Problems to Programs: A Humanistic Approach
(An Introduction to ABL Methodology)

Libero Ficocelli

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August, 1983

ABSTRACT

PROBLEMS TO PROGRAMS: A HUMANISTIC APPROACH
(AN INTRODUCTION TO ABL METHODOLOGY)

Libero Ficocelli

This thesis directs attention to a multitude of difficulties both conceptual and technical involved with the transformation of problems to programs. In so doing it covers such areas as: programming languages, software tools and environments, the psychology of problem solving and programming as well as software production and validation techniques. A methodology and environment is proposed which it is hoped can more fully exploit the positive attributes of current software technologies while trying to minimize and alleviate difficulties inherent to these conventional approaches.

ACKNOWLEDGEMENTS

Foremost I would like to thank my thesis supervisor (and friend) Dr. Wojciech M. Jaworski for introducing me to the topic of ABL and for his continual help, guidance and motivation throughout the course of my research. More importantly I thank him for having taught me the futility of passive acceptance and to appreciate the value of innovation and active investigation (QUESTION EVERYTHING).

I would like to express a special thank you to Kevin O'Mara for his generous help during the research and especially during the actual writing of the thesis. Without his time, intellect, wit, proding and superior vocabulary this thesis would not exist in its present form. He has earned my sincerest gratitude.

I thank Mrs. Dianne Eddy for allowing me to make use of her ABL display programs and to all the other members of the ABL discussion group who have contributed to the growth and evolution of the ABL concept.

I would like to thank my friends Nick Grande (for helping produce the figures) and Agostino Alleva for making my time at Concordia an enjoyable one.

My gratitude to my parents for their confidence and support (moral and otherwise) during my academic carreer.

Last but most definitely not least I wish to thank Helen Galley (my fiancee) for not only typing most of the thesis but also for her patience and understanding during even my most obsessed periods.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Problem solving has existed since the dawn of consciousness and through the ages man has shown himself to be extremely adept at controlling his own environment. During which time he has learnt to manufacture and utilize successively more powerful and sophisticated structures. Concomitant with this process he has also learnt, mostly via the method of trial and error, to synthesize and manipulate abstractions. As his problems became more complex, mathematics, and more particularly computations, evolved to encompass his technology.

The military has played a significant role in the development of computer technology: man needed greater computational resources in order to engineer progressively more accurate and destructive weapons. Although modern day digital computers still fulfill these military needs industrial applications of computers have begun to dominate the rapid dissemination and acceptance of these machines.

The production of software necessary to effect computational processes has evolved at a considerably slower pace then its associated hardware counterpart. In fact the term software crisis sprang from the realization that, though man possessed advanced and complex machine architectures, the software essential to reliably control data transformations did not exist. Over a decade after the computing community directed its attention to resolving this imbalance we must concede that this inequality still prevails. This thesis is an attempt to provide a healthy environment with which to address some of the factors which may have contributed to the present dilemma.

Unlike most techniques ABL does not purport to be a "best" problem solving technique, "best" language, nor even a "best" representation. However it does provide a flexible medium with which a user may mold his own best solutions. This avoids the psychological connotations that are intrinsic to the use of conflicting absolutes. When researchers state that a technique is "best", this automatically implies that other techniques are inferior. This cannot help but precipitate confrontation. The emotional trauma triggered by such unnecessary value judgements initiates defensive attitudes and predisposes the community to an unjustified critical perspective of the methods or techniques in question.

This thesis directs attention to a multitude of difficulties, both conceptual and technical, involved with the transformation of problems to programs. The determinants essential to this study are as follows: (a) psychology of problem solving, (b) psychology of programming, (c) programming languages, (d) software tools, (e) software environments, (f) software production methodologies, (g) software validation and testing, and (h) software reliability and complexity. These aspects of software production form the topics through which we will characterize the state of the art of software technology.

Chapter 2 presents an historical perspective on the evolution of programming and software systems.

Chapter 3 provides an overview of human problem solving paradigms. These paradigms are examined from the context of known psychological behavior and their inherent functional correlates (limitations and advantages). This review is essential to the acquisition of a cognizant understanding of the psychological factors fundamental to an objective analysis of software and associated methodologies.

Chapter 4 introduces a technique (ABL) with which to approach the production, analysis and subsequent testing of software. Although this technique is language independent, it can be embodied within a single control construct. Its implementation however, implies a very unconventional approach. This chapter will also describe a suitable ABL environment and a philosophy for its use.

Chapter 5 is a critical synopsis of conventional programming methodologies. The chapter will outline deficiencies which can be ascribed to the aforementioned psychological considerations. Consequent to this analysis we present ABL in an environment which exploits its virtues.

Chapter 6 discusses software validation and testing, the most costly phase of the software development cycle. The chapter will address general problems inherent to this area of software production in which testing, validation and proofs are essential prerequisites to the more global issue of reliability and verifiability. The ABL methodology simplifies reliability considerations by extending standard facilities and methods to those which are currently infeasible or unavailable.

Chapter 7 attempts to put the application of the ABL approach into the perspective of an era which has become dominated by the computer.

Boehm in 1976 [Boeh76] characterized the typical industrial programmer as follows: "2 years college-level education, 2 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, 'in over his head', and undermanaged". He also points out that "given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who must use it". Under the assumption that interest in software engineering will migrate from academia to industry we must develop viable software methodologies and environments which will permit users to tackle problems in this economically driven context.

The thesis will also illustrate how features fundamental to ABL can be utilized. For example, the fact that the ABL methodology and philosophy are not limited to a class of user. Furthermore ABL's language independence also provides a unifying medium with which to approach the entire spectrum of language bounded issues. This feature enhances the quality of software by providing an integrity of description in which there is no need to partition the source code from its virtual description (documentation). In fact it is possible to insure that these two entities are isomorphically mapped unto each other. The issue of integrity within the ABL environment is further exploited by the use of database technology.

CHAPTER 2

SOFTWARE DEVELOPMENT

A program can be described as the vehicle by which a person can specify the sequence of steps that he wishes a computer to perform [Trac79]. Thus a computer language is in effect a formal definition of a syntax through which we may communicate our wishes to a computer.

2.1 HIGH LEVEL LANGUAGES

Early computing systems (first generation computers) required that humans define their programs in machine language. Such languages consist of numerical representations which map onto the internal state of the computer. As the processes to be defined and performed became lengthier and more demanding, assembler languages were introduced to relieve some of the human tedium that this type of programming required. During these early years of computing the major costs of data processing went into hardware [Enos81]. As long as the machines remained expensive, limited in power and relatively unreliable then the costs of producing programs were buried and forgotten.

With the arrival of second generation computers, hardware became less expensive and more powerful, hence cost considerations quickly shifted the industrial emphasis from hardware issues to production of software. This drastically altered the programmers tasks. Whereas he used to produce short and simple pieces of code, using clever algorithms to overcome restrictions on memory and speed, he was now being asked to create larger programs designed to solve problems of ever increasing complexity [Grie79].

Fortunately rising computing capability also created an improved environment for programmers [Hunt82]. This was mainly due to the introduction of high level procedural languages which relieved the error prone demands of machine/assembler language coding. These procedural based languages soon evolved into programming languages which resemble subsets of the natural languages.

Developers naively assumed that having a tool which resembled our natural means of communication would make programming easier [Wino77,Naur75]. The problem with this approach resides with the fact that the strength of natural languages lies in their diversity and inherent ambiguity [Wass82,Naur75]. As such, any attempt to use a relatively large subset of a natural language as a programming language would by defintion be diametrically opposed to its ultimate objective: being able to communicate unambiguously with the computer.

One paradigm [Ehrm80] of software language development is that the higher level programming languages evolved to meet the global objective of minimizing the human effort required to successfully control a desired computation. In other words the stimulus was programming "ease", which required that researchers simplify the mechanics of translating our wishes into actions for the machine to obey. Under such a paradigm, specific application needs lead to an incredible proliferation of computer languages, whereas machine and/or user dependent requirements created a considerable number of dialects.

Language developers hoping to consolidate and standardize these application dependent languages, often attempted to incorporate as many features as possible into one all encompassing language. Thus, they hoped to avoid a tower of Babel by eliminating the problem of needing many different languages. Unfortunately, these comprehensive languages were lumbering giants and as such were much too awkward and difficult to use. There were so many language dependant details that programmers were prone to both syntactic and semantic errors; PL/1 which is an example of such a language, was once described as a "baroque monstrosity" [Dijk72]. The magnitude of its shortcomings is demonstrated by its lack of popularity, even after 15 years on the market with the entire weight of IBM behind it [Phil77].

Concomitant with this consolidation process was the "forced" evolution of varieties of control constructs, programming constructs and syntax variations in an attempt to create "the programming language" [Litv82]. The result of which is that there are now literally hundreds of different computer languages in existence. If we choose "popularity" as our measure of evolutionary fitness, we must conclude that this two dimensional approach, consolidation and forced evolution, does not appear to have succeeded [Phil77]. The hope of finding "the" language with the ideal number and type of control constructs is no longer considered an issue [Shaw80]. The only consistent finding among researchers was that the most popular languages are usually the ones with the sparsest number of control constructs [Bens73].

2.2 STRUCTURED PROGRAMMING METHODOLOGY

The advent of third generation computers, unfortunately created little significant improvement in the programming environment [Enos81]. Emphasis was centered around the concept that the product (the program) was the single most important item of concern and the sole measure of its "goodness" was at the level of " If it works leave it alone. If it executes quickly, its outstanding" [Enos81].

In "the late 1960's as one after another of the ambitious large scale systems faltered, failed or produced less then satisfactory results" [Enos81], it became apparent that the computer industry was in the midst of a "software crisis" [Dijk79]. There was confusion as to what was meant by a software crisis, but it was generally understood to mean that we did not know how to produce programs well enough. In other words, [Wein71] although systems of programs were being produced they were "not good" because they:

(a) did not meet specifications (did not do what they were supposed to do or were loaded with errors)

(b) did not meet schedules (simply not on time)

(c) were not sufficiently adaptable

(d) were not efficient.

To add insult to injury, researchers [Broo74,McCr73] revealed that industry norms, in terms of productivity was as low as 2,3 lines of code per man day. It became obvious that current software technology was not sufficient to meet software production demands.

The verdict was that the time-proven, systematic approach utilized by other engineering disciplines should be applied to the area of software production [Wass78,Grie79]. Structured programming was introduced as the nucleus of this software engineering approach. The premise involved three essential ingredients [Dijk69,Dijk68,Dona73]: (1) the elimination of the GOTO, (2) the notion of top down design and (3) emphasis on program quality.

The following three sections will examine how these three principals were intended to be a preventative measure for insuring "good" software.

2.2.1 ELIMINATE THE GOTO

The elimination of the GOTO was by far the most controversial of the three. For a long time this "rule" was mistaken to be the sole objective of the proposed methodology. This prompted a rather extended debate between radical factions. On the one side, there were those who insisted that the GOTO should be purged entirely from our programming language vocabularies [Wulf72], while on the other, there were those who insisted that the GOTO was useful and simply too valuable to be disposed [Hopk72]. At one point, the arguments had been so throughly thrashed out, and people so bored of hearing about them , that some authors began writing humourous papers on the topic [Clar73,Zoet79]. The heart of the problem was that no one group was able to make any generalizations, there always seemed to be good counter examples, both for and against use of the GOTO [Knut79,Kern74,Leav72]. Eventually, the consensus view grew to be the compromise position, that it was acceptable to employ "somehow" restricted use of GOTOs.

The arguments against the GOTO can be summarized by the statement: it is detrimental to the production of desirable computer code because it is too easy to use the GOTO in ways which will obscure the logical structure of a program. It was implied that the GOTO was such an unstructured or "primitive" [Dijk65,Dijk68] control flow construct, that a programmer could not help but mess up his program by inadvertantly creating "spaghetti code". The control flow manuverability afforded by the GOTO provided an environment appropriate for succumbing to the temptation of the GOTO shortcut: code as you go - think later! In other words the versatile yet hamfisted power of the GOTO, gave the programmer the opportunity to create pieces of code, in a disorganized fashion and then later patch in the appropriate control flow.

The more salient arguments proffered in defence of the use of the GOTO are as follows:

(1) the GOTO is such an unconstrained control structure that it allows synthesis of more advanced and elegant control constructs

(2) the GOTO's ability to direct and override control flow in a program, offers it's users an escape from almost any awkward situation.

The proponents of the latter statement claim that structured programming devotees camouflage these awkward escapes by creating new or rather synonomous control constructs such as "leave", "exit", "escape" [Hopk72,Lask79]. These authors made no judgements as to whether these awkward situations were due to programmer sloppiness, or if they were simply inherent to the algorithm and hence unavoidable. Thus, it would seem that the fundamental power and utility of the GOTO makes it both desireable and undesireable.

Since it had been proved that any program could be expressed using only the one-in one-out control structures (structured control constructs), the recommendation was made that they be used to supplant conventional unstructured (GOTO laden) control flow elements. It was argued [Dijk76,Holt75,Mill73] that use of structured control structures would yield programs with greater readability and maintainability. The reason cited was that it allowed programmers to better follow the flow and hence the logic of a program. Programmers would also be able to trace backwards through a program. This is something which was difficult to do with GOTO programs because all labeled statements could be reached by all appropriately labeled GOTO's. The dilemma of which GOTO (if any) summoned the labelled statement in question, would no longer be a predicament.

Another feature of structured constructs was that it allowed for more distinct and meaningful blocking criteria [Dijk65].

Recognizing how sections of code (blocks) were related to one another would also improve the programmer's ability to understand a program's logical function. Dijkstra claimed that these benefits would make control flow errors easier to isolate, since the only way that a program could fail was through infinite recursion or through faulty loop controls.

Earlier it was mooted that many researchers felt that elimination of the GOTO construct was the necessary and sufficient "prescription" for alleviating the debilitating symptoms of so called "bad" software. On this premise, several authors [Ashc71,Pete73,Kosa73,Kosa76,Lipt75] presented algorithms, restructuring engines, for replacing all GOTO's by structured programming language equivalents, constructs displaying single-entry single-exit control flow. They discovered that these "new" structured versions of program code were not significantly better in terms of being more understandable, maintainable or reliable then the original "unstructured" code [Knut79].

As more programmers started using structured languages, it became readily apparent that it was just as easy to create "bad" code with structured programming constructs, as it had been with languages containing the GOTO construct [Berg79a]. The GOTO was thus partly exonerated: it was neither the cause of the symptoms of "bad" software, nor was its use merely the effect of sloppy reasoning.

It is surprising that in the multitude of papers on the GOTO controversy, only few mention that the underlying deficiency of the GOTO does not reside in the fact that it allows the programmer to explicitly define control flow. Rather, problems occur as a result of the fact that labels (entry points from GOTOs) may be placed almost anywhere in the body of the program and as such control may be transferred into the middle of a previously well defined* section of code. These jumps which typically allow for a reduction in the amount of redundant code may result in confused logic because the established purpose of a particular piece of code is no longer applicable if entry occurs at a place other then the beginning.

Several authors have argued, albeit indirectly, [Knut74,Kenn79] that the difference in quality between structured and unstructured programming, should not be attributed to the use of the new control constructs. Rather, it should be regarded as a byproduct of the extra thought required in avoiding the frequent use of crude and powerful expedient, the GOTO! Current attitudes toward the GOTO, fall more in line with a realization that the GOTO can be easily abused and that the programmer must be careful when using such constructs. The statement: "where time or intelligence are lacking, a GOTO may do the job" [Hopk72],

----

* In terms of logical function

is no longer an accepted criticism of the GOTO.

2.2.2 TOP DOWN DESIGN

Top down design is a methodology in which, by starting with the problem statement, we construct a hierarchy of algorithms [Dahl72]. The higher levels are concerned with isolating important partitions of the problem statement, while lower levels involve actual written code. Movement through the hierarchy, from top to bottom, should show a gradual increase in the detail of specific algorithms, their data structures and operations. The bottom level of such a design methodology specifies the machine executable version of the algorithm.

Using Shaw's definition of the term abstraction as: "a simplistic description or specification, of a system that emphasizes some of the system's details or properties while suppressing others" [Shaw80], we can safely say that "abstraction" is the cornerstone of the top-down design methodology. This top down methodology provides a hierarchy of such abstractions, in which the highest and lowest levels show respectively the least and the most implementation dependant detail.

Other authors [Parn72,Dijk72] have also stressed the fact that the suppression of detail is essential for a thorough understanding of the solution to a problem. Information hiding [Parn72] revolves around the idea of making visible only those properties of a module needed to interface with other modules. This requires the user to clearly define and enforce module boundaries. Stepwise refinement [Wirt71] is defined to be a technique by which an individual approaches a program solution by slowly evolving and adding onto an initial crude piece of code (design documentation).

2.2.3 PROGRAM QUALITY

The last major component of the structured programming methodology is that of program quality. Although the word quality is rather broad and very vague it is generally understood to mean [Hoar72,Dona73,McCr73,Your79] that software should be:

(1) as easy as possible to read and understand

(2) easy to test,debug and integrate

(3) easy to maintain - both modifiable and extensible

(4) reliable - does what is required without unpleast surprises

(5) efficient in terms of -

   (a) development costs for initial implementation

   (b) run time resources

(c) maintenance costs

(6) easy for HUMANS to use - user friendly

(7) as rugged as possible - especially with reference to recovery from errors

(8) produced on or ahead of schedule

(9) extremely portable -

(a) execute properly on different machines

(b) easily spliced into other pieces of software - good library routine

Numerous methods have been proposed on how to achieve these goals. The following is a list of some of the more popular methods*:

(a) inclusion of sufficient and necessary documentation both internal and external to the code

(b) high degree of modularity - both inter and intra procedural

(c) meaningful variable names - mnemonic aids to help remember what each variable stands for

(d) proper and consistent indentation - visual aid for

----

* The last four entries (i,j,k,l) are not part of the structured programming methodology, but have been included for completeness.

control flow and modularity

(e) reduce and simlify the number of control paths - through the use of restricted control constructs and decreased levels of nesting

(f) routines of limited size

(g) restricted use of global data

(h) Chief programmer teams - increase the effectiveness of communication by working in teams rather then as individuals+

(i) software metrics - to measure and control complexity.

(j) prototyping - making available to the user a subset or a simplified version of the total system

(k) extensive static and dynamic software testing

(l) complete and definitive requirements definition

Although the extent of the effectiveness of these adjuncts is still disputed, it is generally agreed that they are not detrimental activities.

One of the most important characteristics of the techniques listed above is that they consistently stress simplicity of design and the use of methods that hinge on standard representations. It seems that the overall goal was really

----

+ These teams also make use of the other features of Structured Programming to aid understanding of the code and related products by presenting them in familiar structural context. Researchers have attempted to build onto the Structured Programming methodology by elaboration of one or several of the software quality factors listed above.

2.3 SOFTWARE LIFECYCLE MODEL

The software development environment was enhanced by the inception of the software lifecyle model [Boch79] which involves several serially interdependant phases, the progression of events in the Lifecycle is graphically represented in figure 2.1. This approach is derived from the manufacturing concept of lifecyle in which "products are first concieved, specified in detail, designed and then built and maintained until they are no longer useable" [Wass80a].

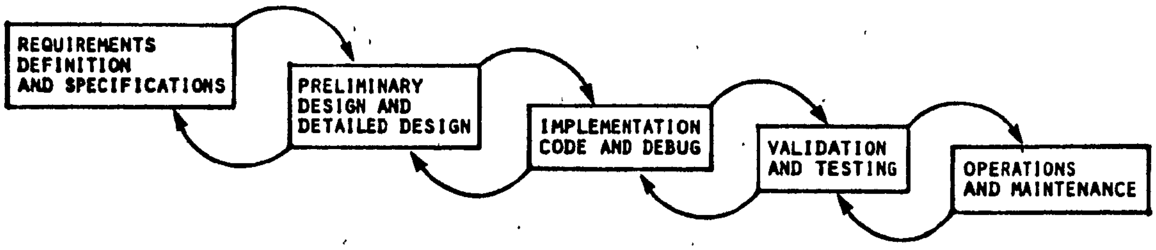

REQUIREMENTS DEFINITION AND SPECIFICATIONS
PRELIMINARY DESIGN AND DETAILED DESIGN
IMPLEMENTATION CODE AND DEBUG
VALIDATION AND TESTING
OPERATIONS AND MAINTENANCE

FIGURE 2.1 A model of the Software Lifecycle (Boeh79)

The most important characteristic concerning the lifecycle is the fact that for the first time researchers and developers stressed that intermediate products were at least as important, if not more important than the finished software [Wass78]. Although Structured Programming methodology emphasized top down design; the major impetus for its use was concentrated at the level of improving code. Researchers were still trying to treat the symptoms rather then attempting to eradicate the disease of "poor" software.

2.3.1 REQUIREMENTS DEFINITION/SPECIFICATIONS

Requirements specification is a set of documents whose purpose is to describe what a software package is expected to do. It should also describe any interaction with other systems, people, and device. This is fundamentally a process of data gathering (systems analysis) followed by a formalization procedure [Howd82]. Specifications should be complete, consistent and unambiguous enough so that they can serve to demonstrate to interested parties+ that the problem is properly understood.

These specifications are also used as a guideline for the

+ Software production companies often use these documents as legally binding contractual agreements [Boeh79].

subsequent design phase [Pete81]. If the proposed system is not defined in a "sufficiehtly precise" way then the final system cannot be checked for correct implementation. Thus we see that the requirements definition and specification documents are functional at both pre and post implementation stages. Hence possibilities for misunderstanding exist both between and within the various groups.

Currently most software specifications are informal and expressed in free form English [Boeh79]. They are often merely notes which have been assembled in a cut and paste mode [Howd82]. This adhoc approach is reflected in the fact that requirements specification documents suffer from numerous critical deficencies [Leve82]:

(a) omission of key information

(b) ambiguously stated information (results in terms such as : suitable, sufficient, flexible)

(c) wrong or infeasible statements of requirements

(d) untestable statements of requirements (terms such as : optimum, 99.9% reliable)

Likely reasons for the above defects are due to the fact that the approach is too informal, and does not have the appropriate support tools.

In attempts to overcome these deficiences and to more effectively fulfill the role of requirements definition and specification documents, numerous formal languages have been developed: SADT [Ross77a,Ross77b], PSL/PSA [Teic77], SREM and SDS [reported in Leve82]. These tools can have a fixed predefined specification language and/or facilities which allow for language definition.

2.3.2 SOFTWARE DESIGN

A software design specification is a document whose purpose is to model a solution for a proposed problem. The design model should then be able to "guide" the implementation of the software. Thus each design document is a "blueprint" of the method by which to map an algorithm onto a computer program.

Until recently (1975) not much effort was being expended on this area of software development [Grif79,Berg79a]. However research studies have revealed that not only were design errors more prevalent than other types of errors, they were also the most difficult to correct [Boeh79]. These statistics prompted renewed interest in the area of software design with particular emphasis being placed on the top-down approach (described earlier). A currently accepted practice of system design is to approach the design problem in the following two phases [Pete81,Boeh79,Free80a]:

PRELIMINARY DESIGN (LOGICAL ARCHITECTURE): is a high level system model which describes the method by which tasks are to be performed, unencumbered by low level details.

DETAILED DESIGN (PHYSICAL ARCHITECTURE): is an expansion of the preliminary design to include sufficiently low level details to guide implementation.

This does not preclude the possibility of having numerous other levels of refinement (abstraction). These design products are primarily informal prose descriptions and/or diagrams generated using formal methodologies [Howd82].

Other design representation techniques and methodologies present varied approaches to viewing any given problem. Each method emphasizes different aspects of the system which in turn generates varied responses from its viewers. Typical design representations concentrate on one or more of the following areas [Pete81]:

Example: (1) Leighton diagrams [Pete81], (2)Hierarchy, plus Input, Process, Output (HIPO diagrams) [Wass78].

SOFTWARE DESIGN STRUCTURE

Primary concern is to provide a method by which to organize and develop software structure in a manner which addresses the critical issues of integration and interface definition

[Boeh79]. More popular techniques are based on representations which accommodate the hierarchical approach of top-down design and information hiding.

Example: (1) Structured Analysis and Design Technique (SADT) , (2) Design Trees [Pete80], (3) Structure Charts [Stev79], (4) Structured Decomposition Diagrams (SDD) [Rudk79].

SOFTWARE CONTROL FLOW

We wish to provide methods with which to illustrate the paths (execution sequence) that a required computation may follow. This is the oldest and most popular area with regard to design representations. Thus numerous schemes are currently available. The techniques are used primarily for describing low level control flow behavior (algorithms)such as would be required for the detailed design phase (decision points, loop structure, control sequence). Main criticism [Boeh79] of these techniques is that they are used "too much" and hence results in the neglect of the other areas of design.

Example: (1) flowcharts [DeMa74], (2) pseudocode [DeMa74], (3) decision tables [Mont74], (4) Structured Control and Top down (SCAT) [Boch79], (5) Program Design Language (PDL), (6) Problem Analysis Diagrams (PAD) [Futa81], (7) Nassi/Shneiderman diagrams [Yode78], (8) Warnier Orr diagrams [Orr79a,Orr79b], (9) control graphs [Pete81].

SOFTWARE DATA FLOW

These representations are geared towards establishing software designs which attempt to optimize data management. The focus is to provide methods with which to illustrate data characteristics as well as its concomitant flow and transformations. This non procedural approach is a reflection of current software priorities which are basically not computationally bound. Modern day practice is occupied with systems for handling vast volumes of data but which undergo few complex transformations. The following techniques are related to data base design criteria.
Example: (1) Jackson Data Structures [Berg79b], (2) Chen Entity-Relationships [Pete81], (3) DeMarco Data Structure Diagrams [DeMa79], (4) Flory and Kouloumdjian Approach [Pete81].

Software design is a highly creative human endeavor and as such it is not surprising that it is not easily understood or controlled. Software design does not lend itself well to prescriptive or sequential steps that will ensure success, like the directions for assembling a bicycle. Nor is it a deterministic process in which it will be "possible to exhaust all reasonable variations and averages of inquiry to reach an acceptable design" [Pete81]. Thus it is foolish as well as naive to assume that it would be possible to point to any given technique listed above and proclaim it to be the "best". Rather we should realize that each has particular merits which enable it to excel in some specific problem area.

The following is a list of some of the criteria which are important if not essential for the construction of a successful design process [Pete81,Berg79a]:

1) A design process must allow for many levels of abstraction: from the conceptual model to functional details. This could include:

   a) graphical notation (graphs, trees, tables, flowcharts)

   b) prototypes (simplified working model of final system)

   c) natural language documentation (with some form of overlaying structure)

2) The process should be based on an underlying philosophy or rationale which helps the designer to focus his activities. The design philosophy should ensure consistency and provide for more effective communication among designers. Such a philosophy should also form an infrastructure for providing better documentation tools.

3) The design technique should be flexible enough to accomodate persons of varying backgrounds thus allowing them to bring their experience to bear on the problem[Wass80b]. Such flexibility permits more

efficient contribution from each individual within the design group.

4) A good design method should also be easily maintainable, so that refinements to the design model can be made as human understanding of the problem grows. Similarly, we can handle the inevitable changes in the design specifications that occur after implementation is complete.

5) A design process should be formalized to the point where it is possible to build automated tools for design validation and risk analysis.

2.3.3 CODING AND DEBUGGING

This phase of the software lifecycle involves the actual implementation of a piece of source code. The objective is, as it was from the beginning is to produce "quality" software which is a physical realization of the design specification. The fact that the defintion of quality has changed since then is not at issue. The quality techniques discussed in section 2.2.3 for Structured Programming methodology apply equally well here and thus will not be reviewed. Although unstructured languages are still in wide use [Phil77] it is generally accepted that languages which support structured control constructs are the languages of choice. Examples of such languages include ADA, Pascal, and C as well as other languages such as structured Fortran and Cobol which have been retrofitted to accept these constructs.

Specifications for the languages of the eighties include the adoption of concurrent processing and data flow languages [Back78,Litv82]. More interestingly other research innovations point to the ability to create source code directly from software specifications. The former research is bounded by the development of appropriate computer architecture while the latter work underscores the trend of shifting the emphasis of our endeavors away from technicalities and necessary evils.

2.3.4 SOFTWARE TESTING AND VALIDATION

The objective of this phase of the software lifecycle is to determine that the software produced is functional, reliable, and correct (in terms of user specifications). Testing is a very costly stage of the lifecycle, incurring approximately 40 to 50% of the initial development costs, figure 2.2a [Boeh73]. It is extremely time consuming and frequently requires reworking of code and/or modifications to design and specifications. Typical stages that occur in testing are as follows [Wass80a]:

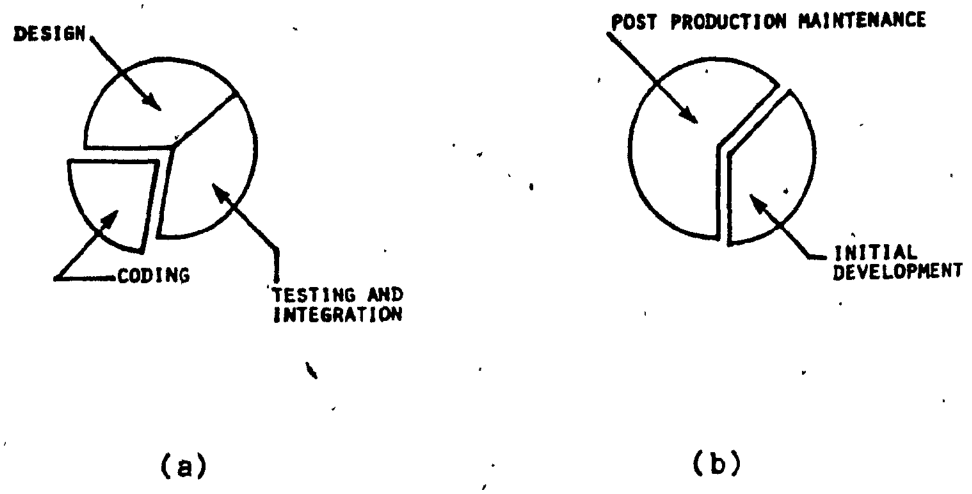


FIGURE 2.2a Costs incurred during initial development
2.2b Costs incurred during software system lifetime

UNIT TESTING

Individual program modules are tested for correctness. This level of testing usually reveals common programming errors such as: uninitialized variables, inappropriate array bounds, inadequate housekeeping of variables, poor I/O formats. Usual tests include exercising every statement, branch and as many parts of the code as is economically feasible.

INTEGRATION TESTING

Program modules are assembled to ascertain if they function properly together. Integration testing usually reveals errors committed during the design phase, such as inconsistent procedure interfaces or inappropriate module boundaries.

ACCEPTANCE TESTING

The intended user/customer evaluates the system with respect to his initial specifications. Errors detected at this level may reflect errors in the initial specifications and can result in extensive reconstruction of the system. These errors are usually attributed to incomplete, inconsistent, incorrect or ambiguous statements of what the system was intended to perform.

The following is a list of some of the more popular testing procedures and methods which will be discussed in detail in chapter 6: 1) Peer Code Review, 2) Code Walk Through, 3) Fault Isolation, 4) Test Data Generation, 5) Program Mutation 6) Symbolic Execution, 7) Program Proofs.

It is important to note that our present methodologies detect errors in what is perhaps the worst possible manner. Errors are detected in inverse relation to when they were committed (LIFO behavior): coding errors are detected first whereas specification errors are encountered last. The cost incurred by this phenomena is probably the principle reason behind the move to increase "quality" of products in the early stages of the lifecycle.

2.3.5 POST PRODUCTION MAINTENANCE

This stage of the software lifecycle involves making changes to software which is already operational. Maintenance is usually the most costly activity incurred by any major piece of software. Average costs of maintenance run in the order of 50 to 75% of all costs associated with the functional lifetime (from initial development to final shelving) of the software systems, figure 2.2b [Berg79b].

Software maintenance can be subdivided into two major categories [Boeh79]:

a) Software Update: includes those changes which result in a modified functional specification. This usually involves adding or deleting capabilities to or from a software system.

b) Software Repair: includes only those changes which result in an unaltered functional specification. This would include modifications such as: algorithm enhancement in terms of performance/readability/adaptability, minor I/O changes or corrections to previously overlooked errors.

To be able to program either of the above maintenance tasks effectively requires the ability to execute the following functions:

1) understand existing software
   Implies: proper documentation, good mapping of code to requirements, well constructed code

2) modify existing software
   Implies: easily modifiable documentation/software, minimal side effects from required changes

3) revalidate modified software
   Implies: software which can allow for selective retesting and/or tools for selective retesting

The above factors make evident the fact that programming involves a great deal more then just coding, thus advocating that perhaps problem to program transformations should no longer rely primarily on tools to support coding. The concept of the Software Lifecycle has provided a framework within which to coordinate the different techniques for software production, as well as a structure with which we may consider the devopment process as a single entity rather then just individual unrelated stages.

CHAPTER 3

PSYCHOLOGY OF PROGRAMMING

The production of computer programs has been considered an art form [Knut74], and as such it was found to be extremely sensitive to the individuals involved in its production. This sensitivity resulted in products whose intrinsic qualities were highly unpredictable in the large and inconsistent, at best in the small (large - refers to industry in general : a GLOBAL EFFECT; small refers to individuals, small groups or a small company : a LOCAL EFFECT). Researchers have invested much effort in attempting to formulate a discipline whose underlying functional goal would be to curb these artistic tendencies. It was believed that these disciplines would allow the process of computer programming to achieve its crucial transition into the "science of computer programming".

3.1 PROGRAMMING: ART TO SCIENCE

The definition of science has been listed as follows [Knut74]:

"knowledge that has been logically arranged and

> systematized in the form of general 'laws'. The advantage of science is that it saves us from the need to think things through in each individual case;"

Thus it seems that the mandate of software engineering researchers should be to isolate those variables which are thought to play an important role towards "systematization" and for the genesis of "general laws" of computer programming. Theoretically the application of these laws would yield a carefully controlled product which would achieve an acceptable balance between optimum productivity and maximum quality.

As mentioned in the previous chapter, the methodologies which are currently in vogue have not attained their requisite goal of the "science of programming". Abrahams [Abra75] makes the following statement concerning why he believes Structured Programming Methodology has failed:

> "the problem with structured programming lies not so much in its content as in its sociology. There are two baleful aspects of this sociology: the elevation of good heuristics into bad dogma, and the creation of the illusion that difficult problems are easy. If structured programming is treated as a collection of inflexible rules which can replace good judgement, it will ultimately increase rather than decrease our

> efforts, while concealing from us the fact that this increase has occurred."

This statement points to a very important and fundamental oversight on the part of researchers involved with the development of these disciplines.
Programming must not become dehumanized to the extent that inflexible rules prevail over good judgement. In other words, we must avoid statements like: "In the production of computer software the product is more important than the process" [Chap78]. Computer scientists must remember that programming is predominantly, if not uniquely a human activity (excluding artificial intelligence systems and automatic programming). We must remember that it is ultimately the human who must solve the problem and creates the software. Thus we must be careful and more selective of the tools that are developed and enforced for it is these which will ultimately determine the quality (if any) of the end product. This provides a good starting point from which we may reevaluate the process of software production with regard to humanistic criteria.

3.2 PARADIGMS OF PROBLEM SOLVING

Programming has been described as being the study of human problem solving, but at a level which has been relatively unexplored by cognitive psychologists [Shne75]. Though this statement has the obvious ring of truth the literature in computer science has until recently [Shne77,Broo77] been relatively devoid of reference to psychological investigations within the domain of problem solving. Probing into these neglected paradigms of problem solving may yield valuable insights about the psychology of programming [Shie81,Maye81].

Psychological theories of problem solving have developed from both empirical and theoretical explorations. Experimental research has probed heavily into both human and animal studies. The techniques used have spanned the entire gamet of available psychological methodologies, from passive observation of natural behavior to active manipulations of artificial environments. The literature in the area of cognitive psychology and problem solving is quite diverse and a thorough review would be beyond the scope of this thesis. The following section is a brief synopsis of four of the more widely accepted theories of problem solving.

3.2.1 BEHAVIORIST MODEL

This theory was first formulated by Thorndike in 1898 [Maye77]. After extensive study of cats in puzzle box situations he noticed that with increased practice the cats required less time to escape from the boxes. He concluded that the cats were learning on the basis of accidental successes. The model proceeds under the fundamental assumption that problem solving is strictly a trial and error process.

The behaviorist model reduces all problem solving situations into three basic elements:

1) stimulus - a problem solving situation
2) responses - solutions and general problem solving behavior
3) associations - links that are formed between the stimulus and the response

These three elements are combined as follows:

```
           elicits                  creates
STIMULUS ----------> RESPONSE ----------> ASSOCIATION
```

If a given response yields a reward, then the association between this response and the original stimulus is increased. If no reward is present then the strength of the link is decreased.

Thorndike believed that this "law of effect" enabled the problem solver to create a hierarchy of responses which could be used in future problem solving situations. The model subsumes that the brain acts as a storehouse of solution hierarchies which are formed from previous problem solving episodes. Thus when similar problems are presented the subject will likely present the most rewarded response (highest in hierarchy) first and proceed through the hierarchy until a suitable solution is found. If the hierarchy does not contain an answer the subject is forced to find new response via trial and error.

Major criticisms of this model concern the fact that it does not adequately describe the complexity of human thought [Davi73], especially with regards to the fact that the model strips the problem solver of the ability to consciously or deliberately guide the process of problem solving. Another difficulty with the behaviourist approach is that it presumes a great deal of repetition whereas real world problems hardly ever are.

3.2.2 RULE LEARNING MODEL

This model is based on the premise that problem solving is merely a continuous process of hypothesis testing [Maye77,Davi73]. The hypothesis or "rules" are generated through a classification scheme termed CONCEPT LEARNING. This process is subdivided into two types of tasks:

1) concept identification tasks requires the problem solver to discover relevant rules for a problem in

which he already knows the stimulus dimensions (data objects)

2) concept formation tasks require the problem solver to discover relevant rules for problems in which he must also discover the relevant stimulus dimensions.

The thinking process required for concept learning has been characterized into two basic classes of theories. The first, termed CONTINUITY THEORY, is an extension of the behaviorist model and claims that the hypotheses to be tested are derived from response hierarchies. The second, termed NONCONTINUITY THEORY, claims that concept learning involves the ability to induce rules. This implies that new hypotheses are generated, voluntarily and in a conscious manner by the problem solver. The hypothesis is tested and changed only when it fails to work.

Although researchers working with rule induction tasks have amassed an impressive amount of detailed statistics on human problem solving, they have been criticized [Maye77] because the research involves only one type of problem solving situation, hence results may not be readily applicable to more general areas. Another major criticism concerns the finding that subjects use "rules" in problem solving. Some psychologists have argued that this is merely an artifact of the experimental methodology in which the rules are built into the task. In other words, if the experimenter devises a problem situation which requires a subject to discover a rule, it should not be surprising when the subject discovers that rule.

3.2.3 GESTALT MODEL

The Gestalt approach to problem solving is to view the thought process as a highly mental activity (CPU bound!). The theory assumes that the search for solutions is dependent on the problem solver's ability to understand how the parts of a problem can be made to fit together to achieve the goal [Maye77,Davi73]. This phenomena is defined as STRUCTURAL UNDERSTANDING.

The Gestalt psychologist stresses the concept of organization and structures. The central idea is that the structure of the problem should point the way to its solution. The actual process of problem solving is perceived as a continual progression of mental reorganizations of structures into subjectively more ordered components. At some point in the rearrangement when the mind achieves structural understanding, the correct solution is instantly revealed. This sudden flash is termed "insight" (often accompanied by the word, "AHA!" : Martin Gardner reported in [Maye77]).

Through observation and introspective analysis Gestalt theorists have proposed a taxonomy for the problem solving process. The following list provides a chronological perspective of the slowly evolving Gestalt views of these phases [Sack70].

WALLAS (1926)
* Preparation
* Incubation
* Illumination
* Verification

ROSSMAN (1931)
* Observation of a need or difficulty
* Analysis of the need
* Survey of the available information
* Critical analysis of the proposed solutions for advantages and disadvantages
* Birth of the new idea, the invention
* Experimentation to test out the most promising solution: perfection of the final embodiment by repeating some or all of the previous steps

DEWEY (1938)
* Distributed equilibrium, initiation of inquiry
* Problem formulation
* Hypothesis formulation
* Experimental testing
* Settled outcome, termination of inquiry

OSBORN (1957)
* Orientation: pointing up the problem
* Preparation: gathering pertinent data
* Analysis: breaking down the relevant material
* Hypothesis: piling up alternatives by way of ideas
* Incubation: letting up to invite illumunation
* Synthesis: putting the pieces together
* Verification: judging the resultant ideas

The stages are not mutually exclusive nor are they strictly sequential in nature. Since the phases are not distinct it has been extremely difficult to use experimental techniques to quantify both the effort and time required at each stage, Another basic concept in the Gestalt approach is that all problems are not created equal. They claim that problems can be solved using one of two kinds of thinking, either PRODUCTIVE or REPRODUCTIVE. The basic distinction between these is that the first requires a new organization to be developed (creative thought) whereas the latter merely involves reproduction of behavior performed for past solutions. These can be linked to the arguments used for types of learning: rote learning versus learning through understanding.

Based on this qualitative distinction in types of thought, Gestalt psychologists have also proposed and experimentally demonstrated the concept of rigidity in the problem solving set [Maye77]. In other words problem solvers often become FIXATED on inappropriate approaches for deriving a correct solution. The cause of fixations generally falls within one of the following domains [Sche63]:

1) incorrect premise concerning problem requirements
2) solution is not within conventional context
3) problem solver is unwilling to take detour which will delay solution
4) habituation (always attack problem in same way)

Fixation is overcome by recognizing that it exists and by encouraging shifts the way problems are viewed and approached.

Although the Gestalt approach has been praised for recognizing the complexity of human mental thought processes as well as for introducing many provocative ideas, they have been criticized for being too vague. Critics [Maye77] have also pointed out that the general theory is very subjective and hence difficult to test explicitly.

3.2.4 INFORMATION - PROCESSING MODEL

This theoretical model assumes that human thought process functions as an information processing machine. Information theorists believe that their techniques can be used not only to describe but to explain problem solving [Newe72]. The theory states that problem solving by humans requires the execution of Elemental Information Processes (E.I.P.s) and that these operations can be simulated on a computer. The model draws on the fact that the basic human requirements for problem solving have analogous computerized capabilities. The following illustrate some of these fundamental correspondences [Maye77].

HUMAN	COMPUTER
sensory input	keyboard, tape, disk, TV camera, touch sensitive screens, light pens, joy sticks
memory: LTM, STM	disk, tape, cards, ROM,

	RAM (core memory)
decision making ability	preprogrammed decision making criteria (rules)
learning	write into memory
forgetting	erase from memory
output (talking, writing, drawing)	alphanumeric printing, talking, graphing, etc.

The goal of the information processing theorist is to reduce complex problem solving behavior to a set of elementary processes which can be used to either produce, imitate, or duplicate human thinking. To isolate these EIPs they have adopted an experimental procedure which involves asking human subjects to solve problems aloud, hence giving a running description of their thought processes and behavior. Using a careful analysis of the transcript of the subject's comments (known as protocols) and through comparison to other subjects, researchers have been able to distinguish three intrinsic categories of internalized information which humans draw upon to solve problems [Lind73].

FACTS : knowledge which is instantly available.
Example - the multiplication table by which we can answer questions like what is 4 X 8.

ALGORITHMS : sets of rules that allow the user to automatically generate correct answers.
Example - multiplication rules with which we can answer questions like what is 262 times 127.

HEURISTICS : rules of thumb or general plans of action.
Example - "estimation rules" with which we may answer questions like what is the approximate answer to 262 times 127.

Using these approaches researchers have been able to create programs which can solve problems in well defined areas, such as logical proofs and Chess [Newe72], understanding natural language [Wino77], and many others.

Major criticisms of this theory stem from the fact that although computer programs simulate human thinking behavior this does not imply that it has simulated the underlying cognitive processes. Thus information processing theorists should not conclude that they have explained problem solving.

3.3 HUMAN PROBLEM SOLVER

Having examined these paradigms of problem solving we can now turn our attention to those attributes of "mind" which support the process. Cognitive psychologists [Lind73,Shne77,Shie82] have labeled the following cognitive structures as being fundamental to problem solving: Long Term Memory, Short Term Memory and Creativity/Intelligence.

3.3.1 LONG TERM MEMORY (LTM)

LTM is the storehouse of our accumulated knowledge. It provides the means with which to extract and utilize pertinent and necessary background material. The most important feature concerning LTM is that it has, for all intents and purposes, infinite volume. LTM memory is capable of organizing [Newe72] "stimuli or patterns of stimuli, from input channels" into one recognizable symbol which when retrieved represents the entire stimulus. This phenomenon is called "chunking", and represents the ability to form links between individual elements. The classic experiments in chunking were demonstrated by the difference in ability to restore chess configurations between chess masters and novices [Newe72,Lind73]. The masters were often able to perceive complex patterns as single higher level units thus making retrieval easier than the novice who attempted to remember individual player positions.

Two distinct and unfortunate disadvantages of human memory are: (1) LTM is prone to partial memory loss (permanent or temporary), and (2) LTM has comparatively slow and most often incomplete input into storage.

3.3.2 SHORT TERM MEMORY (STM)

The function of STM can be considered analogous to a scratch pad. It allows the user to temporarily store information, currently of value, into an area where it is instantly retrievable. It is thought that STM is where manipulations and reorganizations are performed, hence this is often associated as the focus of consciousness and reasoning: "inner voice" [Trac79]. The acoustic property of inner voice seems to indicate that processing of information in STM proceeds in a sequential fashion.

STM suffers from two major disadvantages: (1) the quantity of new information which can be stored and manipulated is extremely small: (7 +/- 2 'chunks',) and (2) information retained in STM is extremely volatile and will fade if it is not rehearsed frequently,on the order of once every 20 or 30 seconds.

3.3.3 INTELLIGENCE AND CREATIVITY

Psychologists and philosophers have long struggled with the defintion of both these terms. Although conceptually easy to grasp they deal with very abstract, not formally defined and not easily quantifiable properties. Therefore I will not venture anything more than the following brief and purposefully naive definitions:

1) Intelligence represents both the ability and the facility to process information.
2) Creativity represents the ability to combine and restructure elements in new and unique (to the

individual) ways.

These two characteristics of the human problem solver translate into a variety of problem solving techniques [Poly65,Lind73,Rubi75]:

FORWARD REASONING

Conceptually driven approach in which the problem solver starts at the initial state and slowly evolves a solution by working forward towards the desired goal, assessing his progress after each step.

BACKWARD REASONING

Data driven approach is one in which the problem solver studies the desired goal and attempts to ascertain which steps must have preceeded it, thus working backwards towards the initial state.

INSIGHT

As explained earlier insight is not an approach but rather the sudden revelation of a solution which occurs after having spent time studying the details of a problem and performing some mental manipulations on the problem.

PROBLEM MATCHING

This template matching scheme involves the problem solver remembering if he has encountered similar problems in the past. If he has, then he can use his previous solutions as guides to solving the current problem.

DIVIDE and CONQUER

In this approach the problem solver breaks up a complex problem into conceptually more manageable sub-problems which he can solve. These sub-components are then recombined to obtain the solution to the initial problem.

TRIAL and ERROR

When the problem solver has not been successful using other techniques or he has no clue as to how to approach a problem he can always resort to this technique. This involves tackling the problem any way at all (within reason) hoping to accidently uncover the solution.

HEURISTICS

This involves using a general mode to attack a specific problem, with the prevailing philosophy that each specific plan will probably but not always yield a good solution.

Assuming that the human problem solver understands a given problem he cannot consistently deliver the "best" or even a "good" solution. For that matter, he may not even be successful at all. The following are some of the many obstacles and difficulties which can and most often do arise during a problem solving encounter [Rubi75].

1) failure to use known information

   a) inadequate organization can result in an oversight

   b) an overly complex problem model can obscure relevant details

2) unnecessary constraints imposed on the required solution

   a) Association constraint: concerns the manner in which elements are viewed to be related.

      Example: thinking that a problem must be solved in 2-D space when the actual solution exists in 3-D space.

   b) Function constraint: concerns the context in which elements are used.

      Example : not realizing that boxes can support as well as hold things.

   c) World View constraint: concerns the global context in which elements function.

      Example : not realizing that a problem can be solved outside of a self-imposed mathematical reference frame.

3) inadequate knowledge base

   a) state of the art is not sufficiently advanced

   b) individual is lacking in appropriate training

4) limited motivation

   a) inadequate gratification

   b) negative attitudes about achieving solution

   c) insufficient interest

   d) insecurity

   e) lack of confidence

5) fixated solution set

   a) due to conformity: performs in a specific manner because he is "told to do it this way", or "everyone else does it this way".

   b) due to habit and inertia: "it works so why change".

Thus we have observed that problem solving is a very complex and usually unpredicatable process. Fortunately, however by having acknowledged that these attributes do exist we have taken our first step towards improving it.

3.4 PROBLEMS TO PROGRAMS

In chapter two we introduced a popular definition of the term program to be: the vehicle by which we express our wishes to a machine. Based on this definition the term "programming" might be interpreted as: the act of transferring our wishes to a machine. Although this definition can and does apply, it is misleading in that it does not adequately reflect the essential prerequisite of what it is that needs to be transferred. A more functional definition of programming might be as follows: a method of problem solving using a medium whose end product allows a computer to calculate the required answers. Although seemingly more accurate this definition presents programming as being fundamentally a uniform and highly complex problem solving continuim.

3.4.1 ALPHA/BETA PROCESS

We may clarify this ambiguity by viewing programming as a discontinuous process. Program generation can be dissected into the following two stage sequence:

```
                      ALPHA
1)  Problem     ===========>  Algorithm
                     PROCESS


                      BETA
2)  Algorithm  ===========>  Program
                     PROCESS
```

Although the two steps obviously represent problem solving situations, the type of problems each handles is somewhat different. We can continue this analysis by isolating important features in each of the above steps.

ALPHA PROCESS

a) involves establishing a "correct" mental model of the given problem. This requires that the individual be capable of performing the following functions [Lind73,Poly65]:

   1) understand the problem

   2) understand the goal

   3) understand the conditions imposed

   4) understand the data which is given.

b) creative manipulation of the model yielding a plan capable of guiding its user to a correct solution. This requires that the individual be capable of performing the following functions:

   1) capable of creating a finite state machine whose terminal states form the required goal

   2) capable of assessing the logical validity of being able to achieve these terminal states

BETA PROCESS

a) involves translating the correct algorithm into a formally defined syntax which is executable by a machine. This requires that the individual be capable of performing the following functions:

   1) understand the finite state machine which underlies the algorithm

   2) know the translation rules

   3) know the appropriate syntax

This simplistic model ignores the issue that language selection can often be part of the original problem.

The above separation reveals that problems to program transformation is not a uniform process. Although both processes require problem solving skills, there exist distinct differences in both the intellectual requirements and intellectual environment in the two stages. The ALPHA process is obviously a highly creative human endeavor while the latter, BETA process, is more literal and straightforward. In fact these two stages can be considered representative of the two categories of problems which do exist [Rubi75].

SYNTHESIS : consists of a statement of an initial state and of a desired goal. The major effort is in the selection of a solution process to the desired explicit goal, but for which the process as a whole, the complete pattern of the solution, is new to us, even though the individual steps are not.

ANALYSIS : this consists of focusing the application of known transformation processes to achieve a goal. The resultant transformations make clear what was originally obscure or hidden.

3.4.2 PSYCHOLOGY OF PROGRAMMING

Recently researchers have begun to model programmer behavior [Shie82,Shne79,Broo77]. The research has explored various experimental tasks such as program construction [Fitt79,Chry78], program comprehension [Sime77,Shne82,Shep81], maintenance and debugging [Duns78,Shne77] as well as studies on the programmer's ability to acquire new programming skills [Maye79,Maye81b]. Results from these experiments have demonstrated by and large that there does seem to be some sort of "qualitiative" difference between algorithm composition and algorithm implementation.

Shneiderman [Shne79] has proposed a cognitive model of programmer behavior based on syntatic/semantic interactions. The model partitions LTM into language independent (semantic) and language dependent (syntatic) cognitive based structures. Syntatic knowledge is precise, detailed and easily forgotten while semantic knowledge is less specific, deals with higher level concepts and generally is contextual in nature. Both types are acquired and abstracted from experience and instruction.

The model suggests that both of these knowledge sets are tapped in order to create an internal semantic representation (cognitive model) of a given computational prooess. "Semantic knowledge is essential for problem analysis while syntatic knowledge is useful during the coding and implementation phase" [Shne79]. When creating a program, semantic information is brought into play first and then formalized using syntatic knowledge. During program comprehension the converse is true.

Shneiderman did not propose a cognitive structure for either semantic or syntatic knowledge, however he did point out that the programming process will be eased when a language's syntax more closely reflects internal semantic structures. Other researchers [Broo77,Newl72] have proposed a storage schema for these knowledge structures. They suggest that knowledge is stored in the form of a production system: "a production system consists of a set of pairs of conditions and actions to be performed when the conditions are met" [Broo77].

3.4.3 COGNITIVE TOOLS

The issues discussed above have strong implications for the types of cognitive based tools which should be researched. For example LTM tools should highlight organizational techniques which can increase chunking ability. These tools should reduce the effects of slow input and memory loss through the use of techniques which stress conceptual simplicity. These tools should be designed to allow for easier storage and more complete retention. STM tools should also be directed towards creating an organized environment which is amenable to chunking. This would permit more complex information to be held locally in STM, thus enhancing problem solving abilities. Another priority should be to create external aids which can act as effective adjuncts to the resources of STM.

Neurochemistry, biochemistry, and psychology may someday provide a technology with which to increase the efficiency of the aforementioned human attributes necessary to problem solving. Till then we must accept the fact that humans have limited mental capabilities. The computer which has been heralded as a means by which mankind can extend these intellectual capabilities, cannot be expected to occur until researchers provide appropriate computer based tools to aid the human in problem solving. Only then will man have taken a substantive step in his quest for extension of intellect.

CHAPTER 4

INTRODUCTION TO ABL

Several authors [Broo74,Boeh73] have researched cost breakdowns of typical systems and have reported that actual implementation of code, on the average, requires only 20% of the total initial development costs of the software system, figure 2.2a. Given this statistic it might seem rather misguided that researchers have invested so much time and effort investigating and developing computer languages. Has computer code because of its high visibility (after all it's what makes everthing go) attracted more then its fair share of concern? The answer to this question is an emphatic NO! The reasoning becomes evident when we note the following statistics from figures 2.2a and 2.2b:

1) testing, debugging, integration and validation consumes 40 to 50% of initial development cost.
2) post production maintenance usually incurs 50 to 75% of total costs incurred during the functional lifetime of the system.

Since both of the above phases involve actually working with or changing computer code, we may appreciate why so much energy has been focused on computer languages. Not surprisingly then, the technique to be presented, though not restricted to being a computer language is nevertheless based on a functional yet language independant control construct. Several applications of the technique, coined ABL+ have been described elsewhere [Jawo82,Horv82,Lebe82,Morg81,Hint82].

This chapter will focus on some of the more fundamental and not yet documented characteristics of an ABL approach to the production and maintenance of software. Particular emphasis will be placed upon ABL's tabular notation, because it is this feature which most conveniently characterizes and underlies its function and varied capabilities.

4.1 THE ABL CONSTRUCT

ABL has only one control structure and it is related to selection: the KOMPUT. This control structure is illustrated in figure 4.1. The symbol S in figure 4.1 represents an ordered set of zero or more actions. The circle leading into each KOMPUT contains a number which represents the cardinality of that specific KOMPUT, known as the CLUSTER NUMBER. Each selection path (different decision points) is known as an ALTERNATIVE. The circle at the end of each selection path contains a value which is used to

----

+ Alternative Based Language

indicate control flow ordinality, NEXT CLUSTER. This number is used as a control flow pointer who's purpose is to link the given alternative to a subsequent KOMPUT or to process termination which is equated to the cardinal number: ZERO (0).

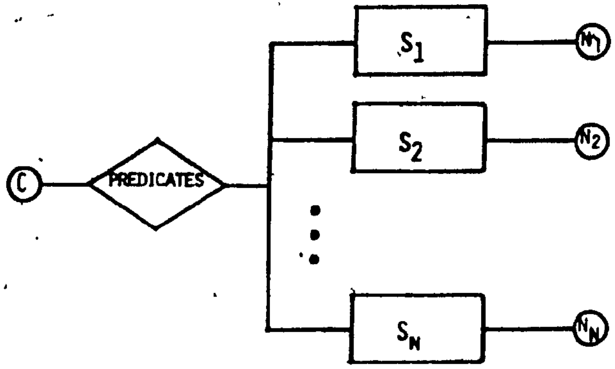


FIGURE 4.1 Structural description of the ABL KOMPUT construct

Some very important features to be noted concerning the ABL construct are as follows:

(1) the flow of control allowed by ABL is not restricted, apriori, to the single entry single exit control flow constraint which is a requirement of the structured programming descipline
(2) each KOMPUT may utilize single or multiple predicates
(3) the sequence construct is unnecessary.

Pertaining to the latter statement, we find that the underlying reason for this is that the KOMPUT construct has embedded within it the properties of the sequence construct. Therefore, sequence per se as an independent construct is foreign to ABL.

4.1.1 SUFFICIENT LANGUAGE CONSTRUCTS

It is an accepted fact that flowcharts are suited to representing computer programs. These diagrams are usually constructed from two elemental units (Boeh 66):

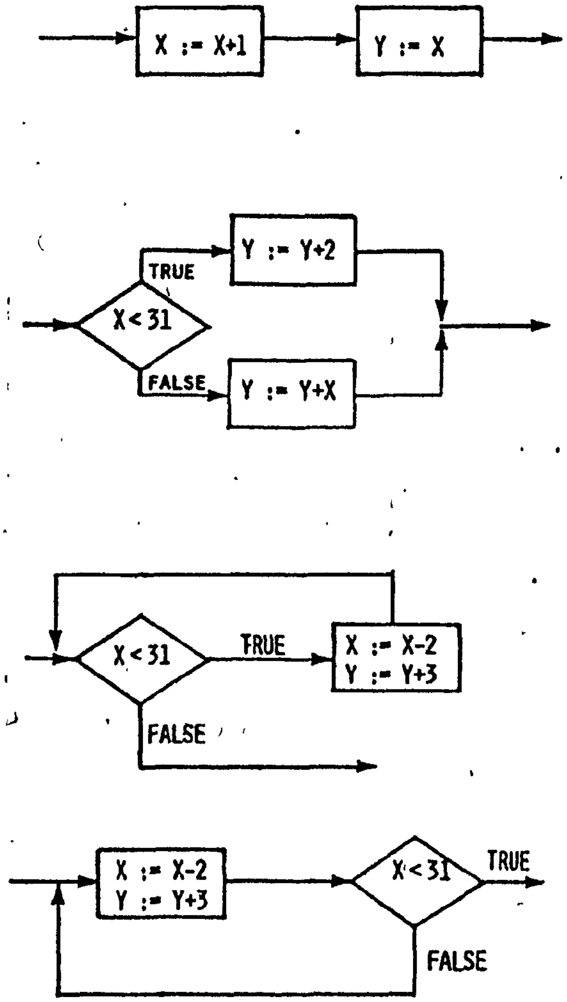
(1) Functional type boxes which represent elementary operations

(2) Predictive type boxes which permit selection (conditionals).

Bohm & Jacopini looked at the possibility of combining combinations of these elemental units into higher order control constructs called "base diagrams". In their classical paper (Boeh 66) they showed that although it is not possible to decompose all flow diagrams into a finite number of given base diagrams, they were able to prove that an equivalent algorithm could be constructed from three base diagrams: (1) sequence (2) selection (3) iteration. These structures later became known as structured control constructs and formed the basis of structured programming languages. Figure 4.2 illustrates flowchart diagrams of these three constructs and provides their structured programming language equivalents.

The theoretical work by Bohm & Jacopini showed that judicious use of restrictive control constructs can enable a reduction of gross structural complexity (spaghetti to linear code) without diminishing global expressive capabilities.

The author believes that the KOMPUT can be utilized in much the same way as the three structures control constructs. By changing the restriction that the construct be (1-in, 1-out) to (1-in, many-out) we can replace the three constructs by the KOMPUT. This in in keeping with the argument stated earlier that it was not the GOTO itself which was to blame for bad code, rather it was jumps into the middle of blocks of previously well-bounded logic that resulted in hopelessly complicated code.



```
X := X+1
Y := X
```

```
IF X<31
    THEN
      Y := Y+2
    ELSE
      Y := Y+X
```

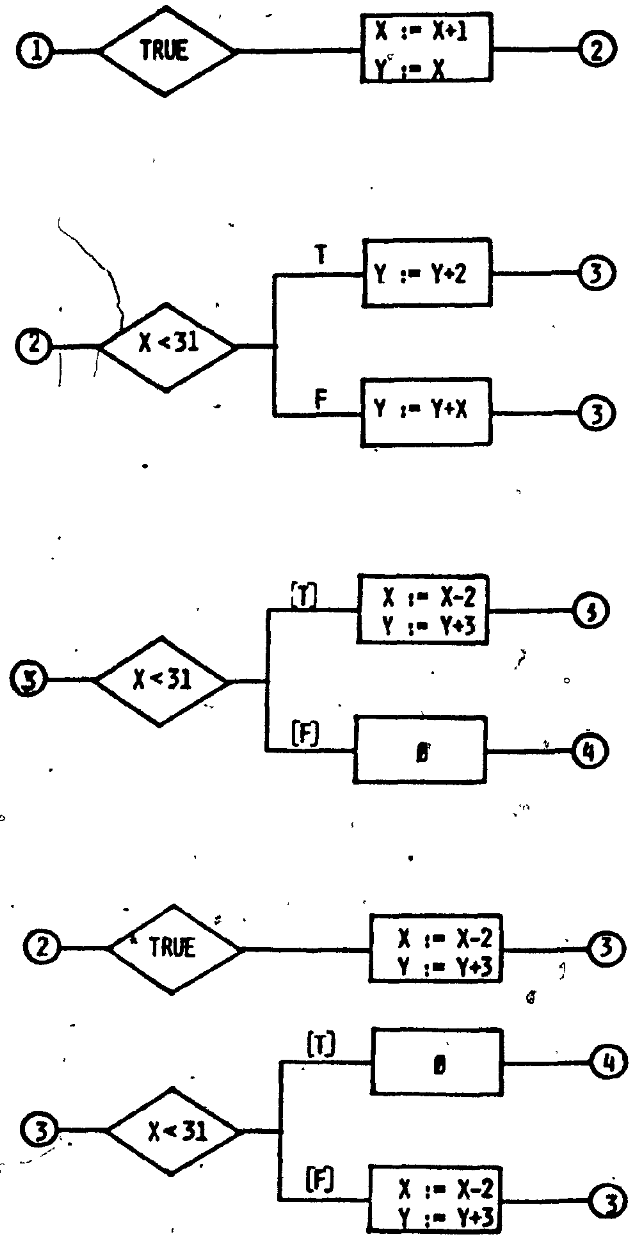
```
WHILE X<31 DO
    BEGIN
      X := X-2
      Y := Y+3
    END
```

```
REPEAT
    X := X-2
    Y := Y+3
UNTIL  X<31
```

FIGURE 4.2 Bohm and Jacopini control constructs and Structured programming language equivalents

4.1.2 THE KOMPUT: A SUFFICIENT CONSTRUCT

In order to show that ABL can represent any program (recursively enumerable function [Lew 82]) it is necessary and sufficient to show its equivalence to the three Bohm and Jacopini structures. Figure 4.3 illustrates how ABL can emulate each of these structures.



```
X := X+1
Y := X
```

```
IF X<31
    THEN
      Y := Y+2
    ELSE
      Y := Y+X
```

```
WHILE (X<31) DO
  BEGIN
    X := X-2;
    Y := Y+3;
  END
```

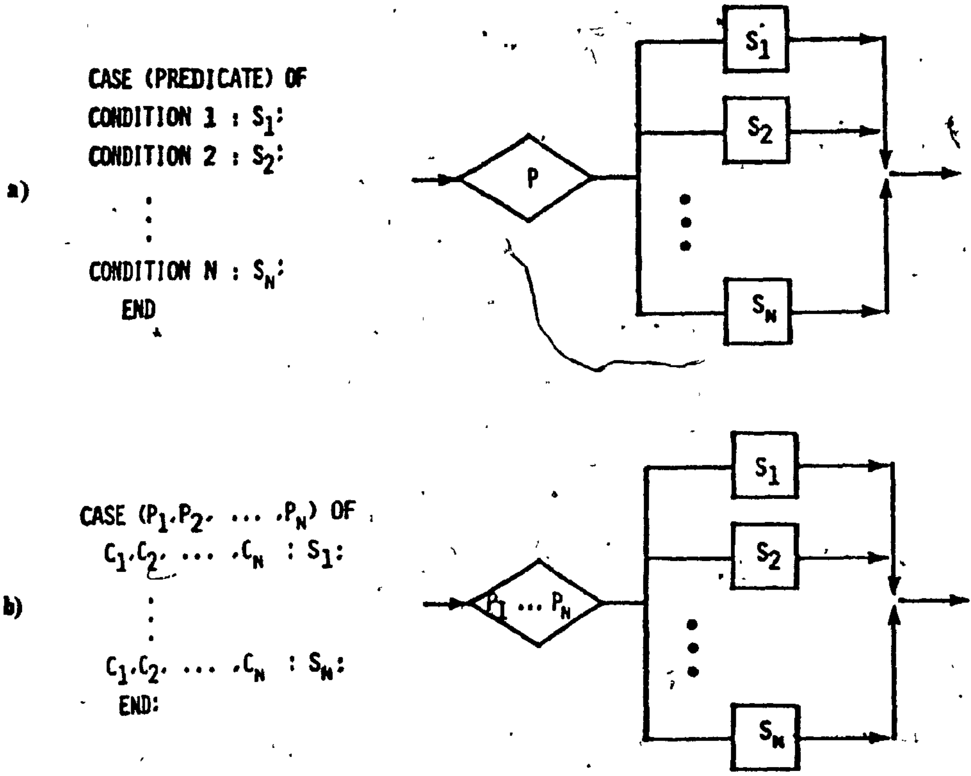
```
REPEAT
  X := X-2;
  Y := Y+3;
UNTIL (X<31)
```

FIGURE 4.3 KOMPUT emulation of the Bohm and Jacopini control constructs

Some confusion may arise concerning the sequence construct emulation given above particularly as to why each separate action (x:= x+1 and y:= x) is not prefaced by a null predicate: a predicate which is set to TRUE. The answer to this is that each block is intended to represent a series of actions. The concatenation of two series of actions yields one larger series of actions which is still represented by only one block of actions.

Another question which may have been raised concerns why the iteration construct, repeat - until, required that two KOMPUTs be used. The reason for this is that with repeat - until, the sequence of actions within the block is executed at least once before the conditional phrase is evaluated. The sequence of events illustrated in figure 4.3 for the "REPEAT"-type interation requires that the number 2 KOMPUT preceed the number 3 KOMPUT: which emulates the function of the UNTIL conditional. In a more sophisticated and realistic ABL application, the block of actions associated with the number 2 KOMPUT, could have been appended to the ordinal sequence of actions for each pathway connected to the number 3 KOMPUT.

Figure 4.4a illustrates a Pascal case statement, with its appropriate flow diagram. It is a popular [Ledg75] structured programming construct and does not result in a binary decision. An extension of the standard Pascal case statement [Ledg75] can be made so as to allow the utilization of multiple conditionals. The flowchart in figure 4.4b demonstrates that the skeletal structure of this extended case statement is unchanged from the original. Figure 4.4c illustrates a simple example of the extended case and its corresponding flow diagram. A flowchart demonstrating how it functions is illustrated in figure 4.4d. By altering the control flow so that it no longer exhibits the one-out property, as shown in figure 4.4e we realize that this is functionally equivalent to the KOMPUT construct illustrated in figure 4.1.



a)

```
CASE (PREDICATE) OF
CONDITION 1 : S1;
CONDITION 2 : S2;
      .
      .
      .
CONDITION N : SN;
   END
```

b)

```
CASE (P1,P2, ... ,PN) OF
 C1,C2, ... ,CN  : S1;
      .
      .
      .
 C1,C2, ... ,CN  : SN;
 END;
```

FIGURE 4.4a Pascal case statement with flow diagram
4.4b Extended Pascal case with flow diagram

These KOMPUT based implementations of structured programming constructs seem very awkward, however it is improper to use these emulations as "the" benchmark for the capabilities of the KOMPUT. The KOMPUT is intended for use in a different purpose other then for the elegant emulation of functionally inferior control constructs [Ledg75,Fanc76]. The preceeding arguments have presented the ABL construct and a rationale of its suitability as a programming language equivalent. Hence, it should be apparent that the ABL construct although different can be viewed as a non radical departure from standard programming language constructs.

4.2 GUARDED COMMANDS

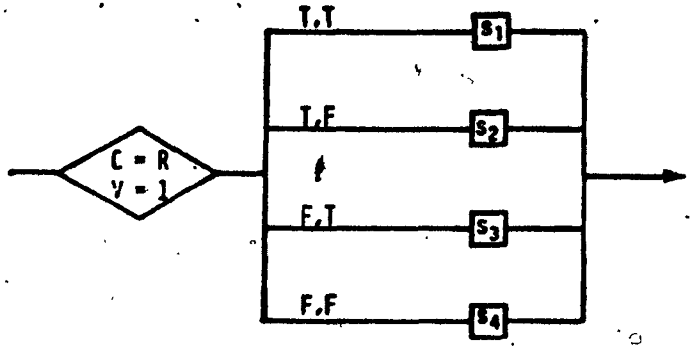
The concept of Guarded commands was put forward to provide a formal syntax of program expression which would be highly amenable to a Calculus: "axiomatic definition of programming language semantics via predicate transforms" [Dijk75]. The syntax of a subset of the Guarded Command is presented in figure 4.5.

As an example let us consider the program represented by the following three Guarded Commands:

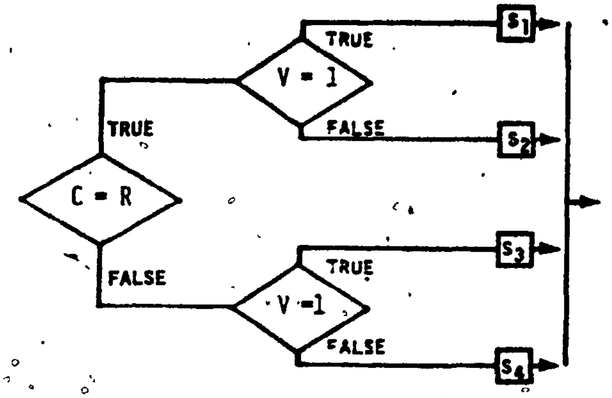
```
(1)     if A > B --> M := X; X := Y; I := I + 1; fi.
(2)     if A <= C --> M := Y; Y := X; fi.
(3)     if C > B --> Y := X; I := I + 1; fi.
```

c)

```
CASE ((C=R),(V=1)) OF
 T,T : S1;
 T,F : S2;
 F,T : S3;
 F,F : S4;
  END;
```



d)



e)

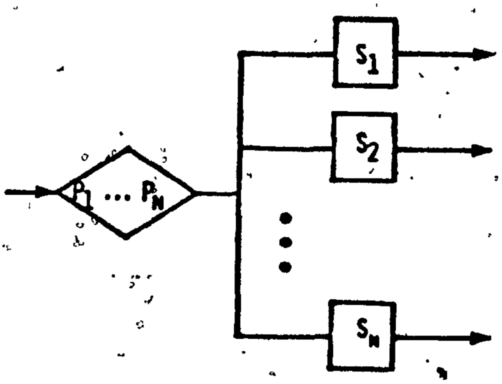


FIGURE 4.4c Example of extended case with flow diagram

4.4d Flowchart equivalent of case statement in figure 4.4c

4.4e Extended case without the "one-out" restriction

The terms "if" and "fi" is used to denote the logical bounds of a Guarded Command.

```
<GUARDED COMMAND> ::= <GUARD> <GUARD LIST>

          <GUARD> ::= BOOLEAN EXPRESSION

    <GUARDED LIST> ::= <STATEMENT> { ; <STATEMENT> }

      <STATEMENT> ::= <ASSIGNMENT STATEMENT> | <PROCEDURE
                      CALLS> | <FUNCTION CALLS>
```

FIGURE 4.5 BNF description of a subset of Dijkstra's guarded command

The function of a Guarded Command is relatively straight forward. The statement list of each Guarded Command is executed only if the appropriate guard is satisfied. The guarded list is the set of actions which follows the symbol: " --> ". Thus in command 2 if the guard, (a <=), is true then the actions: m := y ; y := x, will be executed. The same strategy applies to all other Guarded Commands.

4.2.1 TABULAR NOTATION

As an introduction to general tabular format, figure 4.6 illustrates how the three Guarded Commands given above would be expressed as a table. Figure 4.6 should not be viewed as a functional decision table, but rather generic precursor of the ABL format.

Each Guarded Command is represented by a single column in the table. Relevant predicates for each Guarded Command indicated by either a "Y" or "N" placed in the row next to

	(1)	(2)	(3)
PREDICATES			
A > B	Y	-	-
A <= C	-	Y	-
C > B	-	-	Y
ACTIONS			
M := X	1	-	-
M := Y	-	1	-
I := I+1	3	-	2
X := Y	2	-	-
Y := X	-	2	1

FIGURE 4.6 Tabular form of guarded commands

the appropriate predicate. This would also indicate whether the guard required a TRUE or a FALSE response, in order to be satisfied. A hyphen "-" indicates that the corresponding predicate is not relevant to the given Guarded Command. For example, from column 2 we can deduce that the predicates: A > B and C > A, are not relevant and that the predicate A <= B is relevant and requires a TRUE response.

The corresponding set of actions to be performed for each guard, and the sequence in which they are to be executed is indicated by the integer value in the row of the required action. A hyphen in the column opposite an action would indicate that the action is not to be performed. For example, in column 1, when the guard is satisfied, (A > B) is TRUE; then the sequence of actions to be exectued is as follows: M:=X, X:=Y and I:=I+1.

4.3 ABL TABULAR NOTATION

The ABL table form is not vastly different from the tabular notation presented in figure 4.6. It includes the addition of a matrix called CLUSTER, as well as a vector called NEXT. These four sections consisting of a CLUSTER matrix, a PREDICATE matrix, an ACTION matrix and a NEXT vector serve as a structural template for any ABL algorithm. The ABL structural template is illustrated in figure 4.7. The only quantitative limitation as to the number of permissible clusters, predicates or actions are those imposed by actual physical constraints such as page size and other machine dependant restrictions. However, it is advisable for reasons of "intellectual manageability" [Dijk72,Fros75] that ABL algorithms occupy no more than a single page.

Before describing these structures in more detail, we may profit by noting the following definitions:

Logical Structure : is the user's view of the way in which elements are perceived to be functioning.

Logical Unit : is the user's view of the way in which ALTERNATIVES should be combined to form a functional set. In other words a set of alternatives which the user believes will form a useful decision point.

	A_1	A_2	A_3	...	A_n
CLUSTER					
C_1					
C_2					
C_3					
.					
.					
.					
C_i					
PREDICATE					
P_1					
P_2					
P_3					
.					
.					
.					
P_j					
ACTIONS					
a_1					
a_2					
a_3					
.					
.					
.					
a_k					
NEXT	N_1	N_2	N_3	...	N_n

ASSUMPTION: $0 < i <= n$

Each alternative can only belong to 1 cluster.

FIGURE 4.7 ABL structural template

4.3.1 ALTERNATIVES

Each of the columns in the ABL table is referred to as an ALTERNATIVE. This term is intended to reflect a central theme behind ABL, that being, that at any given CLUSTER (decision point) we have the option to choose from several different paths, hence the term alternatives.

```
<ALTERNATIVE> ::= <CLUSTER NUMBER>  <GUARDED COMMAND>  <NEXT VALUE>
<CLUSTER NUMBER> ::= CARDINALITY OF THE CLUSTER (KOMPUT)
<NEXT VALUE> ::= CONTROL FLOW POINTER TO SUBSEQUENT CLUSTER (KOMPUT)
<GUARDED COMMAND> ::= AS PER DIJKSTRA'S GUARDED COMMAND
```

FIGURE 4.8 BNF description of the logical structure of an alternative

Figure 4.8 presents the BNF description of the "logical structure" of alternatives. From this BNF it should be quite evident, at least at the functional level, that ABL and Guarded Commands are generically related.

4.3.2 CLUSTERS

The purpose of the cluster is to deliniate which alternatives are to be used to form a "logical unit". In other words it groups together alternatives which the program designer feels will best convey a meaningful local decision. Each of these in turn will contribute to a clearer global view of the automata used to execute the required process.

The cluster matrix is used as a visual indicator of which alternatives are to be evaluated as a single unit. Each CLUSTER need not be, nor is it recommended that it be binary. Two or more predicates may be used in conjunction to express the required logic at a particular decision point.

4.3.3 NEXT CLUSTER

The effect of the NEXT vector is to connect the control flow from each alternative, to the next cluster to be executed. A small yet typical ABL table is presented in figure 4.9. This particular table is the ABL representation of a Binary Search algorithm. Special note should be made of the fact that for this example no structured documentation has been included. Thus in figure 4.9 if we had just completed the sequence of actions associated with the guard [test_value > middle] equal TRUE in cluster 3, then the subsequent decision point would be cluster: 2. As in the KOMPUT description, a next value of 0 means that we terminate execution of the current ABL Table.

4.4 ABL FLOW GRAPHS

Current flowchart techniques are ineffective for properly expressing control flow of an ABL table. However, we can borrow from discrete mathematics the concept of directed

	-PROGRAM-						-MACHINE-
C1	V	.	.	.	.	.	
C2	.	V	V	V	.	.	
C3	.	.	.	.	V	V	
P1	-	-	Y	N	-	-	P1 TEST_VALUE = LIST[MIDDLE]
P2	-	-	-	-	Y	N	P2 TEST_VALUE > LIST[MIDDLE]
P3	-	Y	N	N	-	-	P3 TOP <= BOTTOM
A1	1	.	.	.	.	.	A1 TOP := MAX
A8	.	1	.	.	.	.	A8 FOUND := FALSE
A7	.	.	1	.	.	.	A7 FOUND := TRUE
A6	.	.	.	.	1	.	A6 BOTTOM := MIDDLE
A5	.	.	.	.	.	1	A5 TOP := MIDDLE
A2	2	.	.	.	.	.	A2 BOTTOM := 1
A3	3	.	.	.	.	.	A3 READ(TEST_VALUE)
A4	4	.	.	.	2	2	A4 MIDDLE := (TOP + BOTTOM) DIV 2
NEXT	2	0	0	3	2	2	

FIGURE 4.9 ABL tabular representation of the Binary Search algorithm

flow graphs to represent ABL control flow. To create ABL directed flow graphs, the following descriptions need to be adhered too:

CLUSTER : represented as nodes

CLUSTER NUMBER : represented as a number contained within each of the depicted nodes

ALTERNATIVES : represented as directed arcs

NEXT : represented as the node to which the directed arcs connect too.

In situations where the NEXT cluster value is zero, then the arcs would connect to terminal nodes which are represented as darkened nodes. It is important to note that the basic elements of the ABL flow graph closely resemble the listed BNF syntax definition of the logical structure of alternatives. In the flow graph each alternative is represented as a node, an arc and an arrow, these correspond to the terms CLUSTER, GUARDED COMMAND and NEXT value respectively, figure 4.10.

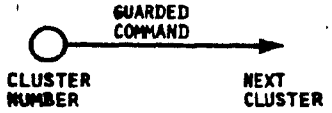


FIGURE 4.10 Flowgraph representation of an Alternative

The directed flowgraph for the binary search algorithm presented in figure 4.9 is illustrated in figure 4.11a. For purposes of comparison the skeletal structure using KOMPUTs is presented in figure 4.11b. The primary purpose of the flow graph is to provide a compact visual representation of possible control flows through an algorithm. As such, the amount of additional information which can be placed on the flow graph is somewhat arbitrary. For example in figure 13b, I have added the requisite guard elements and the corressponding ordered sequence of actions for each alternative.

4.5 ABL INTERPRETER

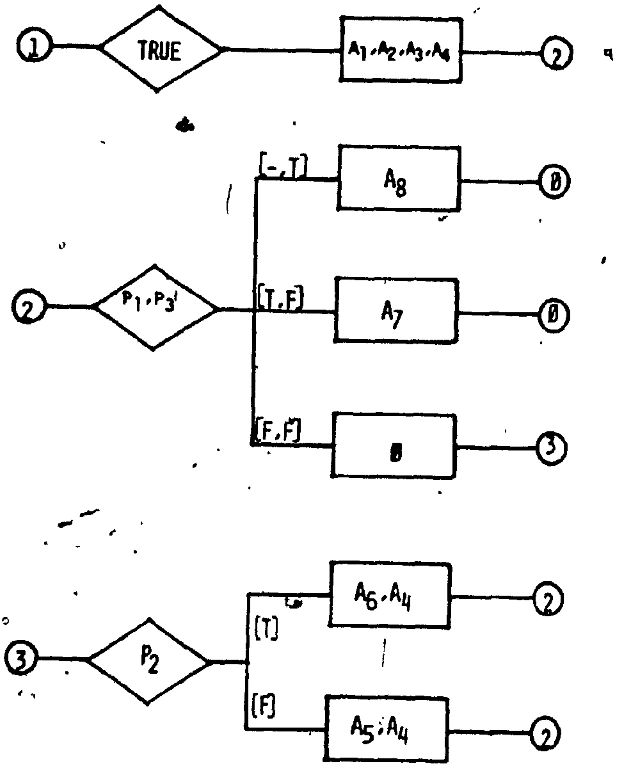


FIGURE 4.11a KOMPUT structural description of the Binary Search algorithm

1.1 INITIALIZE

2.1 TOP <= BOTTOM
"NO SUCCESS"

2.2 TEST_VALUE = LIST (MIDDLE)
"SUCCESS"

2.3 TEST_VALUE < > LIST (MIDDLF)
TOP > BOTTOM

3.1 TEST_VALUE > LIST (MIDDLE)

3.2 TEST_VALUE <= LIST (MIDDLE)

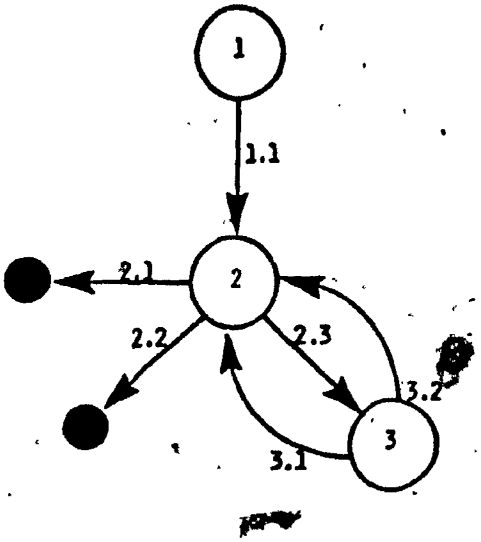


FIGURE 4.11b Flowgraph representation of Binary Search algorithm

Execution of an ABL program can be accomplished using the algorithm described in figure 4.12a and whose flow graph is illustrated in figure 4.12b. This interpreter model makes two fundamental assumptions about any ABL algorithm to be executed by it. These assumptions have been made solely for the purpose of keeping the modelled interpreter as simple as possible, not because of any deficiency in the general ABL model:

1) Each CLUSTER must be complete: no missing rules.
2) Nondeterminism is not allowed.

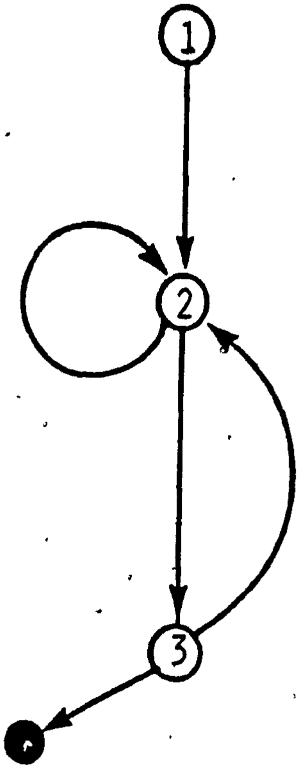
4.6 ABL PROGRAM PARTITIONS

So far we have dealt with ABL by describing and defining distinct partitions within a standard ABL table. This tabular approach, however is only one of many currently available representations of an ABL program. The intrinsic characteristics and strengths of the ABL approach are derived from an even higher order partition of an ABL PROGRAM into: ABSTRACT PROGRAM and ABSTRACT MACHINE.

An ABL Program is the functional entity created by mapping an appropriate Abstract Program unto an Abstract Machine.

```
CLUSTER 1   INITIALIZE
            1.1 GUARD IS SET TO TRUE
                -- CURRENT_CLUSTER = 1
                -- ALTERNATIVE SET = ALTERNATIVESIN CURRENT_CLUSTER
                -- PROCEED TO CLUSTER 2

CLUSTER 2   SELECT AN ALTERNATIVE FROM ALTERNATIVE SET
            2.1 GUARD CONDITIONS MATCH PREDICATE STATES
                THUS AN "OPEN GUARD"
                -- EXECUTE APPROPRIATE ACTIONS
                -- CURRENT_CLUSTER = NEXT_CLUSTER
                -- PROCEED TO CLUSTER 3
            2.2 GUARD CONDITIONS DO NOT MATCH PREDICATE
                STATES THUS "CLOSED GUARD"
                -- DELETE CURRENT_ALTERNATIVE FROM ALTERNATIVE_SET
                -- PROCEED TO CLUSTER 2

CLUSTER 3   EVALUATE THE EXECUTION STATE OF INTERPRETER
            3.1 CURRENT_CLUSTER = 0
                -- TERMINATE EXECUTION
            3.2 CURRENT_CLUSTER <> 0
                -- ALTERNATIVE SET = ALTERNATIVESIN CURRENT_CLUSTER
                -- PROCEED TO CLUSTER 2
```

(a)



(b)

FIGURE 4.12a Algorithm for an ABL interpreter

4.12b Flowgraph of the algorithm in 4.12a

An Abstract Program is defined to be the control flow part of an ABL Program, whereas an Abstract Machine is defined to be the statement part* of an ABL Program. Figure 4.13 illustrates the concept of an ABL PROGRAM using BNF notation.

* Actions and Predicates

```
<ABL PROGRAM>      ::=  <ABSTRACT PROGRAM>
                                        MAPPED ONTO
                        <ABSTRACT MACHINE>

<ABSTRACT PROGRAM> ::=  MATRIX REPRESENTATION OF CONTROL FLOW

<ABSTRACT MACHINE> ::=  SET OF <PREDICATES>
                        SET OF <ACTIONS>

<PREDICATES>       ::=  BOOLEAN EXPRESSION

<ACTIONS>          ::=  STATEMENT (AS PER DIJKSTRA)
```

FIGURE 4.13 BNF description of an ABL program

The immediate implications of this partition is that both Abstract Machines/Programs can be treated as different forms of data and as such can be manipulated with relative ease. Software based on conventional programming languages have global control flows which are determined as the sum of the implicit control flows of the language constructs used, the end result of which are extremely static and inflexible programs. This is obvious from the fact that almost any program change will alter its control flow. ABL on the other hand does not suffer from this problem because control flow is explicitly rather than implicitly defined.

A second important feature concerning the above ABL Program definition is that a number of Abstract Programs can be mapped unto a single Abstract Machine. Other consequences and advantages of the ABL program partition will become apparent as this thesis progresses.

Figure 4.14 illustrates the binary search algorithm from figure 4.9 in the form of an Abstract Machine and an Abstract Program. The Abstract Program is displayed in a compact row form which corresponds to the following convention:

CLUSTER NUMBER/PREDICATE NUMBERS/ACTION NUMBERS/NEXT CLUSTER

(a) the predicate and action sections may be left blank and are considered to be null predicate list, null action list respectively.

EXAMPLE

3/ /3,4,2/3 (null predicate list)

3/2,3/ /3 (null action list)

(b) predicate numbers may be preceeded by a negative sign which implies that a false response to that predicate is required.

EXAMPLE

2/-1,3,-4/2,1,5/0

predicate 1 and 4 must be false and predicate 3 must be true in order to satisfy the guard

4.7 ABL PHILOSOPHY

```
    PREDICATES
P1     TEST_VALUE = LIST (MIDDLE)
P2     TEST_VALUE > LIST (MIDDLE)
P3     TOP< = BOTTOM

    ACTIONS                                    1/ /1.2.3.4/2
A1     TOP := MAX                              2/3/8/0
A2     BOTTOM := MIN                           2/1.-3/7/0
A3     READ (TEST_VALUE)                       2/-1.-3/ /3
A4     MIDDLE := (TOP+BOTTOM) DIV 2            3/2/6,4/2
A5     TOP := MIDDLE                           3/-2/5.4/2
A6     BOTTOM := MIDDLE
A7     FOUND:= TRUE
A8     FOUND:= FALSE
```

FIGURE 4.14 Abstract Machine and Abstract Program of the Binary Search algorithm from figure 4.9

The definition of an ABL program as set out above is intended to be language independent and hence will accomodate both machine executable and human interpretable statements. This allows ABL programs to be written at any level of abstraction: natural languages to microcode [Lina82]. To aid in the logical construction and maintenance of ABL programs we propose a general question and answer methodology. The underlying philosophy of this methodology is to induce the user to focus his attention, increase his awareness of relevant factors and help to systematically organize important pieces of information [Ross77a,77b,Neiv79].

When designing an ABL program each cluster requires the builder to answer the following questions:

(1) Where did I come from?

(2) Why am I here?

(3) What will I do?

(4) Where will I go?

These four questions are geared primarily at the cluster and/or the alternative levels. Figure 4.15 illustrates how ABL components may be involved when answering the 4 Questions. Each question need not be answered fully before proceeding to the next, in fact they should be considered concurrently. Partial responses create a skeleton on which to build successively more complete answers. The purpose of the questions is to guide not to "force".

	ALTERNATIVE	CLUSTER
QUESTION 1 WHERE DID I COME FROM?		☆
QUESTION 2 WHY AM I HERE?	☆	☆
QUESTION 3 WHAT WILL I DO?	☆	
QUESTION 4 WHERE WILL I GO?		☆

FIGURE 4.15 Table showing the relationship of the FOUR Questions to ABL structural components

Question 1: WHERE DID I COME FROM?

This question is intended to remind the analyst/designer/programmer of the state which existed prior to the current one. This is an attempt to create an association between the two clusters and hence reestablish the environment which led to the present decision point.

Question 2: WHY AM I HERE?

This question is actually intended to answer two more specific questions: (1) What is the purpose of the current Cluster within the context of the algorithm (2) Which alternatives will I need to fulfill the requirements of the Cluster? To answer these questions the user is required to ascertain not only the usefulness of the decision point but also to adequately delinate the possible Alternatives. The latter also requires the user to determine the exact predicate(s) necessary to enforce appropriate selection of the given alternatives.

Question 3: WHAT WILL I DO?

This question is an extension and refinement of the previous question. Its purpose is to force the user to re-think his set of Alternatives and to supply the consequences (actions) which need to be executed should the appropriate guards be satisfied.

Question 4: WHERE WILL I GO?

Answering this question should cause the user to extrapolate his train of thought for each alternative to the next logical decision point.

The most obvious characteristics from among these questions is that there is a fair amount of overlap in their scope. This redundancy is intended to be the means by which to clarify and establish meaningful problem partitions. Compelling the user to constantly review and justify his reasoning helps the user overcome human limitations of short term memory by establishing Cluster and Alternative "chunks". Since these chunks are more easily handled by the human problem solver, by repetitively answering each of the questions the user can form a progressively more complete "conceptual understanding" of: the algorithm, individual clusters, as well as the alternatives within each cluster.

4.8 CHAPTER REVIEW

At the beginning of the chapter it was stated that there was reason to account for why researchers have devoted so much in both time and resources towards developing "good" computer languages. However, we are slightly hard pressed to provide justification as tc why there was so much redundancy in effort and consequently in the number of computer languages. Does the mere fact that a language can exist mean that it should? The answer is obviously no, but the multitude of available languages indicates that perhaps researchers are answering yes.

In this chapter, it was shown that ABL could duplicate all of the control constructs required of a structured programming language. With extensions such as data declarations and procedure calls ABL will implement a fully functional computer language [Lew 82]. However, now that we have built the foundation for yet another computer language, does this justify its creation? Criteria pertinent to this question will be elaborated on in the next chapter.

Over a decade after the computing community realized that it was in the midst of a software crisis, we must concede that it still exists. The most publicized attempt to curb the symptoms of the software crisis has been through the use of the Structured Programming Methodology. Although much effort has gone into creating "the" structured programming language none has yet proved itself to be outstanding. In fact there is as yet no general concensus as to the fundamental set of control constructs which a language is expected to possess. It is not sufficient to assume that language has been the sole factor responsible for the software crisis, hence it is not sufficient to propose merely a new language. Thus with this in mind the following chapter will critically address the issue of software production from the general perspective of methodological deficiencies.

CHAPTER 5

AN ABL APPROACH TO SOFTWARE TECHNOLOGY

The success of both the Structured Programming Methodology and the Software Lifecyle Methodology can be attributed, at least in part to the fact that they acknowledged man's individualistic and multifaceted approach to programming. Researchers have attempted to curb problems related to and arising from these human characteristics by imposing restrictions on the way software was produced and maintained. It was hoped that these techniques would introduce consistency to programming style thereby also increase productivity levels. Unfortunately, although the rate of software production and software quality are both significantly higher then they were [Boeh73], they have not yet attained levels which can be considered adequate [Boeh79,Bran81].

An important and surprising finding is the fact that even though experimental observation has shown Structured Programming Methodology to be relatively effective, it has not been readily accepted by industry [Holt77]. Holton states that this may be related to the more significant problem of morale; when Structured Programming Methodology is used productivity usually goes up but morale does not.

This finding seems to indicate that perhaps not enough time and effort has yet been expended on humanistic aspects of programming [Wein71]. This latter phrase is intended to encompass those attributes which are essential to human problem solving skills (with which we can create algorithms) as well as for the subsequent transformation of these algorithms into viable pieces of computer software.

This chapter will study three rather broad areas of current programming systems which researchers have pointed out as being deficient: tools, documentation and software development environments. Particular interest will be placed on those factors which deter from human programming needs.

5.1 SOFTWARE TOOLS

The introduction of software methodologies prompted strong interest in developing tools to aid software personnel at each of the different phases of the software lifecycle. Laubber [Laub82] lists the following general criteria for defining software tools.

A software tool should:

(1) make use of the computer to design/develop/maintain software

(2) aid communication among and between the different

groups involved in a software project (users/managers/technical staff)

(3) facilitate understanding and/or use of the software system

(4) facilitate quality assurance/control as well as promote project management.

Software tools can then be further classified into the following three types:

COGNITIVE TOOLS

The purpose of this type of tool is to provide problem solving support by enhancing the intellectual capabilities of the developers. This would include any technique which allows for a more effecient transition of problems to algorithms. The more important areas amenable to cognitive tools were discussed in the chapter 3: organization, focus, methodology.

AUGMENTATIVE TOOLS

These are used to augment the efficiency of the developer. They allow users to work faster and more efficiently by automating tedious activities. Examples of Augmentative tools would include tools such as automatic documentation generators, optimizing compilers, automated program proofs, symbolic execution, etc.

NOTATIONAL TOOLS

These are "languages and other means of expressing information evolving during development" [Laub82]. Notational tools have numerous applications and are widely used throughout the information processing spectrum. These tools can vary from the informal, imprecise and ambiguous (natural languages) to very strict formal and precise notation (programming languages).

INFORMAL -------------------- FORMAL
(for humans) (for machines)

The above categories are not mutually exclusive, the introduction of any new tool will be expected to have numerous repercussions through each of the above classes. For example, a new notational tool may enhance intellectual capabilities while at the same time be ineffective as an augmentative tool.

5.1.1 NON INTEGRATED TOOLS

The number of tools has grown quickly, especially with the introduction of interactive computing and the software lifecycle [Howd82]. This has created a situation similar to that which occurred with the introduction of high level languages. There exists a large collection of tools but most of which are only locally successful: specific to the environment for which the tool was generated [Barn80]. Reasons for failure include the fact that most tools are either not portable, provide only modest results, are incompatible or are simply poorly engineered for humans (not user friendly) [Evan82].

Researchers have assumed, perhaps a bit naively, that given adequate tools data processing personnel would use them extensively. This subsumes a rather idealistic view of human nature, wherein people will go out of their way to find the "best" solution. In reality while a myriad of different tools exist only a few are ever used extensively let alone synergistically.

5.1.2 INTEGRATED TOOLS

Software designers have collectively been guilty of overlooking one very basic and important factor in the psychology of human behavior, that being : any shaped response will extinguish if insufficient reinforcement is presented. If use of a given software tool (shaped response) requires a great deal of overhead for little gain (reinforcement) then the user's enthusiasm for using the technique will quickly ebb away (extinguish).

In order to decrease this overhead computer scientists have recently proposed a new approach, known as "programming environments", to complement current software development methodologies [Bran81]. This is an attempt to increase enthusiasm and promote more effective tool usage by decreasing the overhead for their use. The core of this approach consists of an integrated tools system, which can be further subdivided into the Toolbox and the Development Support System (DSS) approaches. "A development support system is a collection of individual tools appropriately interfaced, with a user front end and an underlying database. With a toolbox, the ensemble of tools, the tool application and the tool output must be more directly managed by the user" [Bran81].

The toolbox is more easily assembled but is considered inferior to the DSS. The problem with a simple collection of tools is that it places a great deal of responsibility on the user to properly coordinate tool usage. The toolbox approaches also suffer from the problem of inflexible development methodology. In other words, the tool environment may impose a fixed and complex structure of interconnected tools, allowing for the possibility to create convoluted protocols such as: "to use tool A, you must first use tool B, and the output from A is always processed by C and D unless there is input from E" [Howd82]. This rigidity constrains the user and will likely deter him from making further use of those tools which are merely beneficial and not absolutely essential.

The DSS on the other hand provides for a more flexible and less constricted environment, thus being more conducive to extensive tool usage. The most important feature of the DSS is the fact that all tools and facilities work on a common data object. This provides a medium by which to interface tools without forcing them into complex interrelationships. Use of this database approach also reduces the problems of inconsistencies and synchronizations which can occur as a result of needing multiple copies of the same data for use with different tools.

5.1.3 ABL/DSS AN INTEGRATED TOOLS APPROACH

The basic features of ABL: sufficient control construct, language independence and tabular template are flexible enough to accomodate the requirements of a development support system. Figure 5.1 illustrates the general configuration of a hypothetical ABL/DSS. This system would provide a set of tools for generating, manipulating and integrating both the algorithm process and its computational descriptions. The assumption made is that access to both tools and data is properly and adequately regulated.

A Relational data base approach seems to be a natural data model for this system. Earlier it was stated that the tabular format is what underlies ABL's functional characteristics. The Relational data model does not detract

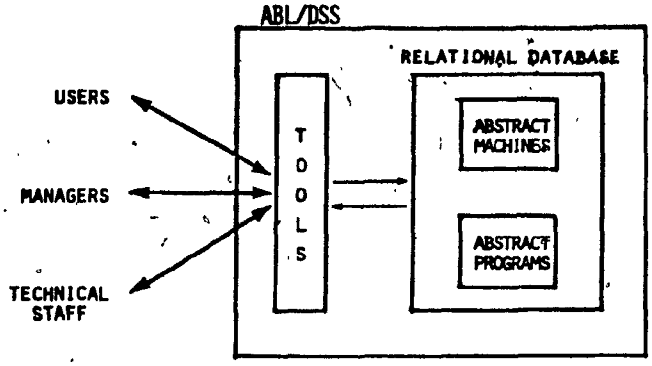
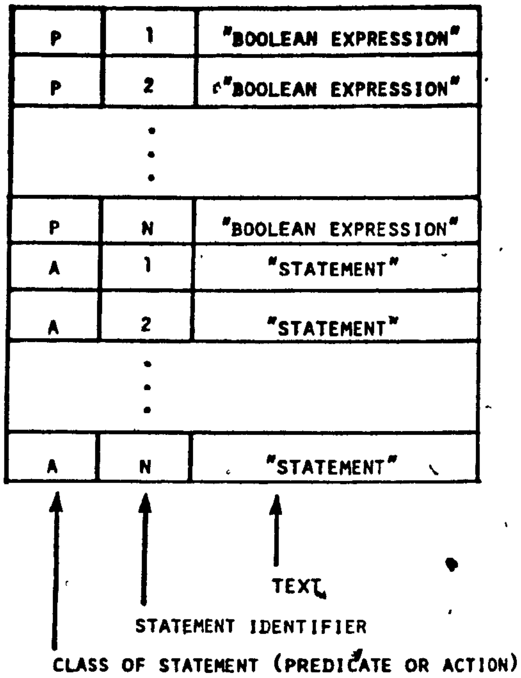


FIGURE 5.1 Configuration of Hypothetical ABL/DSS

but rather complements these features. The reason for this is that the entities and associations in the Relational database are represented in a single uniform manner, namely in the form of tables [Date77].

The data base would act primarily as a repository of Abstract Programs and Abstract Machines. The data dictionary would keep track of allowable mappings between these individual entities. Plausible relational schemas for the Abstract Program/Machine are as follows:

ABSTRACT MACHINE RELATION: Each tuple (row) in the relation illustrated in figure 5.2 is equivalent to an atomic element from the Abstract Machine. The attributes (columns) of the relation consist of the following: class of statement ('A' for action, 'P' for predicate), statement number (integer representing cardinality of the statement) and a program statement (as defined in chapter 4).



P	1	"BOOLEAN EXPRESSION"
P	2	"BOOLEAN EXPRESSION"
	. . .	
P	N	"BOOLEAN EXPRESSION"
A	1	"STATEMENT"
A	2	"STATEMENT"
	. . .	
A	N	"STATEMENT"

FIGURE 5.2 Relation Abstract Machine

ABSTRACT PROGRAM RELATION: Each tuple in the relation illustrated in figure 5.3 corresponds to an alternative in the abstract program. The tuple is defined as consisting of four attributes: cluster number (cardinality of the cluster group), predicate list (string representing the ordinal sequence of predicates to be evaluated), action list (string representing the ordinal sequence of actions to be evaluated), next (the ordinal value of the subsequent cluster to be evaluated).

Conventional "programming environment" proposals

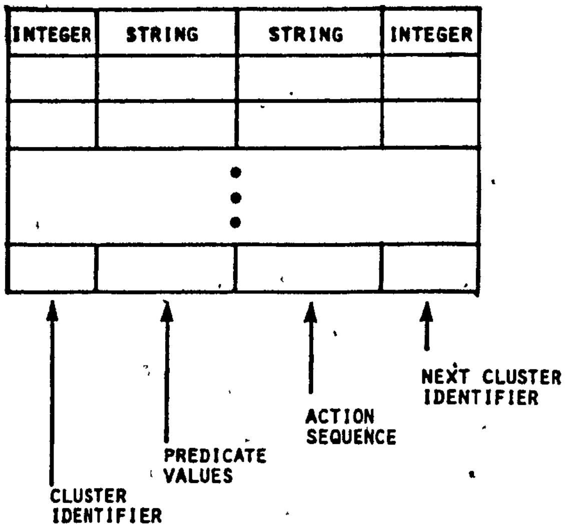


FIGURE 5.3 Relation Abstract Program

[Lamb82,Bran81,Howd82] view the data base as a unifying medium through which to coordinate tool usage and provide inventory control of available software products. The data base facilities themselves are not used extensively as direct development and maintenance tools. ABL/DSS does not stop at this "classical" perception of the function of a database facility.

The ABL approach allows us to utilize the data base utilities in a relatively new context. This context reflects the component structure of the contents of data base store. With conventional programming languages the data base would have to work at the level of non divisible entities such as modules. For example, typical storage and manipulation operations such as INSERT, DELETE, RETRIEVE and

UPDATE would not actually be concerned with the contents (source code/text) of the data base element being manipulated. If the user wanted to change a module external tools would have to be involved to perform the actual editing.

With ABL/DSS stored software products are not static entities. Individual modules are decomposed into units which can more effectively utilize the data base utilities. For example changing code/documentation for an ABL program can be accomplished directly by retrieving the appropriate relations and then using the update facility to modify the relevant tuples.

Higher level manipulative operators based on Relational Algebra can also be used as effecient techniques for dynamic reorganization of information at the program level [Codd82]. This can be extremely useful for increasing comprehension+ as well as for testing and reliability. The report generation features of the data base system may also be used as additional documentation tools.

Advantages of the proposed ABL/DSS system over other current systems stem primarily from the fact that it allows more effective usage of the data base utilities. Thereby

----

+ Discussed later in the chapter.

minimizing the redundancy and complexity of any additional software tools. Since this environment offers a potentially less demanding milieu it should also decrease fear of failure. These technical and psychological factors should provide early acceptance of the system and ensure against disenchantment with the DSS system.

5.2 PROGRAM INFORMATION

A program can be regarded as an element which must satisfy the needs of three different environments (COMPUTER/READER/WRITER), figure 5.4 illustrates this relationship [Trac79]. Of these the most easily satisfied are the COMPUTER requirements; programs must be machine exectuable/understandable. While efficiency of execution ("fast algorithm") can be thought of as a machine requirement, efficiency may be more accurately viewed as a management requirement or as a parameter in a cost benefit analysis.

The diverse and variant needs of the READER and WRITER are by far more challanging and difficult to satisfy. For the READER the basic exigent is to minimize the amount of effort needed to assimilate an adequate understanding of the solution. While the WRITER's goal is to minimize the "cognitive stress" [Shne75] incurred while attempting to

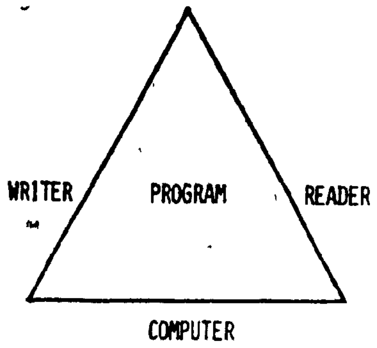


FIGURE 5.4 A program's environmental relationship

convey an adequate solution. The fundamental difference between these two is that the former wishes to understand while the latter is attempting to communicate.

These perspectives enforce a certain modality of communication, more specifically they stereotype the WRITER as being active and the READER as passive. This assumption promotes the impression that cogent understanding is primarily dependant on the WRITER. Further complicating factors include the fact that not only must the WRITER cater to the computer he also has to accommodate different classes of READERS (users/managers/technical staff). It should be obvious that this unidirectional flow of communication imposes severe constraints which likely handicap both the WRITER's and the READER's performance.

5.2.1 LANGUAGE CONSTRAINTS

In a previous chapter the process of creating a program was described as consisting of two distinct stages. The purpose of the problem solving stage (ALPHA) was to discover a solution to a given problem with preference going to the simplest and/or the easiest model of the solution. The second stage (BETA) involved the transformation of the given algorithm into a computer program. In practical situations however the distinctive characteristics of these two phases is often blurred by technical coding considerations.

The most visible and controversial of these technicalities involves the very domain of programming languages themselves. The problem, which is of merit in itself, stems from the fact that "the structure of a programming language affects the way a programmer using it thinks about algorithms" [Shaw80]. A related difficulty with similar arguments can be seen concerning the use of certain design techniques. Both programming languages and design techniques themselves involve the introduction of new and irrelevant yet redundant constraints to the actual problem situation. Such technicalities hamper problem solving and usually produce nonoptimal results because the solution has to massaged to fit within the limits of these superfluous constraints.

5.2.2 RESTRICTED VIEWS

A highly touted characteristic of structured control constructs was that they lent themselves well to mathematical proofs. In fact "quality" was intended to be a function of the correctness of a program, as derived from this mathematical approach to program validation [Dijk76,Dijk79]. Simpler (restricted) control paths along with appropriate blocking made it feasible, albeit nontrivial, to construct these mathematical proofs of programs. Program proofs would eliminate the need for program testing because this technique would allow for a complete check of a program's validity. These control structures also offered hope for an automated process.

Several researchers reasoned [Dijk72,Hoar69] that simply requiring programmers to write a proof of termination of a loop before actually coding it would save considerable debugging time. It was argued that this gain would be achieved even without the aid of automated program proofs because programmers would gain a more intimate understanding of their proposed solution and hence be less prone to making thoughtless errors.

The following is an example of how these authors believed proofs should be written and the manner in which programmers should approach problems [Shie82]. The algorithm to be formulated is intended to solve the following:.

PROBLEM: given a set of N positive numbers contained within a vector, A, find the largest value, M.

SOLUTION: "start by formulating a loop invariant. For this loop the appropriate invariant is that, at the end of the Kth pass through the loop, M >= A sub j for each j, 1 through K. Once formulated, that invariant can be proved by induction. Having been proved by induction it can be instantiated for K = N and now constitutes a proof that M is a maximum of the set, which is the desired result".

These authors suggest that by treating programming as a branch of mathematics programmers can make use of well established mathematical properties to produce "better" programs. The problem is that nobody does it this way, "it takes too long and is far too complicated" [Shie82].

Program proofs may yield quality software, but at excessive and perhaps exorbitant costs. Reasons for this includes the fact that the majority of programmers do not have adequate mathematical training. This approach may be invaluable for important projects with high safety or economic considerations, but social factors [DeMi77] are not likely to make this methodology a "product for the masses".

Programming is not mathematics, nor should it be [Wein71].

Higher level programming systems must emphasize the use of descriptive languages for communication. "The concentration must be on those aspects which aid in giving a person a clear overall understanding rather than on those aspects which increase the mathematical tractability of the descriptions" [Wino77].

5.2.3 REPRESENTATIONAL DISTORTION

One of the more important goals of any programming system should be to diminish misunderstanding and facilitate communication [Wino77]. To do so requires minimizing the amount of representational distortion incurred because of the communication media. Representational distortion is defined as follows: objects are changed by the medium through which they are viewed. Thus minimization of distortion requires that the logical structure of the program reflect directly the structure of the problem [Fitt79].

Structured programs are fundamentally tree like structures [Chap78,Turn80]. These are theoretically easier to handle (by humans) then networks and unrestricted latice-like structures. However, as Knuth [Knut79] remarks it is often hard to believe that the serial string of alpha numeric characters that make up these programs are in fact trees. Indeed a fair amount of processing is required to be able to extract this supposedly useful treelike property of program structure. When reading strings, our visual processing capabilities are reduced to operating at a very rudimentary, one-dimensional capacity. Making more effective use of innate visual processing capabilities would increase comprehension and decrease workload.

An appropriate analogy might be to look at any natural language text. The information contained in a text , can still be had even if it is presented as a linear string, however by the addition of visual cues such as formatting into paragraphs, pages and chapters we reduce the amount of processing that needs to be performed. Thus we have not only convenience but also increased understanding.

This concept of data reduction was and continues to be the motivating force behind crude attempts to add on an extra level of dimensionality to program text. This is done via pretty printers, which stagger text across page according to blocking level. Experimental investigation of the effects of pretty printing on both maintenance and debugging tasks have shown small but statistically significant increases in programmer performance [Sayw81,Shep81] .

Flowcharts were also intended to serve as visual aids to program understanding and development but unfortunately experimental studies have not yet been able to confirm this hypothesis [Curt81,Fitt79]. Several researchers [Yode78,Orr 79] have stated that flowcharts are not as effective as they should be because they are too distant in form from current software programming techniques: conventional flowcharts have not reflected the change to structured programming concepts. Numerous hand drawn and automated structured flowcharting techniques (Nassi/Shneiderman chart, PAD, Warnier Orr diagrams FSM [Salt76], DD paths [Paig75], MTR [DeBa78] have been introduced but have yet to be thoroughly tested for effectiveness.

Unfortunately, in spite of potential for improving programmer performance none of these adjuncts is gaining acceptance in the marketplace. The problem may be that hand drawn methods are much too tedious to create and maintain, while the automated techniques are primarily post hoc aids which are usually cumbersome to use (restrictive).

5.3 DOCUMENTATION

Documentation is a vital and necessary aspect of the software production and maintenance process [Wils81]. Its purpose is to provide information which can facilitate understanding and/or illustrate the appropriate use of a piece of software. The classical and solely structural definition of documentation includes: internal program documentation (comments within a program) as well as user guides and technical manuals [Gray82]. With the acceptance of Structured Programming Methodology and the Software Lifecycle Model the above definition has grown to include documentation at other stages of software development, such as requirements specification and design documents.

5.3.1 DOCUMENTATION DEFICIENCIES

The problem is that current documentation practices do not yield products which are effective communication aids. The documentation generated is usually "notoriously ambiguous, verbose and redundant" [Your82]. The large number of inconsistencies and errors which always seem to crop up in documentation have prompted some [DeMi79] to proclaim that the only truly accurate documentation is the code itself.

As a development aid it does not fare any better. This is probably due to the ad hoc and aposteriori manner in which it is prepared. Classically documentation is produced after the fact [Horn75] and hence is "not formally recorded until the end of the project, by which time it is of historical significance only" [DeMi79]. This ineffectiveness is reflected by situations such as where users will prefer to have someone explain the system, rather than reading the appropriate manuals. A similar instance would be when users do read the manuals yet still find themselves perplexed as to how a program or package is supposed to work and what exactly it is supposed to do [Wils81]. Even worse are the problems encountered when a systems analyst does not clearly communicate what it is that the user wants or the case of a programmer who must implement an algorithm using vague design documents [Teic77].

Other difficulties arise from the fact that each stage of the Software Lifecycle has its own form of documentation. Therefore people will generally not be as proficient in handling documentation from other stages as they would with their own. Such inconsistencies allow error and ambiguity to easily propagate from one stage of the Software Lifecycle to the next.

Some researchers [DeMa79,Your82] have suggested that manual production and editing are among probable causes of bad documentation. People may simply be reluctant to write documentation when they know that it will continually change during the course of production. Automatic methods of documentation would alleviate the wide variance in documnetation style as well as reducing inconsistencies due to documentation efforts which did not keep up with code, design and requirements changes.

Another part of the problem may lie with the attitude that many professionals seem to have towards writing documentation. It is produced in an ad hoc manner, withthe rationalization that "we do it because we have to" [DeMa79]. Others simply avoid the task by proclaiming that documentation is "beneath them" [Wils81]. This lack of motivation and enthusiasm is certainly a major contributing factor in the creation of documentation which is of poor quality and of questionable value. Perhaps people do not enjoy writing documentation because they lack confidence in their ability to produce it. Their attitude may simply be a function of the fact that they may not know where, or even how much, or at what descriptive level to begin writing documentation. These deficiencies in documentation are recognized as serious drawbacks to the software development and support environment.

5.3.2 ABL DOCUMENTATION

How does the ABL methodology help to alleviate problems associated with documentation? It does so in a three fold manner: consistency, modularity and maintainability.

5.3.2a CONSISTENCY

ABL/DSS provides a powerful representation and storage structure which can be used ubiquitiously throughout the software lifecycle. In that ABL is language independent, all system documentation can actually be written and represented as ABL programs. The abstract machines in such cases would of course be comprised of natural language statements. The ABL format provides a flexible standard which can satisfy specific needs of each stage of the software lifecycle. This syntatic and semantic consistency should improve information access between groups throughout the stages of the software lifecycle via a common database.

5.3.2b MODULAR BUILDING BLOCKS

Internal source code documentation has been classified into two basic functional types: (1) high level procedural descriptions; these will normally proceed a given procedure and highlight salient features of the algorithm and attempt to explain the processes involved, (2) low level detailed comments; these are usually found scattered throughout the source code and describe local "noteworthy" characteristics of the program. It has been reported [Wood81,Duns78] that generally the more effective of these two types of source code documentation is the high level comment. The utility of detail level comments may have been severely underestimated due to the multitude of problems associated with its judicious application. ABL adopts a very conservative stance and easily accommodates both.

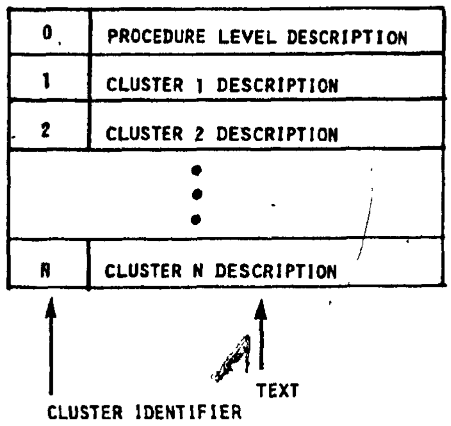
ABL's intrinsic structure reduces the problem of deciding WHERE, HOW MUCH and at WHAT LEVEL to write documentation into the source code. The ABL format allows documentation to be clearly and succintly mapped unto specific structural components of an algorithm. ABL programs allow for three different levels of documentation for source code.

> PROGRAM LEVEL description is the highest level and is equivalent to a procedural description. As in conventional programming language techniques it preceeds and describes the program.

> CLUSTER LEVEL decription involves describing the logical units within the programs. As defined earlier logical units are intended to be meaningful decision points. Documentation at this level should describe the purpose ("WHY") of the decision point.

> ALTERNATIVE LEVEL descriptions define each of the alternatives within a program. These descriptions concentrate on describing the effect of executing the actions associated with each alternative.

These latter two levels taken together should illustrate the relationship between any two clusters or between a cluster and termination. Storage of this documentation using the Relational data model is also a relatively trivial affair. Higher level documentation such as procedural or cluster level description can be stored in a relation seperate from either the Abstract Machine or program relations. The structure of the Relation ABSTRACT DOCUMENTATION, is illustrated in Figure 5.5. Lower level documentation, at the Alternative level, can be stored as an extension to the Abstract Program Relation as shown in figure 5.6. This can be best understood by considering the fact that since each tuple in the relation already represents an Alternative, the Alternative description can be added as an equivalent attribute of the tuple.



0	PROCEDURE LEVEL DESCRIPTION
1	CLUSTER 1 DESCRIPTION
2	CLUSTER 2 DESCRIPTION
• • •	
N	CLUSTER N DESCRIPTION

FIGURE 5.5 Relation Abstract Documentation

At higher levels of system design where the elements of the abstract machine consists of modules rather than statements, another level of documentation can be added. The abstract machine can be replaced by an equivalent abstract machine which contains descriptions of the modules rather than just the module name. This process effectively provides an ACTION LEVEL description. When combined with the appropriate abstract program this description creates a fourth level of documentation.

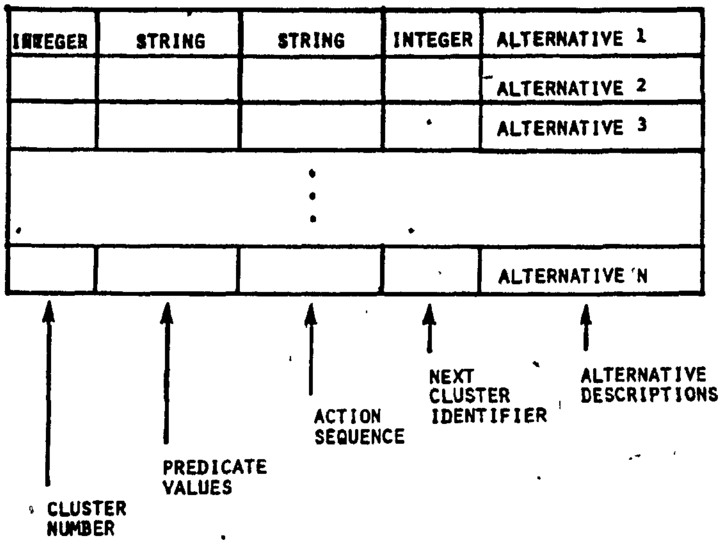


FIGURE 5.6 Relation Abstract Program with Documentation

Another major advantage of actually having these specific levels of documentation means that we have improved locality of reference. Changes in a program no matter how frequent, "minor" or "important" can be easily and adequately recorded. Thus ABL should help reduce the number of inconsistencies between "what the code does" and "what the documentation says it does".

5.3.2c AUTOMATED DOCUMENTATION

It provides automated aid for documentation generation, both textual and graphical. Unlike most conventional automated systems [Horn75,Arch82] these documents are not extracted via preprocessors from the body of the software products. Rather these documents are built up from the individual components described above. Thus not only can we generate numerous types and forms of documentation we are also able to actively manipulate these various levels of documentation. Figure 5.7 illustrates three of the available forms of documentation.

```
ABL-ROUTINE  BINARY_SEARCH;

1.  1. INITIALIZE FOR BINARY SEARCH ALGORITHM
        <  2 >

      1.1P  NO CONDITIONS

          1.1INITIALIZED SEARCH TO MIDDLE ELEMENT IN ARRAY        ----> 2
              1.1A TOP := MAX
              1.1A BOTTOM := 1
              1.1A READ(TEST_VALUE)
              1.1A MIDDLE := (TOP + BOTTOM) DIV 2
2.  2. DETERMINE STATUS OF SEARCH
        <  0, 0, 3 >

      2.1P TOP <= BOTTOM
          2.1ENTIRE ARRAY HAS BEEN SEARCHED AND ELEMENT IS NOT FOUND        ----> 0
              2.1A FOUND := FALSE


1)    2.2P TEST_VALUE = MIDDLE  AND
           NOT TOP <= BOTTOM
          2.2ELEMENT HAS BEEN FOUND THEREFORE TERMINATE SEARCH        ----> 0
              2.2A FOUND := TRUE


      2.3P  NOT TEST_VALUE = MIDDLE  AND
           NOT TOP <= BOTTOM
          2.3NOT FINISHED SEARCHING AND NOT YET FOUND        ----> 3
3.  3. DETERMINE WHICH HALF OF THE UNTESTED ARRARY IS TO BE ELIMINATED
        <  2, 2 >

      3.1P TEST_VALUE > MIDDLE
          3.1CONTINUE SEARCH IN TOP HALF OF CURRENT PARTITION        ----> 2
              3.1A BOTTOM := MIDDLE
              3.1A MIDDLE := (TOP + BOTTOM) DIV 2


      3.2P  NOT TEST_VALUE > MIDDLE
          3.2CONTINUE SEARCH IN LOWER HALF OF CURRENT PARTITION        ----> 2
              3.2A TOP := MIDDLE
              3.2A MIDDLE := (TOP + BOTTOM) DIV 2
```

ABL-ROUTINE BINARY_SEARCH;

THE PURPOSE OF THIS PROCEDURE IS TO DETERMINE IF SOME ARBITRARY TEST VALUE IS ALREADY PART OF A SORTED LIST OF NUMBERS. IF THE VALUE EXISTS THE PROCEDURE WILL SET A FLAG (FOUND) TO TRUE ELSE TO FALSE.

THE APPROACH USED TO PERFORM THE SEARCH IS CALLED THE BINARY SEARCH ALGORITHM. IN EFFECT THIS APPROACH ALLOWS US TO ELIMINATE HALF OF THE REMAINING UNSEARCHED LIST OF ELEMENTS WITH EVERY COMPARISON OF THE GIVEN TEST_VALUE TO A SELECTED LIST ELEMENT.

```
=================================================================
         CLUSTERS-
1.0 INITIALIZE FOR BINARY SEARCH ALGORITHM

  1.1 INITIALIZED SEARCH TO MIDDLE ELEMENT IN ARRAY
```

2)

```
2.0 DETERMINE STATUS OF SEARCH

  2.1 ENTIRE ARRAY HAS BEEN SEARCHED AND ELEMENT IS NOT FOUND
  2.2 ELEMENT HAS BEEN FOUND THEREFORE TERMINATE SEARCH
  2.3 NOT FINISHED SEARCHING AND NOT YET FOUND


3.0 DETERMINE WHICH HALF OF THE UNTESTED ARRARY IS TO BE ELIMINATED

  3.1 CONTINUE SEARCH IN TOP HALF OF CURRENT PARTITION
  3.2 CONTINUE SEARCH IN LOWER HALF OF CURRENT PARTITION
```

ALTERNATIVE SET

1	INITIALIZED SEARCH TO MIDDLE ELEMENT IN ARRAY	X					
2	ENTIRE ARRAY HAS BEEN SEARCHED AND ELEMENT IS NOT FOUND		X				
3	ELEMENT HAS BEEN FOUND THEREFORE TERMINATE SEARCH			X			
4	NOT FINISHED SEARCHING AND NOT YET FOUND				X		
5	CONTINUE SEARCH IN TOP HALF OF CURRENT PARTITION					X	
6	CONTINUE SEARCH IN LOWER HALF OF CURRENT PARTITION						X

ACTION SET

3)

	1	TOP := MAX	1					
	2	FOUND := FALSE		1				
	3	FOUND := TRUE			1			
	4	BOTTOM := MIDDLE					1	
	5	TOP := MIDDLE						1
	6	BOTTOM := 1	2					
	7	READ(TEST_VALUE)	3					
	8	MIDDLE := (TOP + BOTTOM) DIV 2	4				2	2

FIGURE 5.7 Three forms of program documentation for the Binary Search algorithm

The volume of documentation may seem excessive, but it must be remembered that the amount which is actually seen is determined by the user: all, some, or none of it. This is extremely important since documentation is "good" only if it is useful to the individual reading it.

5.4 COMMUNICATION USING ABL/DSS

The ABL approach and the relational data model each help support the most crucial measure of communication, namely comprehensibility. Some of the more pertinent aspects of comprehensibility are listed below [Date77,Codd82,Barn80].

(1) The number of basic constructs are few. In fact the relational data base is built out of one single construct, the relation (table) which is both simple and highly familiar. Similarily the ABL approach is also based on its single control construct, the Komput. Each Komput is represented as a table within the larger table of the program.

(2) Distinct concepts are clearly separated. The relational model provides an optimal medium with which to separate distinct concepts. The ABL methodology is

also based on clear and precise building blocks. In fact it even allows for different levels of building blocks: the program is a set of clusters which in turn is a set of alternatives. The four questions which describe the ABL philosophy enforce and demand distinct seperations.

(3) Symmetry is preserved. "It should not be necessary to present a naturally symmetric structure in an asymmetric manner" [Date77]. Both the relational model and the ABL approach are flexible enough to accomodate most logical structures.

(4) Redundancy is carefully controlled. The data base allows a normalization procedure which can ensure that the same fact will not appear more than once. One purpose of the ABL abstract machine is to make available (with minimal redundancy) all the different actions and predicates.

These characteristics enhance communication for both WRITER and READER and are generally effective through all of the stages of the lifecycle. As a WRITER the benefits of the ABL technique and the DSS tools approach might be summarized as follows: ABL/DSS provides both tools and a notation which are consistent throughout the software lifecycle.

REQUIREMENTS DEFINTION AND SPECIFICATIONS

Decision table techniques which are generic precurssors of ABL. Decision tables were originally developed for systems analysis as an adjunct to natural language text [Maes78,Mont74]. They help to isolate and organize relevant pieces of information [McDa80,Cant71,Wart79].

DESIGN

It supports the design process because it can be used to illustrate and store various levels of abstraction and refinement (see section 4.7 ABL Design Philosophy). Translation from most other design methods (graphs, pseudocode, flowchart) into ABL notation is also quite trivial.

IMPLEMENTATION/MAINTENANCE

It was shown in an earlier chapter that ABL has the necessary and sufficient constructs to allow for a fully implementable language. ABL has also successfully been used to replace the control flow constructs of other languages [Hint81,Fanc76]. Relevant data base utilities were shown to complement the implementation and maintenance process.

TESTING, DEBUGGING and VALIDATION

This topic will be discussed in the next chapter.

For the READER the biggest advantage of ABL/DSS is that it safely enlivens communication by permitting the READER to take a more active role in the exchange of information. Rather than being shackled to the representation chosen by the WRITER, the READER is given the opportunity to interactively seek out a more suitable representation.

"Since the understanding of a component comes from having multiple viewpoints, no single organization of the information on a printed page will be adequate. The programmer needs to be able to reorganize the information dynamically, looking from one view and then another, going from great generality down to specific detail, and maneuvering around in the space of descriptions to view the interconnections" [Wino77]. Three points stand out from Winograd's statement: (1) going from generality to specific detail, VERTICAL MOVEMENT in a top down design, (2) allowing for multiple views of the same information, LATERAL MOVEMENT, and (3) allowing for DYNAMIC REORGANIZATION. In the next three sections of this chapter we shall discuss how the ABL/DSS environment acknowledges and exploits these three requirements of the human problem solver.

5.4.1 VERTICAL MOVEMENT

Vertical movement involves being allowed to navigate through different levels of software abstractions. This is easily accomplished by ABL/DSS through its fundamental property of language independence, which allows it to treat all abstract machines in a uniform manner. The contents of an ABL Abstract Machine, whether natural language text or computer code, is of no consequence to the DSS. To go from a rudimentary design specification to a functional implementation simply requires that the data base sublanguage retrieve the appropriate Abstract Machine and Program. Since ABL/DSS only uses one basic data structure we therefore create a pseudo-standard for all phases of the software lifecycle. The additional benefit would be that since everyone is communicating and working with the same conceptual data model it would reduce the amount of confusion which currently plagues people who are not using the same methods.

A previously stated premise was that the use of abstraction as a problem solving tool involved the suppression of superfluous detail. This however creates a dilemma "abstraction is in the eye of the beholder" [Pgn 79]. What one person may consider extraneous "too many details to be useful" may be absolutely essential for someone else, "not enough detail to be useful". The flexibility afforded the ABL/DSS system enhances personal communication and productivity, as well as that of the systems group. Each member of a team can consider a given problem at whatever level of detail the professional feels most comfortable with. This is of vital importance since abstraction, like documentation, is good only if it is useful.

5.4.2 LATERAL MOVEMENT

Lateral movement involves the ability to view objects from varied perspectives. It is naive to believe that one view can possibly capture all the information about a given object. Varying the viewing frame reveals and accentuates different aspects of the same object. This is important because as the adage goes "a picture is worth a thousand words". We may reap tremendous benefits by providing easy access to as many views of a problem and its solution as possible.

Conventional programming techniques are relatively static and not highly amenable to lateral maneuvering. This is evident from the tremendous efforts which researchers have expended to obtain even minimal changes in perspective. ABL on the other hand is structured in a manner which makes it very malleable and well disposed for generating varied representations. The following are some of the many visual transformations which can be performed on the basic tabular ABL format.

The following views are based on the bubble sort algorithm as presented by [Paig75], figure 5.8.

```
SUBROUTINE BUBBLE (A,N)
   BEGIN
   FOR I = 2 STEPS 1 UNTIL N DO
      BEGIN
      IF A(I) GE A(I-1) THEN GOTO NEXT
      J = 1
   LOOP: IF J LE 1 THEN GOTO NEXT
      IF A(J) GE A(J-1) THEN GOTO NEXT
      TEMP = A(J)
      A(J) = A(J-1)
      A(J-1) = TEMP
      J = J - 1
      GOTO LOOP
   NEXT: NULL
      END
   END
```

FIGURE 5.8 Bubble sort algorithm

ABL TABLE FORM

Figure 5.9 is a direct translation of the Bubble Sort algorithm into ABL tabular format.

KOMPUT FORM

By utilizing the Komput construct we offer the ability to view ABL programs in non-tabular format, figure 5.10. As described in the previous chapter the Komput is the only construct in ABL and is an evolved form of the ALGOL case construct. Different alternatives are chosen by matching current state vector conditions to appropriate guard descriptions. The guard description is contained within a set of matched square brackets [ ]. An addition sign , "+", indicates that a true response to the guard element must be encountered, while a negative sign, "-", is used to indicate that a false response is required. Similarly, the asterix,

					-PROGRAM-					-MACHINE-
C1	V	.	.	.	.	.	.	.	.	
C2	.	V	V	.	.	.	.	.	.	
C3	.	.	.	V	V	.	.	.	.	
C4	.	.	.	.	.	V	V	.	.	
C5	.	.	.	.	.	.	.	V	V	
P1	-	Y	N	-	-	-	-	-	-	P1 I > N
P2	-	-	-	Y	N	-	-	-	-	P2 A[I] >= A[I-1]
P3	-	-	-	-	-	Y	N	-	-	P3 J <= 1
P4	-	-	-	-	-	-	-	Y	N	P4 A[J] >= A[J-1]
A1	1	.	.	.	.	.	.	.	.	A1 I := 2
A2	.	.	.	1	.	.	.	1	.	A2 I := I + 1
A3	.	.	.	.	1	.	.	.	.	A3 J := I
A4	.	.	.	.	.	1	.	.	1	A4 TEMP := A[J]
A5	.	.	.	.	.	.	.	.	2	A5 A[J] := A[J-1]
A6	.	.	.	.	.	.	.	.	3	A6 A[J-1] := TEMP
A7	.	.	.	.	.	.	.	.	4	A7 J := J - 1
NEXT	2	0	3	2	4	2	5	2	4	

FIGURE 5.9 ABL tabular representation of the Bubble Sort algorithm from figure 5.8

"*" indicates that this is a don't care condition. The Komput form is less compact than the ABL tabular form because redundant actions are not removed. The Komput, however is close enough to conventional programming language format+ to be easily accepted by the computing community while still retaining the ABL clustering scheme.

FLOW GRAPHS

ABL flow graphs offer the user the opportunity to view explicit control flow of the program in graphical form. It

----

+ The author has implemented the Komput Form representation on a compiler which executes a subset of the Pascal language (Appendix A).

```
KOMPUT 1 [ TRUE ]

   [ + ] :  I := 2
            NEXT 2

KOMPUT 2 [ I > N ]

   [ + ] :  (no actions)
            NEXT 0

   [ - ] :  (no actions)
            NEXT 3

KOMPUT 3  [ A[I] >= A[I-1] ]

   [ + ] :  I := I + 1
            NEXT 2

   [ - ] :  J := I
            NEXT 4

KOMPUT 4  [ J <= 1 ]

   [ + ] :  I := I + 1
            NEXT 2

   [ - ] :  (no actions)
            NEXT 5

KOMPUT 5  [ A[J] >= A[J-1] ]

   [ + ] :  I := I + 1
            NEXT 2

   [ - ] :  TEMP : = A[J]
            A[J] := A[J-1]
            A[J-1] := TEMP
            J := J - 1
            NEXT 4
```

FIGURE 5.10 KOMPUT representation of the Bubble Sort algorithm

is a compact representation because it strips away most of the computational details, thus allowing the user a general overview of the program in question. ABL flow graphs are in fact functionally equivalent to DD graphs (Decision to Decision) [Paig75,Paig77].

The information necessary to produce the flow graphs can be extracted from the CLUSTER MATRIX and the NEXT VECTOR. Using ABL/DSS to produce Flow Graphs would simply require that it retrieve the attributes (cluster number,next) from

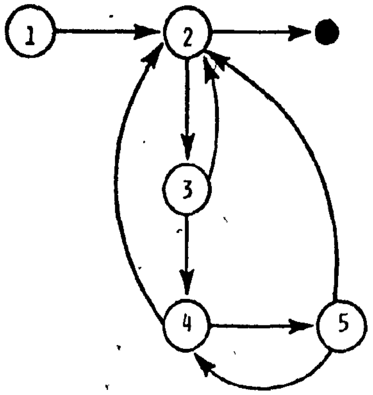


FIGURE 5.11 Flowgraph representation of the Bubble Sort algorithm

the relation Abstract Program.

NORMALIZED TREE FORM

Unlike the flow graph which gives a static picture of the algorithm, the TREE representation provides a temporal projection of the computational process, figure 5.12a. Progression down the tree from the root to the leaves is equivalent to the flow of computation spread across time. A similar method has proven to be successful as a design and scheduling tool in other areas of decision theory. The technique is known as Pert Charts and was originally developed for the US military to curb costly scheduling mistakes [Broo74].

The TREE form is composed of two basic types of leaves: Terminal and Re-entrant. Terminal nodes are equivalent to

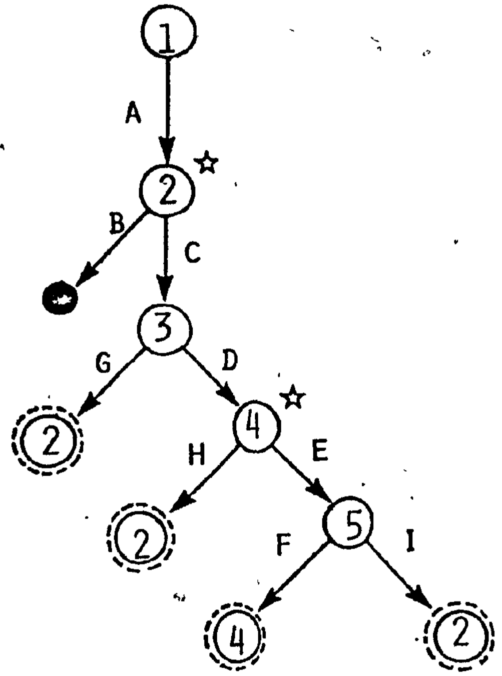


FIGURE 5.12a Normal Tree form of the bubble sort program

program exits while Re-entrant nodes depict algorithm loops. The TREE form is constructed using the algorithm illustrated in figure 5.12b which makes use of a node split process (duplication of nodes).

CONVENTIONAL CODE

Using the information available from the NORMAL TREE, we can easily convert ABL algorithms into conventional code, figure 5.13a. For ease of illustration I have chosen to convert the TREE into an ALGOL like language based on Ledgard's

```
ASSUMPTIONS:

   1) We have available a stack which is global to the program
   2) node U is initialized to Cluster 1

 DEFINITIONS:

      TERMINAL NODE:
      Is a node whose next value was zero (0).  In the
      graph they are represented as darkened nodes.

      RE-ENTRANT NODE:
      If the current node number can already be found
      on the stack then this implies that a loop exists
      and thus is labeled as a Re-entrant node.


 PROCEDURE NORMAL_TREE(U:node)

 (* U is a value parameter *)

 1.  Initialize

     1.1  Node U is visited
          - print node
          - PUSH node U onto the stack
            NEXT 2

 2.  Select a node V connected to U

     2.1  if node V is a TERMINAL node
          - print node V
            NEXT 3

     2.2  if node V is a RE-ENTRANT node
          - print node V with two concentric circles
          - label matching node on the tree with an
            asterix
            NEXT 3

     2.3  if node V is an INTERNAL node (not a leaf)
          - NORMAL_TREE(V)
            NEXT 3

 3.  Visited all nodes connected to U

     3.1  all nodes have been visited
          - POP stack
            NEXT 0

     3.2  all nodes not yet visited
            NEXT 2
```

FIGURE 5.12b Algorithm to Derive Normal Tree Form from Flow Graphs

[Ledg75] REC2 structures and the extended case (as described in chapter 4). The algorithm to do this conversion is described in figure 5.13b.

Subsequently it is a simple matter to proceed to lower level control structures.

```
BEGIN
  I := 2
  REPEAT
    IF ( I > N )
       THEN EXIT (1)
       ELSE BEGIN
              IF ( A[I] >= A [I-1] )
                 THEN BEGIN
                        I := I + 1
                        CYCLE (1)
                      END
                 ELSE BEGIN
                        J := I
                        REPEAT
                          IF ( J <= I )
                             THEN BEGIN
                                    I := I + 1
                                    CYCLE (2)
                                  END
                             ELSE BEGIN
                                    IF ( A[J] >= A[J-1] )
                                       THEN BEGIN
                                              I := I + 1
                                              CYCLE (2)
                                            END
                                       ELSE BEGIN
                                              TEMP := A[J]
                                              A[J] := A[J-1]
                                              A[J-1] := TEMP
                                              J := J - 1
                                              CYCLE (1)
                                            END
                                  END
                         END (* REPEAT *)
                      END
                END
   END (* REPEAT *)
END.
```

FIGURE 5.13a Conventional code representation of the bubble sort algorithm

PRODUCTION RULES

Production rules provide a compact notational representation based on formal language theory, figure 5.14a. This representation can be derived from information provided by the NORMAL TREE form. It partitions the Tree into segments (productions) which can then be used to permit the user to generate all possible path expressions for a program. An algorithm to perform the transformation of Tree to Production rules is described in figure 5.14b.

ASSUMPTIONS:

1) We have available all information concerning the structure of the tree.
2) We have available a file into which we can copy and or add program statements (this is file with conventional code).

By utilizing and maintaining the nesting structure depicted by the TREE form we can convert an ABL program into a conventional program by making use of the following information:

a) Each branch in the tree corresponds to an alternative.
b) There are two types of Leaves on the tree (Terminal and Re-entrant).
c) There are two types of Internal nodes in the tree (Normal and Internal Re-entrant)

```
PROCEDURE CONVENTIONAL_CODE (U:node)

(* U is a value parameter *)

1.  Determine the status of node U

    1.1  node U is an Internal node
         NEXT 2

    1.2  node U is a Leaf node
         NEXT 4

2.  The node being checked is an Internal node

    2.1  node U is INTERNAL RE-ENTRANT
         - bound all branches associated with
           node U with : "REPEAT ... END;"
         NEXT 3

    2.2  node U is a NORMAL internal node
         - bound all branches associated with
           node U with : "BEGIN ... END;"
         NEXT 3

3.  Determine the outdegree status of node U

    3.1  node U has outdegree = 1
         - copy the actions associated with that branch
         - node V = NEXT value of the current alternative
         - CONVENTIONAL_CODE(V)
         NEXT 0

    3.2  node U has an outdegree = 2  (binary branch point)
         - use the IF_THEN_ELSE construct and copy the actions
           associated with the appropriate branches as follows:

            IF (predicate)
               THEN  - copy 'U.alternative_1 actions'
                     - node V = NEXT value of U.alternative_1
                     - CONVENTIONAL_CODE(V)
               ELSE  - copy 'U.alternative_2 actions'
                     - node V = NEXT value of U.alternative_2
                     - CONVENTIONAL_CODE(V)
         NEXT 0

   3.3  node U has an outdegree > 2 (multiway branch point)
        - use the extended CASE construct (section 4.1.2) and
          copy the actions associated with the appropriate
          branches as follows:

          CASE [predicate_1,predicate_2,...,predicate_i]
          [c1,c2,...ci] :  - copy 'U.alternative_1 actions'
                           - node U = NEXT value of U.alternative_1
                           - CONVENTIONAL_CODE(V)
                .
                .
                .
          [c1,c2,...ci] :  - copy 'U.alternative_N actions'
                           - node V = NEXT value of U.alternative_N
                           - CONVENTIONAL_CODE (V)
        NEXT 0

4.  The node being examined is a Leaf element

    4.1  the leaf is a TERMINAL node (darkened node)
         - count the number of RE-ENTRANT internal nodes
           which must be traversed in order to return to
           the root
         - if the count > 1 then add the statement
           'EXIT'(count)'
         NEXT 0

    4.2  leaf is RE-ENTRANT (concentric circles)
         - count the number of RE-ENTRANT internal nodes
           which must be traversed in order to arrive at
           the re-entrant internal node having the same
           value as the leaf node being examined (including
           arriving at the internal node as part of the
           traversal count)
         - add the statement 'CYCLE (count)'
         NEXT 0
```

FIGURE 5.13b Algorithm to derive conventional code from the Normal Tree form

a)

$$S \rightarrow aB$$

$$B \rightarrow b \mid cgB \mid cdC$$

$$C \rightarrow hB \mid efC \mid eiB$$

ASSUMPTIONS:

1) We have the TREE form available and all branches are appropriately labelled (in this case I have used lower case alphabetic characters).
2) Productions are represented as a string of terminals [lower case alphabet] and non-terminals [upper case alphabet].
3) We have available a stack which is global to the program.

Note in the following procedure that when a rule is said to be finished tthe production is then associated with the non-terminal currently at the top of the stack. Wherein the top of the stack is initialized to the start symbol S.

b)

```
PROCEDURE PRODUCTION_RULES (U:node, CURRENT_RULE:string)

(* U is a value parameter and CURRENT_RULE is a variable parameter *)

1.  Determine the status of node U

    1.1  node U is a TERMINAL leaf
         - finish rule
         - CURRENT_RULE = null string
         NEXT 0

    1.2  node U is a RE-ENTRANT TERMINAL leaf
         - append to CURRENT_RULE the non-terminal which
           corresponds to the node number
         - finish rule
         - CURRENT_RULE = null string
         NEXT 0

    1.3  node U is an INTERNAL RE-ENTRANT node
         NEXT 2

    1.4  node U is an INTERNAL NORMAL node
         NEXT 3

2.  Determine whether node U has been assigned a non-terminal
    character

    2.1  node U has been assigned a non-terminal character
         - append to CURRENT_RULE the non-terminal which
           corresponds to the node number for U
         - finish rule
         - CURRENT_RULE = null string
         NEXT 0

    2.2  node U has not been assigned a non-terminal character
         -  node U is assigned to the successor of upper case
            alphabet
         -  append to CURRENT_RULE the non-terminal which has
            been assigned to node U
         -  finish rule
         -  PUSH the non-terminal assigned to node U unto the stack
         -  CURRENT_RULE = null string
         NEXT 3

3.  Determine the status of paths leading from node U

    3.1  all branches leading from node U have been traversed
         - if node U is INTERNAL RE-ENTRANT then POP a non-
           terminal symbol from the stack
         NEXT 0

    3.2  all branches leading from node U have not been traversed
         -  select a path leading from node U
         -  TEMP = CURRENT_RULE
         -  append the path selected to CURRENT_RULE
         -  node V = NEXT value of current branch
         -  PRODUCTION_RULES(V,CURRENT_RULE)
         -  if node U is an INTERNAL NORMAL node
            then CURRENT_RULE = TEMP
         NEXT 3
```

FIGURE 5.14a Production Rules for the bubble sort program

5.14b Algorithm to derive Production Rules from the Normal Tree form

The above list of transformations is not exhaustive, they have been included to illustrate the point that each transformation highlights different features. By allowing access to many such transformations, user's will be able to quickly assimilate information pertinent to their own particular needs. This greater depth of scope should enable the user to more fully appreciate and evaluate the complexity of the problems at hand.

5.4.3 DYNAMIC REORGANIZATION

The term dynamic reorganization in a broad sense means being able to interactively manipulate the information which is being presented. Lateral movement obviously satisfies this definition and is an acceptable method for accomplishing this. ABL/DSS however can also provide other more elegant means with which to perform dynamic reorganization. Given the fact that the Relational database model is soundly based on mathematical set theory we can make use of high level operators based on Relational Algebra [Date77,Codd82].

Thus we have at our disposal well defined operators such as:

PROJECT, JOIN, UNION, INTERSECT, DIFFERENCE, DIVISION. Furthermore, the manipulation of relations under these operators is closed: thus we know that we can specify operations on relations which must yield other relations. In addition use of these relationally closed operators gives us direct and immediate access to specific program attributes not easily obtainable from conventional programming languages. For example we may retrieve information of the following type :

a) List all alternatives with a specific cluster number

b) List only the alternatives which use specific predicates and/or actions

c) List all alternatives with common next values

d) List common attributes of a specific set of alternatives

e) Retrieve all documentation pertinent to a given cluster.

Other queries would allow the user to more fully explore similarities and/or differences between various algorithms. The addition of data flow information, which will be introduced in the next chapter, will provide for even more finely detailed algorithm analysis.

The purpose of this chapter has been to provide an overview of some of the more significant areas and of the more visible deficencies inherent in conventional programming environments. A Relational database model was used in conjunction with the ABL approach to examine how these deficiencies could be reduced to less detrimental levels. Emphasis has been placed on attempting to overcome problems associated with human weaknesses and failings rather then on computationally dependant criteria.

CHAPTER 6

SOFTWARE TESTING AND VALIDATION

The penultimate step in the software lifecycle is the testing and validation of the software product produced. This basically involves assessing the "correctness" of the program. If we take the definition of a program to be a function which describes the relationship of an input element (domain element) to an output element (range element), then "correctness" is interpreted to mean that a given program will faithfully realize the appropriate mapping [Rama82]. The problem which arises is that no known software methodology can guarantee that the program it generates will precisely embody its required function. Hence testing and validation procedures are concerned with the manner in which we may ascertain the veracity of any given function.

6.1 VALIDATION TECHNIQUES

There are three basic and quite distinct modalities with which we may approach this task of verification: exhaustive comparison, program proofs, testing.

6.1.1 EXHAUSTIVE COMPARISON

The simplest and perhaps the most obvious is the technique of Exhaustive Comparison [Adri82,Rama77]. This would involve verifying that for each domain element the program produces a result which matches the expected (true) output. If each instance succeeds we can be assured of the validity of the program, this technique however is not eminently practical. This can be attributed to the fact that if we had access to such complete information, it would obviate the need for a program in the first place. Answers could be acquired faster and with less trouble via a table look up rather than through a program defined computational process.

6.1.2 PROGRAM PROOFS

The second technique called Program Proving involves a highly technical and optimistic approach based on the machinery of mathematical logic [Hoar69]. We may recall that program proofs were discussed in the previous chapter in a somewhat different context. There program proofs were intended as a conceptual framework with which to guide the building of "correct" programs. Program verification in the current context is in fact the converse of this. The purpose is not to be a building tool but rather to provide an abstract notational medium with which to dissect and hopefully isolate logical inconsistencies from a given program.

"Program proving involves expressing the program specifications as a logical proposition, expressing individual program execution statements as logical propositions, expressing program branching as an expansion into separate cases, and performing logical transformations on the propositions in a way which ends by demonstrating the equivalence of the program and its specification" [Boeh79]. Normally the issues of termination and potentially infinite loops are handled seperately via inductive reasoning.

This approach however has just barely begun to penetrate the world of production software [Boeh79,Demi77,Dijk78]. Primary reasons for this include several important deficiencies. First and foremost is the fact that even trivial programs are complicated and time consuming to prove [Adri82,Boeh73]. Automated systems are not much help either because they do not work on the more common languages such as Fortran or Cobol. These languages cannot be accommodated because they do not permit precise axiomatic definitions which are essential for the expression of program statements as logical propositions.

An often quoted remark implying the superior effectiveness of program proofs is that other techniques for "program testing can be used to show the presence of bugs, but never to show their absence" [Dijk72]. It should be noted however, that program proofs can be shown to be consistent with specifications which are themselves incorrect, and the fact that nothing guarantees that the proof itself is complete and or correct, as demonstrated by [Good77]. Given that these two limitations are considered errors we must conclude that "program proving can be used to demonstrate the presence of errors but never their absence" [Boeh79].

6.1.3 TESTING

The last and most pragmatic approach involves a wide range of techniques known as Program Testing. Unlike program proofs which provide conclusions about program behavior in a postulated environment, testing provides the user with "accurate information about a program's actual behavior in its actual environment" [Good77]. The purpose of testing is to provide a means with which we may assess various levels of algorithm performance and thus provide the user with a sense of confidence and assurance concerning the reliability of the software. In general the techniques involved are posthoc and rely on both analytic and non-analytic means to provide measures of correctness.

6.2 PROGRAM TESTING

Techniques for testing and test data generation fall into one of two complimentary categories, either Functional or Structual [Rama77,Adri82]. Functional testing is also known as the black box approach, which is primairly concerned with user visible attributes. Test data are derived apriori from software specifications rather than from posthoc algorithm analysis based on implementation dependant details.

Structural testing on the other hand is almost wholly dependant on the study of the internal organization of the software. Since this requires that we actually examine the software, this technique is alternately known as the glass box approach. The purpose of this form of testing is to assure that all internal operations perform according to specifications.

6.2.1 FUNCTIONAL TESTING

Functional testing is basically a refinement on the method of exhaustive comparison and utilizes more realistic paradigms. Rather than testing the whole input domain, a selected, non random, stratified and hopefully sufficiently complete data sample is used. Results from the chosen data sets are then interpreted and extrapolated to the general problem domain. "Test data must be derived from an analysis of the functional requirements and include representative elements from all the variable domains. These data should include both valid and invalid inputs" [Adri82]. Program data can also be partitioned into both input and output classes. Tests are then run on representative data samples which lie within each of the classes (non-extremal) and at the boundaries of each class (extremal). Testing based on these sets is often also known as boundary value analysis.

6.2.2 STRUCTURAL TESTING

Structural testing is by far the most widely used technique for software validation [Mill74,Bagg78,Good77]. The traditional manual method is called desk checking: going over a program by hand. Desk checking however is more a debugging task than an actual testing technique. In order to overcome limitations of the individual the approach was expanded to include two or more participants: egoless programming [Wein71], peer code reviews [Glas80], and chief programmer terms [Bake72]. Desk checking also evolved into more formal and disciplined procedures called code walkthroughs and inspections [Adri82], where several people read the software product and run through a simple simulation evaluating it against predetermined criteria.

In order to describe and produce more efficient protocols for assessing desirable levels of confidence from structural analysis, researchers have developed software tools and software which permits a greater level of measurability. Within the realm of measurability we often encounter the phrase "software complexity". The phrase denotes a numerical term which is usually determined as a function of some quantifiable software feature. Increased complexity is inversely proportional to software "quality", thus complexity measures provide a means with which to guage relative amounts of testing needed to insure reliability.

This area of software research is currently basking in the aura of "great importance" and has been deemed valuable to both academia and members of the business community [Paig80,Wals79,Schn79]. Researchers have produced a plethora of techniques with which to enumerate and ultimately predict physical characteristics of software.

Typically, measurability is also a function of accessibility. In other words how easy is it to extract relevant information from the appropriate software products. Software testing and validation already consumes 40 to 50% of initial development costs, therefore we must be extremely conscious not only to develop different software testing tools but also to decrease the overhead incurred by their use.

To do justice to this very rich and diverse field of software physics would be far beyond the scope of this thesis. As such the reader is referred to the accompanying annotated bibliography which provides a somewhat more than introductory overview of several of the more salient papers encountered within the domain of software complexity.

6.3 STATIC STRUCTURAL ANALYSIS

The basis of static analysis resides in the ability to extract and interpret both control flow and data flow information of a given program. This work is either done by hand or through the use of automatic tools such as parsers, translators, and preprocessors [Rama77]. The purpose of extracting control flow is to permit for a visual multidimensional representation of program behavior, whereas data flow is used to analyze transformations made on program variables.

Data flow techniques are conventionally used for discovering program anomolies such as undefined or unreferenced variables, as well as for performing program optimization [Cock69,Lowr69]. Data flow is also the basis of numerous design and programming techniques [Dwey79,Gust77] for the generation of conventional code based on data flow considerations. Recently there has also been a fair amount of research into functional programming, a technique whereby code is completely divorced from control flow [Back78,Back82,Ache82,Davi82]. The introduction of these data flow languages has sparked interest in complementary research geared to the implementation of data flow based machine architectures [Wats82].

6.3.1 COVERAGE BASED TESTING

Most structural testing is primarily centered about control flow and is examined via coverage based testing. The underlying philosophy is that selective execution of program paths is a viable alternative to exhaustive search. The errors this technique attempts to isolate can be categorized into the following classes [Good77]:

(a) missing control flow paths - incomplete logic
(b) inappropriate path selection - due to incorrect logic
(c) inappropriate or missing actions - due to oversight

Coverage based testing can be used at several levels of refinement which exercise successively more comprehensive structural components. The accepted standard is as follows [Bagg78]:

(1) execute every statement
(2) execute every conditional branch
(3) execute all possible program paths

Paige [Paig75] has introduced an intermediate step called decision to decision (DD) paths: "It is a portion of a program, path beginning with the execution of a branch predicate and including all statements up to the evaluation, but not execution, of the next branch predicate" [Paig75].

Thus the above list can be expanded by adding the following.

(2.5) execute every DD path.

All of the above require that a program flowchart or flow graph be extracted from the program so that appropriate test cases may be prepared [Rama74]. These graphical approaches further allow the user to uncover two other types of errors:

(a) syntactically disjoint code (dead code) in which the control for the program does not allow a link to certain sections of code

(b) semantically disjoint code in which it is logically impossible for certain paths to be executed

A somewhat different technique for obtaining test data is called symbolic execution and is based on symbolically defining data. These symbols are used instead of actual data values when executing the program. In symbolic execution the effect of an assignment statement would be to replace the value of the left hand side variable by a string representing the unevaluated expression on the right hand side. Thus the result of the execution would produce a complex expression which can then be used to derive appropriate test data.

Mutant analysis is a method by which programs are interactively seeded with errors (mutants) and then run with test data which has been previously executed. The runs are checked against each other to reveal errors. The success of this approach rests upon two fundamental hypothesis:

(1) competent programmer hypothesis - in which it is assumed that the program which is to be tested is nearly "correct"

(2) coupling effect - that tests which uncover simple errors will also uncover more complex errors

Although relatively new and quite costly both these techniques have been shown to be reasonably effective [Budd78a,Budd80,Adri82], and may be extensively used in the future.

6.3.2 STATIC TESTING USING ABL/DSS

At this point in the thesis it should be clear that ABL is extremely well suited for static analysis and testing. The most essential step towards the analysis procedure is extracting the appropriate control flow from a program. Unlike conventional programming languages with implicitly defined control flows, ABL does not require complex and costly parsing techniques to obtain the relevant information. As explained in section 4.5 of this thesis ABL flow graphs are easily derived from the appropriate abstract programs.

Testing of each of the four levels of structural components using ABL is also a trivial process. Without the constraints of embedded control flow, the data analysis necessary to satisfy complete execution at each level of testing is minimal. Of course the assumption is made that we have available an interactive monitor and test driving procedure. For example:

(1) to execute each statement it is necessary to invoke each action or predicate element in the abstract machine

(2) to test each conditional branch requires that we satisfy the guards for each alternative in the abstract program

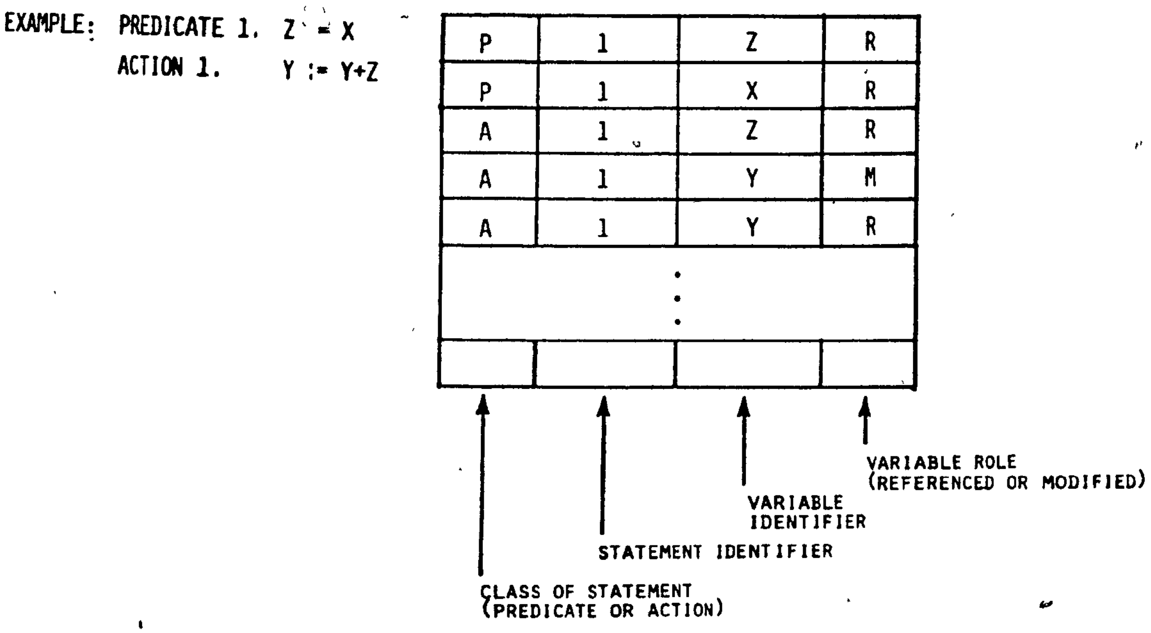
(2.5) to treat every DD path merely requires that each alternative in the ABL program be executed

(3) to test every path requires that we traverse each path expressions drawn from the normal tree form of the ABL flow graph

All that is required is that we define the following relation to hold relevant data flow information.

ABSTRACT DATA FLOW RELATION

Each tuple in the relation illustrated in figure 6.1 gives information about a single variable from a given action/predicate element in the Abstract Machine. The attributes of this relation consist of the following: class of statement ('A' for action, 'P' for predicate), statement), number (integer representing cardinality of the staement), variable name (alphanumeric string of characters), data role (characteristics of the variable: 'R' for referenced, 'M' for modified).



EXAMPLE: PREDICATE 1. Z = X
ACTION 1. Y := Y+Z

P	1	Z	R
P	1	X	R
A	1	Z	R
A	1	Y	M
A	1	Y	R
⋮			

VARIABLE ROLE (REFERENCED OR MODIFIED)
VARIABLE IDENTIFIER
STATEMENT IDENTIFIER
CLASS OF STATEMENT (PREDICATE OR ACTION)

FIGURE 6.1 Relation Abstract Data Flow

Combining the information from the Data Flow relation and the Abstract Program relations via high level relational operators and path expressions gives the user hierarchical, context sensitive data flow information. With the insertion of this data flow relation we have introduced the potential to perform symbolic execution. This can be achieved in a relatively uncomplicated manner through the use of substitution, concatentation, relational algebraic operators and other nececessary utility routines which are synchronized and coordinated with ABL/DSS facilities.

It has been shown that mutation analysis "for special cases like decision tables can be used to verify programs fully" [Budd78b]. Given that a decision table is a special case of an ABL program and that each cluster has at least all the characteristics of a decision table, then it should be possible to apply this technique directly to ABL. However, the question of whether ABL programs are fully verifiable using mutation analysis has yet to be ascertained. The implications of the latter provides a suitable research topic for yet another thesis.

Maintenance costs have been shown to absorb as much as 50 to 75% of all cost incurred during the functional lifetime of any software system. A substantial portion of this amount can be attributed to the testing which must be perfomed to verify that maintenance changes are correct and have not damaged other unrelated program functions. This form of testing is labelled Regression Testing [Adri82]. Part of the reason for these high costs is that conventional programming methods create products which are extremely static, where even minor changes become forced operations. For example, if during testing a change has to made to the conditional which controls a loop then the programmer has to: (1) abandon the compiled version, (2) acquire the source code, (3) call up the text editing facility, (4) make the appropriate changes, (5) recompile the latest version, and (6) resume testing on the program.

More extensive changes or simply trickier ones usually tend to generate a mosaic approach to program refinement, wherein the resultant quality of the new version is usually a degraded version of the original. People in general hate to "redo" something which already exists. In their attempt to "not waste" the previous effort they try to incorporate as much of the old into the new. This reluctance to "redo" tends to promote error propagation because it most often involves introducing housekeeping chores not relevant to the problem at hand, thus reducing the cohesiveness of the previously established program logic. The ABL building block approach minimizes these problems of maintenance and selective retesting.

The fundamental ABL tabular template is another characteristic beneficial to the verification process. Goodenough [Good77] has stated that "most software errors result from failing to see or deal correctly with all conditions and combinations of conditions relevant to the desired operation of a program. An effective methodology for reliable program development must focus on uncovering these conditions and their combinations". He concludes by stating that use of condition tables (extended decision tables) make it easier to "find and analyze" the requisite condition combinations.

Through ABL each action and predicate can be costed and each alternative can be assigned a probability of execution. By combining these two program attributes with the normal tree form representation gives us a means by which to evaluate theoretical program behavior. This static profile can be used to determine both relative and absolute criteria for comparisign and optimization. Some preliminary work on this area has been reported [Lebe82]. The ABL approach can be further utilized to establish a family of metrics which can be used to guide and bound software testing. Figure 6.2 presents a prototype set of the formulations accrued from the above considerations.

A simple logical extension of the formulations given above can allow for a cost effective procedure to meet individual customer specified reliability constraints. This is accomplished via a flexible tree pruning strategy which allows the user to engage in either proportional or type based testing schemes, dependant on branch costs and branch

$$SCP = \sum_{i=1}^{N} CP_i \qquad \text{(EQ.1)}$$

$$CP = \sum_{i=1}^{K} CA_i \qquad \text{(EQ.2)}$$

$$CA = P_i * \left(\sum_{i=1}^{Q} G_i + \sum_{i=1}^{A} S_i \right) \qquad \text{(EQ.3)}$$

$$PP = \prod_{i=1}^{K} P_i \qquad \text{(EQ.4)}$$

$$1 = \sum_{i=1}^{N} PP_i \qquad \text{(EQ.5)}$$

SCP	=	STATIC COST OF A PROGRAM
CP	=	COST OF ANY GIVEN PATH
CA	=	COST OF ANY GIVEN ALTERNATIVE
S	=	COST OF EXECUTING A STATEMENT
G	=	COST OF EXECUTING A GUARD
PP	=	PROBABILITY OF ANY GIVEN PATH
P	=	PROBABILITY OF EXECUTING AN ALTERNATIVE
Q	=	SET OF GUARDS IN AN ALTERNATIVE
A	=	SET OF ACTIONS IN AN ALTERNATIVE
N	=	SET OF ALL PATHS IN A PROGRAM
K	=	SET OF ALTERNATIVES IN A GIVEN PATH

FIGURE 6.2 Prototype set of formulae which can be used to guide software testing

probabilities respectively. This thinning procedure is accomplished through the use of thresholds (Global or Local), and hence can be refered to as the THRESHOLD TESTING TECHNIQUE.

A typical use of a global threshold could be to restrict the length of program path expressions. Thus we could grow a given tree, re-entrant nodes being replaced by appropriate subtrees, until the cumulative probability of the total path falls below some designated value for minimum path probability. An alternative method would be to restrict the size of path expressions by limiting the upper bound of permissible path costs.

The function of the local threshold would be to discard any branch whose probability of execution or functional cost does not surpass some user defined threshold level. This can be used to limit the proportion of "less important" paths in a given tree. Both types of thresholds can be used in conjunction to produce customized testing strategies. Exploitation of this technique deserves further study and provides another topic for a future thesis proposal.

6.4 DYNAMIC STRUCTURAL ANALYSIS

The purpose of dynamic analysis is to verify that program behavior during execution, performs as was anticipated by the static analysis techniques. "Dynamic analysis performs both error diagnosis and verification of performance requirements. It helps to detect and locate errors by displaying the various events that occurred during execution. The amount of code not exercised by any test cases can be used to indicate test ineffectiveness. Code sections that are most frequently executed are identified for optimization purposes" [Rama77]. The methods used to acquire the appropriate information falls into two general categories: execution profiles and assertion checkers.

6.4.1 EXECUTION PROFILES AND ASSERTIONS

An execution profile is primarily a list of frequency counts indicating how each of the program components is executed during any given test run. Besides statement counts, other data which can be collected include: execution time of each statement, distribution of logical success on conditional branches, maximum/minimum values of chosen variables, number of loop iterations [Good77]. Collective statistics on judiciously chosen data samples provides for a reasonably accurate (though not complete) picture of the associated algorithm dynamics [Chev78,Hals73].

To obtain the required information, software probes in the form of source language statements are inserted into the source code [Rama77,Paig74]. However, since these counters significantly burden the execution of the program, it is desirable to minimize the number of counters used. Dynamic analysis techniques often make use of graph theory for partitioning the system into smaller subsystems in order to identify important attributes as well to determine locations for placing test monitors. A great deal of research has been expended on producing instrumentation which can both locate and automatically insert probes into the software.

The principal advantage of dynamic analysis is that it provides for more effective management of testing procedures. "In general, the most active or frequently executed portions of a program are thoroughly tested, while the less active portions receive inadequate testing. Program segments with zero frequency counts could be given more attention in testing and singled out for early and intensive testing" [Rama77].

Although formal program proofs have been shown to be non feasible for larger programs, a more viable option for improving reliability has been to place assertions within the body of the program [Boeh79]. "The assertions may take the form of range checks, state checks, reasonableness checks, and inverse checks. A range check ensures that values of data are within the specified range during execution. A state check verifies that certain conditions hold among the program variables. A reasonableness check is applied to the input data to avoid system misbehavior resulting from abnormal ihput data. An inverse check is used to ensure the correct operation of the system" [Rama77].

These assertions are then executed during the course of a regular test run with violations being reported to the user. The drawback to this technique is that we need to define a high level command language as well to have available a preprocessor which can convert the assertions into executable source code. Overall however this approach seems to be a practical step towards the goal of achieving more reliable software.

6.4.2 DYNAMIC TESTING USING ABL/DSS

ABL/DSS is highly amenable to techniques for dynamic structural analysis. In terms of execution profiles the following data is easily assembled without the excessive overhead encountered with conventional approaches. ABL solves some of these problems because "probes" per se are unnecessary. The proposed function of a probe is rendered redundant by ABL's more abstract mapping of control flow to source code. For example action and predicate counts or costs can be computed directly from frequency counts of alternatives. The following is a partial compilation of typical quantitative dynamic characteristics currently in vogue, and easily obtainable using ABL techniques:

(1) frequency of each statement
(2) frequency of each action/predicate
(3) frequency of each cluster

(4) frequency of each alternative (DD path)

(5) minimum/maximum of any assignment statement/variable

Other more complex and detailed statistics may be obtained by either comparing or decomposing the parameters given above. A non trivial example would be the facility with which ABL can track path expressions. Most of this information can be recorded directly into the appropriate relations through the addition of new attributes.

Furthermore ABL can also be made to handle assertions. The extension of its infrastructure via postconditions permits us to perform a Boolean evaluation functionally equivalent to the assertion. These post conditions may be implemented through the addition of two new structural components to each of the Alternatives: (1) matrix of post condition predicates, (2) a NEXT vector associated with the post condition predicates. Figure 6.3 illustrates the concatenation of post-conditions into the ABL tabular template. Besides flagging errors the post-condition can also be used to override primary control flow thereby allowing the program to compensate for certain types of errors, thus bringing us into the fringes of fault tolerant computing.

	A_1	A_2	A_3	...	A_n
CLUSTER C_1 C_2 C_3 . . . C_i					
PREDICATE P_1 P_2 P_3 . . . P_j					
ACTIONS a_1 a_2 a_3 . . . a_k					
POSTCONDITION PREDICATE PC_1 PC_2 PC_3 . . . PC_L					
POSTCONDITION NEXT	PN_1	PN_2	PN_3	...	PN_n
NEXT	N_1	N_2	N_3	...	N_n

FIGURE 6.3 The addition of Post-Conditions to the ABL tabular template

CHAPTER 7

CONCLUDING REMARKS

The purpose of this thesis has been to focus attention on the multitude of conceptual and technical problems encountered by the computer scientist. We have approached these problems by characterizing the state of the art of software technology. This endeavor has forced us to look at both the deficiencies and successes of: language, software production methodologies, software tools, software testing and validation techniques as well as the psychology of programming and more importantly the psychology of problem solving.

Implicit to the entire domain of software engineering are the issues of perspective, cueing and communication. Although "difficulty of communication has become recognized as the biggest obstacle to progress in the use of computers" [Date75], I feel that this is not the most crucial aspect of the art of computing. Rather our focus should be to enhance understanding and ameliorate problem solving of which communication is an essential component.

If computer science is to be an extension of intellect then we need to recognize that the principal component in the melding of man to machine is the human mind. Linguistic ability and communication alone are certainly an insufficient basis with which to ensure the quality of computational ability. Dijkstra [Dijk72] has stated that "one of the most important aspects of any computing tool is its influence on the thinking habits of those who try to use it". Realistically we cannot hope to extend intellect if we continually and casually ignore the impact of mind.

Science in general is the acquisition of knowledge derived from the study of things which occur. Researchers formulate theories and abstractions which court "why" questions. The practice of engineering is driven by man's attempts to build things which fulfill needs. This discipline is generally guided by a body of knowledge which stress the question of "how". Art on the other hand is an extension of human expression that need not question its environment yet projects the subjective prespective of its creator. Computer science is an analgamation of all of the above and though possessing properties of each also has perspectives unique unto itself. It is this hybridization which may underlie its refreshing but intractable vitality.

Traditional methods of software development have been "relatively ineffective" because each has attempted to directly apply techniques garnered from these parent disciplines. Given that computer science heralds the next industrial and cultural revolution, can the ABL methodology be expected to significantly contribute to this process? We believe that ABL is not only an academic exercise but will provide a medium which will aid not only the discipline of programming but its users as well.

ANNOTATED BIBLIOGRAPHY

The articles covered in this bibliography are on the topic of software complexity and comprehensibility. The papers reviewed include several review articles as well as theoretical and experimental works. The references listed do not represent a complete review of the papers in my own bibliography much less an exhaustive search of the literature. Rather, the purpose has been to provide a collection of papers which while restricted in number would nevertheless span the spectrum of issues involved in this field of research. It is hoped that the following reviews will give the casual reader an adequate overview of the area as well as providing a suitable starting point for anyone wishing to begin research on the topic of complexity.

[Berl80] Does a nice review of the state of the art in complexity measure including the methodology for calculating each of the following : 1) Hellerman (H - entropy function) 2) McCabe (V - cyclomatic number) 3) Meyers (small change to McCabe's V) 4) McTap program features compared to reference base 5) Chapin Q - index of either program or module complexity 6) Chen Z - based on the topological properties of the graph 7) Mohanty C - entropy loading measure 8)

Halstead: V, V*, L, E, I - volume, potential volume, program level, program effort, information content) 9) Sullivan may measure. Last but not least he presents his measure which is called M - which is supposed to refuse total information contained in a program as H - language entropy. This measure is based on the probability distributions with which various tokens of a program are used. Mention is made of Zipf's law.

[Broo77] The following points deserve mention : 1) difficulties in programming are clearly linked to specific features or properties of programming languages (ie. note Shneiderman paper on Cobol vs Fortran) 2) experienced programmers make same number of mistakes initially but find the mistakes faster 3) better conceptual organization involves reducing the cognitive load 4) offers a three step interpretation of programming (problem understanding,method finding, coding) 5) claims that human problem solving behavior is a production system based on pairs of conditions and actions to be performed 6) gives an ABL like appoach to what he feels should be a better methodology for programming.

[Chap79] The author provides for a brief and somewhat restricted review of some of the complexity measures currently available including: McCabe, Zolnowski, McClure, Myers, McTap. He then goes on to describe his own measure which he calls Q. The claim is made that Q is an "index of the difficulty people have in understanding the function implemented by the software", where function is defined as follows : "a function is a correspondence between sets of input data of specified domains, and sets of output data of specified ranges". A ten step computational procedure (complete with example) for calculating the Q of a program is given. Chapin then goes on to do a comparison of his measure to the previously mentioned ones in which he points out some of the strengths and weaknesses of both his and theirs. He includes the recommendation that the McCabe measure might be of greater use if to obtain firm counts of lines, nodes and branches. The author concludes by making the statement that unlike the other measures which require code, Q can be used during design, implementation and maintenance with or without code. Also that this measure provides for "complexity density" rather than the McCabe "complexity extent".

[Chen78] The purpose of this paper is an attempt to demonstrate an empirical relationship between his chosen complexity factor which is a function of control logic, based on a topological analysis and programmer productivity. He mentions that the three basic structures are : 1) serial 2) parallel and 3) recurrent patterns, with the next level of structural entities called "functional blocks" . His results reveal that there seems to be a quantitative relationship between productivity and complexity. The key factor seems to be the existence of "quantum drops or cliffs" in the curve which describes this relationship. There probably exists some sort of cause/effect relationship. We could probably make a point which connects these leaps with our limited abilities of short term memories (ie. the magic number seven plus or minus two). The ideas presented also seem to correlate the concept of "chunking", hence giving us a reason for imposing upper bounds on certain items, such as maintaining some sort of manageability by restricting module size this has already been suggested by others - max 150 lines per procedure.

[Chry78] His opening statement says that if longer development time is related to longer programs, therefore variables associated with development time should also be related to program size. The author then lists fifteen program characteristics and five programmer characteristics which he feels are related to program development time.

Experiments were run (using COBOL programs) from which he attempts to correlate both program and programmer characteristics to the following : 1) data division source statements 2) working storage bytes 3) Procedure division source statements 4) procedure division bytes. Conclusions were as follows : 1) no predictive power from the programmer characteristics 2) three program characteristics were correlated with procedure division bytes at the 0.0005 level.

[Curt79] The terms "computational complexity" and "psychological complexity" are defined. The authors then experimentally attempted to manipulate factors which they believed would change each of the above. Results showed that performance was better on structured programs then unstructured ones. They also found that the metrics were more strongly correlated with time to completion then the accuracy of the implementation (ie. especially on maintenance tasks), also that the metrics were better predictors of performance in the less experienced programmers. This is important because as the authors suggest the more experienced programmers probably reduce the cognitive load "chunking" by structured techniques and commenting. This is probably the reason why people have an incredibly high inertial resistence for acceptence of any new technique. They can no longer use their cognitive load reduction techniques which they have acquired through costly experience.

[Feur79] Authors state that maintenance costs dominate the total cost of a large software system and that emphasis has shifted from "bit fiddling" (my quotes) to issues of clarity and flexibility in program structures. They were seeking a language independant measure so they decided to use flow graphs of the programs. Some of the measures they used include: count of nodes, ratio of decision nodes to total nodes, possible paths through graph as well as mean number of decisions per path. They present an algorithm for reducing the flow graphs to check for strictly well structured programs: reducing flow graph to a single a node. No references are given for this algorithm which I believe varies only slightly from the algorithm given by Hecht and others. Besides the above mentioned variables they also studied the following: length, expected length, volume, level and effort. Results yielded that there was a "striking dependency" between time of repair and node count (time of repair of bugs increased with node count). The authors also revealed that there exists two types of errors (easy and hard to repair), where the easy to repair occur with greater probability then the difficult to repair variety. It was also suggested that the difficult to repair errors required a time component which was exponentially related to node count, whereas the easy to repair errors were linearly related to node count. Other findings include the fact that module size is a good indicator of maintenance performance and that the level variable (adjusted for size) was a fair indicator of the performance of groups of modules.

[Gord79] Attempts to assess the amount of mental effort required to understand a program. Three areas are studied: 1) programmer ability 2) program form 3) program structure. This paper is is basically an experimental study based primarily on Halstead's measure of program complexity. His criteria of complexity is based on the number of operators and operands in a program. The paper includes a good review of the topic of programmer productivity complexity measures in general. Results state Halstead's volume measurement is not a suitable or sufficient indicator of program clarity. Stresses fact that "algorithm" comprehension is a very important factor in program understanding, as assessed from maintenance and debugging tasks.

[Gust77] Talks about the importance of data flow in program testing especially with regard to test data selection and symbolic execution. Mentions briefly the importance of control flow as a basis for determining complexity and how it related to program testing. The author makes extensive use of flow diagrams to express data flow.

[Hous80] The author gives five causes for poor software including : 1) poor functional specification 2) ill chosen internal structure 3) major programming languages are not the best tools 4) programs are not adequately tested 5) magnitude of the task of developing software is not fully recognized. He concludes by giving a six step method of design and coding which he feels would produce better quality software.

[Maes77] Does a review of decision table format to date and finds that non can adequately display sequential information. Important points include : 1) decision tables should be brought back because in our attempt to satisfy the constraints of the old style computer languages (ie. strict flow chart) we were forced to distort the algorithms whereas decision tables are better suited to distinguish the causal relations included in the original problem statement 2) useless to attempt to optimize programs by converting them to decision tables and then back because to really optimize you would have to get rid of the constraints laid down by the control structure of the original program . So Iong as decision tables are not improved they should be used for generating test data predicates and for problem statement.

[McTa80] The author describes the work of Zolnowski and Simmons, he states that although he feels that their work is probably a good measure of program complexity, their measure suffers from four major practical defects : 1) needs reference base 2) measure not static because the reference base of programs can and does change 3) clumsy because you have to have a base before you can use the measure for an individual program 4) cannot be applied to components (ie. procedures or modules). The author then reviews the five criteria used by Zolnowski to identify features which would allow them to qualify as program complexity features. Examples of some of the ones chosen by Zolnowski are : 1) number of imperative instructions 2) average depth of if nesting 3) total lines of code 4) number of entry points 5) total number of parameters passed 6) average number of lines of code jumped by forward transfers of control. The author summarizes the Zolnowski and Simmons measure as being a measure of the deviance or difference of the program being measured from the reference base (ie. from "past programming practices"). McTap's contribution is to add a sixth criteria (to five already used) for qualification as an appropriate feature to measure program complexity.

[Mill75] A rather straight forward paper which dwells on the use of structural based testing, program proving and the automation thereof. Also spends some time on semantic complexity. Makes the following very important statement: "use of simple programming primitives leads to significantly simpler backtracking operations".

[Salt76] A methodology is presented for transforming system requirements into functional structure and system operating rules. Talks about system decomposed into four structural elements: 1) functions 2) control 3) functional flows 4) data. The paper basically describes the following flows (system requirements) ---> (data processing req) ---> (algorithm construction) ---> (code implementation). Spends a good deal of time talking about finite state machines (FSM) to model control elements. Topics listed include the following : Structured FSM's, start end states, Hierarchy of FSM's, FSM synthesis etc. Brief section on properties of control structure (consistent,complete,reachable) and their relationship to FSM's. The appendix to the paper contains a description of his algorithm to structure and stratify functions using FSM's.

[Schn79] The author attempts to relate structural characteristics of a program (ie. via flow graphs) to a complexity measure which he hopes is a valid indicator (strongly related) to program developement time, program quality and ease of debugging. He uses graph theory to analyse programs where he makes use of adjacency matricies, reachability matricies as well as fundamental circuit matricies. He also makes use of trees which are derived from the graphs by deleteing edges so that no circuits exist. Makes mention of the fact that path analysis (ie complete number of paths) can be found by performing ring sum operations on the independant circuits as described by the fundamental circuit matrix. ABL normal tree form procedure yields the same paths. Author states that all of the techniques mentioned could be completely automated if the problems could be put in decision table form.

[Schn81] This paper is essentially a review of some of the literature on what makes a good complexity measure. The motivation is again attributed to high maintenance costs. Problem that exists is to isolate those features which affect the "complexity" measures of a program. The author then questions some of the sources of performance differences (ie. programmer related or task related). The authors then point out three very important experiments which could be used to support my views of complexity being related to the conceptual complexity of both the program and the problem. It was noted that there was a negative correlation between three complexity measures : Halstead's E, McCabe's V(G, Mills/Halstead program length) and simple recall of the program statements. More importantly, however, is the fact that increasing positive correlations were found when these measures were tested against the time required to : a) modify a program b) debug a single error in the program c) construct a program. Under programmer factors it was reported that there was a bimodal distribution (expert, novice) for bar graph (ie frequency vs correct answers) of both short and long programs . It was concluded that programmer attributes were a significant factor in these experiments. Under task factors he mentions Moher and Schneider's proposed five information classifications : 1) high level semantics 2) low level semantics 3) data structure 4) program flow 5) program modification. Their conclusions on this area are that these data reveal the inadequacy of complexity measures based on program factors alone. An important note is that their experiments did show that increased complexity of program flow did reduce performance of both experts and novices, by about the same amount. General conclusion is that it is futile to attempt to encapsulate the concept of complexity within a single measure, at least for those they studied. Implications are that future research should involve a complexity measure which takes into account the following : 1) program 2) programmer 3) programmer and programming task interrelationships.

[Shep79] . This is an experimental study which attempted to check the relationship of Halstead's E and McCabe's V(G) to three dependant variables : 1) percent correct recall of program statements 2) accuracy of program modifications that were performed 3) time spent implementing the modifications. It was found that contol flow complexity was significantly related to the first two but not the last variable. Basic conclusions were that although there was empirical evidence to support these complexity metrics the assessment of the psychological complexity of software requires more than just a count of operators, operands and contol paths. Also, they state that structured coding techniques and comments reduce the cognitive load of the programmer in ways unassessed by the metrics. This statement supports statements made by others who suggest that "chunking" is an important aspect of human understanding of computer software. This might also suggest that if programmers are allowed to create contol structures which allow them to conceptualize the problems better then we would probably see an improvement in performance on both program construction and maintenance.

[Shne77] The gist of this paper is that "comprehension" has become a vital component in assuring quality in debugging and maintenance tasks. The author gives a short defintion of comprehension and then reviews some studies in the area. Interesting points are as follows : 1) being able to trace a program does not insure comprehension 2) as programmers become more experienced they improve their ability to recode ("chunking") syntax of a program into higher level internal semantic structures 3) semantic knowledge is language independent 4) syntax is retained only briefly. Results include finding that recall is related to modifiability, which suggests that one way of assessing ease of modification is to test for the ease of memorization. It was also noted that FORTRAN with its "concise and syntactically limited form" was better for recall then for COBOL with its "rich syntactic variability". This statement is interpreted as meaning that the greater the syntax requirements the harder it is to recode into higher level semantic structures and that therefore we should restrict the number of control structures which the programmer has to use.

[Sime77] The paper deals with an experimental comparison of what he calls "nest-ine" and "nest-be" structures. Results are as follows : 1) nesting languages seem to be better in terms of relating semantic information but predisposed to trivial syntax omissions 2) syntax mistakes are more likely in nest-be structures 3) redundancy of nest-ine structures make them easier to correct. Personally I feel that their results have been distorted by an interference effect (volume of text) which might have been relieved had the nest-be structures been properly indented. Another important point concerns the task of problem drafting which requires that a mental representation be translated into a programming language, the implication being that language should be as close as possible to the mental representation.

[Srin79] The basic premise of this paper is that the number of decision points in a program is not a true indicator of complexity because the blocks of code may or may not be highly interwoven structures. As should be intuitively obvious the more interwoven it is, the more knots it has hence the more complex it should be. These authors have presented an algorithm which restructures the programs so that they have as few knots as possible. This paper should be read immediately after the Woodward paper which introduces the concept of knots.

[Trac79] To bring his point across the author first describes how the human brain would look like if it was listed as specs for a new computer system (he points out that it has slow memory store, limited instruction and register set, loss of information in I/O buffers -- good point is that it has a virtually unlimited word size). He also points out that programming involves two types of thought processes 1) associated with problem solving 2) man/computer communication using conventional programming languages. States that human memory is the most significant aspect of human thought processess which affects the computer programmer. Other important statements include : 1) a program should be recognized as existing in different environments 2) the biggest pitfall affecting problem solving is fixation (note- in another reference to the effect that experienced programmers were the most resistant to change, probably in defence (subconciously) of their "chunking" ability which they have acquired through long hours of experience. Most people in computer science have one language which they prefer over all the rest 3) "elimination of bushy trees " is a good method of reducing complexity 4) describes a nine step process for describing what programming is. The emphasis of this paper is on improving quality of software by giving the programmer a methodology which would allows them to produce less complex programs, thereby enhancing readability and structure.

[Turn80] This paper is devoted to expounding the virtues of programs which have TREE like structures. The author makes many good points as to why these structures offer a better programming style and decreased complexity: 1) functional strength 2) heirarchial control 3) locality of reference. Some of the advantages of tree like structures : 1) easier to understand 2) high reliability maintenance is easier 4) complexity is decreased because we minimize the interconnections between modules. He even states desirable exceptions to using pure tree like structures, and implications of trees to testing and implementation.

[Wals79] This paper could have been called "In praise of McCabe's measure". He begins by making some good points about how humans must neccesarily break a problem down into smaller components so that the resulting programs will be testable and maintainable. This is essential because the high cost of software is due to it unreliability. The author claims that the measure is simple to use and allows users to compare alternate designs in search of "the best solution". This is accomplished by striving to eliminate "obscure structures, cumbersome decision-making processes and overly complicated control paths". The authors main conclusion is that these types of complexity measures should be applied at the design level because its effects "propagate through all the other phases". In summary he makes six recommendations as to why the McCabe measure should be used by software system builders.

[Wood79] This paper introduces the concept of knots as a complexity measure. The authors point out that a structured program will have zero essential knots. Two interesting statements made are : 1) languages with powerful control structures the layout is an important consideration 2) the McCabe measure does not show increasing complexity with increased nesting level whereas the knot number does.

[Zoln80] The authors make mention of some of the complexity measures available. The measures they cover are based on the following concepts:
control flow, module interaction, data reference, program contol, logical complexity, software science, composite measures of complexity. Authors not already covered in the Berlinger paper include the following : 1) Hansen (Pair(cylomatic number, operator count)) 2) Peterson (multiple entry loops) 3) Woodward (number of knots) 4) Cobb (cyclomatic + number of lines I/O code) 5) Glib (number of subsystems) 6) Thayer (composite measure). He proposes a seven step method for obtaining a complexity measure based on four categories of complexity variables : 1) instruction mix 2) data reference 3) interaction/interconnection 4) structure/control flow. After running tests on COBOL programs he concludes that "there is a large amount of data that indeed differentiates between programs within a language and across language usage".

BIBLIOGRAPHY

The items included in this bibliography represent the majority of papers read during the preparation of this thesis. Of these only the more relevant articles have been referred in the body of the text.

[Abra75] Abrahams, P., "Structured Programming Considered Harmful.", SIGPLAN Notices, Vol.10, No. 4, (April,1975) 13 - 24.

[Acke82] Ackerman, W.B. "Data Flow Languages.", Computer, Vol. 15, No. 2 (February,1982) 15 - 23.

[Adri82] Adrion, W.R., Branstad, M.A. and Cherniavsky, J.C. "Validation, Verification and Testing of Computer Software.", ACM Computing Surveys, Vol. 14, No. 2 (June,1982) 159 - 192.

[Alle76] Allen, F.E. and Cocke, J. "A Program Data Flow Analysis Procedure.", Communications of the ACM, Vol. 19, No. 3 (March,1976) 137 - 146.

[Arch82] Archer, Jr., J. and Conway, R. "Display Condensation of Program Text.", IEEE Transactions of Software Engineering, Vol. SE-8, No. 5 (September,1982) 526 - 527.

[Arsa79] Arsac, J. J. "Syntactic Source to Source Transforms and Program Manipulation.", Communications of the ACM, Vol. 22, No. 1 (January,1979) 43 - 53.

[Ashc71] Ashcroft, E. and Manna, Z. "The Translation of 'go to' Programs to 'while' Programs.", Proceedings of the 1971 IFIP Congress, Vol 1 (1971) 250 - 255.

[Back78] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.", Communications of the ACM, Vol. 21, No. 8 (August,1978) 613 - 641.

[Back82] Backus, J. "Function-level Computing.", IEEE Spectrum, Vol. 19, No. 8 (August,1982) 22 - 27.

[Bagg78] Baggi, D. L. and Shooman, M. L. "An Automatic Driver for Pseudo-Exhaustive Software Testing.", Proceedings of the COMPCON, (Spring,1978) 278 - 282.

[Bake72] Baker, F. T. "System Quality Through Structured Programming.", AFIPS Conference Proceedings, Vol. 41 (Fall,1972) 339 - 344.

[Bake79] Baker, A. L., "A Comparison of Measures of Control Flow Complexity.", IEEE, Proceedings of the COMPSAC (1979) 695 - 701.

[Band81a] Bandyopadhyay, S. K., "A Study on Program Level Dependency of Implemented Algorithms on Its Potential Operands." SIGPLAN Notices, Vol. 16, No. 2 (Febuary,1981) 18 - 25.

[Band81b] Bandyopadhyoy, S. K., "Theoretical Relationships Between Potential Operands and Basic Measurable Properties of Algorithm Structure.", SIGPLAN Notices, Vol. 16, No. 2 (February,1981) 26 - 34.

[Barn80] Barnhardt, R.S. "Implementing Relational Dta Bases.", Datamation, Vol. 26, No. 10 (October,1980) 161 - 172.

[Beck77] Beckman, A. "Comments Considered Harmful.", SIGPLAN Notices, Vol. 12, No. 4, (April,1977) 94 - 95.

[Bela81] Belady, L. A. "Complexity of Large Systems."; in Software Metrics: An Analysis and Evaluation, Frederick G. Sayward, Mary Shaw and Alan J. Perlis (eds.), MIT Press, Cambridge, Massachusetts (1981) 225 - 234.

[Belk76] Belkin, G. and Jaworski, W. M., "Towards Logic and Performance Analysis of Unwritten Programs.", Canadian Computer Conference, Session 1976 (March,1976) 314 - 331.

[Bens73] Benson, J. P., "Structured Programming Techniques.", IEEE, Symposium on Computer Software Reliability (May,1973) 143 - 147.

[Bens75] Benson, J. P., "Some Observations Concerning the Structure of Fortran Programs.", IEEE, Fifth Annual International Symposium on Fault-Tolerant Computing (1975) 156 - 159.

[Berg79a] Bergland, G. D. and Gordon, R. D. "Software Design Strategies.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 1 - 14.

[Berg79b] Bergland, G. D. "Structured Design Methodologies.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 162 - 181.

[Berl80] Berlinger, E. "An Information Theory Based Complexity Measure.", AFIPS Conference Proceedings, Vol. 49 (1980) 773 - 779.

[Bloo75] Bloom, A.M. "The 'Else' Must Go, Too.", Datamation, Vol. 21, No. 5 (May, 1975) 123 - 128.

[Boeh73] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment.", Datamation, Vol. 19, No. 5 (May,1973) 5 - 24.

[Boeh79] Boehm, B. W. "Software Engineering.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 225 - 240.

[Bohm66] Bohm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.", Communications of the ACM, Vol. 9, No. 5 (May,1966) 366 - 371.

[Bowi76] Bowie, W. S., "Applications of Graph Theory in Computer Systems.", International Journal of Computer and Information Sciences, Vol. 5, No. 1 (1976) 9 - 31.

[Bran81] Branstad, M. A. and Adrion, W. R., "NBS Programming Environment Workshop Report.", Software Engineering Notes, Vol.6, No. 4 (August,1981) 3 - 17.

[Broo74] Brooks, Jr., F. P., "The Mythical Man-Month.", Datamation, Vol. 20, No. 12 (December,1974) 44 - 52.

[Broo77] Brooks, R. "Towards a Theory of the Cognitive Processes in Computer Programming.", International Journal of Man - Machine Studies, Vol. 9 (1977) 737 - 751.

[Brow80] Brown, L. R. and Paige, M. R., "A Decision

Point Methodology for the Design of Data Reduction Systems.", IEEE, Proceedings of the COMPSAC (1980) 236 - 241.

[Brow81] Browne, J. C. and Shaw, M. "Toward a Scientific Basis for Software Evaluation.", in Software Metrics: An Analysis and Evaluation, Frederick G. Sayward, Mary Shaw and Alan J. Perlis (eds.), MIT Press, Cambridge, Massachusetts (1981) 19 - 42.

[Budd78a] Budd, T. A. and Lipton, R. J. and Sayward, F. G. "The Design of a Prototype Mutation System for Program Testing.", AFIPS Conference Proceedings, Vol. 47 (1978) 623 - 627.

[Budd78b] Budd, T. A. and Lipton, R. J. "Mutation Analysis of Decision Table Programs.", Proceedings of the 1978 Conference on Information Science and Systems, Twefth Conference (March,1978) 346 - 349.

[Budd80] Budd, T. A., DeMillo, R. A., Lipton, R. J. and Sayward, F. G. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs.", Seventh Annual ACM Symposium on Principles of Programming Language (January, 1980) 220 - 233.

[Butt75] Butterworth, R.A. "Structured Symbols.", Datamation, Vol. 21, No. 4 (April,1975) 79 -83.

[Cant71] Cantrell, H.N., King, J. and King, F.E.H. "Logic-Structure Table.", Communications of the ACM, Vol. 14, No. 6

(June,1971) 272 - 275.

[Cavo74] Cavouras, J. C., "On the Conversion of Programs to Decision Tables: Methods and Objectives.", Communications of the ACM, Vol. 17., No. 8 (August,1974) 456 - 462.

[Chan80] Chand, D.R. and Yadav, S. B. "Logical Construction of Software.", Communications of the ACM, Vol. 23, No. 10 (October,1980) 546 - 547.

[Chap78] Chapin, N. and Denniston, S. P., "Characteristics of a Structured Program.", SIGPLAN Notices, Vol. 13, No. 5 (May,1978) 36 - 45.

[Chap79] Chapin, N. "A Measure of Software Complexity.", AFIPS Conference Proceedings, Vol. 48 (1979) 995 - 1002.

[Chen78] Chen, E. T. "Program Complexity and Programmer Productivity.", IEEE Transactions of Software Engineering, Vol. SE-4, No. 3 (May,1978) 187 - 194.

[Chev78] Chevance, R. J. and Heidet, T., "Static Profile and Dynamic Behavior of COBOL Programs.", SIGPLAN Notices, Vol. 13, No. 4 (April,1978) 44 - 57.

[Chin79] Chinnaswamy, V. "Translation of Decision Tables into Computer Programs with the Use of Carnaugh Maps.", SIGPLAN Notices, Vol. 14, No.5 (May,1979) 21 - 23.

[Chry78] Chrysler, E. "The Impact of Program and Programmer Characteristics on Program Size.", AFIPS Conference Proceedings, Vol. 47, (1978) 581 - 587.

[Clar73] Clark, R. L. "A Linguistic Contribution to GOTO-less Programming.", Datamation, Vol. 19, No. 12 (December,1973) 62 - 63.

[Cock69] Cocke, J. and Miller, R. E., "Some Analysis Techniques for Optimizing Computer Programs.", Proceedings of the Second Hawaii International Conference of System Sciences (1969) 143 - 146.

[Codd82] Codd, E. F. "Relational Database: A Practical Foundation for Productivity.", Communication of the ACM, Vol. 25, No. 2 (February,1982) 109 - 117.

[Coop68] Cooper, D. C., "Some Transformations and Standard Forms of Graphs, With Applications to Computer Programs.", Machine Intelligence, Vol. 2 (1968) 21 - 32.

[Cowe78] Cowell, D. F., Gillies, D. F. and Kaposi, A. A., "Introduction to Flowgraph Schemas.", Proceedings of the 1978 Conference on Information Sciences and Systems (1978) 437 - 440.

[Culi79] Culik, K., "The Cyclomatic Number and the Normal Number of Programs.", SIGPLAN Notices, Vol. 14, No. 4 (April,1979) 12 - 17.

[Curt79] Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A. and Love, T. " Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, (March,1979) 96 - 104.

[Curt81] Curtis, B. "The Measurement of Software Quality and Complexity.", in Software

Metrics: An Analysis and Evaluation, Frederick G. Sayward, Mary Shaw and Alan J. Perlis (eds.), MIT Press, Cambridge, Massachusetts (1981) 203 - 224.

[Curt81] Curtis, B. "Experimental Evaluation of Software Characteristics.", in Software Metrics: An Analysis and Evaluation, Frederick G. Sayward, Mary Shaw and Alan J. Perlis (eds.), MIT Press, Cambridge, Massachusetts (1981) 61 -76.

[Dahl72] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. "Structured Programming.", Academic Press, New York (1972).

[Dail75a] Dailey, W. H. "On Generating Binary Decision Trees with Minimum Nodes.", SIGPLAN Notices, Vol. 10, No. 12 (December,1975) 14 - 21.

[Dail75b] Dailey, W. H. "A Tabular Approach to Program Optimization.", SIGPLAN Notices, Vol. 10, No. 11 (November,1975) 17 - 22.

[Date77] Date, C. J. (ed.) "An Introduction to Database Systems.", Addison-Wesley Publishing Company, London (1977).

[Davi73] Davis, G.A. "Psychology of Problem Solving: Theory and Prectice.", Basic Books, New York (1973).

[Davi82] Davis, A.L. and Keller, R.M. "Data Flow Program Graphs.", Computer, Vol. 15, No. 2 (February,1982) 26 - 41.

[DeBa78] DeBalbine, G. "MTR - A Tool for Displaying the Global Structure of Software Systems.", AFIPS Conference Proceedings, Vol. 47, (1978) 571 - 580.

[DeFe70] DeFerranti, B. Z. "Some Characteristics of the Human Brain in Terms of the Computer and its Relation to the Development of Software.", Man and Computers, Proceedings of the International Conference (1970) 102 - 109.

[DeMa79a] DeMarco, T. Structured Analysis and System Specification.", in Classics in Software Engineering, Yourdan, E. N. (ed.), New York (1979).

[DeMa79b] DeMarco, T. "Concise Notes on Software Engineering.", Yourdon Press Monograph, New York (1979).

[DeMi76] DeMillo, R. A., Eisenstat, S. C., and Lipton, R. J. "Can Structured Programs Be Efficient.", SIGPLAN Notices, Vol. 11, No. 10 (October,1976) 10 - 17.

[DeMi77] DeMillo, R. A., Lipton, R. J. and Perlis, A. J. "Social Processes and Proofs of Theorems and Programs.", Conference Record of the Fourth ACM Symposium on Principles of Programming Language (January,1977) 206 - 214.

[DeMi80] DeMillo, R. A. "Mutation Analysis As a Tool for Software Quality Assurance.", IEEE, Proceedings of the COMPSAC (1980) 390 - 393.

[Dijk65] Dijkstra, E. "Programming Considered as a Human Activity.", Proceedings of the 1965 IFIP Congress,

(1965) 213 - 217.

[Dijk68] Dijkstra, E. "Go To Statement Considered Harmful.", Communications of the ACM, Vol. 11, No. 3 (March,1968) 147 - 148.

[Dijk69] Dijkstra, E. "Structured Programming.", in Classics in Software Engineering, Edward N. Yourdon (ed.), Yourdon Press, New York (1979) .

[Dijk72] Dijkstra, E. "The Humble Programmer.", Communications of the ACM, Vol. 15, No. 10 (October,1972) 859 - 866.

[Dijk75] Dijkstra, E.W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs.", Communications of the ACM, Vol. 18, No. 8 (August,1975) 453 - 457.

[Dijk76] Dijkstra, E. W. "A Discipline of Programming.", Prentice-Hall, Englewood Cliffs, New Jersey (1976) Ch. 14 - 15.

[Dijk78] Dijkstra, E. W. "On a Political Pamphlet from the Middle Ages.", ACM SIGSOFT, Software Engineering Notes, Vol. 3, No. 2 (April,1978) 14 - 17.

[Dijk79] Dijkstra, E.W. "On the Interplay Between Mathematics and Programming.", Lecture Notes in Computer Science, Vol. 69 Program Construction, Bauer, F.L. and Broy, M. (eds.) (1979) 35 - 46.

[Dona73] Donaldson, J. R. "Structured Programming.", Datamation, Vol. 19, No. 12 (December,1973) 52 - 54.

[Duns78] Dunsmore, H. E. and Gannon, J. D. "Programming Factors - Language Features That Help Explain Programming Complexity.", ACM, Proceedings of the Seventy-Eigth Annual Conference (December,1978) 554 - 560.

[Dura81] Durann, J. W. and Ntafos, S. "A Report on Random Testing.", Fifth International Conference on Software Engineering, IEEE Proceedings (March, 1981) 179 - 183.

[Dwye79] Dwyer, B. "Lateral Programming - A Proven Technique.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 214 - 222.

[Dwye81] Dwyer, B. "One More Time - How to Update a Master File.", Communications of the ACM, Vol. 24, No. 1 (January,1981) 3 - 8.

[Dwye81] Dwyer, B. "File Updating - Still Once More.", Communications of the ACM, Vol. 24, No. 8 (August,1981) 536 - 539.

[Dzub65] Dzubak, B. J. and Warburton, C. R., "The Organization of Structured Files.",Communications of the ACM, Vol. 8, No. 7 (July,1965) 446 - 452.

[Ehrm80] Ehrman, J.R. "The New Tower of Babel.", Datamation, Vol. 26, No. 3 (March, 1980) 157 - 160.

[Elsh76a] Elshoff, J. L. "An Analysis of some Commercial PL/1 Programs.", IEEE Transactions on Software Engineering, Vol. SE-2, No. 2 (June,1976) 113 - 120.

[Elsh76b] Elshoff, J. L., "Measuring Commercial PL/1 Programs Using Halstead's Criteria.", SIGPLAN Notices, Vol. 11, No. 5 (May,1976) 38 - 46.

[Elsh77] Elshoff, J. L. "The Influence of Structured Programming on PL/1 Program Profiles.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 5 (September,1977) 364 - 368.

[Elsh78] Elshoff, J. L. and Marcotty, M., "On the Use of the Cyclomatic Number to Measure Program Complexity.", SIGPLAN Notices, Vol. 13, No. 12 (December,1978) 29 - 40.

[Enge71] Engeler, E., "Sturcture and Meaning of Elementary Programs.", in E. Engeler (ed.) : Symposium on Semantics of Algorithmic Languages (1971) 91 - 101.

[Enos81] Enos, J. C. and Van Tilburg, R. L., "Software Design." Computer, Vol. 14, No. 2 (February,1981) 61 - 83.

[Evan82] Evans, M. "Software Engineering for the Cobol Environment.", Communications of the ACM, Vol. 25, No. 12 (December,1982) 874 - 882.

[Fanc76] Fancott, T. and Jaworski, W. M., "Primitive Logic Constructs Considered Harmful in Structured Programs.", Canadian Computer Conference, Session 1976 (March,1976) 332 - 344.

[Feue79] Feuer, A. R. and Fowlkes, E. B. "Relating Computer Program Maintainability to Software Measures.", AFIPS Conference Proceedings, Vol. 48 (1979) 1003 - 1012.

[Fitt79] Fitter, M. and Green, T. R. G. "When do Diagrams Make Good Computer Languages?", International Journal of Man-Machine Studies, Vol. 11 (1979) 235 - 261.

[Fong76] Fong, A. C. and Ullman, J. D., "Finding the Depth of a Flow Graph.", Symposium on Theory of Computing (1976) 121 - 125.

[Form81] Forman, I. R. "On the Time Overhead of Counters and Traversal Markers.", Fifth International Conference on Software Engineering, IEEE Proceedings (March,1981) 164 - 169.

[Free76] Freeman, P., "Software Reliability and Design: A Survey.", IEEE Proceedings, 13th Design Automation Conference (June,1976) 74 - 84.

[Free80a] Freeman, P., "The Context of Design.", in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wasserman (eds.), IEEE Computer Society Press 3rd edition (1980) 2 - 4.

[Free80b] Freeman, P., "The Nature of Design.", in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wasserman (eds.), IEEE Computer Society Press 3rd edition (1980) 46 - 53.

[Fros75] Frost, D. "Psychology and Program Design.", Datamation, Vol. 21, No. 5 (May,1975) 137 - 138.

[Futa81] Futamura, Y., Kawai, T., Horikoshi, H.and Tsutsumi, M. "Development of Computer Programs by Problem Analysis Diagram (PAD).", Fifth International Conference on Software

Engineering, IEEE Proceedings (March,1981) 325 - 332.

[Gerh78] Gerhart, S. L. "A Proposal for Publication and Exchange of Program Proofs.", ACM, Software Engineering Notes, Vol. 3, No. 1 (January,1978) 7 - 17.

[Gill78] Gillies, D. F., Cowell, D. F. and Kaposi, A. A., "A Theory for Flowgraph Schemas.", Proceedings of the 1978 Conference on Information Sciences and Systems (1978) 441 - 446.

[Gill80] Gill, A., "Hierarchical Binary Search.", Communications of the ACM, Vol. 23, No. 5 (May,1980) 294 - 300.

[Glas80] Glass, R. L., "A Benefit Analysis of Some Software Reliability Methodologies.", ACM SIGSOFT, Software Engineering Notes, Vol. 5, No. 2 (April,1980) 26 - 33.

[Good77] Goodenough, J.B. and Gerhart, S.L. "Toward a Theory of Testing: Data Selection Criteria.", in Current Trends in Programming Methodology, Vol. II, Yeh, R.T. (ed.), Prentice-Hall, New Jersey (1977) 44 - 78.

[Gord79] Gordon, R.O., "Measuring Improvements in Program Clarity.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, (March,1979) 79 - 91.

[Grah82] Graham, A.K. "Software Design: Breaking the Bottleneck.", IEEE Spectrum, Vol. 19, No. 3 (March,1982) 43 - 50.

[Gray72] Gray, M. and London, K. "Documentation Standards.", Auerback Publishers, New York (1972).

[Grie79] Gries, D. "Current Ideas in Programming Methodology.", Lecture Notes in Computer Science, Bauer, F.L. and Broy, M. (eds.), Vol. 69, Program Construction, Springer-Verlag (1979).

[Grif79] Griffiths, S. M. "Design Methodologies - A Comparison.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 189 - 213.

[Gust77] Gustafson, D. A. "Contol Flow, Data Flow and Data Independence.", SIGPLAN Notices Vol. 12, No. 10 , (Oct ,1977) 13 - 19.

[Hals73] Halstead, M. and Bayer, R., "Algorithm Dynamics", Proceedings of the ACM (1973) 126 - 135.

[Harr81a] Harrison, W. and Magel, K., "A Topological Analysis of the Complexity of Computer Programs With Less Than Three Binary Branches.", SIGPLAN Notices Vol. 16, No. 4 (April,1981) 51 - 60.

[Harr81b] Harrison, W. A. and Magel, K. I., "A Complexity Measure Based on Nesting Level.", SIGPLAN Notices, Vol. 16, No. 7 (July,1981) 63 - 74.

[Hart80] Hart, R., "Pattern Analysis as a Tool for Inventing Algorithms.", Software Practice and Experience, Vol. 10, (1980) 405 - 417.

[Hech72] Hecht, M. S. and Ullman, J. D., "Flow Graph Reducibility.", SIAM Journal of Computing, Vol. 1, No. 2, (June,1972) 188 - 202.

[Hetz77] Hetzel, W. C. "The Future of Quality Software.", Proceedings of the COMPCON, (Spring,1977) 211 - 212.

[Hint81] Hinterberger, H. and Jaworski, W. M. "Controlled Program Design by Use of the ABL Programming Concept.", Angewandte Informatik (Applied Informatics), Weisbaden, Germany (July,1981) 302 - 310.

[Hoar69] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming."~~,~~ Communications of Computer Science, Vol. 12, No. 10 (October,1969) 576 - 583.

[Hoar72] Hoare, C. A. R. "The Quality of Software.", Software - Practice and Experience, Vol. 2 (1972) 103 - 105.

[Holt75] Holton, J.B. and Bryan, B. "Structured Top-Down Flow Charts.", Datamation, Vol. 21, No. 5 (May,1975) 80 - 84.

[Holt77] Holton, J.B. "Are the New Programming Techniques Being Used.", Datamation, Vol. 23, No. 7 (July,1977) 97 - 103.

[Hopk72] Hopkins, M. E. "A Case for the GOTO.", Proceedings of the 25th National ACM Conference (August,1972) 787 - 790.

[Horn75] Horn, R.E. "Information Mapping.", Datamation, Vol. 21, No. 1 (Januar5) 85

[Horv82] Horvath, A. "Modeling and Implementation of an Information System for the Control of Truancy in the Quebec Comprehensive High School.", Masters Thesis, Concordia University, Quebec (June,1982).

[Hous80] House, R. "Comments on Program Specification and Testing.", Communications of the ACM, Vol. 23, No. 6 (June,1980) 324 - 331.

[Howd82] Howden, W.E. "Contemporary Software Development Environments.", Communications of the ACM, Vol. 25, No. 5 (May,1982) 318 - 329.

[Hunt82] Hunt, J. W. "Programming Languages.", Computer, Vol. 15, No. 4 (April,1982) 70 - 88.

[Jawo82] Jaworski, W. M. and Williams, A. J. "Representation of Therapeutics Strategies in Dermatology.", Canadian Dermotological Association Conference, Edmonton (July,1982).

[Karp80] Karp, R. M., "A Note on the Application of Graph Theory to Digital Computer Programming.", Information and Control, Vol. 3, (1980) 179 - 190.

[Kenn77] Kennedy, K. and Zucconi, L., "Applications of a Graph Grammar for Program Control Flow Analysis.", Fourth ACM Symposium on Principles of Programming languages, (January,1977) 72 - 85.

[Kern74] Kernighan, B. W. and Plauger, P. J.

"Programming Style: Examples and Counterexamples.", ACM Computing Surveys, Vol. 6, No. 4 (December,1974) 303 - 319.

[Knut74] Knuth, D.E. "Computer Programming as an Art.", Communications of the ACM, Vol. 17, No. 12 (December,1974) 667 - 673.

[Knut79] Knuth, D. E. "Structured Programming with GOTO Statements.", in Classics in Software Engineering, Yourdon, E. N. (ed.), New York (1979).

[Kosa73] Kosaraju, S. R., "Analysis of Structured Programs.", ACM Symposium on Theory of Computing (1973) 240 - 252.

[Kosa76] Kosaraju, S. R., "On Structuring Flowcharts: Preliminary Version.", ACM Symposium on Theory of Computing (1976) 101 - 111.

[Kowa79] Kowalski, R. "Algorithm = Logic + Control.", Communications of the ACM, Vol. 22, No. 7 (July,1979) 424 - 436.

[Lask79] Laski, J. W., "On Readability of Programs with Loops.", SIGPLAN Notices, Vol 14. No. 1 (January,1979) 73 - 83.

[Laub82] Lauber, R.J. "Development Support Systems.", Computer, Vol. 15, No. 5 (May,1982) 36 - 46.

[Leav72] Leavenworth, B. M. "Programming With(out) the GOTO.", Proceedings of the 25th National ACM Conference (August,1972) 782 - 786.

[Lebe82a] Lebensold, J. "A Language for Describing Office Information Systems.", Masters Thesis, Concordia University, Quebec (December,1982).

[Lebe82b] Lebensold, J., Radhakrishnan, T. and Jaworski, W. M. "A Modelling Tool for Office Automation Systems.", ACM-SIGOA conference on Office Information Systems, Philadelphia, Pennsylvannia (June,1982).

[Ledg75] Ledgard, H. E. and Marcotty, M. "A Genealogy of Control Structures.", Communications of the ACM, Vol. 18, No. 11 (Novemeber,1975) 629 - 639.

[Leve82] Levene, A.A. and Mullery, G.P. "An Investigation of Requirement Specification Languages: Theory and Practice.", Computer, Vol. 15, No. 5 (May,1982) 50 - 59.

[Levy82] Levy, M.R. "Modularity and the Sequential File Update Problem.", Communications of the ACM, Vol. 25, No. 6 (June,1982) 362 - 367.

[Lew 82] Lew, A. "On the Emulation of Flowcharts by Decision Tables.", Communications of the ACM, Vol. 25, No. 12 (December,1982) 895 - 904.

[Lien81] Lientz, B.P. and Swanson, E.B. "Problems in Application Software Maintenance.", Communications of the ACM, Vol. 24, No. 11 (November,1981) 763 - 769.

[Lina82] Linares, J.. "A Comprehensive Support System for Microcode Generation.", Masters Thesis, Concordia University, Quebec (August,1982).

[Lind73] Lindsay, P.H. and Norman, D.A. "Human

Information Processing: An Introduction.", Academic Press, New York (1973).

[Lipt75] Lipton, R. J. and Eisenstat, S. C., "The Complexity of Control Structures and Data Structures.", ACM Symposium on Theory of Computing (1975) 186 - 193.

[Lisk72] Liskov, B. H. "A Design Methodology for Reliable Software Systems.", AFIPS Conference Proceedings, Vol. 41 (1972).

[Litv82] Litvin, Y. "Parallel Evolution Programming Language for Data Flow Machines.", ACM Sigplan Notices, Vol. 17, No. 11 (November,1982) 50 - 58.

[Love77] Loveman, D. B. "Program Improvement by Source-to-Source Transformation.", Journal of the Association for Computing Machinery, Vol. 24, No. 1 (January,1977) 121 - 145.

[Lowe70] Lowe, T. C. "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing.", Communications of the ACM, Vol. 13, No. 1 (January,1970) 3 - 9.

[Lowr69] Lowry, E.S. and Medlock, C.W. "Object Code Optimization.", Communications of the ACM, Vol. 12, No. 1 (January,1969) 13 - 22.

[Maes78] Maes, R. "On the Representation of Program Structures by Decision Tables : A Critical Assessment.", The Computer Journal, Vol. 21, No. 4 (1978) 290 - 295.

[Maes80] Maes, R. "An Algorithmic Approach to the

Conversion of Decision Grid Charts into Compressed Decision Tables.", Communications of the ACM, Vol. 23, No. 5 (May,1980) 286 - 293.

[Mage81] Magel, K., "Regular Expressions in a Program Complexity Metric.", SIGPLAN Notices, Vol. 16, No. 7 (July,1981) 61 - 65.

[Mart77] Martin, J. (ed.) "Computer Data-Base Organization.", Prentice-Hall, New Jersey (1977).

[Maye77] Mayer, R.E. "Thinking and Problem Solving: An Introduction to Human Cognition and Learning.", Scott, Foresman and Company, Illinois (1977).

[Maye79] Mayer, R. E. "A Psychology of Learning BASIC.", Communications of the ACM, Vol. 22, No. 11 (November,1979) 589 - 593.

[Maye81] Mayer, R. E. and Bayman, P. "Psychology of Calculator Languages: A Framework for Describing Differences in Users' Knowledge.", Communications of the ACM, Vol. 24, No. 8, (August,1981) 511 - 520.

[Maye81] Mayer, R.E. "The Psychology of How Novices Learn Computer Programming.", ACM Computing Surveys, Vol. 13, No. 1 (March,1981) 121 - 141.

[McCa76] McCabe, T. J., "A Complexity Measure.", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4 (December,1976) 308 - 320.

[McCr73] McCracken, D. "Revolution in Programming: An

Overview.", Datamation, Vol. 19, No. 12 (December,1973) 50 - 52.

[McDa80] McDaniel, M. (ed.) "Applications of Decision Tables.", Brandon Systems Press, New York (1980).

[McMa78] McMaster, C. L. "An Analysis of Algorithms for the Dutch National Flag Problem.", Communications of the ACM, Vol. 21, No. 10 (October,1978) 842 - 847.

[McTa80] McTap, J. L. "The Complexity of an Individual Program.", AFIPS Conference Proceedings, Vol. 49 (1980) 767 - 771.

[Mekl80] Mekly, L. J. and Yau, S. S. "Software Design Representation Using Abstract Process Networks,", IEEE Transactions on Software Engineering, Vol. SE-6, No. 5 (Sept,1980) 420 - 435.

[Mill63] Miller, J. C. and Maloney, C. J., "Systematic Mistake Analysis of Digital Computer Programs.", Communications of the ACM, Vol. 6, No. 2 (Febuary,1963) 58 - 63.

[Mill73] Miller, E. F. and Lindamood, G. E. "Structured Programming: Top - Down Approach.", Datamation, Vol. 19, No. 12 (December,1973) 55 - 57.

[Mill74a] Miller, Jr., E. F., Paige, M. R., Benson, J. P. and Wisehart, W. R., "Structural Techniques of Program Validation.", IEEE, Proceedings of the COMPCON (Spring,1974) 161 - 164.

[Mill74b] Miller, Jr., E. F., "Automatic Generation of Software Testcases.", Proceedings of 1974 European Computing Congress (May,1974) 1 - 12.

[Mill75] Miller, E. F. Jr. "Engineering Software for Testability.", Proceedings of the COMPCON, (Spring,1975) 7 - 10.

[Mont74] Montalbano, Michael "Decision Tables.", Science Research Associates Inc. (1974).

[Naur69] Naur, P., "Programming by Action Clusters.", BIT, Vol. 9, No. 3 (1969) 250 - 258.

[Naur75] Naur, P. "Programming Languages, Natural Languages, and Mathematics.", Communications of the ACM, Vol. 18, No. 12 (December,1975) 676 - 683.

[Newe72] Newell, A. and Simon, H.A. "Human Problem Solving.", Prentice-Hall, New Jersey (1972).

[Niev79] Nievergelt, J. and Weydert, J. "Sites, Modes and Trails: Telling the User of an Interactive System Where He Is, What He Can Do, and How to Get to Places.", ETH (January,1979) 1 - 12.

[Olde83] Oldehoeft, R.R. "Program Graphs and Execution Behavior.", IEEE Transactions on Software Engineering, Vol. SE-9, No. 1 (January,1983) 103 - 108.

[Orr 79a] Orr, K. T. "Structured Programming with Warnier-Orr Diagrams, Part 2: Coding the Program.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D.

Gordon (eds.), IEEE Computer Society Press (1979) 65 - 71.

[Orr 79b] Orr, K. T. "Introducing Structured Systems Design.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 72 - 82.

[Ovie80] Oviedo, E. I., "Control Flow, Data Flow and Program Complexity.", IEEE, Proceedings of the COMPSAC (1980) 146 - 152.

[Paig73] Paige, M. R. and Balkovich, E. E., "On Testing Programs.", IEEE Proceedings of Conference on Computer Software Reliability (1973) 23 - 27.

[Paig74a] Paige, M. R. and Benson, J. P., "The Use of Software Probes in Testing FORTRAN Programs.", Computer, Vol. 7, No. 7 (July,1974) 40 - 47.

[Paig74b] Paige, M. R., "Software Testing: An Overview.", IEEE, Fourth Annual International Symposium on Fault-Tolerant Computing (1974) 5-18 - 5-21.

[Paig75] Paige, M. R. "Program Graphs, an Algebra, and Their Implication for Programming.", IEEE Transactions on Software Engineering Vol. SE-1, No. 3 (September,1975) 286 - 291.

[Paig77] Paige, M. R. "On Partitioning Program Graphs.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 6, (November,1977) 386 - 393.

[Paig77] Paige, M. R. and Holthouse, M. A., "On Sizing Software Testing for Structured Programs.", IEEE Computer Society, Seventh Annual International Conference on Fault-Tolerant Computing (1977) 212.

[Paig78] Paige, M. R., "An Analytical Approach to Software Testing.", IEEE, Proceedings of the COMPSAC (1978) 527 - 532.

[Paig80] Paige, M. R., "A Metric for Software Test Planning.", IEEE, Proceedings of the COMPSAC (1980) 499 - 504.

[Pape80] Papentin, F. "On Order and Complexity. I. General Considerations.", Journal of Theoretical Biology, Vol. 87 (1980) 421 - 456.

[Parn72] Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules.", Communications of the ACM, Vol. 5, No. 12 (December,1972) 220 - 225.

[Parn79] Parnas, D. L. "Designing Software for Ease of Extension and Contraction.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2 (March,1979) 128 - 137.

[Pete73] Peterson, W. W., Kasami, T. and Tokura, N. "On the Capabilities of While, Repeat, and Exit Statements.", Communications of the ACM, Vol. 16, No. 8 (August,1973) 503 - 512.

[Pete79a] Peters, L. J. and Tripp, L. L., "A Model of Software Engineering.", IEEE Proceedings, Third International Conference on Software Engineering (May,1979) 4 - 10.

[Pete79b] Peters, L. J. and Tripp, L. L. "Comparing Software Design Methodologies.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 183 - 188.

[Pete81] Peters, L.J. "Software Design: Methods and Techniques.", Yourdon Press, New York (1981).

[Pgn 79] Pgn "A Note on the Psychology of Abstraction (PGN).", ACM SIGSOFT, Software Engineering Notes, Vol. 4, No. 1 (January,1979) 21.

[Phil77] Philippakis, A.S. "A Popularity Contest for Languages.", Datamation, Vol. 23, No. 12 (December,1977) 81 - 87.

[Poll65] Pollack, S. L. "Conversion of Limited-Entry Decision Tables to Computer Programs.", Communications of the ACM, Vol. 8, No. 11 (November,1965) 677 - 682.

[Poly65] Polya, G. "Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving.", John Wiley and Sons, New York (1965).

[Prat75] Pratt, T., "Four Models for the Analysis and Optimization of Program Control Structures.", ACM Symposium on Theory of Computing (1975) 167 - 176.

[Pres65] Press, L. I. "Conversion of Decision Tables to Computer Programs.", Communications of the ACM, Vol. 8, No. 6 (June,1965) 385 - 390.

[Pres75] Presser, L., "Structured Languages.", SIGPLAN Notices, Vol. 10, No. 7 (July,1975) 22 -

[Rama77] Ramamoorthy, C.V. and Ho, S.F. "Testing Large Software with Automated Software Evaluation Systems.", in Current Trends in Programming Methodology, Vol. II, Yeh, R.T. (ed.), Prentice-Hall, New Jersey (1977) 112 - 150.

[Rama82] Ramamoorthy, C.V. and Bastani, F.B. "Software Reliability - Status and Perspectives.", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4 (July,1982) 354 - 371.

[Robi77] Robinson, S. K. and Torsun, I. S., "The Automatic Measurement of the Relative Merits of Student Programs.", SIGPLAN Notices, Vol. 12, No. 4 (April,1977) 80 - 93.

[Ross75] Ross, D. T., Goodenough, J. B. and Irvine, C. A., "Software Engineering: Process, Principles, and Goals.", Computer, Vol. 8, No. 5 (May,1975) 54 - 64.

[Ross77a] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (January,1977) 16 - 34.

[Ross77b] Ross, D. T. and Schoman, K. E. "Structured Analysis for Requirements Definition.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (January,1977) 6 - 15.

[Rubi75] Rubinstein, M.F. "Patterns of Problem Solving.", Prentice-Hall, New Jersey (1975).

[Rudk79] Rudkin, R.I. and Shere, K.D. "Structured Decomposition Diagram: A New Technique for System Analysis.", Datamation, Vol. 25, No. 190 (October,1979) 130 - 146.

[Sack70] Sackman, H. "Man Computer Problem Solving: Experimental Evaluation or TimeSharing and Batch Processing.", Auerback Publisher, New York (1970).

[Salt62] Salton, G., "Manipulation of Trees in Information Retrieval.", Communications of the ACM, Vol. 5, No. 2 (Febuary,1962) 103 - 114.

[Salt76] Salter, K. G. "A Methodology for Decomposing System Requirements into Data Processing Requirements.", IEEE Proceedings, Second International Conference on Software Engineering.", (October,1976).

[Sayw81] Sayward, F. G. "Design of Software Experiments.", in Software Metrics: An Analysis and Evaluation, Frederick G. Sayward, Mary Shaw and Alan J. Perlis (eds.), MIT Press, Cambridge, Massachusetts (1981) 43 - 60.

[Sche63] Scheerer, M. "Problem Solving.", Scientific American, Vol. 204, No. 4 (April,1963) 118 - 128.

[Schn73] Schneck, P. B., "A Survey of Compiler Optimization Techniques." Proceedings of the ACM, (1973) 106 - 113.

[Schn79] Schneidewind, N. F. "Software Metrics for Aiding Program Development and Debugging.", AFIPS Conference Proceedings, Vol. 48 (1979) 989 - 994.

[Schn79] Schneidewind, N. F. and Hoffman, H. M. "An Experiment in Software Error Data Collection and Analysis.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 3 (May,1979) 277 - 286.

[Schn81] Schneider, G. M. and Sedlmeyer, R. L. and Kearney, J. "On the Complexity of Measuring Software Complexity.", AFIPS Conference Proceedings, Vol. 50 (1981) 317 - 322.

[Shaw80] Shaw, M. "The Impact of Abstraction Concerns on Modern Programming Languages.", Proceedings of the IEEE, Vol. 68, No. 9 (September,1980) 1119 - 1130.

[Shei81] Sheil, B.A. "The Psychological Study of Programming.", ACM Computing Surveys, Vol. 13, No. 1 (March,1981) 101 - 120.

[Shep79] Sheppard, S. B. and Curtis, B. and Milliman, P. and Borst, M. A. and Love, T. "First-year Results from a Research Program on Human Factors in Software Engineering.", AFIPS Conference Proceedings, Vol. 48 (1979) 1021 - 1027.

[Shep81] Sheppard, S.B., Kruesi, E. and Curtis, B. "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications.", IEEE Fifth International Conference on Software Engineering (March,1981) 207 - 214.

[Shne75] Shneiderman, B. "Cognitive Psychology and Programming Language Design.", SIGPLAN Notices Vol. 10, No. 7, (July,1975) 46 - 47.

[Shne77] Shneiderman, B. "Measuring Computer Program

Quality and Comprehension.", INternational Journal of Man - Machine Studies, Vol. 9, (1977) 465 - 478.

[Shne79] Shneiderman, B. and Mayer, R. "Syntactical/Semantic Interactions in Programmer Behavior: A Model and Experiment Results.", International Journal of Computer and Information Sciences, Vol. 8, No. 3 (1979) 219 - 238.

[Shne82] Shneiderman, B. "Control Flow and Data Structure Documentation: Two Experiments.", Communications of the ACM, Vol. 25, No. 1 (January,1982) 55 - 63.

[Shoo77] Shooman, M. and Laemmel, A. "Statistical Theory of Computer Programs - Information Content and Complexity.", Proceedings of the COMPCON, (Fall,1977) 341 - 347.

[Sime77] Sime, M. E. and Green, T. R. G. and Guest D. J. "Scope Marking in Computer Conditionals - A Psychological Evaluation.", International Journal of Man - Machine Studies, Vol. 9 (1977) 107 - 118.

[Simo73] Simon, H. A. "The Structure of Ill Structured Problems.", Artificial Intelligence, Vol. 4 (1973) 181 - 201.

[Site78] Sites, R. L. "Programming Tools: Statement Counts and Procedure Timings.", SIGPLAN Notices, Vol.13, No. 12 (December,1978) 98 - 101.

[Smyt74] Smyth, M. B., "Unique-Entry Graphs, Flowcharts, and State Diagrams.", Information and Control, Vol. 25 (1974) 20 -29.

[Srin79] Srinivasan, B. and Gopalakrishna, V. "Control Flow Complexity and Structuredness of Programs.", SIGPLAN Notices, Vol. 14, No. 11, (Nov. 1979) 110 - 115.

[Stam81] Stamen, J. and Costello, W. "Evaluating Database Languages.", Datamation, Vol. 27, No. 8 (August,1981) 116 - 122.

[Stev79] Stevens, W. P., Myers, G. J. and Constantine, L. L. "Structured Design.", in Tutorial: Software Design Strategies, Glenn D. Bergland and Ronald D. Gordon (eds.), IEEE Computer Society Press (1979) 84 - 92.

[Stic78] Stickney, M. E. "An Application of Graph Theory to Software Test Data Selection.", ACM Proceedings of the Software and Assurance Workshop: Functional and Performance Issues." (November,1978) 111 - 115.

[Stig74] Stigall, P. D. and Tasar, O. "A Review of Directed Graphs as Applied to Computers.", Computer, Vol. 7, No. 10 (October,1974) 39 - 47.

[Tani81] Tanik, M. M., "Prediction Models for Cyclomatic Complexity.", SIGPLAN Notices, Vol. 16, No.4 (April,1981) 89 - 97.

[Tarj73] Tarjan, R., "Testing Flow Graph Reducibility.", ACM Symposium on Theory of Computing, (1973) 96 - 107.

[Teic77] Teichroen, D. and Hershey, III, E. A. "A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (January,1977) 41 - 48.

[Thur73] Thurner, R. and Bauknecht, K. "Procedural Decision Tables and Their Implementation.", International Computing Symposium (1973) 259 - 263.

[Trac79] Tracz, W. J. "Computer Programming and the Human Thought Process.", Software - Practice and Experience, Vol. 9, (1979) 127 - 137.

[Turn80] Turner, J. "The Structure of Modular Programs.", Communications of the ACM, Vol. 23, No. 5 (May,1980) 272 - 277.

[Wals79] Walsh, T. J. "A Software Reliability Study Using a Complexity Measure.", AFIPS Conference Proceedings, Vol. 48 (1979) 761 - 767.

[Wart79] Wartak, J. and Miller, T.B. "Decision Tables for Planning Drug Therapy.", Drug Intelligence and Clinical Pharmacy, Vol 13 (February,1979) 100 - 104.

[Wass78] Wasserman, A. I. and Freeman, P., "Software Engineering Education: Status and Prospects.", Proceedings of the IEEE, Vol. 66, No. 8 (August,1978) 445 - 451.

[Wass80a] Wasserman, A. I., "Information System Design Methodology.", Journal of the American Society for Information Science, Vol. 31, No. 1, (January,1980) 25 - 44.

[Wass80b] Wasserman, A. I., "A Srategy for Improving Software Development Practices.", in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wasserman (eds.), IEEE Computer Society Press 3rd edition (1980) 440 - 444.

[Wass82] Wasserman, A. I. "The Future of Programming.", Communications of the ACM, Vol. 25, No. 3 (March,1982) 196 - 206.

[Wats82] Watson, I. and Gurd, J. "A Practical Data Flow Computer.", Computer, Vol. 15, No. 2 (February,1982) 51 - 57.

[Webs82] Webster, R. and Miner, L. "Expert Systems Programming Problem-Solving.", Technology, Vol. 2, No. 1 (January/February,1982) 62 - 73.

[Wegn73] Wegner, E., "Tree-Structured Programs.", Communications of the ACM, Vol. 16, No. 11 (Novbember, 1973) 704 - 705.

[Wein71] Weinberg, G.M. "The Psychology of Computer Programming.", Van Nostrand Reinhold Company, New York (10971).

[Weis81] Weiser, M. "Program Slicing.", IEEE, Fifth International Conference on Software Engineering (March, 1981) 439 - 449.

[Wils81] Wilson, L. "Death, Taxes, and DP Documentation.", Datamation, Vol. 27, No. 2 (February,1981) 73 - 76.

[Wino77] Winograd, T. "Beyond Programming Languages.", Communications of the ACM, Vol. 22, No. 7 (July,1977) 391 - 401.

[Wirt71] Wirth, N. "Program Development by Stepwise Refinement.", Communications of the ACM, Vol. 14, No. 4.(April,1971) 413 - 419.

[Wirt74] Wirth, N. "On the Composition of Well Structured Programs.", Computing Surveys, Vol. 6, No. 4 (December,1974) 247 - 259.

[Wood79] Woodward, M. R. and Hennell, M. A. and Hedley, D. "A Measure of Control Flow Complexity in Program Text.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 1 (January,1979) 45 - 50.

[Wood81] Woodfield, S. N., Dunsmore, H. E. and Shen, V. Y. "The Effect of Modularization and Comments on Program Comprehension.", IEEE, Fifth International Conference on Software Engineering (March,1981) 215 - 223.

[Wulf72] Wulf, W. A. "A Case Against the GOTO.", Proceedings of the 25th National ACM Conference (August,1972) 791 -797.

[Yau 80] Yau, S. S. and Grabow, P. C., "A Model for Representing the Control Flow and Data Flow of Program Modules.", IEEE, Proceedings of the COMPSAC (1980) 153 - 160.

[Yau 81] Yau, S. S. and Grabow, P. C., "A Model for Representing Programs Using Hierarchical Graphs.", IEEE Transactions on Software Engineering, Vol. SE-7, No. 6 (November,1981) 556 - 576.

[Yode78] Yoder, C. M. and Schrag, M. L., "Nassi-Shneiderman Charts an Alternative to Flowcharts for Design.", Proceedings, ACM SIGSOFT/SIGMETRICS Software and Assurance Workshop (November,1978).

[Your79] Yourdon, E. N. "Classics in Software Engineering.", Yourdon Press, New York (1979) .

[ZeiI81] Zeil, S. J. and Whit, L. J. "Sufficient Test Sets for Path Analysis Testing Strategies.", Fifth International Conference on Software Engineering, IEEE Proceedings (March,1981) 184 - 191.

[Zoet79] Zoethout, T. A. and Stein; E. I. N., "Why the GOTO Is Harmful", SIGPLAN Notices, Vol. 14, No. 1 (January,1979) 116 - 122.

[Zoln80] Zolnowski, J. and Simmons, D. B. "Measuring Program Complexity in a COBOL Enyironment.", AFIPS Conference Proceedings, Vol. 49 (1980) 757 - 766.

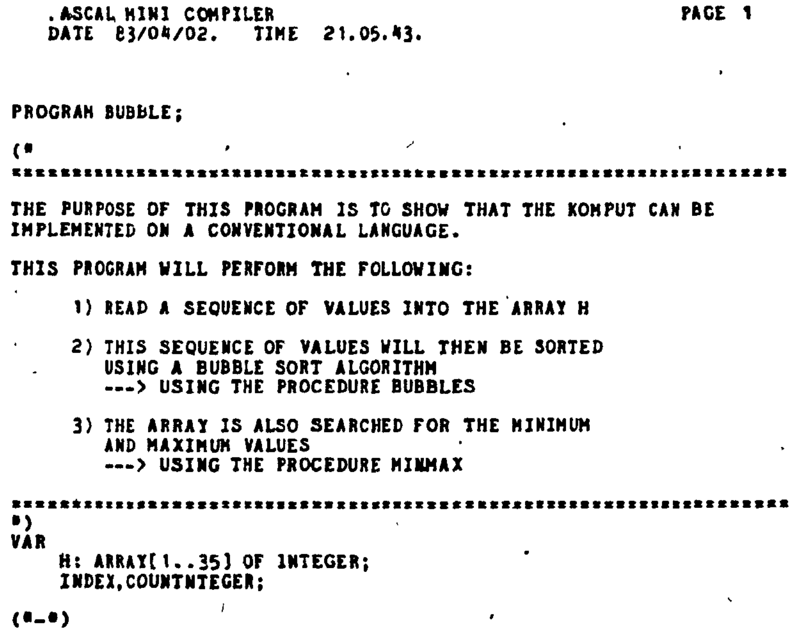
[Zweb77] Zweben, S. H. "A Study of the Physical Structure of Algorithms", IEEE Transactions on Software Engineering, Vol. SE-3, No. 3 (May,1977) 250 - 258.

[Zweb79] Zweben, S. H. and Halstead, M. H. "The Frequency Distribution of Operators in PL/1 Programs.", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2 (March,1979) 91 - 95.

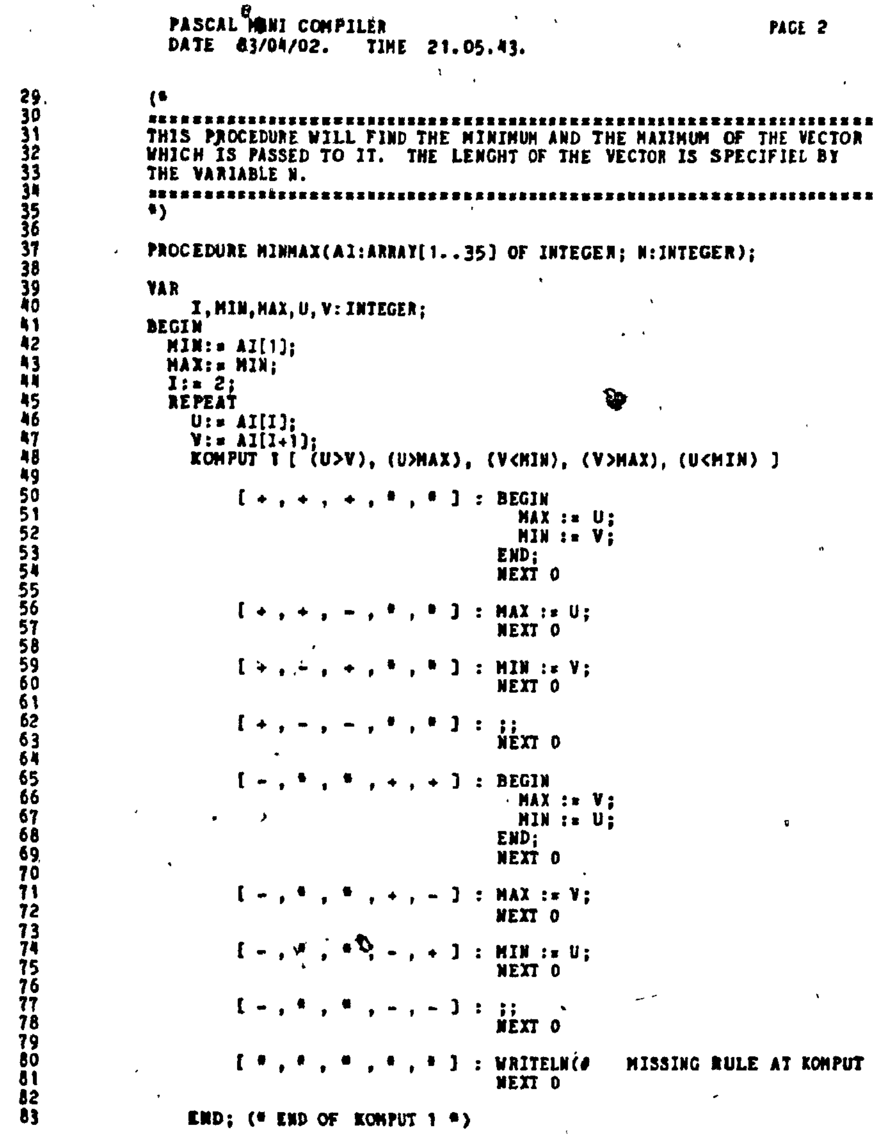
[Zweb79] Zweben, S. H. and Fung, K. C., "Exploring Software Science Relations in COBOL and APL.", IEEE, Proceedings of the COMPSAC (1979) 702 - 707.

APPENDIX A

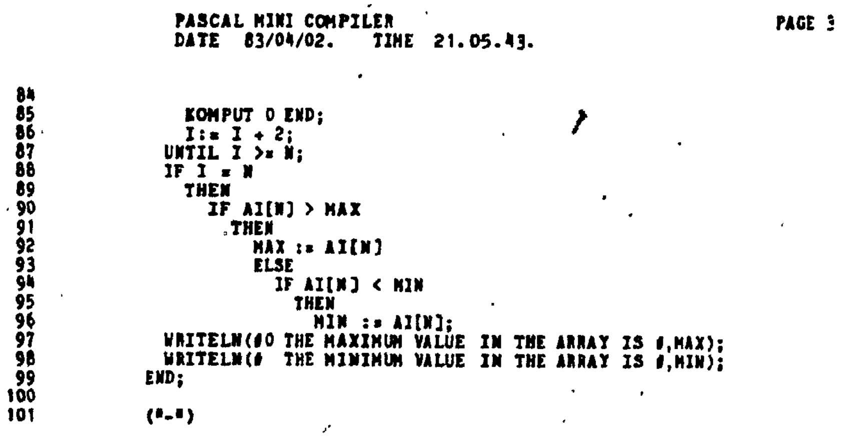
This appendix contains a sample program demonstrating an implementation of the KOMPUT construct on a compiler which executes a small subset of the PASCAL language. The compiler was originally written (by the author) for the course: COMPILER CONSTRUCTION (N 743) and was latter expanded to include the KOMPUT construct.



```
.ASCAL MINI COMPILER                                          PAGE 1
DATE 83/04/02.   TIME  21.05.43.


PROGRAM BUBBLE;

(*
*****************************************************************

THE PURPOSE OF THIS PROGRAM IS TO SHOW THAT THE KOMPUT CAN BE
IMPLEMENTED ON A CONVENTIONAL LANGUAGE.

THIS PROGRAM WILL PERFORM THE FOLLOWING:

      1) READ A SEQUENCE OF VALUES INTO THE ARRAY H

      2) THIS SEQUENCE OF VALUES WILL THEN BE SORTED
         USING A BUBBLE SORT ALGORITHM
         ---> USING THE PROCEDURE BUBBLES

      3) THE ARRAY IS ALSO SEARCHED FOR THE MINIMUM
         AND MAXIMUM VALUES
         ---> USING THE PROCEDURE MINMAX

*****************************************************************
*)
VAR
    H: ARRAY[1..35] OF INTEGER;
    INDEX,COUNTNTEGER;

(*-*)
```



```
PASCAL MINI COMPILER                                    PAGE 2
DATE  83/04/02.   TIME  21.05.43.

(*
***************************************************************
THIS PROCEDURE WILL FIND THE MINIMUM AND THE MAXIMUM OF THE VECTOR
WHICH IS PASSED TO IT.  THE LENGHT OF THE VECTOR IS SPECIFIEL BY
THE VARIABLE N.
***************************************************************
*)

PROCEDURE MINMAX(AI:ARRAY[1..35] OF INTEGER; N:INTEGER);

VAR
    I,MIN,MAX,U,V:INTEGER;
BEGIN
  MIN:= AI[1];
  MAX:= MIN;
  I:= 2;
  REPEAT
    U:= AI[I];
    V:= AI[I+1];
    KOMPUT 1 [ (U>V), (U>MAX), (V<MIN), (V>MAX), (U<MIN) ]

        [ + , + , + , * , * ] : BEGIN
                                  MAX := U;
                                  MIN := V;
                                END;
                                NEXT 0

        [ + , + , - , * , * ] : MAX := U;
                                NEXT 0

        [ + , - , + , * , * ] : MIN := V;
                                NEXT 0

        [ + , - , - , * , * ] : ;;
                                NEXT 0

        [ - , * , * , + , + ] : BEGIN
                                  MAX := V;
                                  MIN := U;
                                END;
                                NEXT 0

        [ - , * , * , + , - ] : MAX := V;
                                NEXT 0

        [ - , * , * , - , + ] : MIN := U;
                                NEXT 0

        [ - , * , * , - , - ] : ;;
                                NEXT 0

        [ * , * , * , * , * ] : WRITELN(#   MISSING RULE AT KOMPUT
                                NEXT 0

    END; (* END OF KOMPUT 1 *)
```



```
PASCAL MINI COMPILER                                    PAGE 3
DATE  83/04/02.   TIME  21.05.43.


    KOMPUT 0 END;
    I:= I + 2;
  UNTIL I >= N;
  IF I = N
    THEN
      IF AI[N] > MAX
        THEN
          MAX := AI[N]
          ELSE
            IF AI[N] < MIN
              THEN
                MIN := AI[N];
  WRITELN(#0 THE MAXIMUM VALUE IN THE ARRAY IS #,MAX);
  WRITELN(#  THE MINIMUM VALUE IN THE ARRAY IS #,MIN);
END;

(*-*)
```

```
(*
*****************************************************************
THIS PROCEDURE WILL SORT THE ARRAY "LIST" USING THE BUBBLE SORT
METHOD.  THE NUMBER OF ELEMENTS IN THE ARRAY IS CONTAINED IN THE
VARIABLE MAX.  THE ARRAY IS OF VARIABLE TYPE PARAMETER.
*****************************************************************
*)

PROCEDURE BUBBLES(VAR LIST:ARRAY [1..35] OF INTEGER; MAX:INTEGER);

VAR
    I,J,TEMP:INTEGER;
BEGIN
MINMAX(LIST,MAX);

KOMPUT 1 [TRUE]
   [ + ] : BEGIN
              I:= 2;
              J:= MAX;
           END;
           NEXT 2
         END;

KOMPUT 2 [ LIST[J-1] > LIST [J] ]
   [ + ] : BEGIN
              TEMP:= LIST[J-1];
              LIST[J-1]:= LIST[J];
              LIST[J]:= TEMP;
              J:= J - 1;
           END;
           NEXT 3
   [ - ] : J:= J - 1;
           NEXT 3
         END;

KOMPUT 3 [ J  < I]
   [ + ] : I:= I + 1;
           NEXT 4
   [ - ] : ;;
           NEXT 2
         END;

KOMPUT 4 [ I  > MAX ]
   [ + ] : ;;
           NEXT 0
   [ - ] : J:= MAX;
           NEXT 2
         END;

KOMPUT 0 END;

END;
(*-*)
```

```
PASCAL MINI COMPILER                                     PAGE 5
DATE  83/04/02.   TIME  21.05.43.


             (*  M A I N   P R O G R A M  *)

         BEGIN
           READ(INDEX);
           WRITELN(#1 THE NUMBER OF ELEMENTS TO BE READ ARE#,INDEX:5);
           COUNT:= 1;
           REPEAT
             READ(H[COUNT]);
             WRITELN(#  THE #,COUNT:2,# ELEMENT READ IS #,H[COUNT]:5);
             COUNT:= COUNT + 1;
           UNTIL COUNT > INDEX;
           BUBBLES(H,INDEX);
           COUNT := 1;
           WRITELN(#0     THE SORTED ARRAY IS AS FOLLOWS#);
           WRITELN(#      ------------------------------#);
           WRITELN(#0#);
           REPEAT
             WRITELN(#  ARRAY ELEMENT #,COUNT:2,# IS EQUAL TO #,H[COUNT]:5);
             COUNT := COUNT + 1;
           UNTIL COUNT > INDEX;
         END.

```

```
PASCAL MINI COMPILER                                     PAGE 6
DATE  83/04/02.   TIME  21.05.43.


          GLOBAL VARIABLES FOR PROGRAM BUBBLES

              COUNT            -----> INTEGER
              H                -----> INTEGER ARRAY    [  1 .. 35 ]
              INDEX            -----> INTEGER

          VARIABLES LOCAL TO PROCEDURE MINMAX

              I                -----> INTEGER
              MAX              -----> INTEGER
              MIN              -----> INTEGER
              U                -----> INTEGER
              V                -----> INTEGER
          PARAMETER VARIABLES FOR PROCEDURE MINMAX

              AI               -----> INTEGER ARRAY    [  1 .. 35 ]
              N                -----> INTEGER

          VARIABLES LOCAL TO PROCEDURE BUBBLES

              I                -----> INTEGER
              J                -----> INTEGER
              TEMP             -----> INTEGER
          PARAMETER VARIABLES FOR PROCEDURE BUBBLES

              LIST             -----> INTEGER ARRAY    [  1 .. 35 ]
              MAX              -----> INTEGER
COMPILATION TIME ----->       1345 MILLISECONDS
```

```
THE NUMBER OF ELEMENTS TO BE READ ARE 21
THE   1 ELEMENT READ IS     -5
THE   2 ELEMENT READ IS     34
THE   3 ELEMENT READ IS     52
THE   4 ELEMENT READ IS     65
THE   5 ELEMENT READ IS    765
THE   6 ELEMENT READ IS    123
THE   7 ELEMENT READ IS      0
THE   8 ELEMENT READ IS   -376
THE   9 ELEMENT READ IS     67
THE  10 ELEMENT READ IS    -11
THE  11 ELEMENT READ IS    -76
THE  12 ELEMENT READ IS    -32
THE  13 ELEMENT READ IS     43
THE  14 ELEMENT READ IS     56
THE  15 ELEMENT READ IS     71
THE  16 ELEMENT READ IS     98
THE  17 ELEMENT READ IS    194
THE  18 ELEMENT READ IS     63
THE  19 ELEMENT READ IS     22
THE  20 ELEMENT READ IS      1
THE  21 ELEMENT READ IS     26

THE MAXIMUM VALUE IN THE ARRAY IS         765
THE MINIMUM VALUE IN THE ARRAY IS        -376

     THE SORTED ARRAY IS AS FOLLOWS
     ------------------------------

ARRAY ELEMENT   1 IS EQUAL TO  -376
ARRAY ELEMENT   2 IS EQUAL TO   -76
ARRAY ELEMENT   3 IS EQUAL TO   -32
ARRAY ELEMENT   4 IS EQUAL TO   -11
ARRAY ELEMENT   5 IS EQUAL TO    -5
ARRAY ELEMENT   6 IS EQUAL TO     0
ARRAY ELEMENT   7 IS EQUAL TO     1
ARRAY ELEMENT   8 IS EQUAL TO    22
ARRAY ELEMENT   9 IS EQUAL TO    26
ARRAY ELEMENT  10 IS EQUAL TO    34
ARRAY ELEMENT  11 IS EQUAL TO    43
ARRAY ELEMENT  12 IS EQUAL TO    52
ARRAY ELEMENT  13 IS EQUAL TO    56
ARRAY ELEMENT  14 IS EQUAL TO    63
ARRAY ELEMENT  15 IS EQUAL TO    65
ARRAY ELEMENT  16 IS EQUAL TO    67
ARRAY ELEMENT  17 IS EQUAL TO    71
ARRAY ELEMENT  18 IS EQUAL TO    98
ARRAY ELEMENT  19 IS EQUAL TO   123
ARRAY ELEMENT  20 IS EQUAL TO   194
ARRAY ELEMENT  21 IS EQUAL TO   765
                EXECUTION TIME ----->        374 MILLISECONDS
```