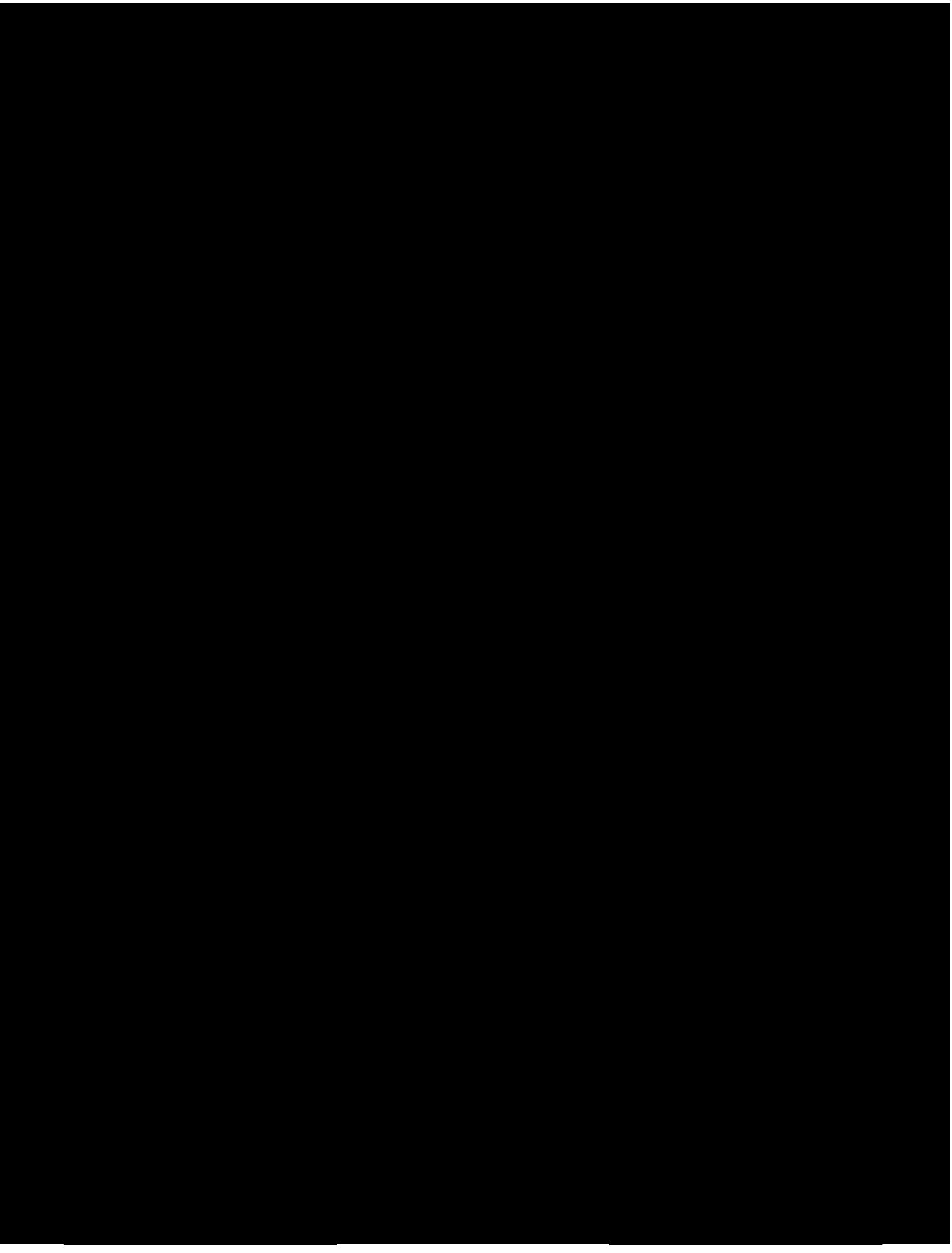


This blank page was inserted to preserve pagination.

COST ANALYSIS OF DEBUGGING SYSTEMS

Bruce P. Lester

September 1971

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

COST ANALYSIS OF DEBUGGING SYSTEMS*

Abstract

A general method is presented for performing cost analysis of interactive debugging systems. The method is based on an abstract model of program execution. This model is derived from the interpreter used in the Vienna method of semantic definition of PL/I. A brief discussion of the overall operation and significance of the Vienna interpreter is included.

Four assumptions are made which allow execution times to be calculated for algorithms of the Vienna interpreter. A notion of absolute cost is developed which requires the use of these execution times for cost analysis of features of debugging systems. A set of eight interactive debugging operations is thoroughly analyzed using the method of cost analysis. Some overall conclusions are drawn about the relative costs of various types of debugging operations and some suggestions are made for minimal cost debugging system design.

---

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science, January 1971.

Acknowledgment

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001.

TABLE OF CONTENTS

		Page
I.	Introduction	4
II.	History of Debugging Systems	8
III.	The Vienna Method of Semantic Definition	14
	A. Why the Vienna Method?	15
	B. Abstract Syntax	16
	C. State of the Machine	25
	D. The Interpreter	30
IV.	Execution Time for the Vienna Machine	36
	A. Tree Operations	37
	B. Propositions (predicates)	40
	C. Implementation Dependencies	43
	D. Parameterization of Algorithms	44
	E. Examples	46
V.	Overview of Cost Analysis	54
	A. Incremental vs. Absolute Cost.	54
	B. Debugging Systems	57
	C. The Principle of Cost Analysis	59
VI.	Cost Analysis of Debugging Operations	61
	A. The Debugging Operations	61
	B. Cost Analysis	65
VII.	Conclusions	85
Appendix A - List of Debugging Operations		88
Appendix B - Abstract Syntax of Program		91
Appendix C - Abstract Syntax of State		95
Appendix D - Execution Time of Interpreter.		98

I. Introduction

One of the major problems in computer software design is the inability to properly forecast the cost of various features. When initiating the design of a large software system, the programmer is faced with a wide variety of possible features to include without any idea of the cost of these features. Ideally, he would like to weigh the degree of usefulness of each feature against its cost in order to choose a set of features for inclusion. From previous experience with the use of similar software systems, the programmer usually knows how useful the features are. But it is usually impossible to determine the cost of features without actually implementing them. For example, a compiler designer may have the option of including label variables in a language. However, he has no basis for a decision because of the lack of knowledge about the cost of label variables. (In this case, the "cost" of label variables would be the increased execution time, larger program storage, and increased compilation time.)

Since the cost of features is never known in advance, the designer must use other criteria for choosing them. So most designers resort to such things as whether existing systems have these features and whether features "sound good" or are aesthetically pleasing. This of course leads to inclusion of extremely costly features that may be almost useless. Many times it becomes apparent during the actual coding that

certain features are highly costly. But more often, the entire software system must be completely implemented before inefficiencies are discovered. By that time, it is difficult to determine which features are causing the inefficiency (i.e. which features cost the most); and it is also usually too late to remove these costly features.

Of course, it is not completely true that nothing is known about the cost of various features of software systems. In some cases, experienced programmers and designers may have a good intuition for the cost. However, when designing languages like PL/I or operating systems like MULTICS, it is impossible for even the best of programmers to visualize the entire software system in sufficient detail to do cost analysis in advance. The tremendous problems encountered in designing these two software systems in particular clearly demonstrate the need for cost analysis in advance of implementation. It is the purpose of this thesis to make the first attempts at developing a method for performing this cost analysis.

Software cost analysis is of course an extremely vast topic. This thesis is concerned with only a limited area of cost analysis — the cost of debugging features for higher level languages. A general method is presented for determining the cost of any language feature. This method is then applied to find the cost of a set of eight interactive debugging

features for a subset of PL/I. The cost criterion used is the increased execution time of the program.

The overall approach of the method for cost determination is to first specify a machine which executes the higher level language program. The cost of a particular feature can then be expressed in terms of the requirements it places on this machine. The machine must be complicated enough to implement the many complex features of the language but be simple enough to clearly and concisely express the algorithms used for these features. Although a conventional computer with a compiler is of sufficient complexity, it by no means satisfies the simplicity requirement. This is the reason that software cost analysis is in such a primitive state. Currently, all cost determination is performed using a conventional computer as the base machine. However, this thesis uses a gedanken machine that is derived from the Vienna method of semantic definition.[6,7,8]

The cost of debugging features is determined by analyzing the requirements placed on the Vienna machine in order to implement these features. The requirements on the machine are initially expressed as particular data bases and algorithms needed by the machine. These requirements are then translated into cost by examining how their presence effects the execution time of the program. For the specific

cost analysis presented in this thesis, a set of twelve parameters is derived that completely characterizes a program.

Section II presents an overview of debugging systems in general followed by a discussion of the Vienna method in Section III. Section IV then outlines a method for estimating the execution time of the Vienna machine. Section V is a general discussion of cost analysis; and Section VI performs the final cost analysis on the debugging features deriving the twelve program parameters.

II. History of Debugging Systems

The use of debugging aids for computer programs is as old as programming itself. Even the first computers had accumulator lights and program counter indicators on the consoles to help debug programs. However, with the rapidly rising cost of software development, program debugging has received a great deal more attention lately. Most of the attention unfortunately has been in the area of implementation of specific debugging systems, and little theoretical work has been done.

Due to the lack of a firm theoretical foundation, debugging system evolution has proceeded in a rather haphazard manner. Each new debugging system was always similar to the existing ones except for possibly a few additional features. Not enough consideration has been given to the utility or cost of features. The following discussion outlines the history of debugging systems.

When the large batch processing systems of the late fifties and early sixties were popular, debugging aids consisted of TRACE type facilities. These TRACE facilities could be used to print the value of a certain variable every time it was referenced. This debugging aid characteristically produced voluminous quantities of computer output. Variations of TRACE allowed sets of variables to be considered.

The TRACE could be turned ON or OFF at different points in the program. Also, the programmer could insert extra I/O statements in critical parts of the program to print the values of certain variables. Because of the lack of direct interaction with the computer, TRACE and extra I/O were the limits of the debugging aids in the batch environment.

With the advent of time-sharing in the mid-sixties, much more sophisticated debugging aids became possible. At first, these debugging aids took on the character of the old computer console aids like accumulator values and program counters for assembly language programs. But with the increasing use of higher level languages and the unique quality of the time-sharing environment, a new type of debugging aid began to evolve. Evans and Darley[3] presented a good survey of this development in their paper of 1966. Ryan[15] wrote a summary of one such debugging aid at the same time. These new debugging aids are the ones with which most programmers are currently acquainted. They contain features like breakpoints, displaying and setting variables, single-step execution of the program, and insertion and deletion of statements in the program text. The overall structure of this type of system is shown in Figure 1.

The source program text may be edited in Stage 1. It must then be compiled and the object program passed to Stage 2. Stage 1 and Compilation may be combined into an incremental compiler or they may be

replaced entirely by an interpreter in Stage 2 that executes the source program directly. Stage 2 is where the breakpoints are implemented. A breakpoint allows the user to stop execution at a specific statement, on reference to a certain variable, or upon entry to a particular subroutine, etc. After execution is temporarily halted by the breakpoint, the user may then go to Stage 3 where selected data variables may be examined or changed. Execution can then be continued in Stage 2 or control can be passed to Stage 1 where a bug can be fixed by patching the source program. The type of configuration shown in Figure 1 will be referred to in this thesis as a "debugging system". This type of debugging aid is flexible and has proven extremely useful. Almost all current time-sharing systems use this type of debugging system.

Recently, there has been some work in designing debugging systems that use graphic displays to communicate with the users. Many of these new debugging systems are similar in form to that shown in Figure 1 except that the actual communication is done through the display rather than a teletype.[2] Some systems have become more sophisticated like HELPER[10] which combines a simulator and compiler into one large interactive runtime system. However, only one debugging system seems to have really taken full advantage of the graphic display medium - EXDAMS.[1] EXDAMS is something of a breakthrough in debugging aid design. It allows the user to slowly run the program forward or backward and display the values of selected variables on different parts of the

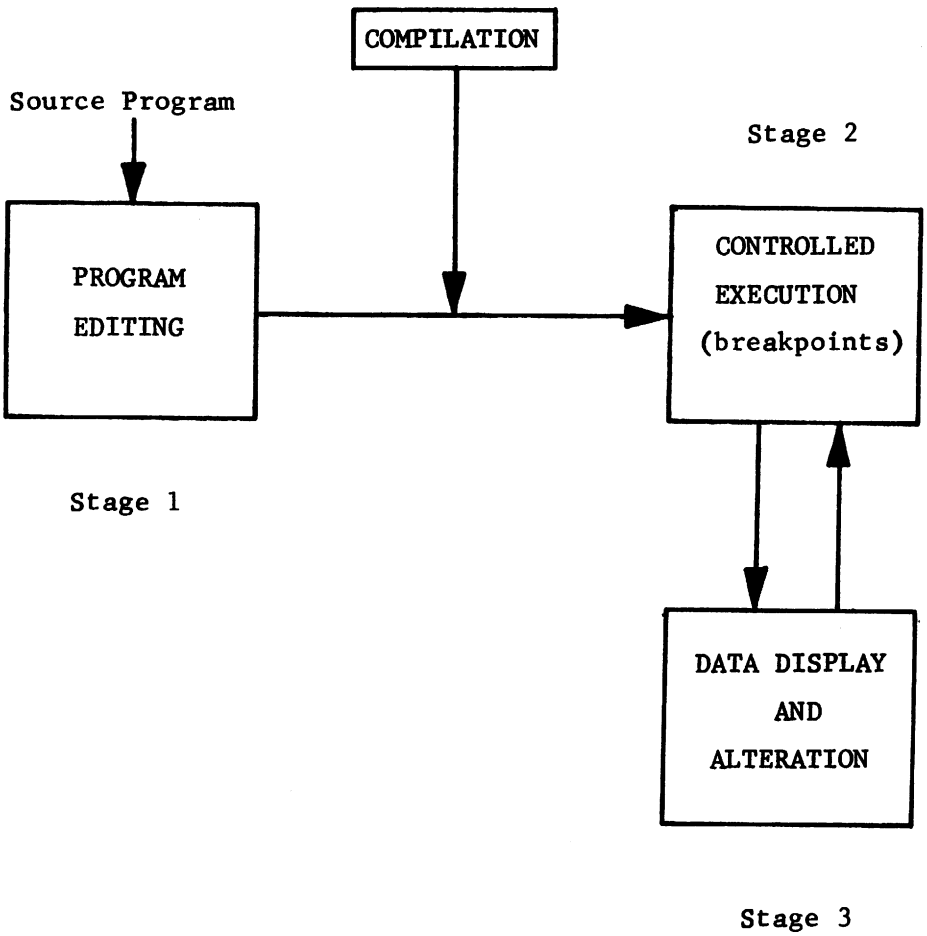


Figure 1

Interactive Debugging System

screen concurrent with execution. The result is a sort of "motion picture" of the executing program with variables constantly changing as a pointer moves from one statement to the next.

Although the more recent debugging systems are much more sophisticated and useful, they are also extremely costly. EXDAMS in particular requires that the entire execution history of the program be written on tape and used during debugging. When debugging systems begin to have such a high cost, there is some question as to whether the extra utility of these systems is really cost effective. The cost of EXDAMS is probably too high for most uses. So the designer of a debugging system has the problem of choosing some small set of useful debugging features that have a lower cost. However, in most cases, it is difficult to know in advance what the cost of a particular set of features will be. For example, a designer may decide to allow breakpoints on procedure entry only to discover that every procedure call and return in the program must be interpreted causing slow execution. Also, he may not realize that as long as calls are already being interpreted, the new feature of reading the subroutine stack could be added at a low additional cost.

Of course, the exact cost of the debugging system depends on the details of the particular implementation. Nevertheless, there definitely appears to be some implementation independent cost relationships between the various possible debugging features. There is so much

difference between the costs of different features, that even an order of magnitude type prediction of the cost would be extremely useful in the early stages of design. This thesis attempts to develop a method for implementation independent cost prediction which is hopefully much better than order of magnitude.

A very limited set of debugging features has been chosen for detailed analysis to illustrate the method. These features are derived from the type of debugging system shown in Figure 1. Features of debugging systems will henceforth be referred to as debugging operations. A debugging operation is one particular action that a user may take while debugging his program. For example, setting a breakpoint is a debugging operation and displaying the value of a variable is a debugging operation.

The next section gives an overview of the Vienna method. This is followed by a definition of the eight debugging operations and cost analysis in Section VI. See Appendix A for a summary of all debugging operations currently available in debugging systems.

III. The Vienna Method of Semantic Definition

The Vienna method of semantic definition has been successfully applied to create a complete formal definition of PL/I.[8] The method of definition has two aspects. The first is to specify an abstract form for the program using a group of "predicates". The second aspect is to specify an interpreter which executes this abstract program. This overall method has been called "semantic definition by interpretation" as differentiated from semantic definition by translation. This section of the thesis describes the applicability of the Vienna method to defining debugging operations in part A. The actual Vienna method itself is surveyed in parts B, C, and D. Part D is the most important part since it is needed to understand the cost analysis of Section VI. It is strongly recommended that the reader study Part D carefully.

For a thorough understanding of this thesis, the reader must know more about the Vienna method than the overview contained in parts B, C, and D. The best reference for this is the Annual Review in Automatic Programming, Vol. 6, Part 3.[12] The entire document is only 75 pages long and contains an excellent presentation of the entire Vienna method.

A. Why the Vienna Method?

In order to analyze debugging operations, it is necessary to somehow represent or express these operations. Debugging operations have implementation dependent representations in existing debugging systems. The actual code of the debugging system completely defines and represents its debugging operations. However, this type of representation is extremely cumbersome and is not well suited to cost analysis. What is needed is a way of representing debugging operations which is closer to the way people think of them. A debugging operation that displays program variables should be represented in terms of objects like programs, identifiers, data types, etc. and not index registers and bits. The advantage of this type of representation is that it represents debugging operations with the same objects people use to represent them in their minds.

The Vienna definition of PL/I specifies an interpreter which executes PL/I programs in abstract form. This interpreter is just a "machine" and manipulates the objects with which the language deals. Since debugging operations are very similar to actual legal operations within the language, the Vienna machine can be used to represent debugging operations. The Vienna machine is completely defined but is still implementation independent. It allows debugging operations to be expressed concisely and understandably in a form well suited to cost analysis.

The fact that debugging operations can be expressed so well on the Vienna machine is of course no accident as shown by this quote from p. 126 of "On the Formal Description of PL/I"[12]:

"The computation being a step by step process, the initial question is how big the individual steps will be. Guidance is obtained here by considering the questions that can meaningfully be asked at any step of the computation. By a meaningful question we mean one which is posed in terms of the particular program being interpreted, i.e. a question which is an inquiry about an entity which has been named in the program, or an inquiry about relationships between such entities."
A debugging operation is usually just a question about a named entity in the program. So it is no surprise that the Vienna machine is so well suited to represent debugging operations since it was designed to clearly show relationships and changes in these named entities.

B. Abstract Syntax

The first step in the Vienna method is the definition of the abstract syntax of a program. The concrete syntax of a program is the actual source text and is translated into the abstract syntax via a translator. The abstract syntax is a tree structure so that no punctuation marks are needed. Also, all default and implicit data declarations are evaluated and combined with other declarations to form a declaration part for each block. An understandable and thorough dis-

(note: * is used to denote functional composition of selectors)

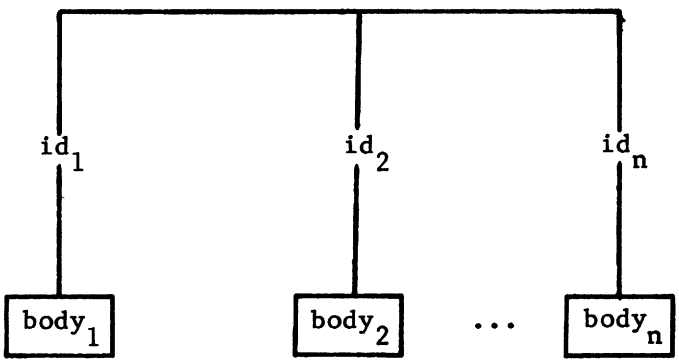
(1) s-b(t) = Z

(2) s-c * s-a(t) = X

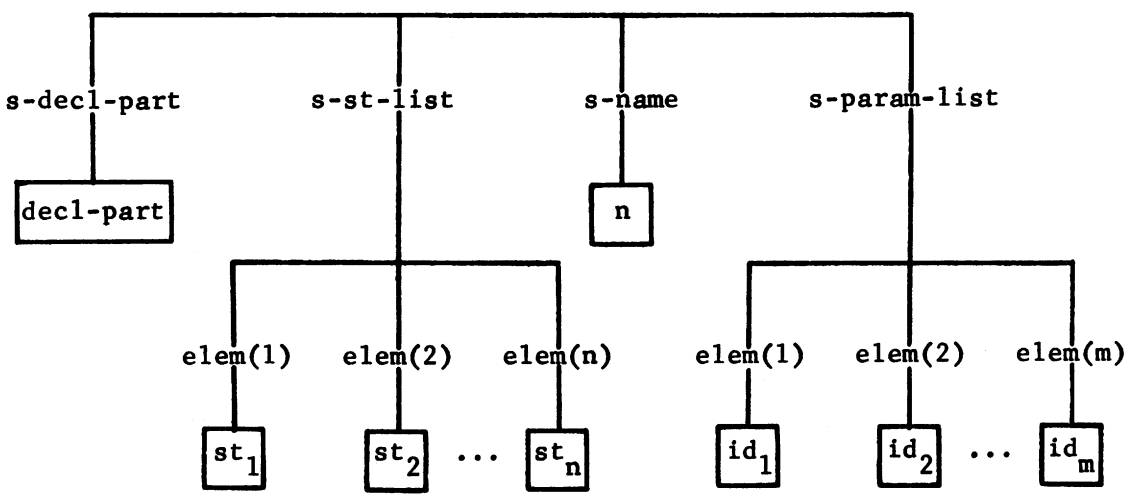
(3) if t' = s-a(t), then s-d(t') = Y

The abstract syntax of a program is shown in Figure 2A. A program consists of several procedure bodies each of which can be accessed by applying its name to the program. For example if the second procedure body is called SQRT and the program is a tree t, then SQRT(t) yields the body of the SQRT procedure. The structure of a procedure body is shown in Figure 2B. The declaration part contains explicit declarations for all labels, data variables, and internal procedures within the procedure body. The name "n" is a unique name that is used to identify this procedure for debugging operations. This s-name component is not part of the abstract syntax for PL/I. However, it has been added here for debugging purposes; it is the only addition made to the abstract syntax. Of course, the addition of this component is accounted for in the cost of debugging operations which use it.

The statement list component of the procedure body contains the executable statements. A statement may be a simple statement like a GOTO, CALL, etc., it may be another statement list, or it may be a begin block. Since begin blocks are encountered and entered as part of the stepwise flow from statement to statement, they must be part of the



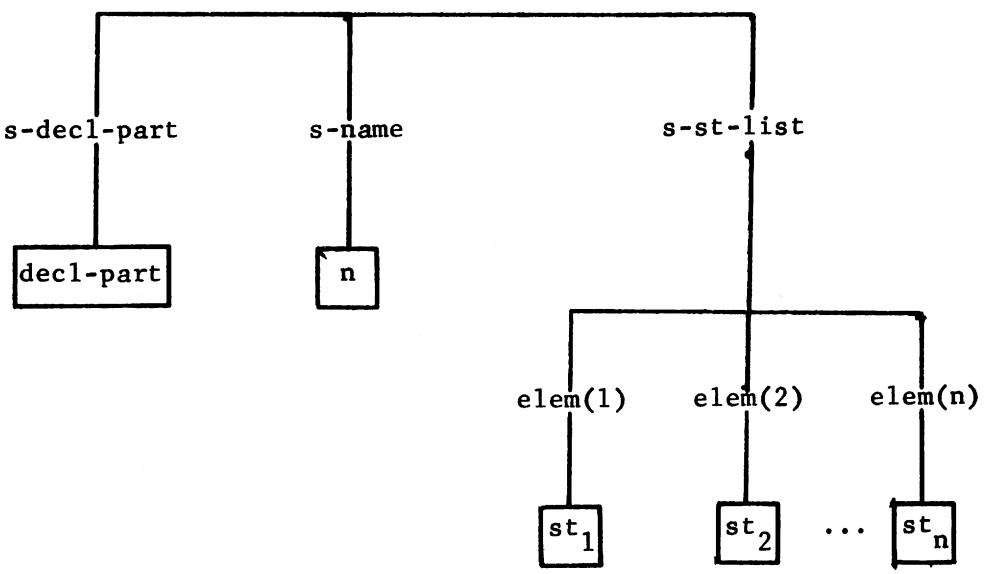
A. Program (with n main procedures)



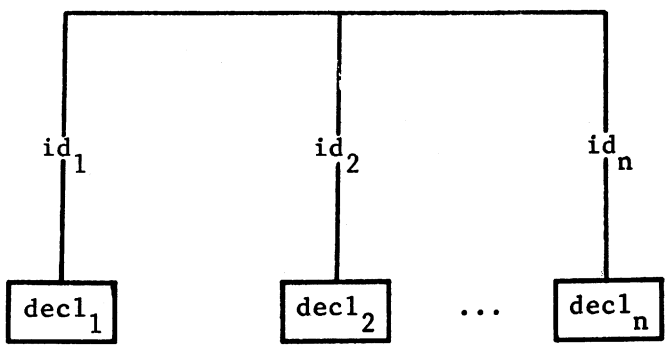
B. Procedure Body

Figure 2

Abstract Syntax

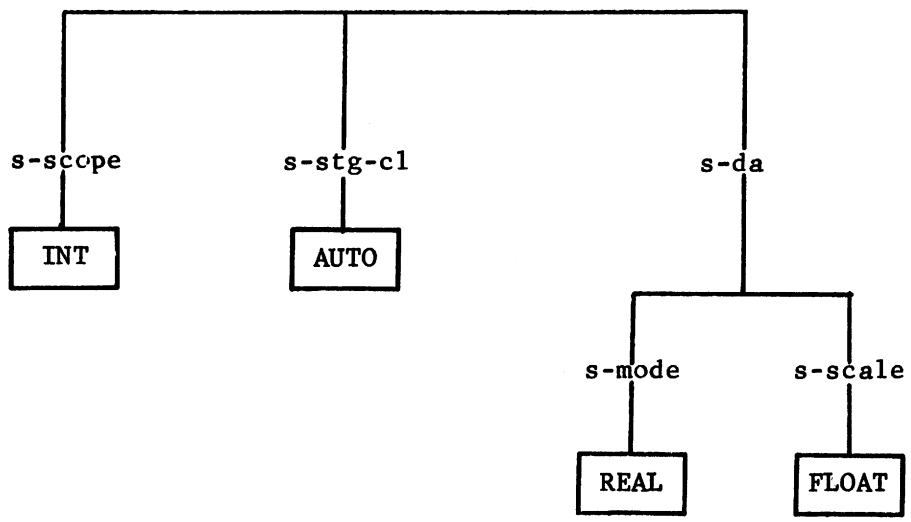


Ca. Block

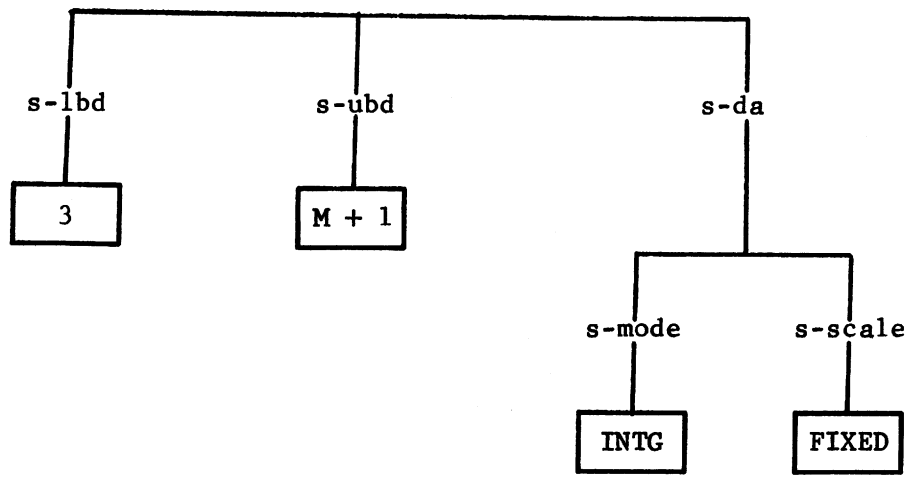


Cb. Declaration Part

Figure 2 - con't

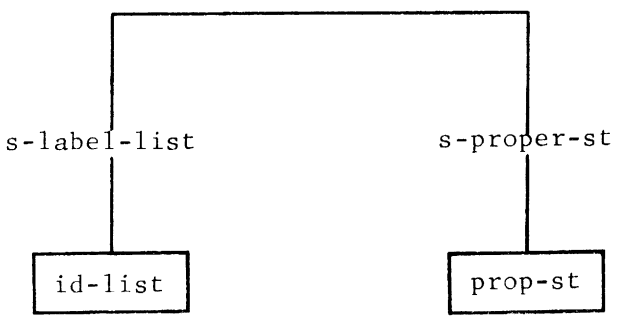


D. Sample Scalar Declaration

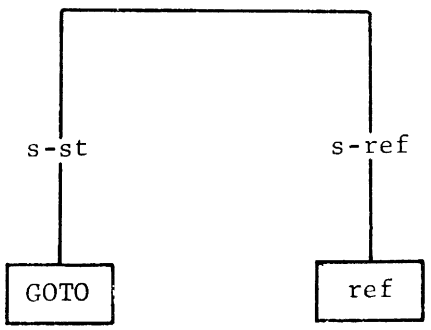


E. Sample Array Declaration

Figure 2 - con't

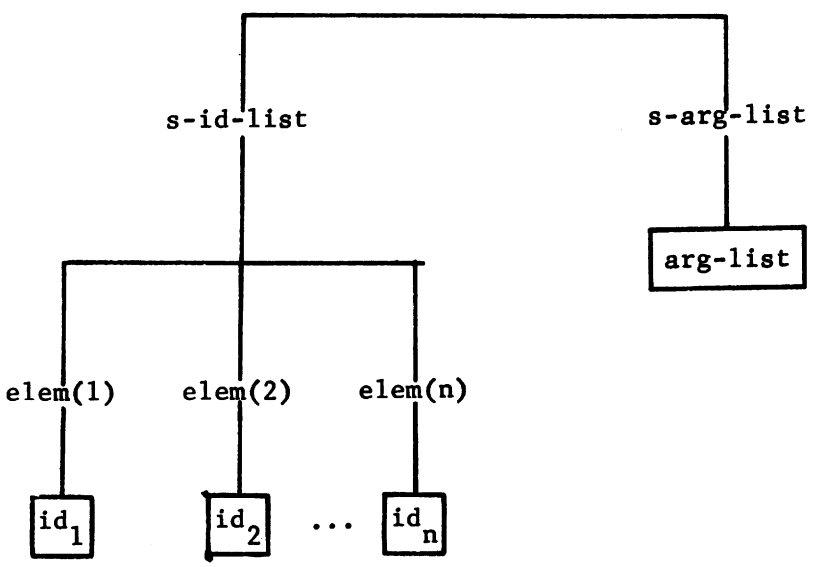


F. Statement

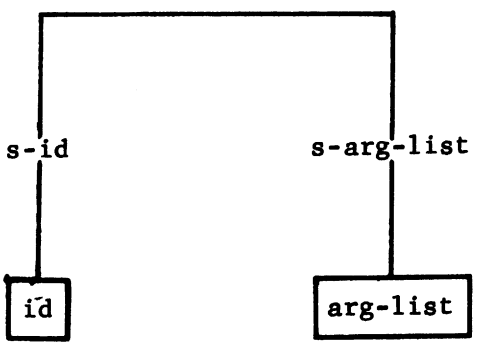


G. Goto Statement

Figure 2 - con't



H. Basic Reference



I. Call Statement

Figure 2 - con't

main statement list. A statement will be a whole new statement list in the case of DO groups in PL/I. The declaration part, name, and statement list described so far are the same for a begin block as shown in Figure 2Ca. The procedure body differs only in the addition of the parameter list which lists the input arguments for the procedure. These input arguments are of course declared with scope PARAM in the declaration part.

The declaration part is organized as in Figure 2Cb with each identifier selecting its own declaration. The general form of declarations is complicated so two examples are shown in Figure 2D and 2E. The scope of a scalar may be INT(internal), EXT(external), or PARAM (parameter). The subset of PL/I used in this thesis allows only AUTO (automatic) storage class. The s-da component is the data attribute and is used during execution to help access the variable from storage. The array of 2E is one dimensional, has a lower bound of 3, and a variable upper bound of M+1. The current value of M when the block is entered will determine this upper bound. Each element of the array is a scalar of mode INTG(integer) and scale FIXED(fixed point).

A simple statement as seen in Figure 2F has a label and a proper statement. Two examples of this proper statement are shown in Figures 2G and 2I. A basic reference is illustrated in Figure 2H. The identifier list is a fully qualified structure name or a simple identifier

for a non-structure reference. The argument list contains the subscripts needed for array reference. A reference to a simple scalar would have one element in s-id-list and a null s-arg-list part.

C. The State of the Machine

The data base of the Vienna machine is a tree structure called the state (ξ). The state contains the abstract program, block activation information, identifier associations, storage, etc. The state is the complete data base for the interpreter and all actions of the machine are described in terms of state transitions (changes in the state). The state has ten basic components. The complete abstract syntax of the state is defined in Appendix C.

Dump Mechanism

One of the major functions of the state is to help keep track of block and procedure activations. This is accomplished through the dump component s-d(ξ) = $\underline{D}$. Each time a new block or procedure is entered, all the current identifier associations and the control information are saved in the dump. A new set of identifier associations and the new control are then installed. After the termination of the new block or procedure, the old information is reinstalled from the dump. Thus, as the program executes, the state takes on the appearance of a stack with one dump on top of the other. The stack grows and shrinks as blocks and procedures are activated and terminated.

Only certain components of the state need to be stacked. These are as follows:

$\underline{E}$ = s-e(ξ) environment

$\underline{C}$ = s-c(ξ) control

$\underline{CI}$ = s-ci(ξ) control information

$\underline{EI}$ = s-ei(ξ) epilogue information

The environment contains local identifier associations and so must be stacked. $\underline{C}$ and $\underline{CI}$ contain control information like the statement counter; and the epilogue information contains information for performing the block epilogue at termination. For the subset of PL/I in this thesis, this consists of freeing the automatic variables and popping the dump. Figure 3A shows a typical dump configuration.

Identifier Associations

Another function of the state is to record the association of identifiers with data attributes and areas of storage. Five of the state components are used for this purpose. They are as follows:

$\underline{E}$ = s-e(ξ) environment

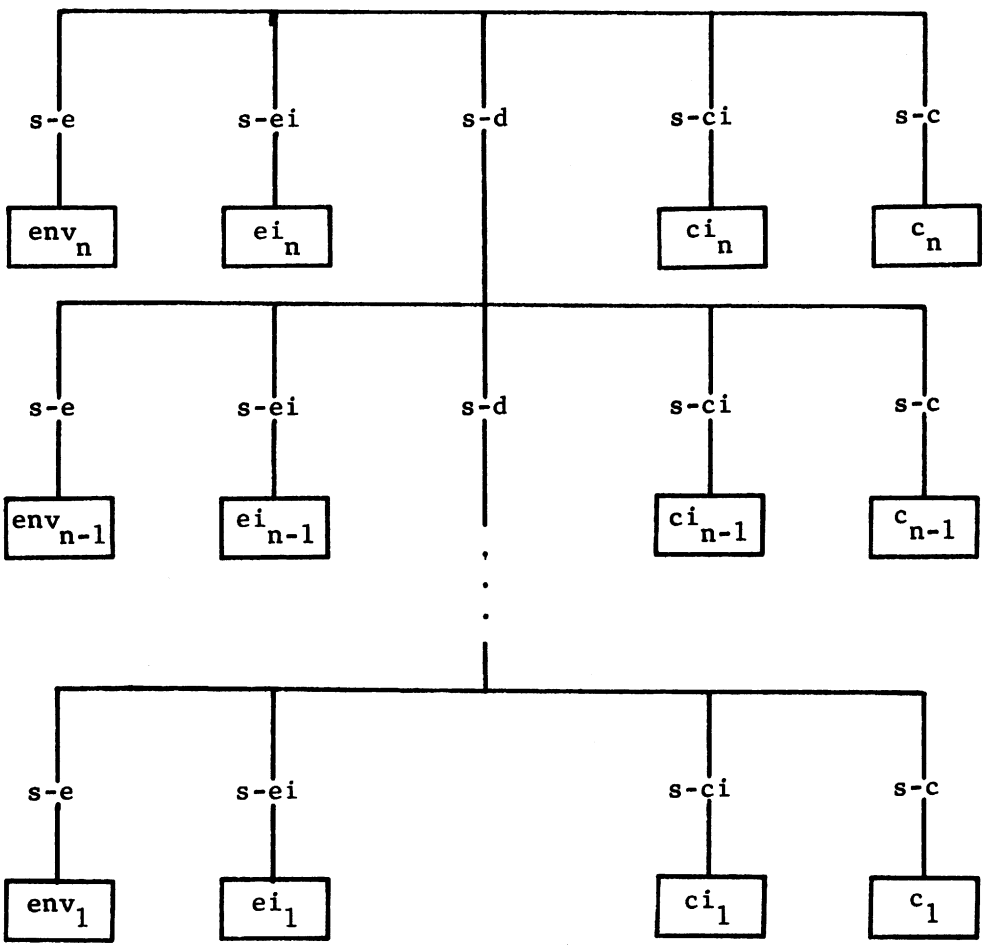
$\underline{DN}$ = s-dn(ξ) denotation directory

$\underline{AT}$ = s-at(ξ) attribute directory

$\underline{AG}$ = s-ag(ξ) aggregate directory

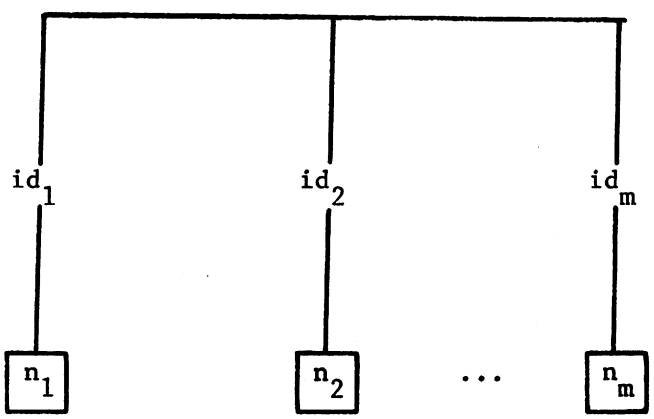
$\underline{S}$ = s-s(ξ) storage

The branches of the environment are named with identifiers so that an identifier applied to the environment will yield a unique name that



A. The Dump

Figure 3



(The $n_1, n_2, \ldots, n_m$ are unique names. A unique name is an integer. A unique name counter is a state component and contains the highest integer used so far. If a new unique name is needed, the next highest number is used and the counter is incremented by one.)

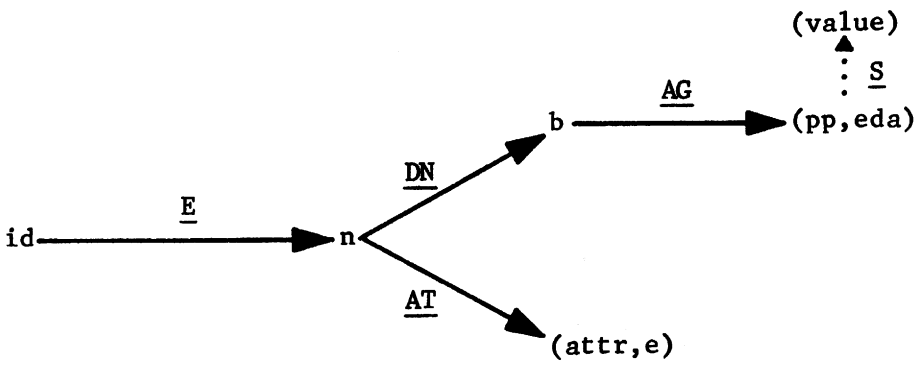
B. The Environment

Figure 3 - con't

can be used to access the DN and AG. See Figure 3B for an environment with m local identifiers. The denotation directory entry depends on the type of identifier. For data variables, the DN contains another unique name which selects a component of the aggregate directory (AG). The aggregate directory contains a pointer into the storage (S) and the evaluated data attributes. The entries of the aggregate directory are sufficient to access storage and retrieve the values of variables. Thus, to associate a data identifier with its pointer and eda(evaluated data attribute) the following application of selectors is required:

id (E) (DN) (AG) = (pp , eda)

This association can be represented in the following diagram:



A discussion of the use of directories for identifiers can be found in pages 131-138 of "On the Formal Description of PL/I".[12] A more complete description is contained in chapter 5 of Informal Introduction to the Abstract Syntax and Interpretation of PL/I.[7]

D. The Interpreter

The interpreter of the Vienna machine is specified using a LISP-like notation called instruction schemata. An instruction schema is just a way of representing a series of transformations on the state of the machine. The theoretical foundation of instruction schemata is the "control tree." Control trees and their relation to instruction schemata is discussed in pages 154-164 of "On the Formal Description of PL/I." A more thorough presentation can be found in chapter 4 of Method and Notation for the Formal Definition of Programming Languages.[6] Since instruction schemata are so similar to some conventional programming languages, their meaning can be understood in a superficial way without control trees.†

An instruction schema has the following basic format:

$in(f_1, f_2, \ldots, f_n) =$

$prop_1 \rightarrow group_1$

$prop_2 \rightarrow group_2$

.

.

.

$prop_m \rightarrow group_m$

† Instruction schemata are compared to programming languages in this thesis for purposes of simplicity. However, they are really quite different in meaning and character than conventional programming languages; and it would be a gross injustice to the Vienna method to assume they are just another programming language.

The meaning of this notation is similar to a function call in LISP. The f_1, f_2, ..., f_n are dummy arguments and are used in the groups and props below. The $prop_1$, $prop_2$, ..., $prop_m$ are propositions which yield true or false when evaluated. When the instruction is executed, the propositions are evaluated starting with $prop_1$ and proceeding to $prop_m$ until one yields true. If $prop_i$ is the first one to evaluate to true, $group_i$ is then executed.

A group may have two possible forms. It may be a call to another instruction schema as shown below:

$in_1(f_1, f_2, \ldots, f_n)$ =

$T \rightarrow in_2(f_1, f_2)$

In this case, the execution of in_1 would cause $in_2(f_1, f_2)$ to be executed. The second possible format of a group involves a value returning mechanism:

PASS: $m\text{-}expr_0$

$s\text{-}sc_1$: $m\text{-}expr_1$

$s\text{-}sc_2$: $m\text{-}expr_2$

.
.
.

$s\text{-}sc_r$: $m\text{-}expr_r$

$s\text{-}sc_i$ are simple selectors on the state

$m\text{-}expr_i$ are expressions denoting trees

If a group of the above format is executed, m-expr$_0$ is returned as the value of the instruction. At the same time a transformation takes place on the state of the machine. The transformation is specified by the s-sc$_i$ and m-expr$_i$. The s-sc$_i$ specifies which component of the state is replaced by the tree expressed by m-expr$_i$. This replacement is done for all s-sc$_i$.

Substitution and use of extra dummy arguments is also allowed within a group as shown below:

$in_1(f_1, f_2, \ldots, f_n)$ =

$T \rightarrow in_2(f_1, f_2, a, b)$;

a: $in_3(f_2)$,

b: $in_4(f_1, f_2)$

In the above schema, in_3 and in_4 are evaluated first. Their values are then transmitted through the dummies "a" and "b" to argument positions in in_2.

The following is part of an instruction schema used in the Vienna interpreter to evaluate expressions:

```
eval-expr(expr, e) =
     is-infix-expr(expr) →
          eval-infix-expr(op-1, op-2, s-opr(expr));
               op-1:  eval-expr(s-op-1(expr), e),
               op-2:  eval-expr(s-op-2(expr), e)
     is-ref(expr) →
          eval-ref(expr, e)
     is-const(expr) →
          eval-const(expr)
```

The reader who is unfamiliar with the Vienna method would be well advised to carefully study the above schema until he understands its meaning. This schema evaluates infix expressions involving constants and variable references. If the expression is a simple reference or constant, the value is found using eval-ref and eval-const respectively. So the value of eval-expr(5, e) would be eval-const(5) or just 5. If the expression is an infix expression, the two operands are evaluated as op-1 and op-2 using eval-expr again. When op-1 and op-2 are finally evaluated and substituted into eval-infix-expr, they will be simple values.

Another example of an instruction schema is the unstack instruction which pops the dump after a block termination and installs the old

environment, epilogue information, etc. See Figure 3A for the structure of the state (ξ) when unstack is called.

unstack =

PASS: Ω

s-e : s-e(D)

s-ei: s-ei(D)

s-d : s-d(D)

s-ci: s-ci(D)

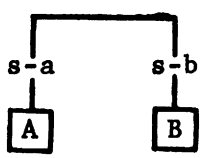
s-c : s-c(D)

(note: D = s-d(ξ))

The effect of this instruction is to return a null value and replace the specified components of the state.

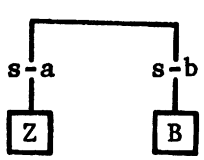
Two primitive operators are used to build and change trees. These are μ and μ_o. μ_o builds trees and μ replaces specified branches with new trees.

Example: μ_o (⟨s-a: A⟩, ⟨s-b: B⟩) ≡



If t is the above two branch tree, then

μ(t; ⟨s-a: Z⟩) ≡



One more property of instruction schemata deserves mention here. The following form may be used as a group:

in_1(arg-list$_1$);
    in_2(arg-list$_2$);
        .
        .
        .
        in_n(arg-list$_n$)

This form specifies a series of instructions that are to be executed. They are evaluated from bottom to top so that the value of the whole group is the value of the top instruction (in_1 in this case). (Note: For the remainder of the thesis, the word "instruction" will be understood to mean instruction schema).

IV. Execution Time for the Vienna Machine

The cost measure used in this thesis is execution time. So the first step in cost analysis is to estimate the execution time of various instructions. Even though the exact algorithms are specified by the instructions, the execution time can only be approximated because the Vienna machine has not been implemented. The same problem would arise in estimating the execution time of an assembly language program without knowing which computer the language was implemented on. It would be difficult to decide how the execution times of different types of primitive instructions compared. For example, it would not be known whether it takes longer to load a register or execute a branch. Nevertheless, some good approximations could be made on the basis of the relative complexity of instructions.

Since a multiply instruction is much more complex than an add, one might arbitrarily decide to assume a multiply takes three times as long as an add. With this sort of logic, it would be possible to get an implementation independent measure of the execution time of a simple assembly language program. Using a similar method, execution times in the implementation independent Vienna machine can be calculated. In the following paragraphs the primitive operations in the Vienna

machine are identified and assigned relative execution time measures. These measures can then be used to estimate the execution time of instruction schemata.

A. Tree Operations

Since the entire data base of the Vienna machine is a tree structure, the most important primitive operations are those that operate on trees. There are four such operations — μ, μ_0, application of selectors, and state transformations. The basis for calculating the relative cost of these operations is the following assumption:

Assumption 1: For all branches emanating from a particular node, one unit of time is required to access a branch, change a branch, or create a branch.

This assumption means that it takes equal time to read or write a branch in a tree independent of the form of the tree. Although this is a rather strong assumption, it is simple and corresponds to intuitive notions of relative read and write times in actual implementations. (Note: Once a node is accessed, Assumption 1 refers to all its branches. This does not mean that any branch in a tree can immediately be accessed in one time unit. Access a branch is used here to mean access the node or elementary object at the lower end of the branch. Also, change a branch means change the object at the lower end.)

Using Assumption 1, execution times can be assigned to the following operations:

Operation	Execution Time
simple selector	1
composite selector	1 for each simple selector in composite
μ_o	1 for each argument
μ	1 for each argument following the semi-colon + time for application of selectors
state transformation	1 for each component of state transformed

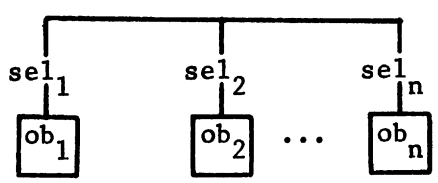
The application of a simple selector is just an access of a branch. See the example in Section IIIB for an explanation of the application of a selector. In that example, (1) has an execution time of one and (2) has an execution time of two. The application of a composite selector consists of a series of applications of simple selectors.

A μ_o operation with n arguments will create a new tree with n branches at the top level thus creating n branches.

Example: μ_o operator - execution time = n

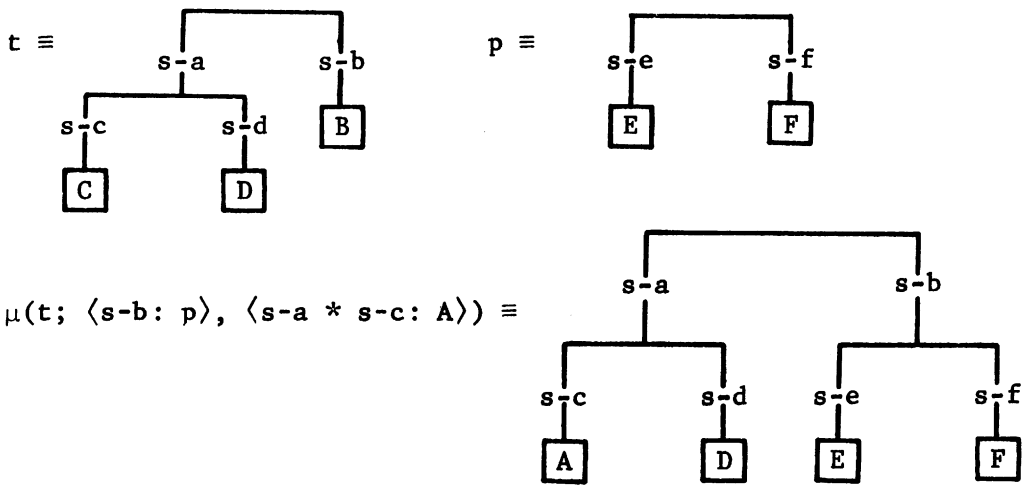
μ_o ($\langle sel_1: ob_1\rangle$, $\langle sel_2: ob_2\rangle$, ..., $\langle sel_n: ob_n\rangle$) ≡

n new branches created



The μ operator changes one branch for each argument following the semicolon. However, these branches may be deep in the tree structure and thus must be accessed using composite selectors. So the time needed to change the branch as well as access the node from which it emanates must be included.

Example: μ operator - execution time = 5



The application of s-b takes one time unit. The composite selector s-a * s-c takes two time units. And the two changes in branches made by μ take two time units.

A state transformation is one of the basic actions of instruction schemata (see Section IIID). It involves changing one or more of the basic components of the state. Changing a basic state component is just changing the lower end of the branch containing that component. There are four other tree operations whose execution time is defined by Assumption 1. These are elem, head, length, and δ.

B. Propositions (predicates)

The other major class of operations performed in instruction schemata is the evaluation of propositions. (A propositions is a logical expression involving predicates.) An estimate must be made about the relative execution times of tree operations and proposition evaluation. This estimate requires assumptions about simple predicates, complex predicates, and logical expressions. In all cases, the guiding principle used is to try and estimate what the execution time would be in a reasonable implementation.

Since a simple predicate merely tests an elementary object, the following assumption can be made:

Assumption 2: One time unit is required for the evaluation of a simple predicate.

Example: Simple predicate - execution time = 1

t ≡ [A]

is-A(t) = True (time = 1)

is-B(t) = False (time = 1)

Complex predicates are used in the instruction schemata in order to clearly and precisely define the propositions. A complex predicate may be testing for the presence of a very highly structured and complicated object like an entire procedure body. However, in the

instruction schema, this predicate might really only be differentiating between the two possibilities of a procedure body and a variable reference. A time consuming check of the structure of the entire procedure body is certainly not necessary to differentiate it from a variable reference.

The implementation of complex predicates is much less time consuming than might be expected since they are only used to differentiate between two or three different alternatives. For example, when an evaluated data attribute is passed as an argument, it must be tested to see if it is a scalar, array, or structure. This determination can be made with one branch access and one simple test. However, the complex predicates appear to require the examination of the entire evaluated data attribute which would be a very time consuming task. Since it is always known that an argument is of a certain general type (i.e. data attribute, statement, etc.), any implementation would only have to choose between two or three possibilities within that general type. The following is a listing of the complex predicates according to which general type of objects they are applied to:

is-scalar	is-scalar-eda	is-ref	is-st	is-gen
is-array	is-array-eda	is-const	is-st-list	is-integer
is-structure	is-structure-eda		is-if-st	

is-proper-var

is-entry

These are all the complex predicates required to define the debugging operations in this thesis. One consequence of the above grouping is for example that when is-array is used, the only other possibilities are is-scalar and is-structure. The following assumption can now be made about complex predicates:

Assumption 3: Two time units are required for the evaluation of a complex predicate.

This assumption is valid since the accessing and testing of only one critical branch is required to verify the truth or falseness of complex predicates.

Logical expressions involving predicates are used to form the propositions of instruction schemata (see Section IIID for role of propositions in instruction schemata). These expressions involve the logical operators &(and) and ∨(or) and the arithmetic operators = (equal to), > (greater than), and < (less than). The following assumption is made regarding these primitive operations:

Assumption 4: One time unit is required for the evaluation of a logical or arithmetic operator in a proposition.

Example: Proposition - evaluation time = 8

expr, q1, da, are passed dummy arguments

is-⟨ ⟩ (q1) (is-⟨ ⟩ (s1) ∨ s1=da) & is-ref(expr)

The simple selector is-⟨ ⟩ takes one time unit. The & and ∨ logical operators take one time unit each, and the arithmetic operator = takes one unit. The complex predicate is-ref requires two time units.

C. Implementation Dependencies

This section is concerned with the implementation of the PL/I subset and not the implementation of the Vienna machine as in some previous sections. The Vienna method formally defines a subset of PL/I. However, this formal definition makes use of a set of storage primitives which are implementation dependent. These primitives are as follows:

el-assign - write value into storage

el-ref - read value from storage

alloc-space - allocate an area of storage

el-free - free an area of storage

map - selects subparts of areas of storage

value - convert a value representation into a value

represent - convert a value into a particular type of representation

For a discussion of these primitives see p. 151-161 of "On the Formal Description of PL/I"[12] or Chapter 2, 6, 8 of Abstract Syntax and Interpretation of PL/I.[8] The Vienna definition states certain properties of these primitives but allows their complete definition to be implementation dependent. In order to estimate the execution time properly,

these details should be known. So the execution time of these storage primitives is implementation dependent. However, for the examples used in this thesis, it is sufficient to assume these primitives require one time unit each.

With assumptions 1-4 it is possible to calculate the execution time of instruction schemata. It is important to note at this point that these particular assumptions are not required for the remainder of this thesis. All that is required is that some set of assumptions be made about the relative execution times of primitive operations in instruction schemata. Assumptions 1-4 represent such a set but any other set could be substituted and used for the remainder of the thesis.

D. Parameterization of Algorithms

Knowing the execution times of the primitives does not solve all the problems of determining the total execution times of instruction schemata. In fact many people would argue that it solves none of the problems. It is extremely difficult to estimate the execution time of an assembly language program even if the exact execution time of every instruction is known. The reason for this is of course the dependency of execution time on input data. The execution time must be given as some function of the input data. For a complicated program with a large volume of input data this is usually impossible.

So merely knowing the execution time for primitive does not necessarily allow calculation of the execution time for arbitrary algorithms expressed with these primitives. What is really needed is an understanding of the nature of the algorithm itself so that it can be parameterized according to certain properties of the input data. It is not really necessary to find an equation for the execution time in terms of the input values. All that is needed is an expression of execution time using parameters that people understand. An example of this might be an information retrieval program. Since there are thousands of input values to this program, an equation involving the input variables is out of the question. However, someone with a thorough understanding of the algorithm might be able to state that the execution time if the number of items retrieved times the logarithm of the density of the stored data base. This parameterized expression of execution time is valid and extremely useful.

The real advantage of the Vienna machine now becomes apparent. The algorithms used in the Vienna machine are so concisely and understandably expressed that parameterization is relatively easy. A compiler and a real computer also form a machine that defines PL/I programs. But the algorithms in this machine are impossible to parameterize because they are expressed with assembly language. The Vienna machine was designed to make it perfectly clear how the named entities of a PL/I program are manipulated. So the execution time of algorithms

in the Vienna machine can be expressed in terms of simple parameters involving these named entities. The following section gives an example of this parameterization.

E. Examples

The control information (s-ci(ξ) = CI) component of the state is shown in Figure 4. The text component (s-tx) holds the current statement list st-list. The s-sc component contains a statement counter sc which specifies the member of st-list currently being executed. The control dump (s-cd) is needed because of nested statement lists. Each time a new statement list is started, the entire control information and the control (C) are stacked and a new control information is created. This new CI contains only the text of the new statement list and an initial statement counter of 0. st-list is the innermost statement list and the one currently being executed. st-list_1 is the complete statement list for the entire block. The statement counter list $sc, sc_n, sc_{n-1}, \ldots, sc_1$ can be used to specify which statement in the block is currently being executed. This is not quite true because it would not be known which branch of an IF statement was chosen. However, this can be solved by the use of T or F as a statement counter in the list.

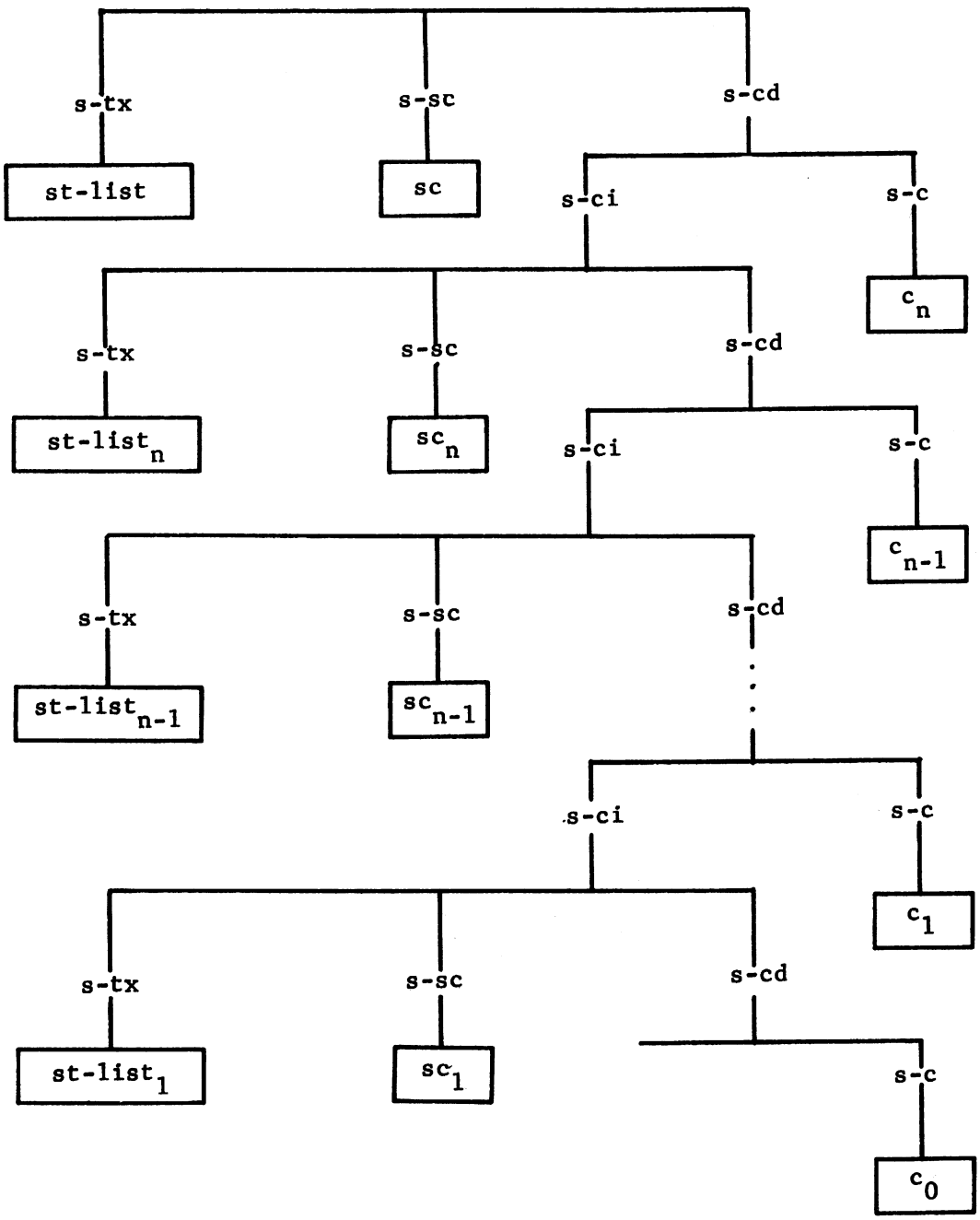


Figure 4

The Control Information (CI)

The instruction int-st-list is used to initialize a new statement in the control information. The new statement list becomes the text component. The statement counter is initialized to 0, and the old control information and control are put into the control dump.
(t is the new statement list.)

int-st-list(t) =

s-ci: μ_o(⟨s-tx: t⟩,
⟨s-sc: 0⟩,
⟨s-cd: μ_o(⟨s-ci: CI⟩,
⟨s-c: C⟩) ⟩)

s-c: int-next-st

Execution Time for int-st-list

Operation	Execution Time
outer μ_o with 3 arguments	3
inner μ_o with 2 arguments	2
application of simple selector s-ci†	1
application of simple selector s-c	1
state transformation on two components (s-c and s-ci)	2
Total execution time for int-st-list =	9 units

(†Note: CI is only an abbreviation for s-ci(ξ) so this simple selector must be included in the time. The same is true for C.)

The instruction int-next-st is used to iterate from one statement to the next in a statement list. If the end of the statement list has not yet been reached, sc is incremented by one and the interpretation of the new statement is started with a call to int-st. If the end of the current list is reached, the control dump is popped.
(Note: length is a tree primitive to determine the number of branches at the root node.)

```
int-next-st =
     s-sc(CI) < length * s-tx(CI) →
          s-ci: μ(CI; ⟨s-sc: s-sc(CI) + 1⟩)
          s-c:  int-next-st;
                     int-st(elem(s-sc(CI) + 1,s-tx(CI)))
     T →
          s-ci:  s-ci * s-cd(CI)
          s-c :  s-c * s-cd(CI)
```

Execution Time for int-next-st

Operation	Execution Time
application of simple selector s-sc	1
application of simple selector s-ci	1
evaluation of < operator	1
application of length primitive	1
application of composite selector s-tx * s-ci	2

Time for evaluation of proposition = 6 units

μ with one argument	1
application of composite selector s-sc * s-ci	2
evaluation of arithmetic operation +	1
application of elem primitive	1
application of composite selector s-tx * s-ci	2
state transition for 2 components (s-ci and s-c)	2
Execution time for first alternative	= 9 units

application of composite selector s-ci * s-cd * s-ci	3
application of composite selector s-c * s-cd * s-ci	3
state transition for 2 components (s-ci and s-c)	2
Execution time for second alternative	= 8 units

Assumptions 1-4 can easily be applied as above to calculate the execution time for the parts of int-next-st. However, in order to express the total execution time in closed form, it is necessary to parameterize the algorithm. The first alternative is chosen once for each statement in a particular statement list. The total execution time for the first alternative is the evaluation time of the proposition (6) plus the execution time of the alternative itself (9) yielding 15. Thus, there is an execution time of 15 for each statement in the list. At the end of the list when the first proposition is no longer true, the second alternative is chosen. Since the first proposition must be evaluated in order to know that it is false, the

execution time for the second alternative is the evaluation time of the proposition (6) plus the execution time of the second alternative (8) yielding 14. This action occurs only once for a particular statement list. So the execution time of int-next-st can now be stated as follows:

Execution time of int-next-st for statement list t

= 15m + 14

where: m = length of statement list t

If the time to initialize the particular list is to be included, int-st-list must be added. This instruction is executed only once for a particular statement list. So the total execution time required to initialize a statement list and iterate from one statement to the next is as follows:

Time = Time for int-st-list + Time for int-next-st

= 9 + 15m + 14

= 15m + 23

Thus, the time required for the mechanism in the Vienna machine that initializes and steps through a statement list is 15m + 23.

Another example of execution time estimation is given in Appendix D along with listings, discussions, and execution time for all instruction schemata needed in Section VI. Following is a summary

of the execution times developed above and in Appendix D. The execution times for many of the instruction schemata are not listed but instead are included in the total times of instructions that use them.

Summary of Execution Times for Instruction Schemata

Instruction schema	
int-st-list(t)	9
int-next-st	15m + 14
where: m = length of statement list	
int-block(t)	22
int-call(body, e, arg-list)	23
int-return-st(t)	3
return	8 for BLOCK, 4 for PROC
unstack	11
int-assign-st(t)	19
eval-ref-gen(ref, e)	36+47q+49a
where: (these parameters refer to the reference ref)	
q = length of qualifier list (= length of id-list - 1)	
a = length of subscript list	

goto-search(id) 11n + 4s +12b + 7c

where: (these parameters refer to search for id)

n = total number of statements searched

s = total number of statement lists searched

b = total number of blocks terminated during search

c = total number of control dumps unstacked during search

goto-jump(scl) 15t

where:

t = length of statement counter list (scl)

(Note: The execution times for eval-ref-gen, goto-search, and goto-jump are abbreviated here. For completely parameterized times see Appendix D.)

the system will require if the feature is included. Although in some situations incremental cost can be very useful, it has two fundamental problems. The first is that it is generally extremely difficult to determine. If a redesign of parts of the system is required to add a new feature, it is hard to predict how this will affect the efficiency of the overall system. In addition, an alternate implementation of the feature might be achieved by a different set of redesigns that are less inefficient. It is difficult to find the minimal cost implementation of a feature as well as estimate the cost once an implementation is chosen.

The second and most important problem with incremental cost is that it does not properly measure the real cost of a feature. For example, if another similar feature is already in the system, then the cost of a new feature may appear to be misleadingly low. If a software system contains a set of features that all use the same mechanisms within the system, the incremental cost of any one of these features will actually be zero. In some cases, it may be desirable to use incremental cost. An example is when an existing system is being slightly redesigned or a few marginally useful features are being added to a set of crucial features. However, in general it seems as though a better method of cost analysis should be used.

The alternative to incremental cost is absolute cost. Absolute cost attempts to measure the inherent or total cost of a particular feature rather than the additional cost. The absolute cost for a feature is calculated by examining the complete mechanism in the system needed to execute this feature. It does not assume that the existing mechanisms are free and so does not underestimate the true cost of a feature. The use of absolute cost seems to solve the two problems encountered in incremental cost. Absolute cost is easier to measure because no alternate designs need be considered. The absolute cost of a feature of a complete system can be calculated by analyzing the mechanisms the feature actually uses. Also, absolute cost does not consider any parts of the system to be free and so does not underestimate the cost of a feature.

Of course, absolute cost also has some problems. The only way absolute cost can be measured is through a complete software system. But the algorithms of a complete system are designed to accommodate all the features of the system. So the absolute cost of one feature may be overestimated because it shares a mechanism with another feature. If one particular feature requires that a certain type of algorithm perform inefficiently, other features that use this algorithm will appear to have a high absolute cost. For example, in a programming language, block entry may appear to have a high absolute cost because the non-local goto feature requires that the old environment be explicitly stacked on block entry. Any attempt to try and calculate

the cost of block entry ignoring non-local gotos begins to run into the problems of incremental cost. The following discussion of cost analysis of debugging systems should help clarify the uses and differentiation of incremental and absolute cost.

B. Debugging Systems

Debugging system design is an area where cost analysis is essential. Without thorough investigation, it is usually difficult to estimate the cost of a feature in a debugging system. The simplest of debugging features that intuitively appear cheap often are the most costly. The cost of a debugging system is measured in terms of the execution time and storage requirements of the program being debugged. If a particular debugging feature causes the program to run more slowly or take more space, this feature has some positive cost associated with it.

Oftentimes, incremental cost analysis is desirable for debugging systems. When debugging features are being added to a language, they are usually added in the context of a particular compiler and run-time system. Since the existing compiler or interpreter is needed to execute the program already, it is in some sense cost free. Thus, the addition of single debugging features can be viewed in terms of their incremental cost. If a debugging feature can be added to a particular interpreter without causing any inefficiency, then the feature actually is cost free. So in practical debugger design, incremental cost

implementation that differs significantly from the model. However, it seems that certain features are inherently more costly than others; and no matter how these features are implemented this cost must somehow be paid. For some debugging systems, the cost may be buried in the algorithms of the compiler or runtime library routines. But the inherent complexity and cost of features must be accounted for somewhere in the total system.

The question of exactly how to determine absolute cost from a general model is not answered in this section. The details of how to calculate absolute cost depends on the particular type of software system being considered and the specific form of the general model of that system. The following section will calculate absolute cost for a few simple debugging operations as a representative method for cost analysis using the Vienna machine.

C. The Principle of Cost Analysis

The following two statements give a general philosophy that can be used for cost analysis of software systems. They are rather vague and require a thorough explanation before they can be applied. However, they do summarize two of the major points of this thesis and represent a new approach to cost analysis:

Principle of Cost Analysis

1. Cost is measured in terms of the relative efficiency with which a system performs its designated function.
2. The cost of a feature is determined by examining the mechanisms that implement this feature in the system.

VI. Cost Analysis of Debugging Operations

A. The Debugging Operations

As demonstrated in Appendix A, there are a wide variety of potential features for any higher level language debugging system. This section chooses eight of these debugging features (operations) and performs cost analysis using the results of Section IV and V. The debugging operations used are intended to be representative of the different types of debugging operations available. They are also primitive in the sense that they can be combined to form many more debugging operations. The eight operations presented here form something of a basis for larger expanded systems because of their primitive and independent nature. The cost analysis in this section has two distinct purposes. The first is to serve as an example of a general method for cost analysis of software systems. The overall philosophy of the method is outlined in Section V, but this section is needed to clarify and illustrate the method.

The second purpose of this section is to serve as a guide to debugging system designers. From a thorough examination of the state-of-the-art of debugging system design, it is apparent that not enough consideration has been given to the cost vs. the utility of various operations. Many relatively useless but costly features are included in new debuggers because other debuggers already have these features.

Some cost analysis of the "standard" features of debugging systems would probably result in the replacement of some of the costly features with cheaper and possibly more useful features. The set of debugging operations used here is representative of quite a number of existing debugging systems so the results should have some relevance to practical debugger design.

One of the most important features useful for debugging is to be able to examine and alter data variables of the program. The operation Display Variable is used to examine the value of a data variable. It requires one argument which is a reference to the variable. Recall that a reference consists of an identifier list and an argument list. The identifier list is used to qualify structure names, and the argument list is a subscript list for arrays. Thus, any scalar or member of an array or structure may be examined using Display Variable.

Another useful debugging feature is the ability to restart execution at an arbitrary statement. This type of feature is offered by the Goto operation which takes a label as an argument and transfers control to the statement with that label.

At times during debugging it is helpful to know the current subroutine name or the specific statement about to be executed. This information can be gathered by the Read Block or Procedure Name

(abbreviated Read Block) operation or the Read Statement Counter List operation (abbreviated Read Statement). The Read Block operation is used to determine the current subroutine or block name. This operation in combination with the Read Statement operation is sufficient to completely identify the current statement. These two operations are good examples of the way primitive operations can be combined to form other features. The successive application of Read Block and Read Statement can be used to read the subroutine stack. (This point is clarified in the cost analysis below.)

Of course, it is implicitly assumed in the above explanation that it is possible to stop execution temporarily. This ability is one of the key features of any debugging system. All debugging operations are used to monitor the execution of the program to locate bugs. The type of execution halting features available determines when and how often this execution can be monitored. Since the Display, Set, and Read operations discussed above can only be applied after execution is temporarily halted, they actually become more powerful as more sophisticated trigger operations are added.

A trigger operation is something that temporarily stops program execution so that other debugging operations can then be applied before execution is continued. These operations are sometimes called conditional breakpoints (see Appendix A). There are three basic trigger

operations allowed: Trigger on Variable Reference, Trigger on Block or Procedure Activation, Trigger on Statement. The Trigger on Reference operation takes a reference as an argument. Any time this reference is used in the program, execution is stopped temporarily. Notice that this operation can only be used to trigger on reference to a specific named data variable. The trigger is not associated with the storage area used for the variable but the name of the variable. Thus, if sharing of storage exists it is possible to change the value of the data variable without the trigger going off. The feature of triggering on reference to storage is a completely different and incidentally much more costly operation.

Recall from Section IIIB that every procedure and block has a unique name associated with it. So it is possible to identify a particular block or procedure for the Trigger on Entry operation (same as Trigger on Activation). Since statement lists are local to a block or procedure, it is necessary to include both the unique name and the statement counter list for Trigger on Statement. This operation is intended to provide the ability to put a trigger on a specific statement in the program. Because of nested statement lists, a statement counter list is needed to localize a statement within a block (see Section IVE for explanation of statement counter lists).

Following is a summary of the eight interactive debugging operations:

Operation	Arguments
Display Variable	reference
Set Variable	reference, expression
Goto	label
Read Block or Procedure Name	—
Read Statement Counter List	—
Trigger on Variable Reference	reference
Trigger on Block or Procedure Activation	unique name
Trigger on Statement	unique name, statement counter list

B. Cost Analysis

The Principle of Cost Analysis stated in Section V can now be used to find the cost of these debugging operations. The designated function of any debugging system is ultimately to execute the program. So the cost can be measured in terms of the relative efficiency with which the debugging system executes the program. Some people would say that the designated function of a debugging system is to perform debugging operations on the program. With this assumption, a different set of costs would be calculated than the ones in this thesis. However, the decision has been made here to define cost in terms of program execution. So the cost of a debugging operation can be calculated by finding how its availability affects the execution time (and storage) of the program. Although there is some storage cost associated with

debugging operations, the really important component of cost is the execution time.

It is an extremely important property of the Vienna machine that, except for a few insignificant changes, its algorithms and operations are sufficient to directly implement debugging operations. This means that for the Vienna machine, the incremental cost of the debugging operations is zero. However, the cost analysis performed here is in terms of absolute cost. The Vienna machine is used as a general model of a debugging system and absolute cost can be calculated by examining the mechanisms of the machine that are required by each debugging operation.

The mechanisms of the Vienna machine are of course the instruction schemata. So it must be determined which instructions are required by each of the eight debugging operations. This is done below. Once the instructions for an operation have been chosen, it is relatively straightforward to estimate the absolute cost. Since the cost is measured by program execution time, the absolute cost of a debugging operation is calculated by finding how its instruction schemata affect the execution time of the program. This absolute cost is just the total amount of execution time used by these instructions for a particular program. The total execution time required by all the instructions of an operation is the cost because absolute cost assumes nothing is cost free.

The one remaining task is to decide which instructions are utilized or required for each debugging operation. The eight debugging operations can be divided into three general classes for this purpose. The classes can be generally described as follows:

Class 1 - operations that use the instructions of the machine directly

Class 2 - operations that use the data base of the machine directly

Class 3 - trigger operations

The first class contains Display, Set, and Goto. These operations perform functions so similar to actual operations in the Vienna machine that certain instruction schemata can be used directly to execute these operations. For this class of operations, the instruction schemata required are just those instructions that are used directly to apply the operations.

The second class of operations is Read Block and Read Statement. These operations do not use any particular instructions but require parts of the data base of the machine. However, certain instructions are required to update and maintain these parts of the data base. These instructions are therefore the ones required by the debugging operations Read Block and Read Statement.

Trigger operations comprise the third class of debugging operations. These operations depend on the detection of certain events within the Vienna machine. The instructions required by these operations are the ones that directly cause the events to occur. All instructions are somewhat indirectly used to cause these events. But the absolute cost of a trigger operation is estimated from the execution times of only those instructions that directly cause the events. The rules for choosing the instruction schemata required by debugging operations can be summarized as follows:

Rule 1 - For class 1 operations, the instructions that are used directly to execute the operation.

Rule 2 - For class 2 operations, the instructions that are used directly to update and maintain the data needed by the operation.

Rule 3 - For class 3 operations, the instructions that may directly cause the event (or events) to occur.

The three rules outlined above for choosing instructions are utilized along with the execution times from the end of Section IV to calculate a parametric cost estimate for each debugging operation.

1. Display Variable

The function performed by this operation is to translate a reference into a value by reading its storage pointer and data attributes

from the aggregate directory and accessing the proper area of storage. This operation is performed by the instruction schema eval-ref-gen. Since Display is a class 1 operation as discussed above, the required instruction is just eval-ref-gen. So the cost of the Display Variable debugging feature is the total execution time required for eval-ref-gen during program execution.

The summary at the end of Section IV shows that the characteristics of the particular variable reference determine the execution time for the instruction. In order to find the total execution time for an entire program, it is necessary to characterize the set of all references used in the program. This can be done with three parameters — one which specifies the total number of data references and two which specify the average characteristics of the references.

Cost of Display Variable = v (36 + 47q + 49a)

where: v = total number of variable references during program execution

q = average length of the qualifier list for all these references

a = average length of the subscript list for these references

The estimate of these parameters is something of an unsolved problem itself. However, this problem is not the concern of this thesis.

It is important only that the cost can be expressed in terms of a reasonable set of well-understood program parameters.

2. Set Variable

The Set Variable operation is exactly the same as an assignment statement. It requires that references be converted to values from storage just as Display. Thus, eval-ref-gen is needed by Set Variable. The additional mechanism of assigning a value to a data reference is also required. This type of function is performed by int-assign-st. The cost of Set Variable has two parts, the cost of evaluating a reference (eval-ref-gen) and the cost of assigning a value to a data reference (int-assign-st).

The total execution time for eval-ref-gen is shown above. The execution time for an individual application of int-assign-st is 19 (see summary at end of Section IV). The total execution time of this instruction for the whole program is 19 times the number of assignment statements executed.

Cost of Set Variable = $19w + v\ (36 + 47q + 49a)$

where: w = total number of assignments during program execution

v, q, a are defined above

It is interesting to note the sharing of eval-ref-gen by Display Variable and Set Variable.

3. Goto

The Goto operation is the last of the class 1 operations. It is exactly the same as a goto statement in the program and so requires the use of goto-search and goto-jump. The goto-search instruction searches the text component of the control information for the specified statement label and records a statement counter list. This statement counter list is used by goto-jump to transfer control to the statement. Since these operations are used by all program gotos, their execution time is characterized by parameters involving these gotos.

Cost of Goto = 11n + 4s + 12e + 7c + 15t

where: n = total number of statements searched during all program gotos

s = total number of statement lists searched during all program gotos

e = total number of blocks terminated during all program gotos

c = total number of control dumps unstacked during all program gotos

t = total length of statement counter lists used during all program gotos

The cost shown above is merely the total time used by all goto statements during program execution. It must be emphasized at this point that these parameters characterize a particular run or execution of the program. It is not important how many goto statements are contained in a program but how many are executed.

4. Read Statement Counter List

Since this is a class 2 operation, it must be decided which in-instructions directly maintain the statement counter list. The two basic instructions that maintain this list are int-st-list and int-next-st. One of these instructions (int-st-list) initializes a new statement list and sets the statement counter to zero. The other (int-next-st) increases the statement counter as successive statements are executed. The two together form the mechanism for building the statement counter list. One additional instruction sometimes updates the list — goto-jump. In order to transfer control to a statement, goto-jump usually has to build up part of the statement counter list.

The execution time of int-st-list depends on only the number of statement lists; but int-next-st depends on the length of each statement list. Since a member of a statement list may be either a statement or a statement list, the time for int-next-st is affected by both the number of statements and the number of statement lists.

Cost of Read Statement Counter List = 15 (m + k) + 23k + 15t

where: m = total number of statements executed

k = total number of statement lists executed

t = total length of statement counter lists used in all gotos

5. Read Block or Procedure Name

Read Block is a class 2 operation. The data base for this operation is part of the epilogue information. The block-activation type component of the epilogue information contains the unique name and type (BLOCK or PROC) for that part of the state. The instructions that create the epilogue information are int-block and int-call. The epilogue information is also changed by the termination of a block or procedure activation. For a procedure termination the instructions used are int-return-st, return, and unstack.

A block may be terminated by a return or by executing the last statement of the block. The instructions used depend on how the termination comes about. If a return causes termination, the instructions unstack and return are needed. In the other case, only unstack is needed.

Cost of Read Block or Procedure Name = 33b + 8r + 41p

The result is that the absolute cost of Display is misleadingly high. Thus, as stated previously, absolute cost must be used in the context of additional information in order to be interpreted properly.

7. Trigger on Block or Procedure Activation

The event Block entry is always caused by int-block and the event Procedure entry by int-call. It may seem at first that the requirements of this operation should be the same as for Read Block or Procedure Name. However, the Read Block operation needs both block entry and block exit information and so includes some additional instructions.

Cost of Trigger on Block or Procedure Activation = 22b + 23p

where: b = total number of block activations

p = total number of procedure activations

8. Trigger on Statement

The Trigger on Statement operation is more complex than the other two class 3 operations since it involves two events. For this operation it makes more sense to refer to "conditions" rather than events. Trigger on Statement is used to halt execution at a specific statement. This statement is specified by a unique name identifying the block and a statement counter list localizing the statement within the block. The first condition is that the specific block be currently active. This condition can be caused by activation which uses

instructions int-block and int-call. It can also be caused by termination since this changes the currently active block. Termination requires the instructions int-return-st, return, and unstack.

The second condition is the occurrence of a specific statement counter list. A statement counter list may be altered by the instructions int-st-list, int-next-st, and goto-jump. Any of these instructions could cause a certain statement counter list to appear. The goto-jump instruction may also initiate block termination and so is required for both conditions. (A statement counter list is the s-sc components of all the control dumps in a certain control information.)

It is important to differentiate Trigger on Statement from Trigger on Block and understand why the former requires block termination instructions where the latter does not.

Cost of Trigger on Statement = $33b + 8r + 41p + 15(m + k + t) + 23k$

where: b = total number of block activations

p = total number of procedure activations

r = total number of block activations terminated via return

m = total number of statements executed

k = total number of statement lists executed

t = total length of statement counter lists used in all gotos

Summary of Cost of Debugging Operations

	Operation	Cost	Instructions Required
1.	Display Variable	v(36+47q+49a)	eval-ref-gen
2.	Set Variable	19w+v(36+47q+49a)	int-assign-st,eval-ref-gen
3.	Goto	11n+4s+12e+7c+15t	goto-search,goto-jump
4.	Read Statement Counter List	15m+38k+15t	int-st-list,int-next-st goto-jump
5.	Read Block or Procedure Name	33b+8r+41p	int-block,int-call,unstack int-return-st,return
6.	Trigger on Variable Reference	v(36+47q+49a)	eval-ref-gen
7.	Trigger on Block or Procedure Activation	22b+33p	int-block,int-call
8.	Trigger on Statement	33b+8r+41p+ 15m+38k+15t	int-block,int-call,unstack int-return-st,return, int-st-list,int-next-st, goto-jump

where:

a = average length of subscript list for all variable references

b = total number of block activations

c = total number of control dumps unstacked during all goto statements

e = total number of blocks terminated during all goto statements

k = total number of statement lists executed

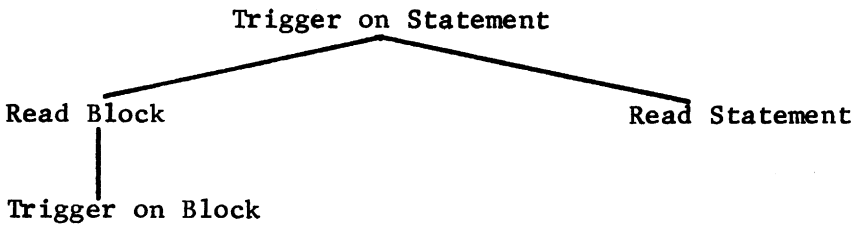
m = total number of statements executed

Primitive Operation	Instruction Schemata
Variable Reference	eval-ref-gen
Block Activation	int-block, int-call
Block Termination	unstack, int-return-st, return
Statement Counter List Maintenance	int-st-list, int-next-st, goto-jump

The use of each of these primitives for several debugging operations causes sharing of cost between the operations. Display Variable, Set Variable, and Trigger on Variable Reference all require the primitive for variable reference and so share this cost. This cost is that associated with the complete interpretation of all variable references in the Vienna machine. The block activation primitive is used by Read Block, Trigger on Block, and Trigger on Statement. Since Trigger on Block uses only the block activation primitive, it is a proper cost subset of Read Block and Trigger on Statement. The block termination primitive is part of Read Block and Trigger on Statement. In addition, Read Statement and Trigger on Statement both require the statement counter maintenance primitive.

Some interesting results can be seen from these combinations of primitives. The Trigger on Statement debugging operation is a highly

costly operation since it contains all the primitives needed for Read Statement, Read Block, and Trigger on Block. These four debugging operations form a simple cost heirarchy as shown below:



(Trigger on Block) < (Read Block)

(Read Block) + (Read Statement) = (Trigger on Statement)

Another result is that Display Variable, Set Variable, and Trigger on Variable Reference are all of approximately equal cost. This fact is true because of the one central mechanism in the Vienna machine that is used to interpret variable references.

Any further conclusions about the relative cost of debugging operations must be based on some assumptions about the parameters of the cost equations. It is noteworthy that a program can be completely characterized for cost analysis with twelve parameters. These parameters are only valid for one particular execution of the program. In general for an arbitrary program, it might be difficult to estimate these parameters since they depend on the input data. One approach to

this problem might be to specify the twelve program parameters with probability distributions. The cost of the debugging operations as expressed above would then be a probabilistic cost which could be very useful in practical terms.

By making some order of magnitude type estimates of some typical program parameters, it is possible to directly compare the relative costs of the debugging operations. For an execution of an average PL/I program, the following parameter values are reasonable:

a = .1	p = 20
b = 20	q = .1
c = 10	r = 10
e = 5	s = 20
k = 100	t = 100
m = 1000	v = 2000
n = 200	w = 200

These values state that during execution there are a total of 1000 statements and 100 statement lists executed. Within these 1000 statements there are 2000 variable references and 200 assignment statements. There are 20 begin block activations, 10 of which are terminated by a return statement and 5 of which are terminated by GOTO statements. During execution, 20 procedure calls are made. The average length of the subscript list for variable references is .1

meaning that 1 out of 10 references has a subscript. Also, 1 out of 10 references has a structure qualifier. Although these values will vary greatly from program to program, this particular sample set can be used as a representative average set for a wide class of programs.

The following absolute costs result from the above parameters: (Costs are rounded to the nearest thousand.)

Debugging Operation	Absolute Cost (thousands)
Display Variable	92
Set Variable	96
Goto	4
Read Statement Counter List	20
Read Block or Procedure Name	1.6
Trigger on Variable Reference	92
Trigger on Block or Procedure Activation	1
Trigger on Statement	22

Some simple conclusions are immediately apparent from these figures. There seem to be three levels of costs. The relatively cheapest operations are Trigger on Block, Read Block, and Goto. These operations can be said to form a minimal low cost debugging system. The next level of costs is an order of magnitude higher. This level contains Read Statement and Trigger on Statement. These operations

are so costly as to make the lower level operations insignificant. So these two debugging operations in combination with the three at the low cost level form an intermediate cost debugging system. Of course, the extremely high cost operations are those associated with the interpretation of variable references: Display Variable, Set Variable, and Trigger on Variable Reference.

Additional insight can be gained by computing the absolute costs of the four primitive operations discussed above:

Primitive	Absolute Cost (thousands)
Variable Reference	92
Block and Procedure Activation	1
Block and Procedure Termination	.6
Statement Counter List Maintenance	20

Thus, the primitives form a definite cost heirarchy which underlies the cost relationships of the debugging operations. One very striking conclusion that can be drawn from the above costs is that debugging operations involving block and procedure activation and termination are extremely inexpensive. This conclusion has definite implications for practical debugger design. Debugging operations like Read Block or Procedure Name and Trigger on Block should be included

in all debugging systems because of their low cost. (Operations to read the subroutine stack are also in this class.) Anyone who has implemented a debugging system will confirm the low cost of these operations. However, these low cost operations are not included in many current debugging systems whereas the much higher cost operation of Trigger on Statement is included in almost every system. This fact illustrates the need for more absolute cost analysis of debugging systems. This practical and useful conclusion also verifies the assertion that absolute cost analysis can be a useful tool in cost prediction.

VII. Conclusions

The purpose of this thesis has been to develop a general method for cost analysis of software systems. The overall philosophy of the method is given by the Principle of Cost Analysis in Section V. This philosophy has been used to perform cost analysis on debugging systems as a specific example of the method. It would be premature to claim that this thesis presents a scientific method for cost analysis. Rather the foundations for the art of cost analysis have been built. Even though a well-defined and exact algorithm for estimating cost would be desirable, an art in the hands of a skilled artist could prove extremely useful.

There has been little or no theoretical work in the area of practical cost analysis. So this thesis represents a first attempt at something that is badly needed for computer software design. The cost figures for debugging operations calculated in Section VI could be of some usefulness in the early stages of debugging system design. Also, the general art of cost analysis could hopefully be applied with similar success to other software systems. However, the main contribution to this thesis seems to be that it has demonstrated that cost analysis is possible on a general theoretical level. The thesis should serve as a stimulation to researchers who might eventually develop a science

in cost analysis. This is the field of "alternate machines". In order to perform accurate cost analysis it is necessary to consider alternate ways of implementing the same operation. To estimate the cost of a particular feature, it is really necessary to determine how the software system could be redesigned if this feature were not present. Thus, alternate machines (i.e. alternate implementations) must be considered. This type of cost analysis is of course incremental cost analysis. Absolute cost can be a help in the early stages of design. But the real question that people usually want to answer is how much will it cost in terms of efficiency to add a particular feature. This question can be accurately answered only by incremental cost analysis which requires a consideration of alternate machines.

Appendix A

List of Debugging Operations

Debugging operations can be divided into four categories: breakpoints, data display, history, and editing. Breakpoints are of the form - execute until a certain event occurs. Data display is just changing and displaying data variables. History is a record of the past execution of the program. It can be a complete history as in EXDAMS[1], or it can just contain a few previous values of certain variables as in HELPER.[10] Editing is sometimes called incremental compiling and refers to making changes in the source program. After a thorough study of interactive debugging systems, the following list has been gathered of possible operations:

I. Breakpoints

Execute program until

1. Certain variable is referenced
2. Certain statement is reached
3. Certain variable is changed
4. Certain variable takes on some specific value
5. A transfer statement is executed.
6. Certain subroutine is entered
7. One of a set of variables is referenced or changed

II. Data Display

1. Display variable
2. Set a variable to a specific value

III. History

A. Audit (summary of events)

1. How many times was a particular subroutine entered
2. Which portions of the program were never executed
3. Which variables were never set
4. Where are all the places that changed a certain variable

B. Execution History

1. Where did a variable take on its current value
2. Read the subroutine stack
3. When was the last time a variable was referenced
4. When was the last time a variable was changed
5. When was the last time a certain statement was executed

IV. Editing

1. Print certain statement
2. Delete certain statement
3. Insert certain statement
4. Move groups of statements
5. Substitute for occurrences of certain patterns
6. Find all references to a certain variable

Appendix B

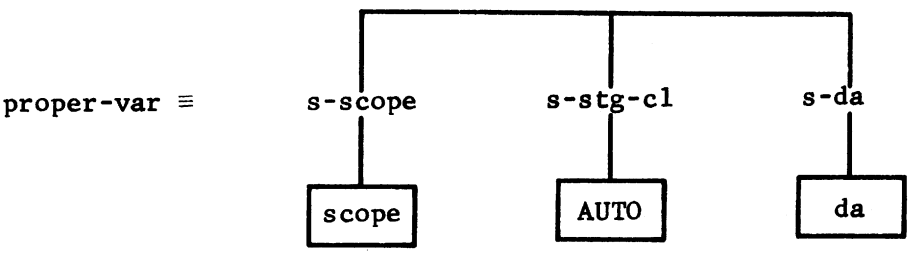
Abstract Syntax of Program

The method of predicates used for abstract syntax is simple and can be clarified by an example shown below:

Predicate

is-proper-var = (⟨s-scope: is-scope⟩,
⟨s-stg-cl: is-AUTO⟩,
⟨s-da: is-da⟩)

Tree representation



The following is the complete abstract syntax of the subset of PL/I used in this thesis. The notation is formally described in pages 115-119 of "On the Formal Description of PL/I".[12]

Abstract syntax of a program

Programs and Blocks

```
is-program = ({⟨id: is-body⟩ || is-id(id)})
is-body = (is-block; ⟨s-param-list: is-id-list⟩)
is-block = (⟨s-decl-part: is-decl-part⟩, ⟨s-name: is-n⟩,
            ⟨s-st-list: is-st-list⟩)
```

Declarations

```
is-decl-part = ({⟨id: is-decl⟩ || is-id(id)})
is-decl = is-proper-var ∨ is-entry ∨ is-LABEL
is-proper-var = (⟨s-scope: is-scope⟩,
                 ⟨s-stg-cl: is-stg-cl ∨ is-Ω⟩,
                 ⟨s-da: is-da⟩)
is-scope = is-INT ∨ is-EXT ∨ is-PARAM
is-stg-cl = is-AUTO
is-entry = (⟨s-scope: is-scope⟩,
            ⟨s-den: is-body ∨ is-n ∨ is-Ω⟩)
is-da = is-array ∨ is-struct ∨ is-scalar
is-array = (⟨s-lbd: is-expr⟩,
            ⟨s-ubd: is-expr⟩,
            ⟨s-elem: is-da⟩)
is-struct = is-succ-list
is-succ = (⟨s-id: is-id⟩,
           ⟨s-da: is-da⟩)
```

```
is-scalar = (⟨s-mode: is-mode⟩,
             ⟨s-scale: is-scale⟩)
is-mode = is-REAL ∨ is-INTG
is-scale = is-FLOAT ∨ is-FIXED
```

Statements

```
in-st = (⟨s-label-list: is-id-list⟩,
         ⟨s-proper-st: is-proper-st⟩)
is-proper-st = is-block ∨ is-call-st ∨ is-assign-st ∨ is-while-group
               ∨ is-if-st ∨ is-goto-st ∨ is-return-st ∨ is-null-st
               ∨ is-st-list
is-call-st = (⟨s-id: is-id⟩,
              ⟨s-arg-list: is-expr-list⟩)
is-assign-st = (⟨s-lp: is-ref⟩,
                ⟨s-rp: is-expr⟩)
is-goto-st = (⟨s-st: is-GOTO⟩,
              ⟨s-ref: is-ref⟩)
is-return-st = is-RETURN
is-if-st = (⟨s-expr: is-expr⟩,
            ⟨s-then-st: is-st⟩,
            ⟨s-else-st: is-else-st⟩)
is-else-st = is-st ∨ is-Ω
is-while-group = (⟨s-while-expr: is-expr⟩,
                  ⟨s-st-list: is-st-list⟩)
```

is-null-st = is-NULL

Expressions

is-expr = is-ref | is-const

is-ref = (⟨s-id-list⟩ is-id-list⟩,

⟨s-arg-list⟩ is-arg-expr-list⟩)

is-arg-expr = is-expr | is-*

Appendix C

Abstract Syntax of the State of the Machine

is-state = (⟨s-s: is-stg⟩,
⟨s-un: is-un⟩,
⟨s-dn: is-dn⟩,
⟨s-at: is-at⟩,
⟨s-ag: is-ag⟩,
⟨s-e: is-e⟩,
⟨s-ei: is-ei ∨ is-Ω⟩,
⟨s-d: is-d⟩,
⟨s-ci: is-ci⟩,
⟨s-c: is-c⟩)

S = s-s(ξ) storage

UN = s-un(ξ) unique name counter

DN = s-dn(ξ) denotation directory

AT = s-at(ξ) attribute directory

AG = s-ag(ξ) aggregate directory

E = s-e(ξ) environment

EI = s-ei(ξ) epilogue information

D = s-d(ξ) dump

CI = s-ci(ξ) control information

C = s-c(ξ) control

Local State Components

is-d = is-Ω ∨ (⟨s-e: is-e⟩,
⟨s-ei: is-ei⟩,
⟨s-d: is-d⟩,
⟨s-ci: is-ci⟩,
⟨s-c: is-c⟩)

is-e = ({⟨is: is-n⟩‖ is-id(id)})

is-ei = (⟨s-block-act: is-block-name⟩,
⟨s-free-set: is-n-set⟩, ⟨s-main: is-Ω ∨ is-ON⟩)

is-block-name = (⟨s-type: is-BLOCK ∨ is-PROC⟩,
⟨s-name: is-n⟩)

is-ci = is-Ω ∨ (⟨s-tx: is-st-list ∨ is-if-st⟩,
⟨s-sc: is-integer⟩,
⟨s-cd: is-cd⟩)

is-cd = (⟨s-ci: is-ci⟩,
⟨s-c: is-c⟩)

is-c = see discussion of control trees in "On the Formal Description of PL/I"[12]

is-n = the set of unique names n_0, n_1, ... which are elementary objects

Global State Components

is-dn = ({⟨n: is-den⟩ || is-n(n)})

is-den = is-n ∨ is-entry-den

is-entry-den = (⟨s-body: is-body⟩,
⟨s-e: is-e⟩)

is-ag = (⟨s-pp: is-pp⟩,
⟨s-eda: is-eda⟩)

is-pp = is-ptr ∨ is-pp-list

is-ptr = see discussion of storage in "On the Formal Description of PL/I"[12] - ptr identifies area of storage

is-at = ({⟨n: (⟨s-attr: is-attr⟩,
⟨s-e: is-e⟩)⟩ || is-n(n)})

is-attr = is-proper-var ∨ is-entry

Appendix D

Execution Time for Interpreter

When a new block is activated by int-block there are several actions that must be taken. The epilogue information (EI) must be initialized with "block-ei", and the old state must be stacked with "stack". Notice that block-ei and stack are not instruction schemata but only notational conveniences so their execution time is part of int-block.

for a block t

int-block(t) =

s-ei: block-ei(t)

s-d : stack(ξ)

s-ci: Ω

s-c : epilogue;

int-st-list(s-st-list(t));

update-dn(s-decl-part(t));

update-at(s-decl-part(t));

update-env(s-decl-part(t))

block-ei(t) =

μ(EI; (⟨s-block-act: μ_o(⟨s-type: BLOCK⟩,

⟨s-name: n_t⟩)⟩,

⟨s-free-set: Ω⟩)

where: n_t = s-name(t)

stack(ξ) =

μ_o(⟨s-e: s-e(ξ)⟩,
⟨s-ei: s-ei(ξ)⟩,
⟨s-d: s-d(ξ)⟩,
⟨s-ci: s-ci(ξ)⟩,
⟨s-c: s-c(ξ)⟩)

Execution time for stack

Operation	Execution Time
μ_o with 5 arguments	5
simple selectors s-e, s-ei, s-d, s-ci, s-c	5
	10 units

Execution time for block-ei

Operation	Execution Time
μ with 2 arguments	2
simple selector s-ei	1
μ_o with 2 arguments	2
simple selector s-name	1
	6 units

Execution time for int-block

Operation	Execution Time
block-ei	6
stack	10
selector s-st-list	1
selector s-decl-part	1
state transformation for 4 state components	4
Total Execution Time for int-block	= 22 units

Observing the control component (s-c) of the block activation, one notices that first the new environment is created. Then the attribute and denotation directories are updated followed by the interpretation of the statements of the block. The last action performed is the epilogue which frees the local variables and unstacks the dump.

```
epilogue =
    unstack;
        free-local

unstack =
    s-e : s-e(D)
    s-ei: s-ei(D)
    s-d : s-d(D)
    s-ci: s-ci(D)
    s-c : s-c(D)
```

Instruction schema	Execution Time
unstack	11
epilogue	0

The execution time for free-local is not needed for this thesis; however, the reader can find this instruction schema and many others in section 7 of "On the Formal Description of PL/I".[12]

When a procedure call is interpreted, actions are taken that are similar to those upon block activation. The initial environment is the environment that existed when the procedure was declared. int-call sets up the new epilogue information with "proc-ei" and stacks the local state components with "stack". Notice the similarity between int-call and int-block.

```
int-call(body,  e, arg-list) =
    s-e : e
    s-ei: proc-ei(body)
    s-d:  stack(ξ)
    s-ci: Ω
    s-c:  epilogue;
              int-st-list(s-st-list(body));
                  update-dn(s-decl-part(body));
                      install-arg-list(arg-list, s-param-list(body),
                                       s-decl-part(body));
```

update-at(s-decl-part(body));

update-env(s-decl-part(body))

proc-ei(body) =

μ_o(⟨s-block-act: μ_o(⟨s-type: PROC⟩,

⟨s-name: n_b⟩)⟩,

⟨s-free-set: Ω⟩)

where: n_b = s-name(body)

Instruction schema	Execution Time
int-call	23

A procedure or block activation may be terminated by a return statement. If s-main (EI) is set this is the outermost procedure and the entire program is terminated. Otherwise, return is called successively to terminate nested block activations until the current procedure is terminated.

int-return-st =

is-Ω* s-main (EI) → return

T → s-c: Ω

return =

is-BLOCK * s-type * s-block-act(EI) →

s-d: μ(D; ⟨s-c: return⟩)

s-c: epilogue

is-PROC * s-type * s-block-act(EI) →

epilogue

Instruction schema	Execution Time
int-return-st	3
return	8 for BLOCK, 4 for PROC

A Goto statement is interpreted by searching the text components for the statement with the indicated label. The search is performed by first searching the current statement list and then successive statement lists in the control dumps (s-cd) of the control information (CI). If the label is not found in the current block activation, the block is terminated and the next block searched. The actual search of statement lists is performed by "search". If the search is successful, the result is a statement counter list that localizes the statement with the current statement list.

goto-search(id) =

¬ is -Ω * search(id, s-tx(CI)) →

goto-jump(search(id, s-tx(CI)))

¬ is -Ω (CI) →

s-ci: s-ci * s-cd(CI)

s-c : goto-search(id)

¬ is -Ω * s-c(D) & is-BLOCK * s-type * s-block-act(EI) →

s-d : μ(D; ⟨s-c: goto-search(id)⟩)

s-c: epilogue

search(id, t) =

is-st(t) & (∃i)(id = elem(i, s-label-list(t))) → ⟨ ⟩

is-st(t) → search(id, s-proper-st(t))

is-st-list(t) → search-l(id, t, 1)

is-if-st(t) & ¬is-Ω * search(id, s-then-st(t)) →

⟨T⟩⌢search(id, s-then-st(t))

is-if-st(t) & ¬ is-Ω * search(id, s-else-st(t)) →

⟨F⟩⌢search(id, s-else-st(t))

T → Ω

search-l(id, t, i) =

i > length(t) →

¬is-Ω * search(id, elem(i, t)) → ⟨i⟩⌢search(id, elem(i, t))

T → search-l(id, t, i+1)

Once the label is located, the sc-list is used by goto-jump to build up the control dump stack until the proper statement is reached.

goto-jump(scl) =

is-integer * head(scl) & length(scl) = 1 →

s-ci: ci_s

s-c: int-next-st;

int-st(st_s)

is-integer * head(scl) →

s-ci: μ_o(⟨s-tx: s-proper-st(st_s)⟩,

⟨s-cd: μ_o(⟨s-ci: ci_s⟩,

⟨s-c: int-next-st⟩)))

s-c: goto-jump(tail(scl))

length(scl) = 1 →

s-ci: s-ci * s-cd(CI)

s-c: int-next-st;

int-st(st_s)

T →

s-ci: μ(CI; ⟨s-tx: s-proper-st(st_s)⟩)

s-c: goto-jump(tail(scl))

where: ci_s = μ(CI; ⟨s-sc: head(scl)⟩)

st_s = (is-integer * head(scl) → elem(head(scl), s-tx(CI)),

head(scl) → s-then-st * s-tx(CI),

T → s-then-st * s-tx(CI))

The execution time for a Goto is extremely complicated and depends on the properties of statement lists searched. Eight parameters are required to characterize this execution time.

Instruction schema	Execution time
goto-search	11n + 4s + 8f - 3e + 12b + 7c
goto-jump	151 - 5t

where: (The following parameters refer to the text that is searched before the statement is found)

n = total number of statements

s = total number of statement lists

f = total number of IF-statements

e = total number of single statement then or else clauses

b = total number of block levels unstacked

c = total number of control dump levels unstacked

(the following parameters refer to the statement counter list that localizes the statement for the Goto)

l = length of statement counter list

t = number of T or F's in list

The parameter n is the total number of statements encountered in the entire search. The parameter e refers to all then or else clauses which are not statement lists but single statements. The parameter b is the number of blocks terminated during the search. The parameter c refers to the total of all control dumps unstacked during the entire search. Although these parameters seem complex, they can be easily found for any particular Goto in a program. All that is needed is a knowledge of the search method of the algorithm. First the current statement list is searched, then the list containing the current list, etc. If the label is not found in the current block, the containing block is searched and so on. By counting the statements, statement lists, IF-statements, and blocks terminated, the parameters can be calculated.

An assignment statement requires the evaluation of the variable reference with eval-ref-gen and the evaluation of the expression with eval-expr. The actual assignment is performed by convert-assign.

```
int-assign-st(st) =
     convert-assign(gen, op);
          op:   eval-expr(s-rp(st), E),
          gen:  eval-ref-gen(s-lp(st), E)

convert-assign(gen, op) =
     assign(gen, op-1);
          op-1: convert(s-da(gen), op)

assign(gen, op) =
     is-scalar * s-da(gen) →
          s-s:   el-ass(s-pp(gen), s-vr(op))

convert(da, op) =
     mk-op(da, vr);
          vr:   represent(da, v)
          v:    value(s-da(op), s-vr(op))
```

represent(da, v) is an implementation defined primitive that gets a value representation with a data attribute da from the value v.

value(da, vr) is an implementation defined primitive that extracts a value from a value representation vr with a data attribute da.

Instruction schema	Execution time
value	1
represent	1
mk-op	1
convert	2
assign	7
convert-assign	1
int-assign-st	4

The total execution time of all these instructions can be included in int-assign-st since their individual identity is not important. The time for eval-expr must also be added to get a total time of 19 for int-assign-st.

For simplicity, the only form of expressions allowed in the PL/I subset is a simple reference or a constant. In eval-expr the instruction eval-ref-gen is used to find a pointer into storage and a data attribute for the variable reference. pass-gen-op is used to access the storage and create an operand (op), an internal form used for transferring values.

```
eval-expr(expr, e) =
    is-ref(expr) →
        pass-gen-op(gen, S)
            gen: eval-ref-gen(ref, e)
    is-const(expr) → PASS:  expr
```

Instruction schema	Execution time
eval-expr	3 for reference, 2 for constant

The instruction eval-ref-gen is completely listed in section 7 of "On the Formal Description of PL/I"[12]. It is essentially the same as the eval-ref-gen needed for this thesis. The only difference is the instruction eval-gen because the PL/I subset in this thesis does not allow controlled or pointer variables. The execution time for all instructions used by eval-ref-gen is included in the total shown below.

eval-gen(ref, e) =

$$\text{is-proper-var}(\text{attr}_r) \;\&\; \text{is-ref}(\text{ref}) \;\&\; \text{is-gen}(\text{gen}_r) \rightarrow$$

$$\underline{\text{PASS}}: \; \text{gen}_r$$

$$T \rightarrow \underline{\text{error}}$$

where: $n_r = \text{head} * \text{s-id-list}(\text{ref})(e)$

$\text{attr}_r = \text{s-attr} * n_r(\underline{\text{AT}})$

$\text{gen}_r = \text{head} * n_r(\underline{\text{DN}})(\underline{\text{AG}})$

Instruction schema	Execution time
eval-ref-gen(ref, e)	$36 + q(39 + 4t) + 49a + s(5c + 13)$

(Recall that a reference (ref) consists of an id-list of qualifiers for structure reference and an argument list of subscripts for array reference. A subscript may be a * in which case the entire crossection of the array is used.)

where:

q = length of qualifier list (= length of id-list - 1)

a = length of subscript list (not including *'s)

s = number of *'s in subscript list

c = crossection size of array at each *

t = number of structure parts at each qualifier level

The parameters c and t are clarified below.

Example:

Dimension A(10,7,25)

If subscript list for A is (x, y, *) where $1 \leq x \leq 10$ and $1 \leq y \leq 7$, then c = 25.

```
Declare X,
     1 A,
       2B,
       2C,
       2D,
     1 E,
       2 F
```

If id-list is X.A.C., then the qualifier list is A.C and t = 2+3 = 5. At qualifier level A there are 2 structure parts, and at qualifier level C there are 3 structure parts.

Bibliography

1. Balzer, R. M., "EXDAMS - Extendable Debugging and Monitoring System," SJCC proceedings, 1961.

2. Bratman, H., Martin, H., Perstein, E., "Program composition and Editing with an On-Line Display," FJCC proceedings, 1968.

3. Evans, T. G., Darley, D. L., "On-Line Debugging Techniques: A Survey," FJCC proceedings, 1966.

4. Glass, R. L., "Splinter - A PL/I Interpreter Emphasizing Debugging Capability," The Computer Bulletin, September 1968.

5. Hassler, E. B., Editing Compilers, Their Feasibility and Effects, University of Illinois PhD Thesis, 1966.

6. IBM Laboratory Vienna, Method and Notation for the Formal Definition of Programming Languages, TR 25.087, June 1968.

7. IBM Laboratory Vienna, Informal Introduction to the Abstract Syntax and Interpretation of PL/I, TR 25.083, June 1968.

8. IBM Laboratory Vienna, Abstract Syntax and Interpretation of PL/I, TR 25.082, June 1968.

9. IBM System/360 PL/I Reference Manual, March 1968.

10. Kulsrud, H. E., "HELPER: An Interactive Extensible Debugging System," ACM National Conference, 1969.

11. Lock, K., "Structuring Programs for Multi-Program Time Sharing on-Line Applications," FJCC proceedings, 1965.

12. Lucas, P., Walk, K., "On the Formal Description of PL/I," Annual Review in Automatic Programming, Vol. 6, Part 3, 1969.

13. McCarthy, J., "A Formal Description of a Subset of Algol," Formal Language Description Languages, T. B. Steel, editor, North-Holland Publishing Co., Amsterdam, 1966.

14. Ramchandani, C., Debugging System to Run Interpretively in Virtual Memory, MIT M.S. Thesis, January 1970.

15. Ryan, J. L., Crandall, R. L., Medwedeff, M. C.,"A Conversational System for Incremental Compilation and Execution in a Time-Sharing Environment," FJCC proceedings, 1966.

16. Warshall, S., "On Computation Cost," Annual Review in Automatic Programming, Vol. 5, Pergamon Press, 1968.

CS-TR Scanning Project
Document Control Form

Date : 1/23/96

Report # LCS-TR-90

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 114 (120 IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOWS TITLE PAGE

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-114) UNUM'ED TITLE, BLANK, ABSTRACT, ACKNOWLEDGMENT, TABLE CONTENTS, 4-112
(115-120) SCANCONTROL, COVER, DOD, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 1/23/96 Date Scanned: 1/25/96 Date Returned: 2/1/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Massachusetts Institute of Technology Project MAC	UNCLASSIFIED
	2b. GROUP None

3. REPORT TITLE

Cost Analysis of Debugging Systems

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

B.S. and M.S. Thesis, Dept. of Electrical Engineering, January 1971

5. AUTHOR(S) (Last name, first name, initial)

Lester, Bruce P.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
September 1971	116	16

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-70-A-0362-0001 b. PROJECT NO.	MAC TR-90 (THESIS)
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

A general method is presented for performing cost analysis of interactive debugging systems. The method is based on an abstract model of program execution. This model is derived from the interpreter used in the Vienna method of semantic definition of PL/I. A brief discussion of the overall operation and significance of the Vienna interpreter is included.

Four assumptions are made which allow execution times to be calculated for algorithms of the Vienna interpreter. A notion of absolute cost is developed which requires the use of these execution times for cost analysis of features of debugging systems. A set of eight interactive debugging operations is thoroughly analyzed using the method of cost analysis. Some overall conclusions are drawn about the relative costs of various types of debugging operations and some suggestions are made for minimal cost debugging system design.

14. KEY WORDS

Programming Linguistics
Interpreters
Interactive Computing
Semantic Definition
Cost Analysis
Debuggers
Time-Sharing

DD FORM 1 NOV 65 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 1/25/1996
M.I.T. Libraries
Document Services