

AD-A 198 753

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NSWC TR 87-181			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Naval Surface Warfare Center		6b. OFFICE SYMBOL (If applicable) K52	7a. NAME OF MONITORING ORGANIZATION Strategic Systems Program Office		
6c. ADDRESS (City, State, and ZIP Code) Dahlgren, VA 22448-5000			7b. ADDRESS (City, State, and ZIP Code) Washington, DC 20376-5002		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Systems Program Office		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) Washington, DC 20376-5002			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO. 64363N	PROJECT NO.	TASK NO. J0951
			WORK UNIT NO.		
11. TITLE (Include Security Classification) Higher Order Software - Evaluation and Critique					
12. PERSONAL AUTHOR(S) Huber, Hartmut G. M.					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 1976 To 1987		14. DATE OF REPORT (Yr., Mo., Day) 1987, August	
15. PAGE COUNT 69					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	Software Specifications, 'Functional Programming,' Abstract Data Type, 'Efficiency, Productivity, Automatic Documentation. (560)		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
Higher Order Software (HOS) is presented by its authors as a methodology for system design and implementation based on a functional view of a system and its development process. This abstract methodology is implemented as a set of integrated tools, collectively called USEIT. This report addresses and evaluates the theoretical aspects of HOS as well as the practical aspects of its implementation in the form of USEIT. <i>Keywords:</i>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Hartmut G. M. Huber			22b. TELEPHONE NUMBER (Include Area Code) (703) 663-7510		22c. OFFICE SYMBOL Code K52

DD FORM 1473, 84 MAR

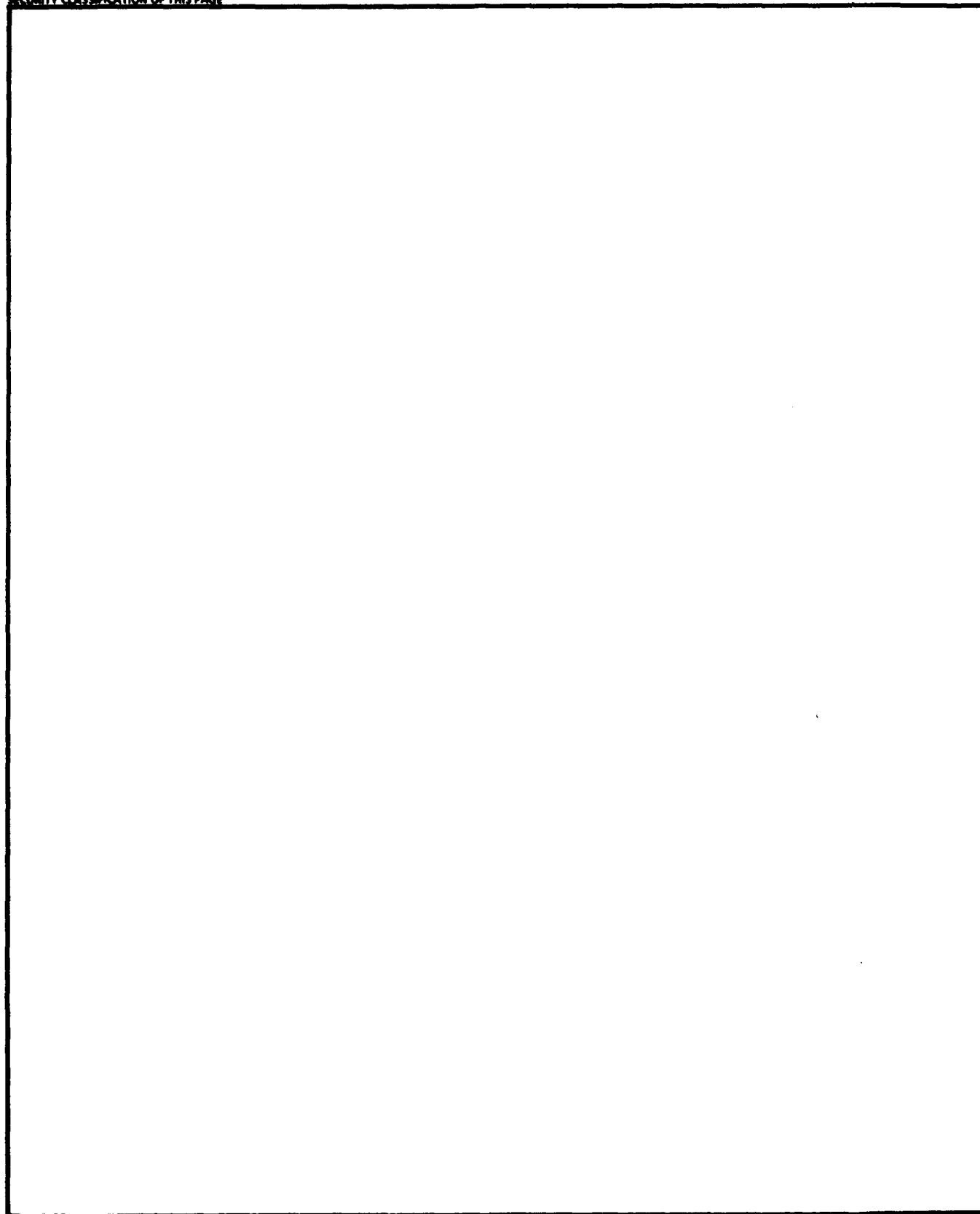
EDITION OF 1 APR 83 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

FOREWORD

This report was written in the Quality Assurance Branch (K52), Submarine Launched Ballistic Missile (SLBM) Software Development Division (K50), of the Strategic Systems Department (K) at the Naval Surface Warfare Center (NSWC), Dahlgren, Virginia.

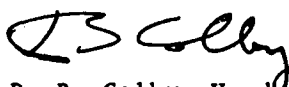
The purpose of this document is to describe, evaluate, and critique the HOS approach for system implementation and its automation in the form of a set of tools called USEIT.

This report was reviewed by personnel in K52, the Operational Support Branch (K53), and the Operational Systems Branch (K54). The help of the reviewers is appreciated, however, the author is solely responsible for the conclusions expressed in this report.

This project was funded by the Strategic Systems Program Office, Washington, D.C. 20376, under task number K36403.05.

Questions, comments, and suggestions concerning the material presented in this document should be directed to the Commander, Naval Surface Warfare Center, ATTN: K52, Dahlgren, Virginia, 22448-5000.

Approved by:



D. B. Colby, Head
Strategic Systems Department

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



CONTENTS

CHAPTER 1	INTRODUCTION	1-1
CHAPTER 2	THEORETICAL ASPECTS OF HOS	2-1
2.1	BACKGROUND	2-1
2.2	AXES	2-2
2.2.1	Functions	2-2
2.2.2	Abstract Control Structures	2-2
2.2.2.1	Composition	2-3
2.2.2.2	Set Partition	2-3
2.2.2.3	Class Partition	2-3
2.2.3	Axioms of Control	2-3
2.2.4	Abstract Data Types	2-5
2.2.4.1	Example of an Abstract Data Type	2-5
2.2.4.2	Derived Functions versus Defined Functions	2-6
2.3	MACHINE INDEPENDENCE OF AXES	2-8
2.4	AXES AND PROGRAMMING LANGUAGES	2-8
2.5	PROOFS OF THEOREMS AND PROOFS OF PROGRAMS	2-9
CHAPTER 3	HOS IMPLEMENTED: USEIT	3-1
3.1	DESCRIPTION OF USEIT	3-1
3.1.1	AXES in USEIT	3-1
3.1.2	Analyzer	3-3
3.1.3	Resource Allocation Tool (RAT)	3-3
3.2	PROBLEM AREAS OF USEIT	3-4
3.2.1	Fundamental Problems	3-4
3.2.1.1	Higher Level Features	3-4
3.2.1.2	Notation for Expressions	3-4
3.2.1.3	Efficiency	3-4
3.2.2	USEIT Design and Implementation	3-5
3.2.2.1	Language Independency of the Analyzer	3-5
3.2.2.2	Control Map Representation	3-5
3.2.2.3	Recursion Restrictions	3-6
3.2.2.4	Automatic Documentation in USEIT	3-7
3.2.2.5	Documentation of the USEIT System	3-8
3.3	PRODUCTIVITY	3-8
CHAPTER 4	CONCLUSION	4-1
CHAPTER 5	REFERENCES	5-1
APPENDIX A	ADDITION OF VECTORS	A-1
A.1	MATHEMATICAL FORMULATION	A-1
A.2	USEIT CONTROL MAPS FOR VECTOR ADDITION	A-1

APPENDIX B	ADDITION OF MATRICES	B-1
B.1	ARRAY BASED ALGORITHM	B-1
B.2	USEIT CONTROL MAPS FOR MATRIX ADDITION	B-1
B.3	USEIT GENERATED DOCUMENTATION	B-4
B.4	LIST BASED ALGORITHM	B-6
APPENDIX C	GREATEST COMMON DIVISOR OF A LIST OF INTEGERS . .	C-1
APPENDIX D	PARSING BY RECURSIVE DESCENT	D-1
D.1	MATHEMATICAL FORMULATION	D-1
D.2	USEIT CONTROL MAPS FOR THE RECURSIVE DESCENT PARSER	D-3
D.3	USEIT GENERATED DOCUMENTATION FOR THE RECURSIVE DESCENT PARSER	D-9
APPENDIX E	THE TOWERS OF HANOI GAME	E-1
E.1	MATHEMATICAL FORMULATION	E-1
E.2	USEIT CONTROL MAPS FOR HANOI, VERSION 1	E-2
E.3	USEIT CONTROL MAPS FOR HANOI, VERSION 2	E-4
DISTRIBUTION		(1)

CHAPTER 1

INTRODUCTION

When software systems were small, no formal discipline for developing them was needed. As software systems grew to enormously complex systems in the late sixties and seventies, the need for a software development process producing reliable economical systems at predictable cost became apparent and urgent. It was in the wake of the search for such a development method that the conceptual framework of Higher Order Software (HOS) was born. Like many other systems with that same goal, it was called a methodology, a system of methods. Unlike many other systems, HOS was conceived as a formalized method based on the mathematical concept function and on the principle of abstraction.

HOS is presented by its authors as a unified methodology for system design and implementation derived from a functional view of a system and its development process. This abstract methodology is implemented as a set of integrated tools, collectively called USEIT. One of the functions of USEIT is to generate code in a programming language such as FORTRAN or PASCAL for a system being designed. Thus, USEIT depends on a target language. In fact, for each target language supported by HOS a different USEIT system exists.

This report addresses the theoretical aspects of HOS, the practical aspects of its implementation in the form of USEIT, and the particular USEIT system with FORTRAN as its target language, here called FORTRAN USEIT.

CHAPTER 2

THEORETICAL ASPECTS OF HOS

2.1 BACKGROUND

HOS and its automation in the form of USEIT were developed by M. Hamilton and S. Zeldin starting in 1975 when they both worked at the Charles Stark Draper Laboratory on the Appollo project. Subsequently they founded a Company called Higher Order Software (HOS). For clarity, the company, in particular its founders, will be referred to as "HOS" and the methodology as "HOS methodology".

While working on the Apollo project Hamilton and Zeldin discovered that most errors in a software system are interface errors between the components of the system. A second realization was that for the different phases of the software development process either no formal methods were used or different, basically unrelated formalisms were used that did not allow automatic analysis. A third recognition was that the von Neumann type architecture of computing machines and of programming languages encouraged machine dependent thinking in terms of representation of data and specific state transition algorithms rather than machine independent thinking in terms of the functional abstract properties of a system. These were the major ideas that led to the HOS methodology, a formalized functional approach for all phases of the program development process [9,10,11,12,14,15] and for all aspects of it including management [13] and documentation [19].

In addition to making full use of the concept function and the principle of abstraction for system development, HOS also recognized the importance and relevance of other ideas already developed in this area, the major one being the concept of abstract data types. HOS advocated these general ideas with eloquence and passion. It is in emphasizing these ideas for a consistent view of a system starting from the top level down to the lowest level that HOS made a contribution to Computer Science. It is in realizing this vision via a formalism and a set of tools supporting this formalism that HOS went aground and foundered.

HOS implemented the general idea of a total functional approach to system building via a formalism called AXES. The purpose of AXES is to serve as a specification language that can be used for specifying the functional properties of systems as abstractly as desired. Ideally, it should be possible to specify a system without making a commitment to use particular data structures to represent data or a particular programming language to express the

algorithms or a particular machine to execute the software system. The formalism AXES, though not strictly and completely defined [9,10], supports, in principle, this approach to system design. However, the formalism, to be useful, must be supported by tools. Unfortunately, the tools provided in the form of the USEIT system fall far short of supporting the goals of AXES. More will be said on this in later sections.

2.2 AXES

Following is a brief review of the essential features of AXES and the relationship of this formalism to programming languages. Unfortunately, no standard definition of AXES has been published. The terminology changed over the years starting with the original papers [9,10] and ending with the USEIT Reference Manual [19]. The following sections describe the essential features of a version of the AXES formalism that served - in the judgment of the author - as a foundation of USEIT. Concepts and terminology lost by the wayside between 1976 and 1983 will not be discussed.

AXES has three fundamental components: functions, abstract control structures, and abstract data types.

2.2.1 Functions

Functions are either primitive functions that operate on data belonging to an abstract data type or are non-primitive. Non-primitive functions are either defined or derived. A "derived function" is specified implicitly via a set of relations with primitive functions (see section 2.2.4.2); a "defined function" is represented explicitly in terms of other functions. Every defined function, in particular the function representing the entire system, is defined in terms of other functions and, ultimately, in terms of primitive functions.

2.2.2 Abstract Control Structures

The means of defining functions in terms of other functions are provided in the form of abstract control structures. The use of abstract control structures to support the definition of functions in terms of other functions is peculiar to AXES. The objective is to allow full interface checking, that is, type checking of the parameters and checking the number of parameters. This led HOS away from the conventional notation for specifying functions using functional expressions and lambda abstraction, to a notation that identifies a function by a name, by a list of input variables, and a list of output variables and defines the effect of a function as the combined effect of lower level functions according to the rules of an abstract control structure.

Abstract control structures are the "forms" of specifying functions in terms of other functions. Primitive abstract control structures are composition, set partition, and class partition.

2.2.2.1 Composition

This is normal function composition. f is defined in terms of g and h by

$$f(x) = h(g(x)).$$

More generally, if the domain of h is a Cartesian product $D_1 \times \dots \times D_k$ then k functions $g_1, \dots, g_k$ can be composed with h to form f .

2.2.2.2 Set Partition

The domain of f is partitioned into two groups by a property P , $D_1 = \{x: P(x)\}$, $D_2 = \{x: \text{not } P(x)\}$ and f is expressed in terms of g and h by

$$f(x) = \text{if } P(x) \text{ then } g(x) \text{ else } h(x)$$

More generally, the domain of f may be partitioned into n ($n > 2$) subdomains and f is expressed in terms of n functions on these subdomains.

2.2.2.3 Class Partition

Both the domain and the range of values of f are represented as Cartesian products $D = D_1 \times D_2 = \{(x_1, x_2): x_1 \text{ in } D_1, x_2 \text{ in } D_2\}$, $V = V_1 \times V_2 = \{(y_1, y_2): y_1 \text{ in } V_1, y_2 \text{ in } V_2\}$, and f is expressed in terms of g and h by

$$f(x_1, x_2) = (g(x_1), h(x_2))$$

More generally, the domain and the range of values can be a Cartesian products $D_1 \times \dots \times D_k$, $V_1 \times \dots \times V_k$ and f is expressed in terms of $g_1, \dots, g_k$ where each g_i contributes one component $g_i(x_j)$ to the value of f .

2.2.3 Axioms of Control

Once the primitive abstract control structures are defined, other abstract control structures can be expressed as groupings of them. Some of the most useful ones are predefined in AXES; other abstract control structures can be defined by the user. With the possibility of defining abstract control structures the question arises: What are the rules for defining such abstract control structures? The answer in the HOS methodology is: The rules are defined by six axioms! The primitive abstract control structures as well

as the predefined ones satisfy these rules. User defined abstract control structures must be specified using the AXES formalism which guarantees that the axioms are satisfied.

The axioms formulate the rules for building up functions from given functions, or, equivalently, for decomposing functions into a structured set of lower level functions, in a peculiar, obscure, and unnecessarily difficult language. According to HOS [11], a "formal control system" is a finite tree in which the relationship between each internal node and its descendents is defined by a control structure. Each subtree represents a function. If the subtree is a leaf node then it represents a primitive function or a function defined by a separate tree. Otherwise, the control structure for the root of the subtree and the set of descendents describes how the function is expressed in terms of the descendent functions. The properties of any such relationship sufficient to guarantee function definitions with clearly defined interfaces are postulated as axioms. A consequence of the rules is that a function definition using defined control structures is always equivalent to a definition of this function using only the primitive control structures ([17], page 33). Systems so defined will be reliable in the sense that complete type checking can be done and no timing conflict can arise [11].

A more severe conflict, however, arises for the conceptual formulation of the HOS approach from the wording of axiom one describing function invocation. This axiom says: "A given module controls the invocation of the set of functions on its immediate, and only its immediate lower level" (wording from [17], page 35). The "module" represents a function whose value is computed as the collective effect of invoking the lower level functions according to the abstract control structure used. The problem is the word "only" since it implies that a module cannot invoke a function if that function is not one of the descendants of the module. Yet, to achieve any significant power in computing one must allow the recursive invocation of functions, and AXES does allow that. In the picture of the function tree this means that a node may invoke a function represented by a higher level node. This is apparently not permitted by axiom one.

If the formulation of axiom one is accepted then the concept of a finite tree becomes inadequate for representing a defined function. Either we introduce a graph for this purpose or an infinite collection of finite trees each finite tree representing the invocation structure for a particular computation. Since, for recursive functions, the invocation structure depends on the input value of the recursive function, there is, in general, a different invocation tree for different input values. For nonrecursive functions the invocation structure is static, that is, independent of the input values. Therefore, in this case, one single finite tree suffices to represent the computation of all function values, hence, to represent the function.

HOS does not address this problem adequately. They simply allow recursion and leave the apparent conflict with axiom one unresolved. It appears that originally HOS modeled the formal control structure after the hierarchical structure of an organization. In fact, on many occasions HOS uses the organizational structure of a Company or a military Command as an example of the hierarchical decomposition of a function. This, however, is misleading in one

essential point which is at the root of the problem with axiom one. The static invocation structure of a function is, in general, not a tree but a graph. Functions are not built up in terms of other functions in a strictly hierarchical manner.

As an afterthought, the question arises whether the strict hierarchy is an adequate organizational form for Companies or the military. Perhaps, the organization of Companies should be modeled after the organization of functions.

2.2.4 Abstract Data Types

HOS uses the algebraic approach for defining abstract data types developed by Guttag [8], Zilles [23], and others [2,3,17] in the mid seventies. The two papers by Cushing, Algebraic Specification of Abstract Data Types, and The Intrinsic Types of AXES, contained as appendices in [10] are a lucid, delightful description of this approach.

An abstract data type is characterized by a set S representing the data being defined, a list of functions called primitive operations, and a set of axioms describing the interactions of the primitive operations with one another. AXES has the most commonly used data types predefined as intrinsic data types: boolean, property (of T), set (of T), natural, integer, rational, line. Both "property (of T)" and "set (of T)" are schemas of data types. The schema becomes a data type for each value of the parameter T which must be a data type. AXES also provides a mechanism for the user to define new abstract data types.

The axioms for a data type define all the relevant characteristics of the data type. Any implementation of the data type must satisfy these axioms and any implementation that does satisfy them is a valid implementation no matter how obscure it is or how different it is from other valid implementations. The differences of valid implementations are irrelevant for the correct use of the data type. They may be important for higher efficiency in execution time or memory use but are irrelevant for their functional use. It is this abstraction of the concept data from representation and implementation considerations that makes the concept "abstract data type" so powerful and so attractive.

2.2.4.1 Example of an Abstract Data Type

The simplest example of an abstract data type is "BOOLEAN" which will serve us to illustrate the use of the axioms of a data type and to explain the difference between primitive functions, derived functions, and defined functions. Following is the specification of the abstract data type BOOLEAN in the syntax of AXES [10]:

```

DATA TYPE: BOOLEAN;
PRIMITIVE OPERATIONS:
  boolean3 = AND(boolean1,boolean2);
  boolean2 = NOT(boolean1);
AXIOMS:
  WHERE True IS A CONSTANT BOOLEAN;
  WHERE False IS A CONSTANT BOOLEAN;
  AND(True,True)   = True;
  AND(True,False)  = False;
  AND(False,True)  = False;
  AND(False,False) = False;
  NOT(True)        = False;
  NOT(False)       = True;
END BOOLEAN;

```

Here boolean1, boolean2, boolean3 are variables ranging over the set BOOLEAN; True, False are constant elements in BOOLEAN and are, in fact, the only elements in this set. The two primitive functions "AND" and "NOT" are specified by a complete listing of all pairs of domain values and corresponding function values. It is well known that any function $F:D \rightarrow \text{BOOLEAN}$ where D is a finite Cartesian product of BOOLEAN can be represented by a finite composition of AND's and NOT's.

The function OR, for example, can be represented as follows:

$$(1) \quad \text{OR}(x,y) = \text{NOT}(\text{AND}(\text{NOT}(x),\text{NOT}(y)))$$

OR can also be defined implicitly by the following two relations:

$$(2a) \quad \text{OR}(x,\text{NOT}(x)) = T$$

$$(2b) \quad \text{NOT}(\text{OR}(x,y)) = \text{AND}(\text{NOT}(x),\text{NOT}(y))$$

Now consider the following relation:

$$(2c) \quad \text{OR}(x,\text{AND}(y,z)) = \text{AND}(\text{OR}(x,y),z)$$

If this relation is added to (2a)+(2b) then the three relations (2a)+(2b)+(2c) will not be compatible with the axioms of BOOLEAN. The combined set axioms+(2a)+(2b)+(2c) is inconsistent. This is easy to verify by standard truth table techniques. However, the three statements (2a)+(2b)+(2c) without the axioms are consistent which can be seen by interpreting NOT as the identity, OR and AND as EQUIV where EQUIV is defined by: $\text{EQUIV}(x,y) = T$ if and only if $x = y$.

2.2.4.2 Derived Functions versus Defined Functions

OR as specified via (1) is an example of a defined operation (function), OR as specified via the relations (2a)+(2b) is an example of a derived operation (function). Both AND and NOT as specified in the axioms are

examples of primitive functions. In general, a defined function is specified explicitly in terms of other functions via control structures, in the case of OR, only using composition; a derived function is specified implicitly via a set of relations.

It would be possible to add the definition of OR via a complete truth table to the axioms. This would mean that OR is considered a primitive function. It is not wrong to do so but unnecessary and therefore undesirable. It would complicate the data type BOOLEAN by apparent postulates about a function OR even though OR can be expressed in terms of AND and NOT without the need for any axioms.

When representing OR in terms of AND and NOT explicitly as in (1) nothing can go wrong. This type of specification says nothing beyond the fact that for each value in the domain a unique function value is defined. However, when specifying a function f implicitly via relations three different cases could occur:

- (1) There is no conflict and f is incompletely specified.
- (2) There is no conflict and f is completely specified.
- (3) There is a conflict. The relations for f are inconsistent with the axioms.

Given the axioms for AND and NOT, it is easy to verify that (2a) by itself is an example of case 1, (2a)+(2b) is an example of case 2, and (2a)+(2b)+(2c) is an example of case 3. (2a)+(2c) is another example of case 3 and (2c) by itself is another example of case 1.

There is a difference in the specification of defined functions and derived functions. Defined functions always exist consistently with the axioms, whereas derived functions must be specified such that no conflict with the axioms arises. In both cases, the function is described explicitly or implicitly in terms of the primitive functions.

It can happen that a desired function cannot be expressed in terms of the primitive functions of a data type. In this case it may be necessary to extend the type by adding a new function and appropriate axioms specifying the behavior of the function relative to the other primitive functions. An example of this situation would be if a data type BOOL would be defined with only one primitive operation, NOT. In this case, OR and many other functions cannot be expressed explicitly or implicitly in terms of NOT. One could extend BOOL to BOOL2 by adding the axioms for OR. Bool2 with NOT and OR as primitives is different from BOOLEAN which has NOT and AND as primitives, but BOOL2 and BOOLEAN are equivalent in the sense that any function that can be expressed in terms of NOT and AND can be expressed in terms of NOT and OR and vice versa.

2.3 MACHINE INDEPENDENCE OF AXES

The significance of using abstract data types in AXES is that they provide machine independence and therefore complete portability of a system specified in AXES. If the system is correctly specified in AXES then it will execute correctly on any machine on which the data types and their primitives are implemented correctly, that is, according to the axioms. Thus, the problem of portability of programs from machine A to machine B is reduced to the correct implementation of one fixed set of operations, the primitive functions, on both machines. This one time effort makes all programs using only these primitives automatically portable.

It must be realized, however, that AXES is not a closed formalism in the sense that it has a fixed set of abstract data types and associated primitive functions as its basis. Therefore, if a system is specified in AXES using newly defined data types not yet implemented on a machine A then an incremental effort to implement these newly defined data types on A is necessary to make the system executable on A.

Finally, a serious flaw in the perfect picture is that some of the most important abstract data types, such as the integers, are almost never implemented validly by any implementor, including HOS. The implemented data type normally uses a finite set of data elements whereas the algebraically defined data type requires an infinite set. Thus, imperfections creep into the methodology and proliferate, and the theoretical beauty of complete portability degenerates in reality to only partial portability.

2.4 AXES AND PROGRAMMING LANGUAGES

AXES is a formalism for specifying functions in terms of primitive functions. As such it is a functional programming language. A program is a function; running a program means applying a function to a set of parameter values. The means of defining a function in terms of other functions is restricted in AXES as compared to functional languages based on the lambda calculus.

Since the AXES formalism is defined rather loosely and incompletely in [10], restrictions will be discussed with respect to the implementation of AXES in USEIT even though these restrictions may apply to AXES as well.

HOS describes AXES as a specification language which is "not a programming language" ([15], page 41, [19]). It appears that there are two reasons for this view. The first reason is a very narrow concept of a "programming language" as a sequence of instructions ([17], page 61]. This, of course, disqualifies all functional languages and logical languages, such as Prolog, as programming languages. A second reason is that the functional elements in an AXES specification may not be executable; therefore, such a specification should not be called a program (private communication). However, this situation can also occur in a standard procedural programming language, such as ADA, by associating only input output properties with the names of the

primitive functions but no executable body. In any case, it appears somewhat miraculous that code can be generated automatically from a specification which is not a program. The implication is that in the HOS methodology all programming and all problems that plague programmers are avoided; only a very high level specification process remains. The truth, however, is that specifying in AXES is programming and, as it turns out, in USEIT, very tedious programming.

A second related issue is the correctness of the specified product. HOS gives credit to its methodology for producing "correct" specifications. The next section addresses the issue of correct programs.

2.5 PROOFS OF THEOREMS AND PROOFS OF PROGRAMS

In Mathematics, theorems are proved. A proof of a theorem is a sequence of statements, the theorem being the last one in the sequence, such that each statement is a logical consequence of previous statements in the sequence and of the basic assumptions about the underlying theory and of theorems already proved. Thus, a proof establishes a connection between the theorem and a "basis" already known to be true. The theorem and the basis are given. Finding a proof means constructing such a connection.

What does it mean when we say that a program is correct? What is a proof of a program? A program is correct if the program computes what the problem statement specifies. More formally, it means that the following statement is a true theorem: "The function P implemented by the program is the function F specified by the problem statement if the domain of P is restricted to the domain of F". A proof of a program, then, is a proof of the above theorem.

HOS considers a system specified in AXES to be correct if the specification is well formed according to the rules of AXES. Such a system is defined as a, possibly very large, structure built up from primitive functions such that all primitive functions involved operate on data of the correct type and produce data of the correct type. Intermediate functions in this structure will operate on and produce data of certain types the same way for all uses.

This is good to know that a specification is at least well formed but it has nothing to do with it being correct as a specification for a system whose properties are defined by some requirements. In the HOS terminology, every system specification that is well formed is correct; the concept "correct" is synonymous with the concept "well formed". The perplexed AXES user, however, will want to know: Which one of all these "correct" system specifications is the the one that implements my system?

Correctness of a program means a relationship between two different formulations, one being the program and the other one being the problem statement. The relationship is: The program computes what the problem statement requires. A proof of correctness is a proof of this relationship. This relationship is traditionally expressed as a proposition in terms of a precondition, a program, and a postcondition. This is what Floyd [6], Hoare

[16], Dijkstra [4], Gries [7], and others have tackled. It is preposterous to compare their methods of correctness proofs for programs with the HOS method of simply establishing well formed AXES specifications [19,20].

Another important aspect of proving programs correct is the proof of the correct implementation of an abstract data type. An abstract data type is specified axiomatically by a set of equations between expressions involving the primitive operations of the type. Implementing the data type means to express the primitive functions in terms of a set of functions which are already implemented and satisfy certain conditions in the form of equations between expressions involving the implementing functions. An implementation is correct if it maps the implemented data type onto the defined data type and expresses the primitive functions in terms of the implementing functions such that the equations for the defined functions become true because of the properties expressed by the equations for the implementing functions. This correctness concept was developed by Zilles [23] and Guttag [8]. Guttag and others have also developed practical methods and tools that assist in the semi-automatic proofs of abstract data type implementations.

Considering the depth and quality of research in the area of proving programs correct, the HOS literature on this subject [19,20,21] with all their unsubstantiated claims is embarrassingly naive and unscientific. In particular, J. Martin's book [20] may be useful as a tutorial for the HOS methodology but is of little value in the area suggested by the title. It is interesting to note that an earlier version of this book with practically identical content was published under the title: "Program Design which is Provably Correct".

CHAPTER 3

HOS IMPLEMENTED: USEIT

The HOS methodology is supported by a set of tools collectively called USEIT. This section first describes the major features of USEIT briefly. Following this is a discussion of some problem areas in USEIT and of the issue of productivity.

It should be noted that until June 1987 HOS was implementing a PC based USEIT system which has been announced as more general than the VAX based USEIT system (private communication). The author does not know in what stage of completion this PC based USEIT system was when HOS went out of business or if the criticism expressed in this paper applies to the new system.

3.1 DESCRIPTION OF USEIT

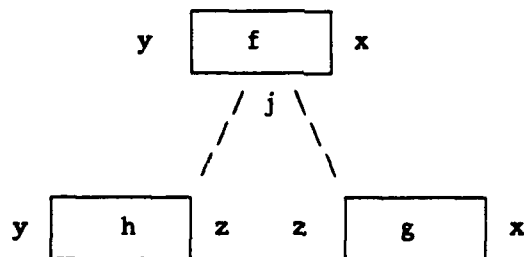
USEIT has three main components: The specification formalism AXES, the Analyzer, and the Resource Allocation Tool (RAT). In more familiar terms, these components correspond approximately to a programming language, the target language independent part of a compiler, and the code generator of a compiler.

3.1.1 AXES in USEIT

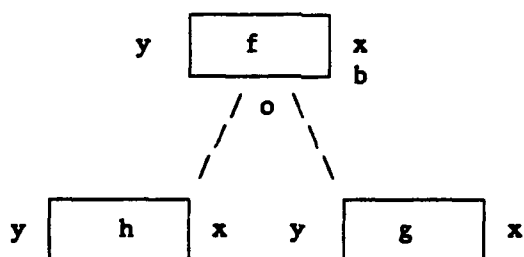
Functions are specified by control maps. A control map is a tree representation of the definition of a function in terms of other functions, ultimately, in terms of primitive functions. Each node represents a function. The root represents the function being defined, the leaf nodes represent primitive functions or functions defined by other control maps or recursive calls of nodes higher in the tree. USEIT also allows leaf nodes to reference external functions which are defined outside the USEIT formalism.

Each control structure is represented as a node together with its descendants and a symbol identifying the type of the control structure: "join" (j) for composition, "or" (o) for set partition, and "include" (i) for class partition. The following diagrams illustrate the primitive control structures:

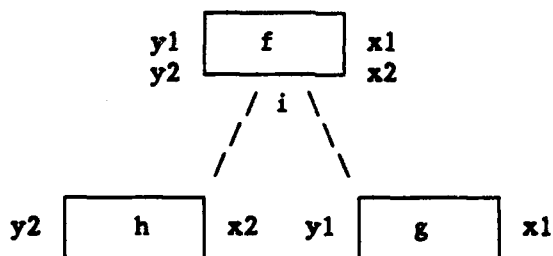
Composition: $y = f(x) = h(g(x))$



Set partition: $y = f(x) = \text{if } b \text{ then } g(x) \text{ else } h(x)$



Class partition: $(y1, y2) = f(x1, x2) = (g(x1), h(x2))$



More general and more convenient control structures are predefined and are directly available to the AXES user. Examples can be seen in the Appendices.

The following data types are predefined for every target language and can be used together with their primitive operations: boolean, natural, integer, rational, terminal. In FORTRAN USEIT, a variety of other types is predefined (see Reference Manual [19]). Also certain generic functions, such as equal and copy (called "clone"), are defined for every type and must be defined for every new type that is added to the system.

Under the FORTRAN USEIT system on the VAX, these control maps are constructed and modified using a graphics editor and are displayed, in the simplest case, on a VT240 work station. There is a variety of utility commands, for example for printing control maps and for extracting "documentation" of a

control map. The usual activity associated with constructing programs occurs in this environment. The difference is that the programs are functional and that they are represented as control maps rather than in textual form. Also, not all information relevant to a control map is visible on it, for example, the types of the variables used for input and output of the functions on all levels of the tree. This information is contained in a file associated with a control map. In the course of constructing, analyzing, "ratting", running, printing, and documenting a control map a large collection of files is generated and associated with the control map file.

3.1.2 Analyzer

The Analyzer performs the following target language independent function: Check if the control map is well formed according to the rules of AXES. This means, in particular:

- (1) Check that all functions are defined and all variables have a type.
- (2) Check for consistent usage of types in function interfaces (arguments and values).
- (3) Make these checks also across independently developed control maps and against libraries.

In the HOS terminology, the Analyzer guarantees completeness by (1) and (3) and consistency by (2) and (3). The concepts "completeness" and "consistency" are used in many HOS publications in a general sense without the definition of a restricted meaning, and the claim is made that the HOS methodology guarantees completeness and "logical consistency" of a specification. The puzzled reader, knowing that logical consistency of a set of statements cannot be proved by a program in general, wonders what exactly these claims mean. Little does he realize that it only means "All symbols are defined and used consistently according to their definition". Of course, this condition is normally guaranteed by a good compiler (and Linker) for any typed language.

3.1.3 Resource Allocation Tool (RAT)

Once a control map is analyzed, target code can be generated. This process is often referred to as "ratting".

For each target language for which USEIT is available, object code can be generated from the control maps. If the target language is FORTRAN, then the translation of a control map that has a number of defined leaf nodes results in the generation of a main program and a number of subroutines. Subroutines correspond to separately defined control maps. This design decision for FORTRAN USEIT has the consequence that the limitations of FORTRAN with respect to recursion and reentrancy become limitations of FORTRAN USEIT.

3.2 PROBLEM AREAS OF USEIT

The problem areas of USEIT fall into two different categories: (1) fundamental and inherent in the HOS approach; (2) arising from USEIT design decisions and from the poor quality of the implementation and documentation of USEIT.

3.2.1 Fundamental Problems

3.2.1.1 Higher Level Features

USEIT does not provide higher level features such as functions as arguments or values of functions. There is no abstraction mechanism in AXES that produces a function from an expression similar to the lambda operator in LISP. Thus, functions are static objects that can only be defined as "constant expressions" in terms of primitives but cannot be created or manipulated during the execution of a program.

3.2.1.2 Notation for Expressions

USEIT does not provide a notation for expressions such as the conventional infix notation. All expressions are functional expressions that must be represented via control maps. This lack of expressiveness in USEIT adds greatly to the apparent complexity of control maps.

3.2.1.3 Efficiency

As a functional language AXES shares the efficiency problems of functional languages caused by the lack of the destructive assignment. This makes these languages cleaner and more amenable to analysis but also less useful, at least currently, for certain problem domains such as linear algebra (Appendices A and B). As a simple example, a direct implementation of the addition of $M \times N$ matrices will generate $M \times N$ different $M \times N$ matrixes, each one being different from the previous one by only one element (Appendix B.1). In this case, a different recursive algorithm can be used that computes the elements of the sum matrix and gradually assembles the resulting sum matrix without any unnecessary copying of elements already computed (Appendix B.4). This algorithm uses a different representation of matrices where elements are not directly addressable. However, for most problems of linear algebra such algorithms have not been developed. The existing algorithms work efficiently by manipulating directly addressable elements of matrices represented as arrays in a language like FORTRAN but become hopelessly inefficient when transliterated into a functional language. While these problems are not specific to USEIT, one must realize that USEIT is, for this reason, not

suitable for many important problem domains. Currently, it is not suitable for the construction of a mathematical library as suggested by HOS (private communication).

3.2.2 USEIT Design and Implementation

Given the theoretical basis of AXES, does the implementation of the HOS approach in the form of USEIT measure up to the promised advantages inherent in this methodology?

3.2.2.1 Language Independency of the Analyzer

A major goal in the HOS approach is to make specifications target language independent. This goal was abandoned in the implementation. The FORTRAN USEIT Analyzer generates FORTRAN code and makes the user adhere to the restrictions imposed by FORTRAN as target language. Thus, general recursive functions cannot be specified and analyzed.

3.2.2.2 Control Map Representation

A control map is a bulky unmanageable two-dimensional representation of a function. The problem is that too little information is spread out too thin in two dimensions. Even very simple function definitions that take only a few lines in a mathematical notation normally extend over one page in each direction. To get the picture of a tree, a printed control map requires cut and paste, a time consuming manual activity. This is unacceptably slow and tedious for the specification of a complex system with frequent changes during the development and maintenance phase. For any program development system, it is absolutely mandatory that a readable printed representation of the program can be produced quickly and automatically without any manual work.

It is hard to envision a readable representation of an AXES specification essentially different from the USEIT control maps. Therefore, this point belongs perhaps into the fundamental category. It is described here since a better control of the shape of the tree could possibly alleviate the situation considerably.

USEIT does provide the capability to plot control maps. This, however, is an expensive operation with a longer turn-around time for a service that is required frequently and should be cheap and fast.

Not only are the USEIT control maps large in size and light in content; they are also, in general, over specified. Some of the information on them is irrelevant and therefore undesirable. Only the root, the leaves, and those internal nodes that are called recursively need to be named. USEIT requires every node to be named, a rather counterproductive requirement. Most internal

nodes do not represent functional units as envisioned by the user; they represent only information such as in which order subfunctions are to be executed and how arguments are to be passed around. There is no need to name them. If the user wishes to name a node, he should be able to do so. If he does not name a node, the system can generate its own name for internal reference purposes.

3.2.2.3 Recursion Restrictions

Recursion is the only way in USEIT to specify an unbounded number of execution steps, the actual number being determined at execution time depending on the input data. FORTRAN USEIT allows only a special form of recursion that can be compiled directly to iteration. This form is commonly called tail recursion. The conditions for it in the USEIT environment are stated incompletely in Appendix I of the Reference Manual. Following is a complete description of the conditions for tail recursion in FORTRAN USEIT.

First we give some general definitions for control map trees:

A path is a sequence of nodes $n_1, \dots, n_k$ where each node n_{i+1} is a descendant of n_i , $1 \leq i < k$. A path is leftmost if each n_{i+1} is a leftmost descendant of n_i , $1 \leq i < k$. A leftmost descendant of a node is one that appears leftmost in the control map except for "or" and "coor" nodes. In these cases all descendants are leftmost. Two paths overlap if they have at least one node in common. A path ending in a terminal node is called a terminal path.

If it was not for "or" and "coor" nodes, then there would be exactly one leftmost path starting at node n_1 and ending in a terminal node. Because of "or" and "coor" nodes, there may be many such paths.

A node n_1 defines a tail recursive function F if the following conditions are satisfied.

- (1) The node n_1 is labeled F .
- (2) There is at least one terminal path starting from n_1 ending in a node n_2 labeled F . Each such path is leftmost. Each such path is called a recursion path for F .
- (3) n_1 and n_2 have the same number of input variables.
- (4) n_2 must have at least one input variable different from the input variables at n_1 .
- (5) n_1, n_2 must have identical output variables.
- (6) Every recursion path for F must contain an "or" or "coor" node.

- (7) The function called by the end node of any terminal path that is not a recursion path for F must not invoke, directly or indirectly, the function F.
- (8) The recursion paths for F must not overlap with any recursion paths for another function F1.

Condition 2 defines tail recursion by requiring that all recursion paths be leftmost. Conditions 4 and 6 are necessary but not sufficient for termination. Condition 7 prevents indirect and mutual recursion of a set of functions. The last condition 8 defines a simple form of nested recursion where the nested function cannot invoke the embracing function.

The Analyzer of FORTRAN USEIT will not check all these conditions but requires the conditions to be satisfied for correct code generation. For example, it will not check the condition for proper nesting or the condition preventing mutual recursion. Thus, it is quite possible to get the blessing of the Analyzer for incorrectly specified control maps and have incorrect code generated. This happened at NSWC when a matrix addition function was specified using incorrect nesting. The generated code did not work, of course. Ironically, HOS acknowledged a "known bug" in their FORTRAN USEIT that sometimes generated incorrect code in nested loops.

The example of a recursive descent parser (Appendix D) uses mutually recursive functions which are not recognized as such by the Analyzer. Incorrect code is generated which, surprisingly, works for many cases because of the simple use of the recursive functions involved. The input $((A*A)*A)$ will force an execution path for which a wrong result will be printed even though the given expression is recognized correctly.

3.2.2.4 Automatic Documentation in USEIT

The automatic documentation facility of USEIT gives some of the information of the formal specification, as represented by the control maps and associated type files, in English with little control by the user. A box corresponding to $y = f(x)$ appears as "f processes x as input and produces y as output". This is advertised as satisfying Mil Standard 2167 [5].

The only reason for mentioning in the documentation of a function the input or output variables of this function is to explain their meaning and role in the function definition. This, of course, cannot be done automatically; it must be supplied by the programmer. But, without it, the words produced by the automatic documenter are useless.

Now, if a systematic documentation of a program is to be produced, then let the system generate a skeleton in symbolic form suppressing primitive functions and any internal nodes considered only cluttering up the documentation. Function definitions should be documented, not each use of them as USEIT does. The user needs control to direct the system accordingly. This

reviewer's opinion is that the string " $y = f(x)$ " is shorter and easier to read than the string "Function f processes data x as input producing y as output" and is, therefore, preferable in the skeleton. This argument becomes much more convincing when one considers the amount of English generated by the USEIT documenter for functions with more input or output parameters. It is instructive to see the voluminous output generated by the USEIT documenter for a simple function definition such as matrix addition (Appendix B.3). It is clear that by adding user text to the USEIT generated skeleton the resulting documentation would become even more voluminous and remain, for the most part, irrelevant.

Documentation is, unlike code generation, not an activity that can easily be automated, at least not with current technology. Good documentation describes the properties of functions and data structures more abstractly, that is, omitting details, than the program specification. It also should describe existing relationships that are assumed in the program but are not expressed explicitly. Therefore, contrary to popular belief, documenting is a creative intellectual activity. It is a challenge to write good documentation. Tools with good, convenient editing and graphics capabilities can help with bookkeeping to make the creative part of documenting easier and more productive. The USEIT documenter is a much too rigid tool to be useful for creating good documentation.

3.2.2.5 Documentation of the USEIT System

The criticism of the USEIT generated documentation applies, in part, to the documentation of the USEIT system itself.

The Reference Manual contains no less than 780 pages. It is a common Manual for USEIT for the three target languages: FORTRAN, PASCAL, and C. It contains much detailed operational information on how to create and modify control maps using Editor commands etc. Yet, there is no concise complete description of the AXES formalism. Syntax of variable names, function names, literals, the scope of names, all these things are left for the user to find out by using the system.

New users of the USEIT system are expected to take a five day training course offered by HOS. The course material consists of 370 viewgraphs [18]. Again, there is no definition of the AXES formalism. Instead, the HOS methodology is presented in general terms as the panacea that requires only the high level specification of a problem and makes all programming in the usual sense obsolete.

3.3 PRODUCTIVITY

Considering the problems described above, how does USEIT compare with a standard programming approach using FORTRAN with respect to productivity?

HOS presents USEIT as a methodology that increases productivity greatly. To demonstrate that, HOS compares the actual time, t_u , for constructing USEIT control maps with an "estimated time", t_e , for constructing equivalent FORTRAN algorithms. This estimated time is $N/10$ in man-days where N is the number of FORTRAN source lines generated by USEIT and 10 is assumed as the number of lines a FORTRAN programmer can code per day ([15], page 61).

If we apply this approach to the Matrix addition (ADD, Appendix B) then $t_e = 18.5$ man-days. In the FORTRAN code for ADD, text comment lines are counted, but blank comment lines and 33 lines for input output are not counted. The author submits that, given the problem statement

Matrix Addition: $C = A + B$ defined by
 $C(i,j) = A(i,j) + B(i,j)$ for $1 \leq i \leq M$, $1 \leq j \leq N$

an average FORTRAN programmer would be able to write something equivalent to the following code in one afternoon, or, more likely, in one hour:

```
* SUBROUTINE ADDM COMPUTES THE SUM OF TWO MxN REAL MATRICES.
* X AND Y ARE THE INPUT MATRICES, Z IS THE OUTPUT MATRIX
  SUBROUTINE ADDM(M,N,X,Y,Z)
    DIMENSION X(M,N), Y(M,N), Z(M,N)
    DO 200 J=1,N
      DO 100 I=1,M
        Z(I,J) = X(I,J) + Y(I,J)
100    CONTINUE
200    CONTINUE
      END
```

Assuming that it takes the FORTRAN programmer one full day, the discrepancy with the HOS estimate is a factor of 18.5. The reason is that, given a problem, no FORTRAN programmer will solve it by writing code as USEIT does. His code is normally much more compact and many times more efficient in execution than the code produced by USEIT. Of course, he will use the assignment statement and a state transition instead of a functional style of coding. This is the style FORTRAN was designed for. Thus, the HOS claims for increased productivity should be examined with great caution.

CHAPTER 4

CONCLUSION

This report is an evaluation and critique of the HOS methodology for system specification and program development.

The HOS methodology is based on abstract data types and on a functional approach to all phases of system development, including design, specification, programming, and project management. The underlying formalism AXES is a restricted functional language; restricted in the sense that high level features such as functions as arguments and values of functions are not allowed. The implementation of this approach is called USEIT. In USEIT, AXES specifications are represented graphically by control maps and are processed by tools for analysis, code generation, and documentation.

The HOS approach has been presented to the Computer Science community via a series of technical papers, mostly by Hamilton and Zeldin, with very little response. Consultants associated with HOS have reviewed and praised the HOS approach, but, apart from some very brief independent reviews [22], the professional world has not reacted to HOS. This report has been written to help fill that gap.

Can USEIT serve as the sole methodology for system development as suggested by HOS ([15], pp. 40-44)? Is USEIT a good practical tool for some part of the system building process?

Developing a system from conception to validation of its implementation is a complex process that can be viewed as a sequence of refinements starting from a general, highly abstract requirements description down to the most concrete description in form of an executable program. It includes documenting, testing, and proving consistency of the products on the various levels of abstraction.

So far, no formalism has been developed that is uniformly adequate for all phases of system development. AXES and the HOS methodology are no exception despite the claims by HOS. The author believes that it is unlikely that such a formalism will be found. The current trend in software engineering appears to aim at a variety of methods and tools, integrated in some sense, to support the variety of activities associated with system development. For example, for requirements and design specifications, state transition diagrams, data flow diagrams, and methods of formal language specification are very useful and powerful description tools. For practical reasons, such as productivity, flexibility, and reliability, the system developer wants to be able to use these tools. He does not want to restrict himself to a single formalism,

even if it would theoretically be possible to do so. It follows that USEIT cannot serve as the sole method for specifying requirements, design, code, and good documentation of a system.

In areas where USEIT is, in principle, sufficient, it has severe shortcomings. The control map representation of a function is uneconomical and hard to manage, the object code is large, and the execution speed can be unacceptably slow, especially for programs operating on arrays. FORTRAN USEIT on the VAX is particularly poor because the analyzer is FORTRAN dependent and incomplete in its checking across different control maps. This violates the HOS goals of a target language independent and "logically complete" analysis of specifications. Also, analysis of control maps and resource allocation are extremely slow operations compared to compiling FORTRAN source code.

The USEIT Reference Manual is voluminous and at the same time quite incomplete. Many things such as syntax of names and scope of names are left to the programmer to find out by trial and error. In general, the HOS literature tends to advertise their ideas and products more than making a contribution in substance to the field of Computer Science.

The author recommends that USEIT not be used in the TRIDENT program or any program development at NSWC. Even for a high level system specification, USEIT is not seen as a good choice. A mathematical functional notation or a PROLOG-like notation appears better suited for that purpose. The examples in the Appendices of this report, especially Appendices C, D, and E, show that a LISP-style mathematical notation is more compact and normally easier to read than the control map notation of USEIT.

On a more positive note, the author considers the functional approach to system development and programming very promising. Systems so conceived and programs so constructed are more amenable to analysis and therefore, in principle, more reliable and better manageable. The literature on functional programming and its relatives, equational and logic programming, is extensive. More work needs to be done in the direction of making these approaches practical and competitive with the traditional approach for program development using imperative languages ranging from FORTRAN to ADA. Specifically, efficient functional algorithms need to be developed in various problem domains, algorithms that are not, in essence, transliterated FORTRAN programs. Major progress in this area would be a significant step toward a solution of the software crisis. It was this crisis that spawned the HOS methodology and many others during the past 15 years. Methodologies abound but, so far, the crisis is still here.

CHAPTER 5

REFERENCES

- [1] Clocksin, W.F. and C.S. Mellish, Programming in PROLOG, Springer Verlag Berlin Heidelberg New York, 1981.
- [2] Cushing, S. "Algebraic Specification of Data Types in Higher Order Software (HOS)", Proceedings, Eleventh International Conference on Systems Sciences, Honolulu, Hawaii, 1978.
- [3] Cushing, S. "A Note on Arrows and Control Structures, Category Theory and HOS", In Axiomatic Analysis, TR-19, Higher Order Software, Inc. Cambridge Ma. 1978.
- [4] Dijkstra, E. W. A Discipline of Programming, Prentice-Hall 1976.
- [5] DOD-STD-2167, Military Standard, Defense System Software Development, 4 June 1985.
- [6] Floyd, R. W. "Assigning Meaning to Programs", Proc. AMS Symposium in Applied Mathematics, 19, pp 19 - 31, 1967.
- [7] Gries, D. The Science of Programming, Springer Verlag, New York 1981.
- [8] Guttag, J.V, E. Horowitz, and D.R. Musser, "Abstract Data Types and Software Validation", Com. ACM Vol. 21, No. 12, December 1978.
- [9] Hamilton, M. and S. Zeldin, "The Foundations of AXES, A Specification Language based on Completeness of Control", Doc. R-964, Charles Stark Draper Laboratory, Inc., Cambridge Ma. 1976
- [10] Hamilton, M. and S. Zeldin, "AXES Syntax Description", TR-4, Higher Order Software, Inc., Cambridge Ma. 1976.
- [11] Hamilton, M. and S. Zeldin, "Higher Order Software - A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976, pp 9-32.
- [12] Hamilton, M. and S. Zeldin, "Integrated Software Development System/ Higher Order Software Conceptual Description", Research and Development Technical Report ECOM-76-0329-F, U.S.Army Electronics Command, Fort Monmouth, N.J.

- [13] Hamilton, M. and S. Zeldin, "The Manager as an Abstract System Engineer", Proceedings, COMPCON 1977 Fall Conference, IEEE Computer Society, Washington D.C.
- [14] Hamilton, M. and S. Zeldin, "The Relationship between Design and Verification", The Journal of Systems and Software, Vol 1, No 1, 1979.
- [15] Hamilton, M. and S. Zeldin, "The Functional Life Cycle Model and its Automation: USE.IT", The Journal of Systems and Software 3, 25-68 (1983).
- [16] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming", Comm. ACM, 12(10), 1969.
- [17] HOS Inc., Axiomatic Analysis, TR-19, Higher Order Software, Inc. Cambridge Ma. 1978.
- [18] HOS Inc., The USE.IT System, Training Course.
- [19] HOS Inc., The USE.IT System, System Reference Manual, VAX - Release 3.0. High Order Software, Inc., Cambridge, Mass. 1985.
- [20] Martin, J. System Design from Provably Correct Constructs. Prentice-Hall, Inc., 1985.
- [21] Mimno, P. "Bug-Free Systems: A new Technology for Mathematically Provable Systems", Computerworld, C.W.Communications, Framingham Ma. Dec 1982.
- [22] Peters, L.J. and L.L. Tripp, "Comparing Software Design Methodologies", Datamation, Nov 1977, pp 89-94.
- [23] Zilles, S.N. "Abstract Specification for Data Types", IBM Res. Lab, San Jose, Calif., 1975.

APPENDIX A

ADDITION OF VECTORS

A.1 MATHEMATICAL FORMULATION

The vectors x , y , z are represented as one dimensional arrays and

$$z(i) = x(i) + y(i) \quad \text{for } 1 \leq i \leq N$$

The sum vector z is computed by a call of `addr`: $z = \text{addr}(0, N, x, y, z_0)$ where z_0 is the null vector $(0, \dots, 0)$. The recursive function `addr` is defined as follows:

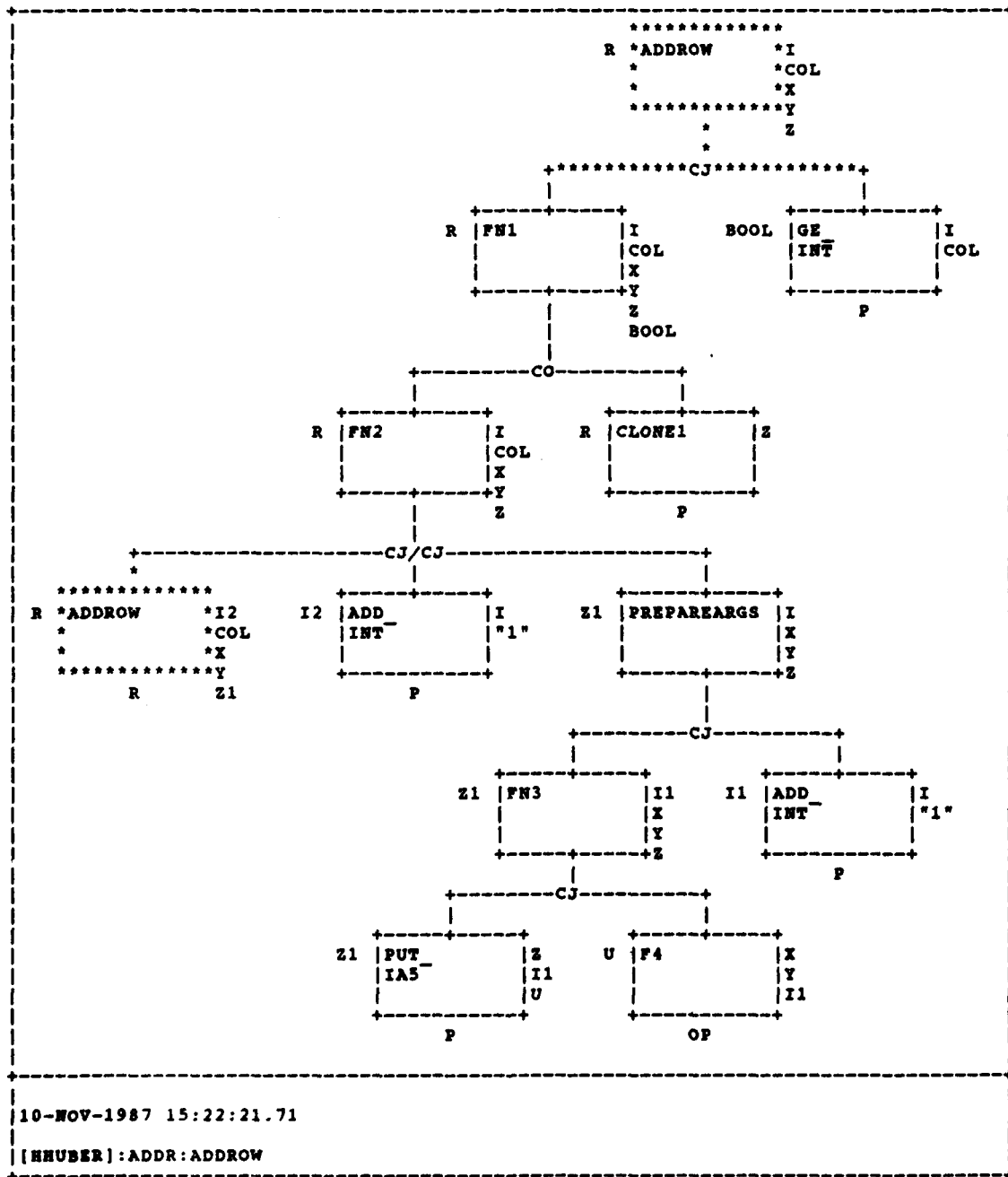
```
addr(i,col,x,y,z) = if i ≥ col then z
                    else addr(il,col,x,y,
                               newrow(z,il,val(x,il)+val(y,il)))
                    where il = i+1
```

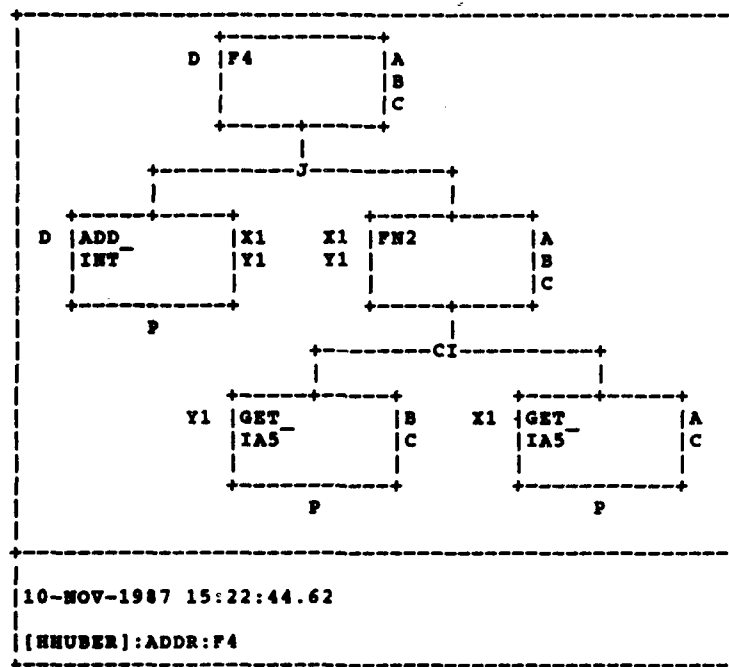
```
newrow(z,i,e) = new vector u,
                u(j) = e for j = i
                u(j) = z(j) otherwise
```

```
val(x,i) = x(i)
```

A.2 USEIT CONTROL MAPS FOR VECTOR ADDITION

The following control maps are the USEIT representation of `addr`.





APPENDIX B

ADDITION OF MATRICES

B.1 ARRAY BASED ALGORITHM

The Matrices x , y , z are represented as two dimensional arrays and

$$z(i,j) = x(i,j) + y(i,j) \quad \text{for } 1 \leq i \leq \text{row}, 1 \leq j \leq \text{col}$$

The sum matrix z is computed by a call of addm: $z = \text{addm}(0, \text{row}, 0, \text{col}, x, y, z0)$ where $z0$ is a row by col zero matrix. The recursive function addm is defined as follows:

```

addm(i,row,j,col,x,y,z) =
    if i > row then z
    else addm(il,row,j,col,x,y,addmr(il,row,j,col,x,y,z))
        where il = i+1

addmr(i,row,j,col,x,y,z) =
    if j > col then z
    else
        addmr(i,row,jl,col,x,y,newm(z,i,jl,val(x,i,jl)+val(y,i,jl)))
            where jl = j+1

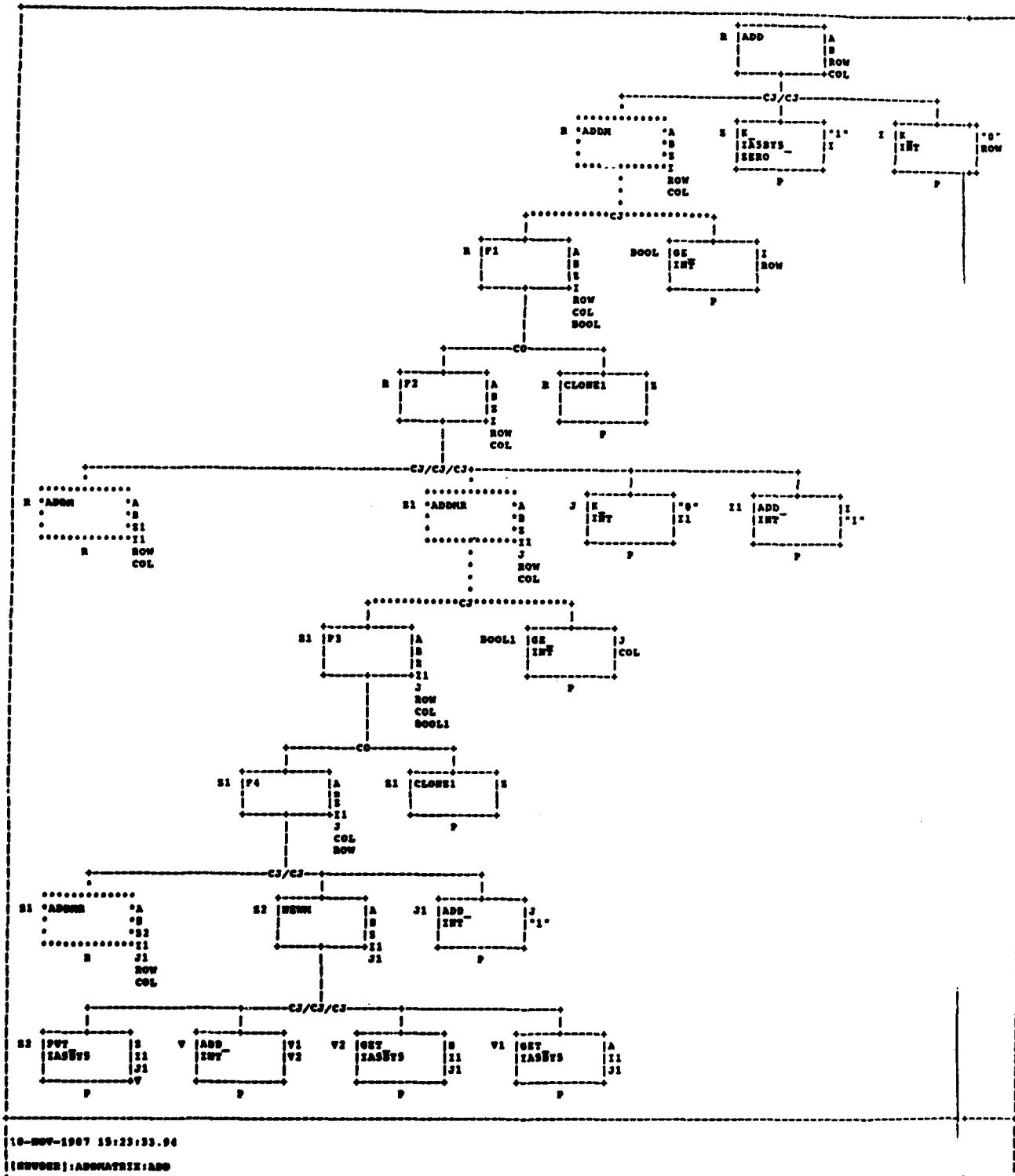
newm(z,i,j,e) = new matrix u,
                u(k,l) = e for k=i and l=j,
                u(k,l) = z(k,l) otherwise

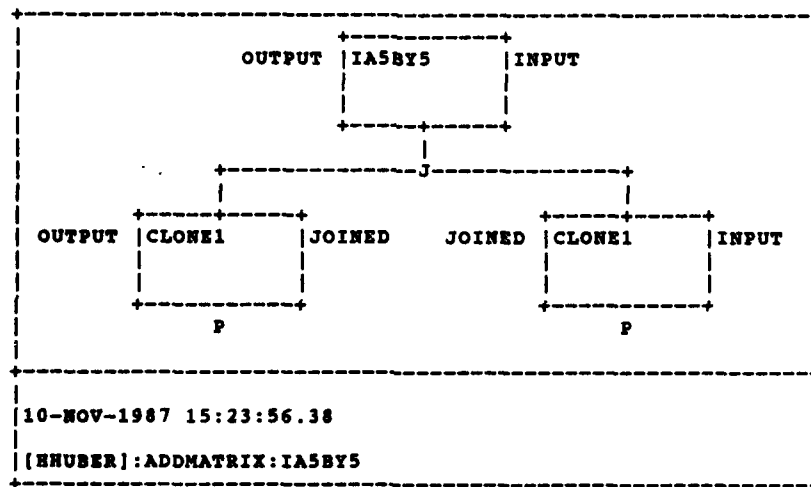
val(x,i,j) = x(i,j)

```

B.2 USEIT CONTROL MAPS FOR MATRIX ADDITION

The following control maps are the USEIT representation of addm.





B.3 USEIT GENERATED DOCUMENTATION

Following is the documentation of the control maps for matrix addition produced by the USEIT documenter.

CHAPTER 1 ADD

1.0 ADD

Function ADD processes data A,B,ROW,COL as input producing R as output. It has 3 children.

It is required that ROW,COL be of type INT.

It is required that A,B,R be of type IA5BY5.

1.1 K_INT

Function K_INT processes data "0",ROW as input producing I as output. Operation K_INT is a primitive operation.

1.2 K_IA5BY5_ZERO

Function K_IA5BY5_ZERO processes data "1",I as input producing Z as output. Operation K_IA5BY5_ZERO is a primitive operation.

1.3 ADDM

Function ADDM processes data A,B,Z,I,ROW,COL as input producing R as output. It has 2 children.

1.3.1 GE_INT

Function GE_INT processes data I,ROW as input producing BOOL as output. Operation GE_INT is a primitive operation.

1.3.2 F1

Function F1 processes data A,B,Z,I,ROW,COL,BOOL as input producing R as output. It has 2 children.

1.3.2.1 CLONE1

Function CLONE1 processes data Z as input producing R as output. Operation CLONE1 is a primitive operation.

1.3.2.2 F2

Function F2 processes data A,B,Z,I,ROW,COL as input producing R as output. It has 4 children.

1.3.2.2.1 ADD_INT

Function ADD_INT processes data I,"1" as input producing I1 as output. Operation ADD_INT is a primitive operation.

1.3.2.2.2 K_INT

Function K_INT processes data "0",I1 as input producing J as output. Operation K_INT is a primitive operation.

1.3.2.2.3 ADDMR

Function ADDMR processes data A,B,Z,I1,J,ROW,COL as input producing Z1 as output. It has 2 children.

1.3.2.2.3.1 GE_INT

Function GE_INT processes data J,COL as input producing BOOL1 as output.

Operation GE_INT is a primitive operation.

1.3.2.2.3.2 F3

Function F3 processes data A,B,Z,I1,J,ROW,COL,BOOL1 as input producing Z1 as output. It has 2 children.

1.3.2.2.3.2.1 CLONE1

Function CLONE1 processes data Z as input producing Z1 as output. Operation CLONE1 is a primitive operation.

1.3.2.2.3.2.2 F4

Function F4 processes data A,B,Z,I1,J,COL,ROW as input producing Z1 as output. It has 3 children.

1.3.2.2.3.2.2.1 ADD_INT

Function ADD_INT processes data J,"1" as input producing J1 as output. Operation ADD_INT is a primitive operation.

1.3.2.2.3.2.2.2 NEWM

Function NEWM processes data A,B,Z,I1,J1 as input producing Z2 as output. It has 4 children.

1.3.2.2.3.2.2.2.1 GET_IA5BY5

Function GET_IA5BY5 processes data A,I1,J1 as input producing V1 as output. Operation GET_IA5BY5 is a primitive operation.

1.3.2.2.3.2.2.2.2 GET_IA5BY5

Function GET_IA5BY5 processes data B,I1,J1 as input producing V2 as output. Operation GET_IA5BY5 is a primitive operation.

1.3.2.2.3.2.2.2.3 ADD_INT

Function ADD_INT processes data V1,V2 as input producing V as output. Operation ADD_INT is a primitive operation.

1.3.2.2.3.2.2.2.4 PUT_IA5BY5

Function PUT_IA5BY5 processes data Z,I1,J1,V as input producing Z2 as output. Operation PUT_IA5BY5 is a primitive operation.

1.3.2.2.3.2.2.3 ADDMR

Function ADDMR processes data A,B,Z2,I1,J1,ROW,COL as input producing Z1 as output. Function ADDMR is a recursive function. See above documentation.

1.3.2.2.4 ADDM

Function ADDM processes data A,B,Z1,I1,ROW,COL as input producing R as output. Function ADDM is a recursive function. See above documentation.

B.4 LIST BASED ALGORITHM

The matrices x, y are represented as lists of rows, each row being a list of elements. The algorithm assumes that there are the same number of rows in x and y and that all corresponding rows of x and y have equal length. The algorithm uses the LISP functions `car`, `cdr`, `cons` for the first of a list, the remainder of a list, and the construction of a new list. The empty list is denoted by `nil`. For convenience, the sum of `nil + nil` is defined to be `nil`.

The sum $x+y$ is computed as the value of `addmat(x,y)`. The recursive function `addmat` is defined as follows:

```
addmat(x,y) = if x = nil then nil
              else cons(addrow(car(x),car(y)),
                        addmat(cdr(x),cdr(y)))

addrow(u,v) = if u = nil then nil
              else cons(car(u)+car(v),addrow(cdr(u),cdr(v)))
```

One could transliterate the specifications for `addmat` and `addrow` into USEIT control maps. However, since FORTRAN does not support lists and the associated functions `car`, `cdr`, `cons`, these control maps would not be executable. In addition, both `addmat` and `addrow` are not tail recursive and FORTRAN USEIT would complain about that even if the user had defined a new abstract data type "list".

APPENDIX C

GREATEST COMMON DIVISOR OF A LIST OF INTEGERS

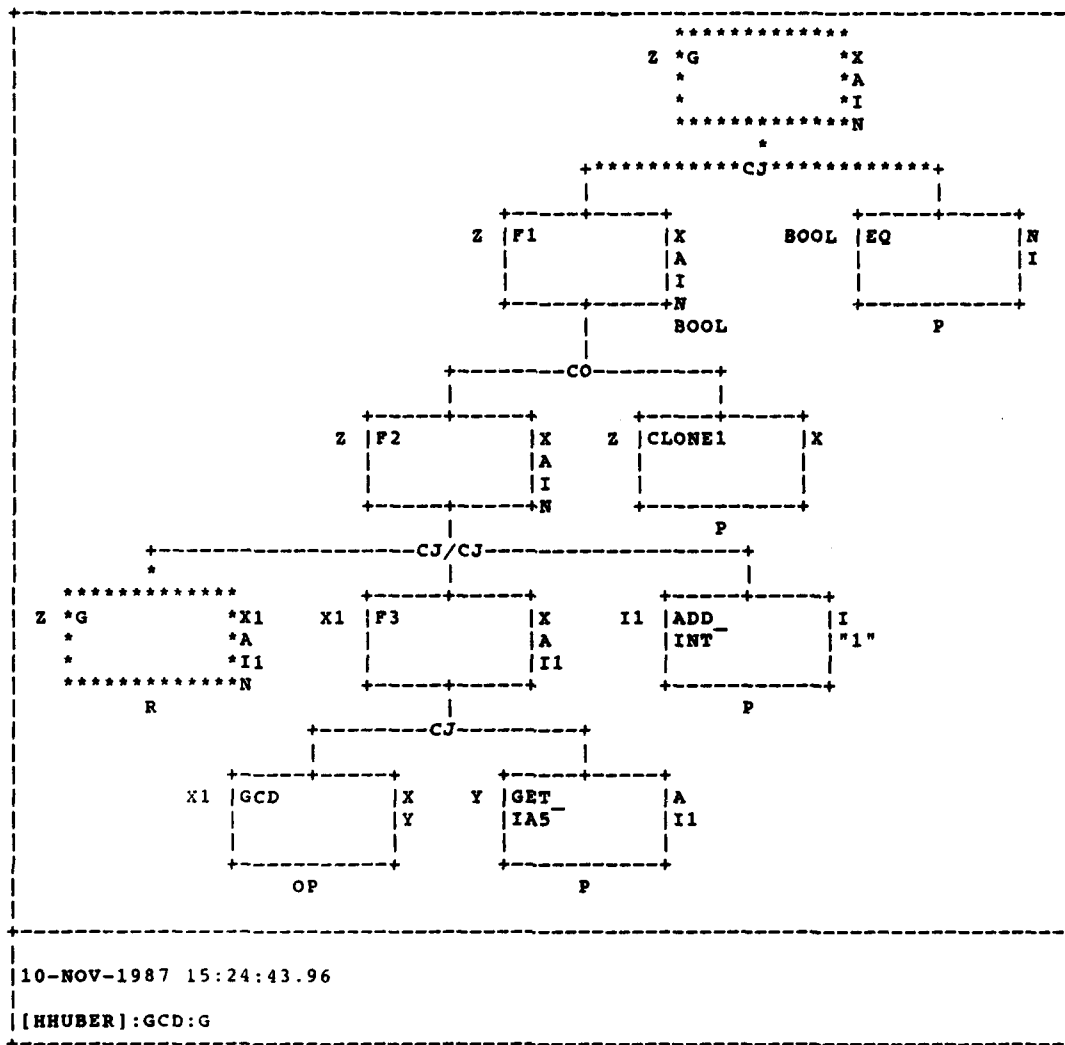
The list of integers is represented by a one dimensional array A of sufficient size and the length $n \geq 1$. The greatest common divisor of (A,n) is computed as follows:

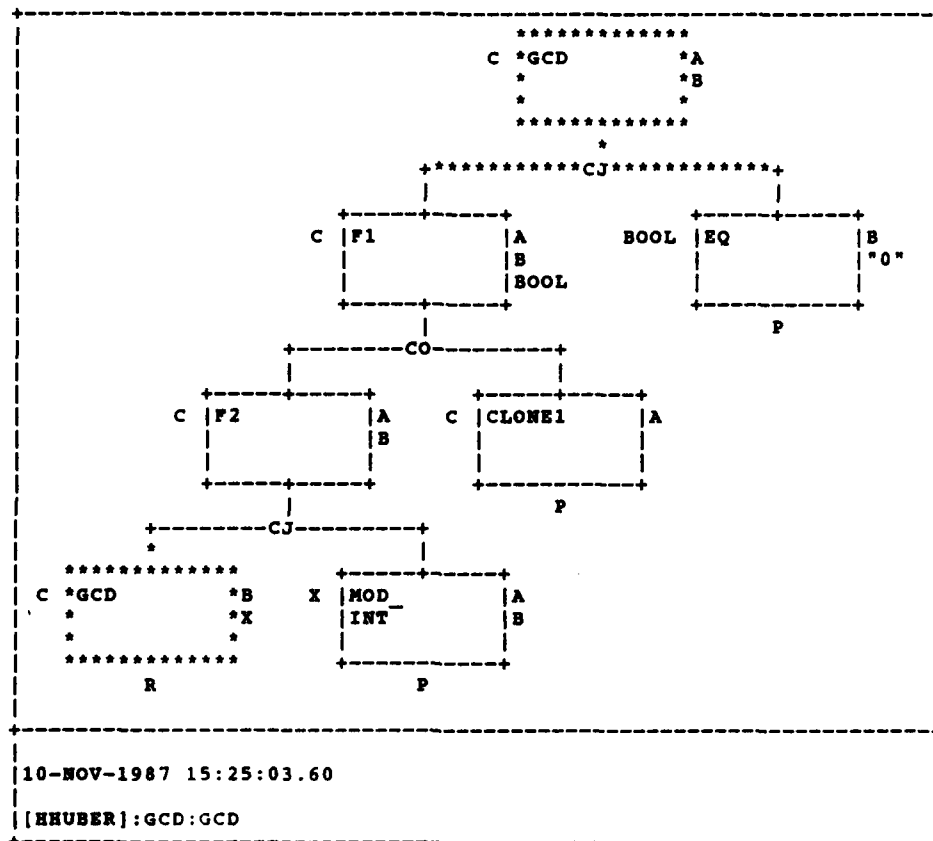
$$\text{gcdlist}(A,n) = g(A(1),A,2,n)$$

$$g(x,A,i,n) = \begin{cases} x & \text{if } i > n \\ g(\text{gcd}(x,A(i)),A,i+1,n) & \text{else} \end{cases}$$

$$\text{gcd}(x,y) = \begin{cases} x & \text{if } y = 0 \\ \text{gcd}(y,x \bmod y) & \text{else} \end{cases}$$

The following control maps are the USEIT representations of the functions g and gcd.





APPENDIX D

PARSING BY RECURSIVE DESCENT

D.1 MATHEMATICAL FORMULATION

Consider as an example the expressions built up from identifiers, parentheses, and the operators + and * as defined by the following grammar:

```

E -> E + T
E -> T
T -> T * F
T -> F
F -> ( E )
F -> id

```

The program will use the following equivalent grammar:

```

E -> T E'
E' -> + T E'
E' ->
T -> F T'
T' -> * F T'
T' ->
F -> ( E )
F -> id

```

An expression e is represented by the sequence of symbols it consists of, stored in a sufficiently large one dimensional array A : $e = A(1)A(2)\dots$. For each syntactic unit $\langle s \rangle$ a function fs : Naturals $\rightarrow$ Integers is defined that recognizes that unit. fs is defined as follows:

```

fs(i) = if A(i)A(i+1)... = <s>A(j)... then j
      else -i

```

The following function definitions can be directly derived from the above grammar:

```

fe(i)  = fep(ft(i))
fep(i) = if A(i) = "+" then fep(ft(i+1)) else i
ft(i)  = ftp(ff(i))

```

```

ftp(i) = if A(i) = "*" then ftp(ff(i+1)) else i

ff(i) = if A(i) = "id" then i+1
        else if A(i) = "(" then
            { let j = fe(i+1)
              if A(j) = ")" then j+1 else -j }

```

The function `fe` will consume as much input as possible, that means, it will stop only on an error. One can fairly easily rectify this situation by introducing a unique terminating symbol. We have omitted this because it would require more conditions in the above function definitions and would make their relationship with the grammar less transparent.

An expression stored in the array `A` is parsed by computing `fe(1)`. For example:

Input string stored in A	Value of <code>fe(1)</code>
<code>((A+A)*A)</code>	10
<code>A(A+A)</code>	2
<code>(A*A</code>	-5

The following control maps are the USEIT representations of the functions `fe`, `fep`, `ft`, `ftp`, and `ff`. The element "id" is represented as the character `A`. One function for printing out a trace was added. It prints, after the recognition of certain syntactic units `s`, the numbers `i,j` and an identifier for `s`, where `i` and `j-1` are the start and end positions of the substring representing `s`.

Compared to the 8 line compact functional notation, the USEIT control maps appear rather bulky and large. The accompanying text in the above description is brief but informative and explains the underlying ideas that are used but are not expressed directly in the function definitions. The documentation produced by the USEIT documenter (last section of this Appendix) is a monument of wasteful documentation void of any useful information. Even user added enhancements could not make a good document out of this monstrosity.

Even though the above functions are mutually recursive, the analyzer does not recognize this. It turns out that with the exception of the `print_line` function all other functions use only values of previous functions and work correctly. The printout, however, can be wrong due to the overwriting of an input variable during a recursive invocation of another function. For example, when parsing the expression `((A*A)*A)`, a portion of the trace will be

... 3 7 F, 8 9 F, 3 9 T, 3 9 E, 3 10 F, ...

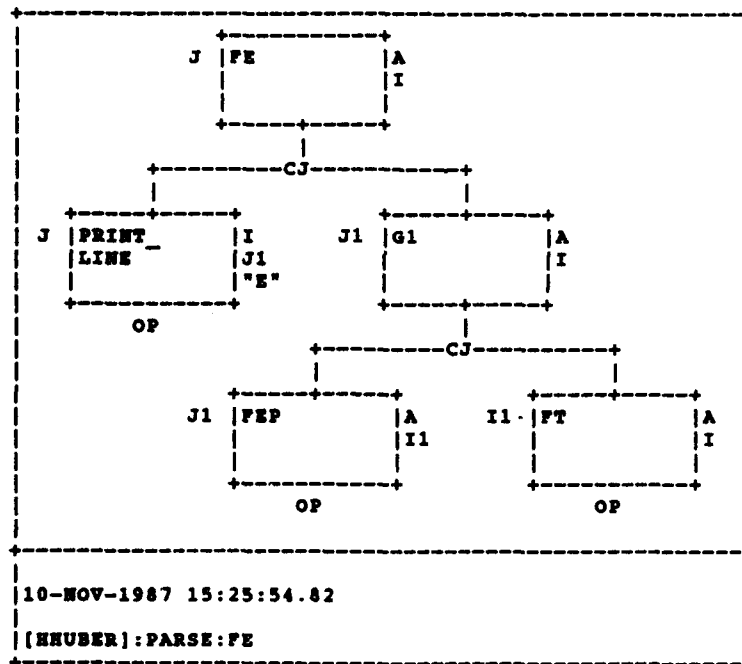
instead of the correct portion

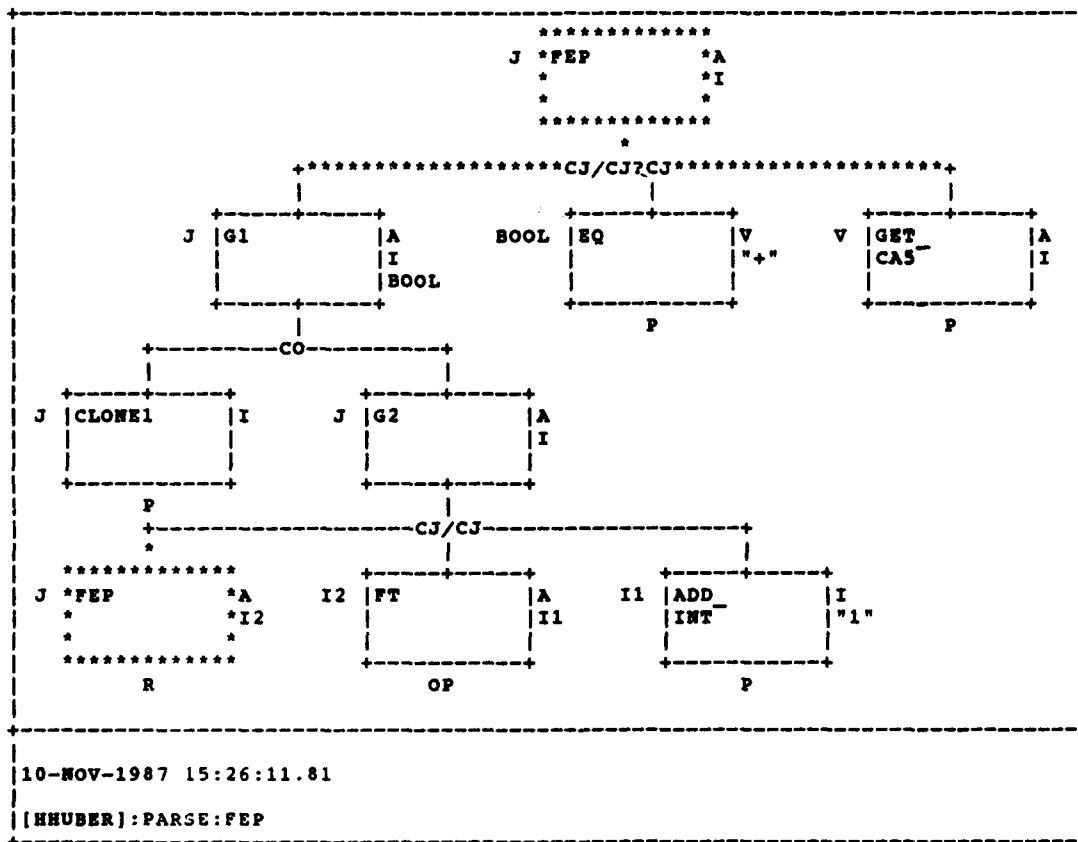
... 2 7 E, 8 9 F, 2 9 T, 2 9 E, 1 10 F, ...

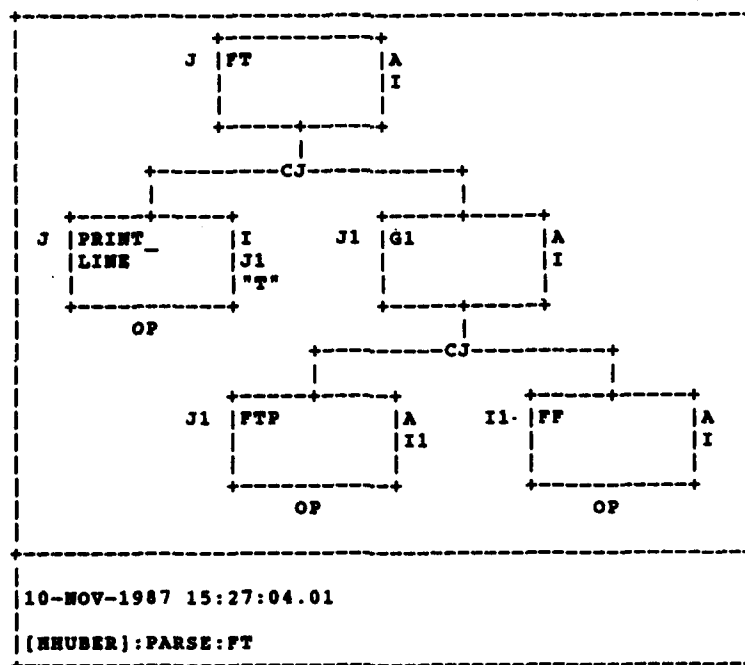
according to the correct parse that was in fact recognized by `fe`.

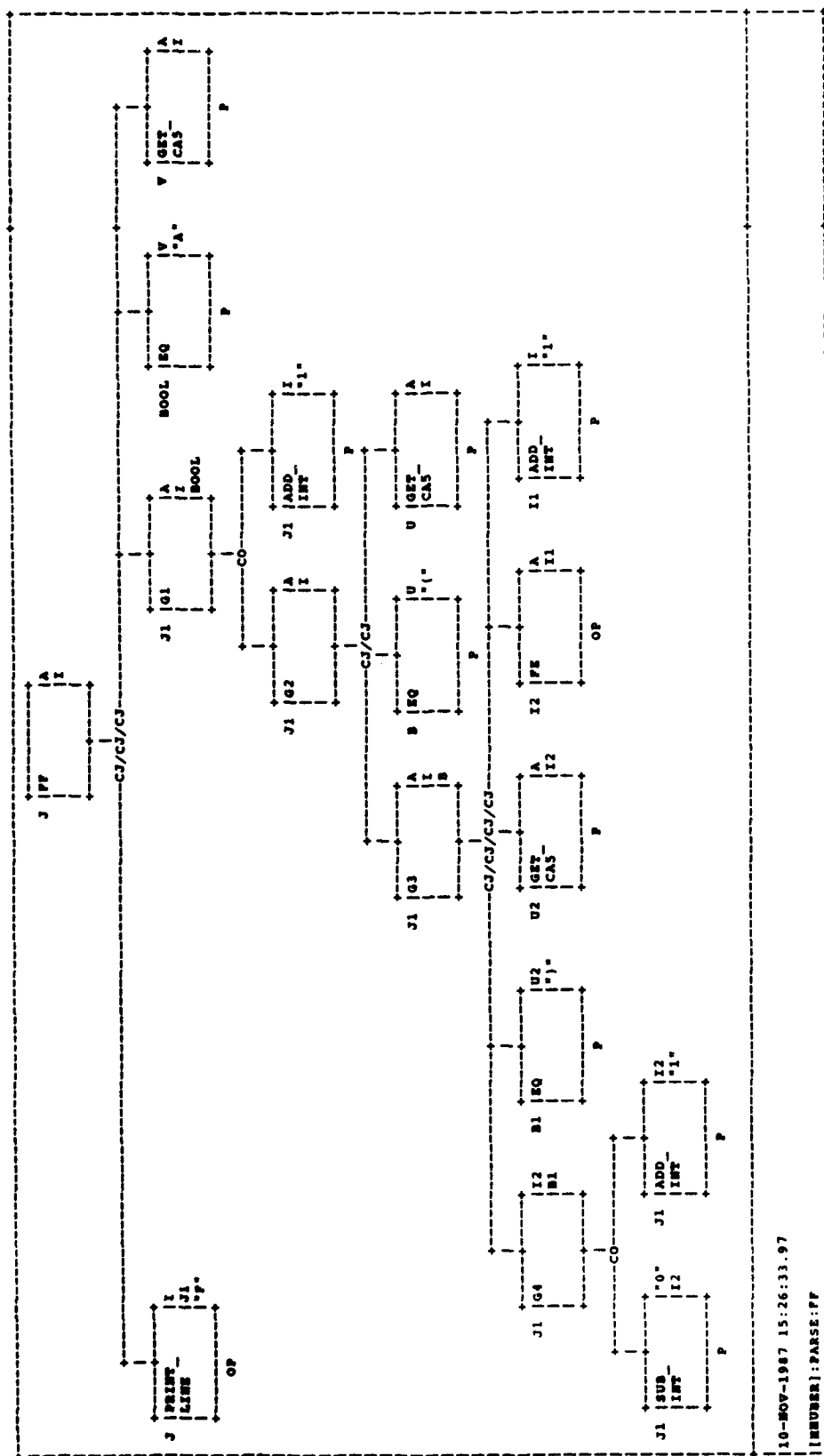
D.2 USEIT CONTROL MAPS FOR THE RECURSIVE DESCENT PARSER

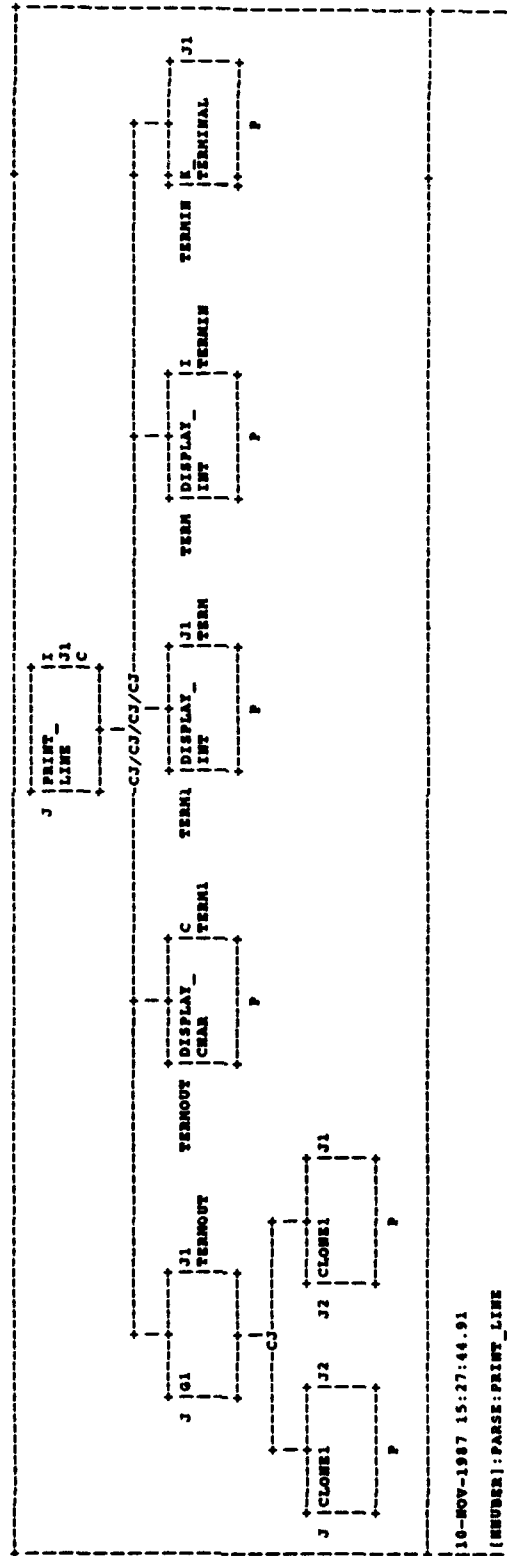
The following control maps are the USEIT representations of the functions fe, fep, ft, ftp, and ff. The additional function print_line prints a trace of the parse.











D.3 USEIT GENERATED DOCUMENTATION FOR THE RECURSIVE DESCENT PARSER

What follows is the documentation produced by the USEIT documenter for the recursive descent parser with the exception of fourteen pages of FORTRAN code.

TABLE OF CONTENTS

CHAPTER	OPERATION	PAGE
1	FE.....	1
2	FT.....	2
3	FEP.....	3
4	PRINT_LINE.....	4
5	FF.....	5
6	FTP.....	7
7	FE.....	8

CHAPTER 1 FE

1.0 FE

Function FE processes data A,I as input producing J as output. It has 2 children.

It is required that RMINEDTYPE>> be of type
CAS* <<(15)MODELCONTAINSVARIABLESOFUNDETERMINEDTYPE>>.

It is required that >> be of type
INT* <<(14)PARENTOUTPUTISINPUTTOOFFSPRING>>.

It is required that I1,J1 be of type INT.
It is required that "E" be of type CHAR.

1.1 G1

Function G1 processes data A,I as input producing J1 as output. It has 2 children.

1.1.1 FT

Function FT processes data A,I as input producing I1 as output. For information on operation FT see chapter 2.

1.1.2 FEP

Function FEP processes data A,I1 as input producing J1 as output. For information on operation FEP see chapter 3.

1.2 PRINT_LINE

Function PRINT_LINE processes data I,J1,"E" as input producing J as output. For information on operation PRINT_LINE see chapter 4.

CHAPTER 2 FT

2.0 FT

Function FT processes data A,I as input producing J as output. It has 2 children.

It is required that A be of type CA3.

It is required that I,J be of type INT.

It is required that I1 be of type INT.

2.1 G1

Function G1 processes data A,I as input producing J1 as output. It has 2 children.

2.1.1 FF

Function FF processes data A,I as input producing I1 as output. For information on operation FF see chapter 5.

2.1.2 FTP

Function FTP processes data A,I1 as input producing J1 as output. For information on operation FTP see chapter 6.

2.2 PRINT_LINE

Function PRINT_LINE processes data I,J1,"T" as input producing J as output. For information on operation PRINT_LINE see chapter 4.

CHAPTER 3 FEP

3.0 FEP

Function FEP processes data A,I as input producing J as output. It has 3 children.

It is required that A be of type CA5.

It is required that I,J be of type INT.

3.1 GET CA5

Function GET CA5 processes data A,I as input producing V as output. Operation GET_CA5 is a primitive operation.

3.2 EQ

Function EQ processes data V,"+" as input producing BOOL as output. Operation EQ is a primitive operation.

3.3 G1

Function G1 processes data A,I,BOOL as input producing J as output. It has 2 children.

3.3.1 G2

Function G2 processes data A,I as input producing J as output. It has 3 children.

3.3.1.1 ADD INT

Function ADD_INT processes data I,"1" as input producing I1 as output. Operation ADD_INT is a primitive operation.

3.3.1.2 FT

Function FT processes data A,I1 as input producing I2 as output. For information on operation FT see chapter 2.

3.3.1.3 FEP

Function FEP processes data A,I2 as input producing J as output. Function FEP is a recursive function. See above documentation.

3.3.2 CLONE1

Function CLONE1 processes data I as input producing J as output. Operation CLONE1 is a primitive operation.

CHAPTER 4 PRINT_LINE

4.0 PRINT_LINE

Function PRINT_LINE processes data I,J1,C as input producing J as output. It has 5 children.

4.1 K_TERMINAL

Function K_TERMINAL processes data J1 as input producing TERMIN as output. Operation K_TERMINAL is a primitive operation.

4.2 DISPLAY_INT

Function DISPLAY_INT processes data I,TERMIN as input producing TERM as output. Operation DISPLAY_INT is a primitive operation.

4.3 DISPLAY_INT

Function DISPLAY_INT processes data J1,TERM as input producing TERM1 as output. Operation DISPLAY_INT is a primitive operation.

4.4 DISPLAY_CHAR

Function DISPLAY_CHAR processes data C,TERM1 as input producing TERMOUT as output. Operation DISPLAY_CHAR is a primitive operation.

4.5 G1

Function G1 processes data J1,TERMOUT as input producing J as output. It has 2 children.

4.5.1 CLONE1

Function CLONE1 processes data J1 as input producing J2 as output. Operation CLONE1 is a primitive operation.

4.5.2 CLONE1

Function CLONE1 processes data J2 as input producing J as output. Operation CLONE1 is a primitive operation.

CHAPTER 5 FF

5.0 FF

Function FF processes data A,I as input producing J as output. It has 4 children.

It is required that A be of type CA5.

It is required that I,J be of type INT.

5.1 GET_CA5

Function GET_CA5 processes data A,I as input producing V as output. Operation GET_CA5 is a primitive operation.

5.2 EQ

Function EQ processes data V,"A" as input producing BOOL as output. Operation EQ is a primitive operation.

5.3 G1

Function G1 processes data A,I,BOOL as input producing J1 as output. It has 2 children.

5.3.1 ADD_INT

Function ADD_INT processes data I,"1" as input producing J1 as output. Operation ADD_INT is a primitive operation.

5.3.2 G2

Function G2 processes data A,I as input producing J1 as output. It has 3 children.

5.3.2.1 GET_CA5

Function GET_CA5 processes data A,I as input producing U as output. Operation GET_CA5 is a primitive operation.

5.3.2.2 EQ

Function EQ processes data U,"(" as input producing B as output. Operation EQ is a primitive operation.

5.3.2.3 G3

Function G3 processes data A,I,B as input producing J1 as output. It has 5 children.

5.3.2.3.1 ADD_INT

Function ADD_INT processes data I,"1" as input producing I1 as output. Operation ADD_INT is a primitive operation.

5.3.2.3.2 FE

Function FE processes data A,I1 as input producing I2 as output. For information on operation FE see chapter 7.

5.3.2.3.3 GET_CAS

Function GET_CAS processes data A,I2 as input producing U2 as output. Operation GET_CAS is a primitive operation.

5.3.2.3.4 EQ

Function EQ processes data U2," as input producing B1 as output. Operation EQ is a primitive operation.

5.3.2.3.5 G4

Function G4 processes data I2,B1 as input producing J1 as output. It has 2 children.

5.3.2.3.5.1 ADD_INT

Function ADD_INT processes data I2,"1" as input producing J1 as output. Operation ADD_INT is a primitive operation.

5.3.2.3.5.2 SUB_INT

Function SUB_INT processes data "0",I2 as input producing J1 as output. Operation SUB_INT is a primitive operation.

5.4 PRINT_LINE

Function PRINT_LINE processes data I,J1,"F" as input producing J as output. For information on operation PRINT_LINE see chapter 4.

CHAPTER 6 FTP

6.0 FTP

Function FTP processes data A,I as input producing J as output. It has 3 children.

6.1 GET_CA5

Function GET_CA5 processes data A,I as input producing V as output. Operation GET_CA5 is a primitive operation.

6.2 EQ

Function EQ processes data V,"*" as input producing BOOL as output. Operation EQ is a primitive operation.

6.3 G1

Function G1 processes data A,I,BOOL as input producing J as output. It has 2 children.

6.3.1 G2

Function G2 processes data A,I as input producing J as output. It has 3 children.

6.3.1.1 ADD_INT

Function ADD_INT processes data I,"1" as input producing I1 as output. Operation ADD_INT is a primitive operation.

6.3.1.2 FF

Function FF processes data A,I1 as input producing I2 as output. For information on operation FF see chapter 5.

6.3.1.3 FTP

Function FTP processes data A,I2 as input producing J as output. Function FTP is a recursive function. See above documentation.

6.3.2 CLONE1

Function CLONE1 processes data I as input producing J as output. Operation CLONE1 is a primitive operation.

CHAPTER 7 FE

7.0 FE

Function FE processes data A,I as input producing J as output. It has 2 children.

It is required that RMINEDTYPE>> be of type
CA5*<<((15)MODELCONTAINSVARIABLESOFUNDETERMINEDTYPE>>.

It is required that >> be of type
INT*<<((14)PARENTOUTPUTISINPUTTOOFFSPRING>>.

It is required that I1,J1 be of type INT.
It is required that "E" be of type CHAR.

7.1 G1

Function G1 processes data A,I as input producing J1 as output. It has 2 children.

7.1.1 FT

Function FT processes data A,I as input producing I1 as output. For information on operation FT see chapter 2.

7.1.2 FEP

Function FEP processes data A,I1 as input producing J1 as output. For information on operation FEP see chapter 3.

7.2 PRINT LINE

Function PRINT_LINE processes data I,J1,"E" as input producing J as output. For information on operation PRINT_LINE see chapter 4.

HIERARCHY OF MODULES

OPERATIONS CALLED BY FE

```

FT
  I1=FT(A,I)[OP]

FEP
  J1=FEP(A,I1)[OP]

PRINT_LINE
  J=PRINT_LINE(I,J1,"E")[OP]
  
```

OPERATIONS CALLED BY FT

```

FF
  I1=FF(A,I)[OP]

FTP
  J1=FTP(A,I1)[OP]

PRINT_LINE
  J=PRINT_LINE(I,J1,"T")[OP]
  
```

OPERATIONS CALLED BY FEP

```

FT
  I2=FT(A,I1)[OP]
  
```

OPERATIONS CALLED BY PRINT_LINE

NONE

OPERATIONS CALLED BY FF

```

FE
  I2=FE(A,I1)[OP]

PRINT_LINE
  J=PRINT_LINE(I,J1,"F")[OP]
  
```

OPERATIONS CALLED BY FTP

```

FF
  I2=FF(A,I1)[OP]
  
```

OPERATIONS CALLED BY FE

FT

I1=FT(A,I)[OP]

FEP

J1=FEP(A,I1)[OP]

PRINT_LINE

J=PRINT_LINE(I,J1,"E")[OP]

APPENDIX E

THE TOWERS OF HANOI GAME

E.1 MATHEMATICAL FORMULATION

The towers of Hanoi game can be viewed as a function of four arguments: `hanoi(n,a,b,c)` returns as its value the sequence of steps ($x_1 \rightarrow y_1, \dots, x_k \rightarrow y_k$) which moves all n discs from pole "a" to pole "b" using pole "c" as a spare. The x_i, y_i are pole names. At no time can a larger disc be placed on top of a smaller disc.

A standard solution [1] expresses `hanoi(n,a,b,c)` as a single step if $n=1$ or as a sequence of the solutions of three simpler games of the same kind if $n>1$:

```
hanoi(n-1,a,c,b)
hanoi(1,a,b,c)
hanoi(n-1,c,b,a)
```

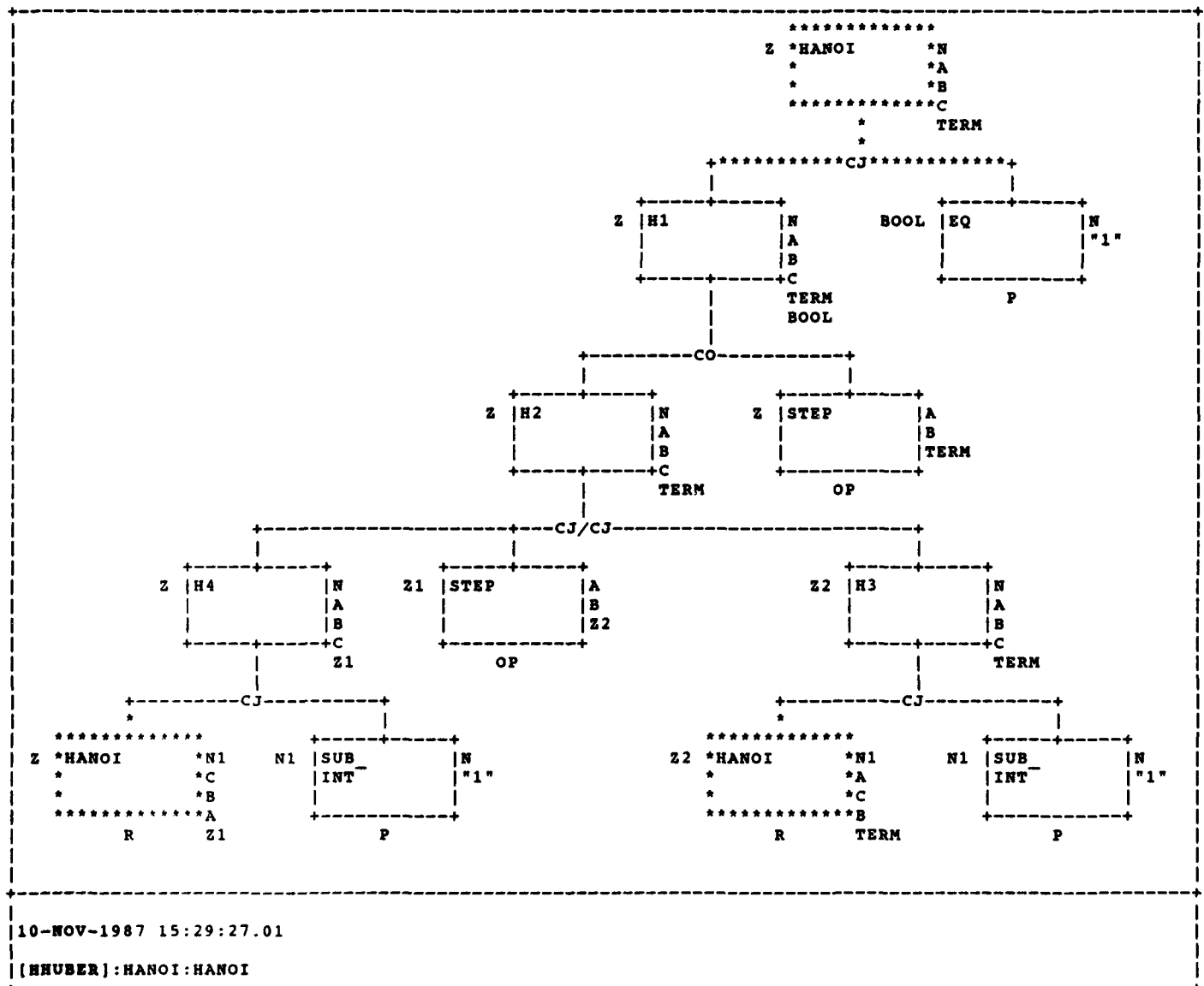
This leads to the following definition of `hanoi`:

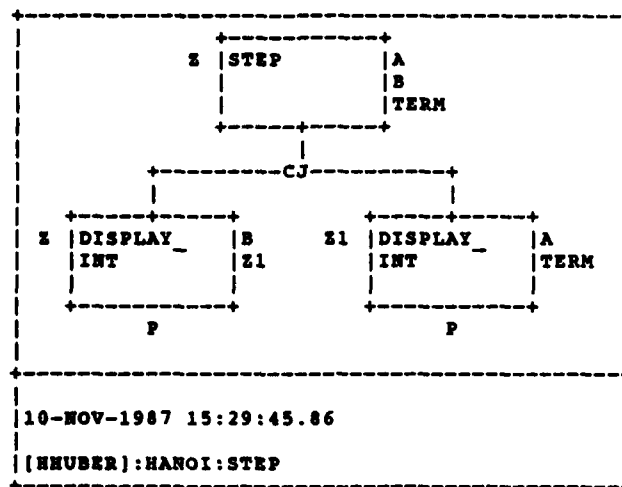
```
hanoi(n,a,b,c) = if n=1 then list(a->b)
                  else append(
                        append(hanoi(n-1,a,c,b),
                              list(a->b)),
                        hanoi(n-1,c,b,a))
```

For simplicity, the USEIT representation of `hanoi` does not compute the sequence of moves as a list but prints each move via a call to `step`. Two versions of USEIT control maps are presented. For the first version USEIT recognizes the non iterative nature of `hanoi` and refuses to "rat". For the second version USEIT does not recognize the recursive nature of `hanoi` and generates code for it which works only for $n=1$ and $n=2$ but goes into an infinite loop for $n>2$.

E.2 USEIT CONTROL MAPS FOR HANOI, VERSION 1

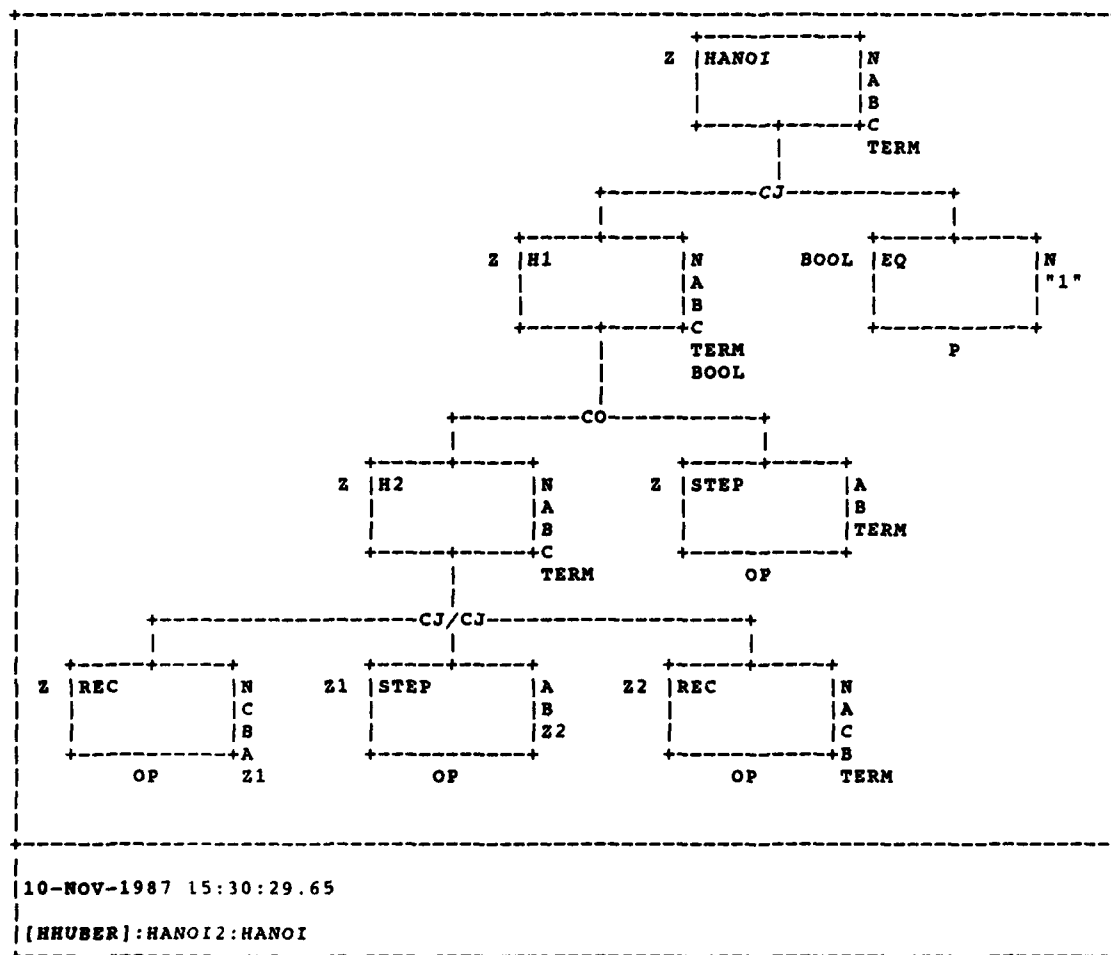
The following control maps are a USEIT representation of hanoi. In this version the control map of hanoi by itself appears as recursive.

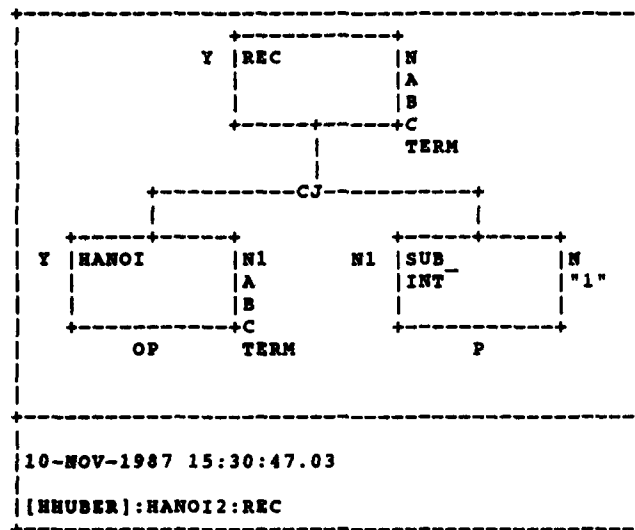


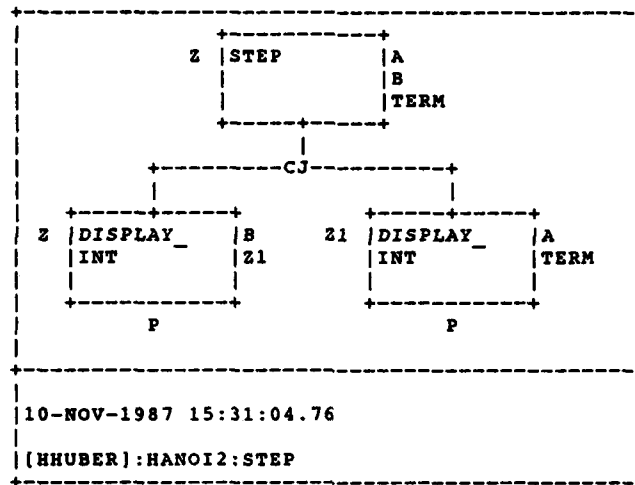


E.3 USEIT CONTROL MAPS FOR HANOI, VERSION 2

The following control maps are a USEIT representation of hanoi. In this version the control map of hanoi by itself appears as non-recursive.







DISTRIBUTION

	Copies
Strategic Systems Project Office	
Attn: SP-23115 (C. Chappell)	1
SP-23423 (H. Cook)	1
Department of the Navy	
Washington, DC 20376-5002	
EG&G Washington Analytical	
Services Center, Inc.	
Attn. IMC	2
P.O.Box 552	
Dahlgren VA 22448-0552	
Internal Distribution:	
E211	1
E231	10
K50	1
K51	2
K52	25
K53	2
K54	4
K301	1