condition number of the technology matrix. If the condition number is ok then we will go ahead and solve the above equation, if not we exit the program.
```

```
.PP
The subroutine we need are part of the LINPACK Subroutine Library for General Matrices. The routines we will be using are  $\text{\fBSGECO\fr}$  (estimate the condition number of the matrix while decomposing it) and  $\text{\fBSGESL\fr}$  to solve a system of linear equations decomposed into an  $\text{\fBLx\fr = \fBD\fr}$  format.
```

```
.PP
```



For a description of the subroutines themselves, the user is referred to chapter one the Linpack User manual: General Matrices.

@( The main driver )

The main driver simply defines the variables required to generate and solve the model; The Leontief matrix is defined and initialized, the solution matrix (which contains the desired surplus production values) is defined and initialized, and the technology matrix is then formed from the Leontief matrix.

.PP

Once the technology matrix has been formed (by calling the TecMat routine) the Linpack subroutine SGECCO is called to do the LU factorization of the technology matrix. If SGECCO returns a non-zero value in the info variable, there is a possibility that the matrix is singular to the working precision of the machine, or that there is a possibility of a divide by zero (0) if SGESL is used to solve the system of equations. If the Info variable is not smaller then the working precision of the computer, the Linpack routine SGESL is called to solve the system of equations, and the results are printed on the terminal screen.

.PP

When Info is returned as non-zero, the user is asked if they wish to test for singularity. If the Matrix is singular to the working precision of the machine, the user is told and the program aborts. If the test for singularity fails (i.e., the matrix is not singular) then the program continues and the SGESL routine is called to solve the system of equations.

.nf

\fBGeneral algorithm:\fR

```

Initialize Leontief Matrix
Initialize Production matrix
Transform Leontief matrix into Technology matrix
Call SGECCO to factor the Matrix
If Technology matrix may be singular
    Warn the user
    Test for singularity
    If the Technology matrix is singular
        Tell the user
        Abort the program
    Endif
Endif
Call SGESL to solve the system of equations
Print the resulting solution

```

\fBCalls:\fR

```

TecMat - routine to form the technology matrix
ReadAR - routine to read an array
PrnWrn - routine that prints the singular matrix warning message
PrnSol - routine to print the solution

```

\fBLibrary routines used\fR

FROM THE LINPACK LIBRARY

```

SGECCO - Factor a matrix and estimate its condition number
SGESL - Solve a system of linear equations

```

\fBCalled by:\fR

Operating system

.bp

\fBVariables:\fR

```

Mat - The technology matrix
Prod - The solution matrix
Info - Holds estimate of singularity
IPvt - LINPACK uses this to store pivot information
Work - Work array for LINPACK
LDM - The leading dimension of Mat
Dim - The Dimension of Work, Prod, and IPVT

```

@(

Program Leontief

```

C
C      This program tests several subroutines that were written to
C      solve variable sized Leontief Input/Output economy models.
C
C      Define the variables:
C
C      Real Mat(3,3), Prod(3), IPVT(3), Work (3), Info
C      Integer LDM, Dim
C      Character Real
C
C      Initialize LDM and N to be 3. Also init REAL to be 'F'
C      Data LDM /3/, Dim /3/, Real/'F'/
C
C      Read the Leontief matrix, product surplus array, and
C      form the technology matrix
C      Call ReadAR(Mat,Dim,Dim)
C      Call ReadAR(Prod,1,Dim)
C
C      Use LINPACK subroutine SGECCO to do LU factorization of Mat
C      call SGECCO (Mat,LDM,Dim,Ipvt,Info)
C
C      Check for singularity and exit if singular
C      If (Info.NE.0.) Call PrnWrn(Info)
C
C      Use LINPACK subroutine SGESL to compute [A]x = b
C      Call SGESL (Mat,LDM,Dim,Ipvt,Prod,0)
C
C      Print the results
C      Call PrnRes(Prod,Dim)
C
C      Stop
C      End

```

a)

#### @( Support Routines )

The following routines are used to support the main driver. This chapter is divided into sections that are used to manipulate data, read data, or write results out to the user.

.nf

The support routines consist of:

TecMat	-	routine to form the technology matrix
ReadAR	-	routine to read an array
PrnWrn	-	routine to print a warning message and exit if necessary
PrnRes	-	routine to print the results

All other support routines are called from the LINPACK Scientific Subroutine Library.

#### @[ Matrix manipulation routines ]

The following routine manipulates the Leontief matrix into a form that can be used to solve the system of equations.

#### @[[ TecMat: Form a Technology Matrix from a Leontief Matrix ]]

The Leontief matrix is subtracted from the Identity matrix, which results in the Technology matrix.

.PP

An Identity matrix is a matrix in which all elements of the matrix are zero (0) except the elements on the diagonal, which have the value one (1).

.PP

It would be inefficient to generate an identity matrix and then call a subroutine to do matrix subtraction. Instead, we can simulate the subtraction of a matrix from its identity matrix by realizing that the characteristics of an identity matrix can be simulated using two do loops. When the looping variables used for each loop are equal, the value of a corresponding element in an identity matrix indexed by those variables would be a one (1). When the looping variables are not equal, the values of a corresponding element in an identity matrix indexed by these

variables would be zero (0). This suggests that, given the dimensions of any square matrix, the following algorithm would solve the problem of subtracting any it from its identity matrix.

```
.nf
\fbGeneral algorithm:\fR

    For Row index in [1 ... NDim] do
        For Column index in [1 ... NDim] do
            If (Row Index = Column index) (the diagonal elements)
                Matrix element = 1 - Matrix Element
            Else
                Matrix element = 0 - Matrix element
            Endif
        EndDo
    EndDo

\fbCalls:\fR None
\fbCalled by:\fR The Main Driver
```

```
\fbArguments:\fR
LeoMat - The Leontief Matrix to be subtracted from the identity matrix
RCDim - The row and column dimension of the Leontief matrix
```

```
\fbLocal Variables:\fR
RowIdx - Row index
ColIdx - Column index
```

```
a(
    Subroutine TecMat(LeoMat,RCDim)
    Integer RCDim, RowIdx, ColIdx
    Real LeoMat(RCDim,RCDim)

C
C    Given the dimensions of a square two dimensional Leontief matrix
C    form a technology matrix by subtracting the Leontief Matrix from
C    its identity matrix.
C
C    Form the Technology matrix (I - A)
C
    Do 20 RowIdx = 1,RCDim
        Do 10 ColIdx = 1,RCDim
            If (RowIdx.EQ.ColIdx) Then
                LeoMat(RowIdx,ColIdx) = 1 - LeoMat(RowIdx,ColIdx)
            Else
                LeoMat(RowIdx,ColIdx) = 0 - LeoMat(RowIdx,ColIdx)
            Endif
        10    Continue
    20    Continue

    Return
End

a)
```

#### a[ Input routines ]

The following routines are used to read information from the user. Information is assumed to be entered from the terminal. On systems with input redirection (DOS, UNIX, Minix, OS/2, Xenix, etc.), the information can be stored in a file and redirected to the program as input.

#### a[[ ReadAr: Read a two dimensional array of unknown size ]]

This routine reads a two dimensional array with unknown Row and Column size. Reading is done using an implied do loop, which is based on the column size of the array. Unformatted input is used to give the user flexibility of input format. The only requirement is that data values for a row of data be consecutive and be separated by at least one space.

```
.PP
The information to be read is assumed to be REAL data.
```

```
.nf
\fbGeneral algorithm:\fR
```

```
    For Row index in [1 ... Row dimension] Do
```

Read a row of the matrix

\fBCalls:\fR None  
 \fBCalled by:\fR The Main Driver

\fBArguments:\fR  
 InArray - Array variable to read information into  
 Rows - Number of rows in the array  
 Cols - Number of columns in the array

\fBLocal Variables:\fR  
 RowIdx - Row index  
 ColIdx - Column index

```
a(
    Subroutine ReadAR(Array,Rows,Cols)
    Integer Rows, Cols, RowIdx, ColIdx
    Real Array(Rows,Cols)

C
C    Given the numbers of rows and columns in any two dimensional
C    array, read the array into the matrix row by row. Assume the
C    input file is in no specific format.
C
    Do 10 RowIdx = 1,Rows
    10    Read (*,*) (Array(RowIdx,ColIdx), ColIdx = 1,Cols)

    Return
End
```

a)

a[ Output Routines ]

The following routines are used to print warning messages or to print the results of the calculations performed by the program.

a[[ PrnRes: Print the results of the calculations ]]  
 PrnRes prints the resulting Product array when the solution has been found.  
 .nf

\fBGeneral algorithm:\fR

For each element in the array  
 Write the element number and its value

\fBCalls:\fR None  
 \fBCalled by:\fR The Main Driver

\fBArguments:\fR  
 Prod - The product array  
 Dim - The dimension of the product array

\fBLocal variables:\fR  
 Index - The index into the array

```
a(
    Subroutine PrnRes(Prod,Dim)
    Integer Dim, Index
    Real Prod(Dim)

C
C    Given the result array from solving the system of equations that
C    make up the Leontief model and its dimension, print the results
C    out for the user.
C
    Do 10 Index = 1,Dim
    10    Print 20, Index, Prod(I)
    20    FORMAT (1x,'X(',12,' )',3x,'=',3x,f10.4)

    Return
End
```

a)

.bp

a[[ PrnWrn: Warn User and Exit If Matrix is Singular ]]

This routine warns the user that the array might be singular, checks the condition number passed to the routine, and if it is smaller than machine

accuracy (i.e., the condition number + 1 is indistinguishable from the condition number) the program is aborted.  
 .nf

General algorithm:\fR

```

    Print the warning message
    If Check for singularity is true
        inform user of singularity
        exit program
    Endif
  
```

\fBCalls:\fR None

\fBCalled by:\fR The Main Driver

\fBArguments:\fR

Info - Condition number estimate of the array (from SGECO)

```

a(
    Subroutine PrnWrn(Info)
    Real Info

C
C    Print a warning message to the user indicating the system of
C    equations may not be solvable. Then test to see if the
C    decomposition routine returned a condition number that
C    indicates the matrix may be singular to the working precision
C    of the machine. If it is, tell the user and abort the program.
C
C    Print warning message
C
    Print *, 'Matrix may be singular to working precision'
    Print *, 'or there is a possibility of a divide by zero'
    Print *, 'during the calculation of the result.'
    Print *, 'Checking for singularity ...'

C
C    Check condition number estimate and exit if matrix is singular
C
    if (Info.EQ.Info+1) Then
        Print *, 'Matrix is singular to working precision: aborting.'
        Print *
        Print *, 'Execution completed, no results generated.'
        Stop
    Endif

    Return
End
a)
  
```

## APPENDIX K

### WHAT THE LIT SYSTEM DOES FOR THE USER

The number of commands and the required parameters for each command that are needed to effectively edit, format, view, print, debug, provide revision control, and run a literate program is large. The idea behind the Lit system is to enable the programmer in the programming and maintenance task. Adding several more complex layers to the programming paradigm would probably defeat this purpose; all of the commands and parameters would just add to the cognitive load of the programmer. Although the same commands are used over and over, with the same options (usually), there is no need for the programmer to be burdened with this extra level of detail. For example, the commands required to write, debug, format, view, run, and print a small literate program might be:

```

co -l project-name.lit
vi project-name.lit
lit -lC -ftroff project-name
mv project-name.src project-name.c
gcc -g -c -o project-name.exec project-name.c >& \
    project-name.compile-errors
vi project-name.compile-errors
vi project-name.lit
lit -lC -ftroff project-name
mv project-name.src project-name.c
gcc -g -c -o project-name.exec project-name.c >& \
    project-name.compile-errors
project-name.exec and some associated parameters
dbx project-name.exec
vi project-name.lit
lit -lC -ftroff project-name
mv project-name.src project-name.c
gcc -g -c -o project-name.exec project-name.c >& \
    project-name.compile-errors
project-name.exec and some associated parameters
groff -me -mlit -Tascii -geqn -gtbl -gpic project-name.doc \
    > project-name.nr
less -ewqd project-name.nr
groff -me -mlit -Tps -geqn -gtbl -gpic project-name.doc \
    | lpr -Ppostscript1
ci project-name.lit ; rm core project-name.bkp

```

Obviously, this is a lot of information to remember, the commands have the possibility of being mistyped, and the commands are quite repetitious. The Lit system will prompt the user when it is invoked, or when a project change is requested, for the relevant information about which compiler to use, etc. The relevant information can also be stored in the user's environment, in which case Lit will only prompt for information that the user has not explicitly defined. Armed with the knowledge of which editor, compiler, libraries, formatter, viewer, and debugger to use, the Lit system significantly reduces the amount of this information which must be remembered by the programmer, and allows the programmer to perform operations in a more natural manner independent of the details of which underlying applications are needed to perform the indicated actions. For example, the sequence of instructions described above would have the following equivalent instructions in the Lit system.

```
Lit project-name
Edit
Compile
errors
Edit
Compile
Run
Debug
Edit
Compile
Run
Format
View
Print
Exit
```



Note that most of the commands (such as Compile) could have been entered by the user as a simple number (1=Edit, 2=Compile, etc.). Also note the use of the command "errors"; it is a predefined alias that allows the user to edit the error file, when one exists. Figures 6 - 15 below outline the operations that are performed by Lit from system invocation, with each menu selection, and when the system is exited.

When the user invokes Lit (e.g., Lit project-name) Lit performs the following actions (see Figure 6):

1. Check out the project from the revision control system.
2. Select a programming environment (e.g., C and associated libraries).
3. Invoke the programming interface at top level menu.

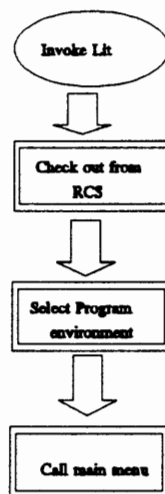


Figure 6. Invoking Lit.

When the user selects the **Edit** Option from the main menu, the following actions are performed (see Figure 7):

1. Create a backup copy of the project file.
2. Edit the project file.
3. When finished editing the project file, return to main menu.

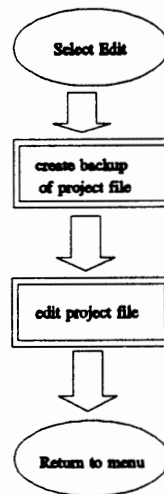
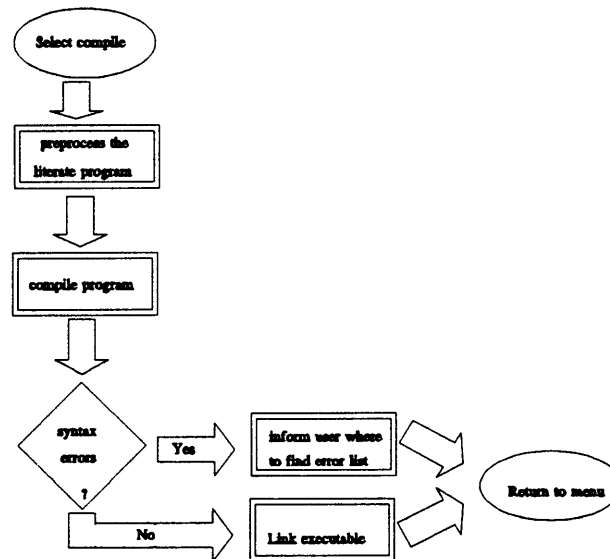


Figure 7. The *Edit* option.

If the **Compile** option is selected, the following actions are performed by

Lit (see Figure 8):

1. Preprocess the literate program file (see Figure 4).
2. Compile the program file.
3. If there were compile-time errors, inform the user about the name of the error message file.
4. If there were no compile-time errors, link the executable file.
5. Return to the main menu.



**Figure 8.** The *Compile* option.

If the user selects **Format** from the menu, Lit performs the following actions (see Figure 9):

1. Preprocess the literate program file (see Figure 4).
2. Format the document for printing or for viewing with code browser.
3. Return to the main menu.

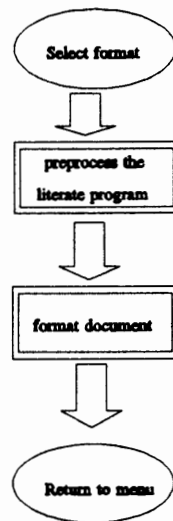


Figure 9. The *Format* option.

If the user selects the **View** option from the main menu, Lit performs the following actions (see Figure 10):

1. If the document is not formatted, format it for viewing.
2. View the document with the code browser.
3. Return to the main menu.

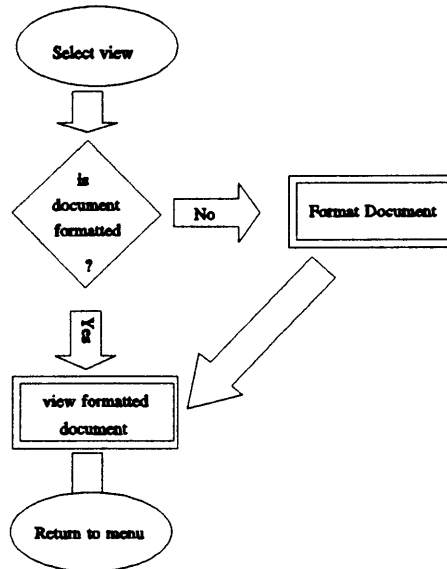


Figure 10. The *View* option.

If the user selects the **Print** option from the main menu, Lit performs the following actions (see Figure 11):

1. If the document is not formatted, format it for printing.
2. Send document to appropriate print spooler.
3. Return to the main menu.

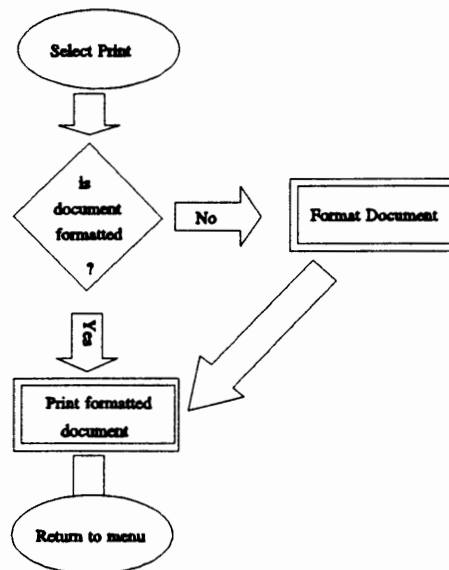


Figure 11. The *Print* option.

If the user selects **Debug** from the main menu, Lit invokes the debugger. When the user has finished, the main menu is redisplayed (see Figure 12). If the user selects the **Run** option, Lit allows the user to enter the required command line parameters, and then executes the linked object file (see Figure 13).

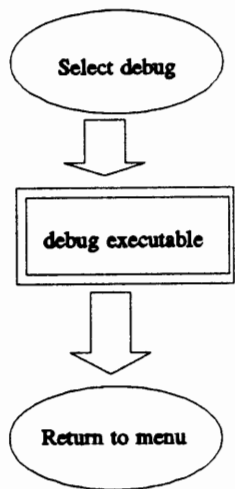


Figure 12. The *Debug* option.

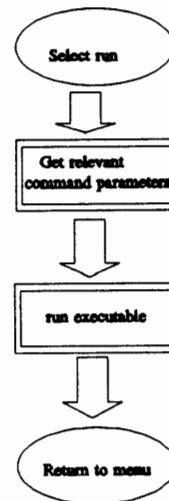


Figure 13. The *Run* option.

If the user wishes to work on a project different from the current project, the **Goto** option is selected. The **Goto** option (see Figure 14) performs the following actions:

1. Check the current project in to the revision control system.
2. Get the name of the next project to open.
3. Perform the startup routine (see Figure 6).
4. Return to the main menu.



Figure 14. The Goto option.



Finally, when the user selects the **Exit** option, Lit performs the following actions (see Figure 15):

1. Check the project file in to the revision control system.
2. Remove any temporary files and/or core dump files.
3. Return control to the invoking process.

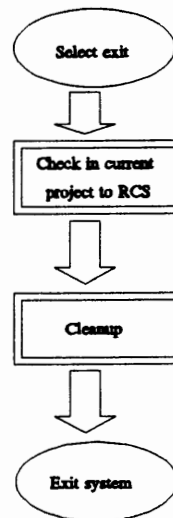


Figure 15. The *Exit* option.

## APPENDIX L

### SPECIFIC FORMATTING RULES USED BY LIT

The following list describes several of the specific document formatting conventions used by the Lit system. All conventions used by Lit have some empirically derived principle associated with them. Lit conventions were derived from the literature on textual comprehension, reading comprehension, and guidelines for documentation from General Electric and other producers of documentation (e.g., IBM). Lit users were asked for input over a period of two years about the format of the documentation, and their observations were used to make modification to it. The results of that process produced the following documentation conventions.

- 1) Line length of 6.5 inches on 8.5" x 11" paper (1 inch margins).
- 2) All text is fully justified between the margins.
- 3) Point size for program name (on title page): 19; always centered.
- 4) Point size for terse description on title page: 9; always centered and bold faced.
- 5) Use a font with well pronounced serifs (Lit uses Times-Roman).
- 6) Point size for entire text body (documentation and code) 8, 9, or 10. Lit defaults to 9.
- 7) Point size for chapter headings: 16; always starts on a new page and is centered and placed at the top of the page margin.
- 8) Point size for section headings: 14; always starts on a new page and is centered and placed at the top of the page margin.
- 9) Point size for subsection headings: 12; always left justified.

- 10) If possible, Lit keeps code sections from being split over a page boundary.
- 11) When (sub)sections are used to separate modules, the module name is placed in the (sub)section title. (e.g., module-name(): title).
- 12) Table of contents lists chapters, sections, and subsections by page and is located at the end of the document.
- 13) Page numbers on every page except title page and introduction; Lit uses page numbers centered 1 inch from the top of the page.
- 14) All chapter, section, and subsection headings include their chapter, section, and subsection number, enclosed in square brackets, in the heading.