DOCUMENT RESUME

ED 315 055 IR 014 169

AUTHOR Scandura, Alice B.
TITLE The INTELLIGENT RuleTutor: A Structured Approach to Intelligent Tutoring. Final Report.
INSTITUTION Intelligent Micro Systems, Inc., Narberth, PA.
SPONS AGENCY Department of Education, Washington, DC.
PUB DATE 1 Feb 89
CONTRACT 400-86-0060
NOTE 85p.
PUB TYPE Information Analyses (070) -- Reports - Descriptive (141)

EDRS PRICE MF01/PC04 Plus Postage.
DESCRIPTORS *Artificial Intelligence; Cognitive Style; *Computer Assisted Instruction; Computer Managed Instruction; Computer Simulation; *Diagnostic Tests; Instructional Systems; *Programed Tutoring
IDENTIFIERS *Generative Computer Assisted Instruction; *Intelligent CAI Systems; Structural Learning

ABSTRACT

This final report describes a general purpose system for developing intelligent tutors based on the Structural Learning Theory. The report opens with a discussion of the rules and related constructs that underlie cognitive constructs in all structural learning theories. The remainder of the text provides: (1) an introduction to the Structural Learning Theory as it relates to simple intelligent computer based instruction (ICBI) systems and authoring; (2) a description and analysis of the MicroTutor II arithmetic tutor; (3) an overview of the Structural Learning Theory and the kinds of intelligent tutor systems that have been developed based on this theory; (4) a discussion of the RuleTutor prototype itself; (5) an explanation of the PRODOC Computer software development system, which is designed to represent content in the form needed for use by any planned modular ICBI system; (6) sample arithmetic rules (constructed using PRODOC) to be used in conjunction with the intelligent RuleTutor; and (7) a review of related research and a summary of major points. A number of Scandura FLOWforms (an extended form of a flow chart) are interspersed throughout the text, and four additional FLOWforms illustrating simulations for addition, subtraction, multiplication, and division are appended. (72 references) (SD)

The INTELLIGENT RuleTutor: A Structured Approach to Intelligent Tutoring*

Alice B. Scandura, Ph. D., Principal Investigator
Intelligent Micro Systems, Inc.
1249 Greentree Lane
Narberth, PA 19072

Contract 400-86-0060
Final Report
February 1, 1989

* This project was funded in part with Federal Funds from the Department of Education under contract numbers 400-85-1020 and 400-86-0060. The contents of this publication do not necessarily reflect the views or policies of the Department nor does mention of trade names, commercial products or organizations imply endorsement by the U.S. Government.

The INTELLIGENT RuleTutor: A Structured Approach to Intelligent Tutoring*

Alice B. Scandura, Ph. D., Principal Investigator
Intelligent Micro Systems, Inc.
1249 Greentree Lane
Narberth, PA 19072

Contract 400-86-0060
Final Report
February 1, 1989

* This project was funded in part with Federal Funds from the Department of Education under contract numbers 400-85-1020 and 400-86-0060. The contents of this publication do not necessarily reflect the views or policies of the Department nor does mention of trade names, commercial products or organizations imply endorsement by the U.S. Government.

The INTELLIGENT RuleTutor:
A Structured Approach to Intelligent Tutoring

ABSTRACT

This report describes a general purpose system for developing intelligent tutors based on the Structural Learning Theory. This system has two distinct but complementary parts: The first part is a general purpose intelligent tutor which is able to perform both diagnostic testing and instruction -- but which does not contain content specific knowledge, either of the problem or tasks to be generated or the cognitive procedures (rules) to be taught. The second part consists of IMS's PRODOC software development system. PRODOC provides an easy-to-use medium for specifying the content to be taught (in terms of rules). Each such rule, in turn, is interpretable by a general purpose tutor, resulting in an operational intelligent tutoring system.

More specifically, we have described a general-purpose Intelligent RuleTutor which can be used in conjunction with ANY cognitive procedural task formulated as a single rule (i.e. as defined in the Structural Learning Theory - e.g., Scandura 1970,1977,1984). The component or atomic rules in PRODOC's rule library might reasonably accommodate essentially any content area. Specifically, the atomic rules in this library have been shown to provide a natural basis for formulating arbitrary rules (corresponding to to-be-learned cognitive procedures) in arithmetic.

THE INTELLIGENT RULETUTOR PHASE II:
A Structured Approach to Intelligent Tutoring

Joseph M. Scandura and Alice B. Scandura
University of Pennsylvania Intelligent Micro Systems, Inc.

INTRODUCTION

Most contemporary computer-based instruction (CBI) authoring systems are of the "fixed content" variety; that is, they require users to input explicitly the instruction and questions to be presented as well as possible answers and feedback that might be given. In addition, the CBI author must specify for each intended application the exact conditions governing the selection and sequencing of information used in diagnosis and instruction.

Unlike instructional systems created with "fixed content" authoring systems, generative authoring systems create instructional systems in which content is generated dynamically as testing and/or instruction proceeds. Generative authoring systems which are intelligent also determine automatically what test items and/or instructions are to be given and when (they are to be given).

At the present time the latter problem is being attacked from two different perspectives. Perhaps the predominant one is based largely on programming techniques associated with "artificial intelligence". The other is more directly associated with cognitive instructional systems.

The former approach is typically characterized by use of the programming languages LISP and, to a growing but lessor extent, Prolog. These languages are especially good for rapid prototyping. More uniquely, their very nature lends them to logical deduction and open ended programming tasks -- that is, tasks where it is infeasible for the programmer to fully anticipate all possibilities during program construction. Unanticipated possibilities may be inferred from relatively small sets of basic assumptions. In this context "giving reasons" for an assertion is equivalent to being able to derive the assertion (from mutually agreed assumptions). Consequently, some working in this tradition actually equate the word "intelligent" with the ability to give reasons.

Educational applications have tended to parallel these characteristics. So called "microworlds", for example, generally provide an open ended environment within which the learner may explore the possibilities inherent in some domain of knowledge. The

programming language Logo (which is based on LISP) is a well known example. Microworlds may be viewed as generative systems in which the learner has full control over the goals to be achieved and how to achieve them. Intelligent tutoring systems in this tradition provide "advice" but tend to stress the idiosyncratic. In large part this is because "learning" within the AI tradition was often equated with fixing "bugs" (rather than the acquisition of new knowledge). More generally, it is because languages like LISP presuppose a certain ordering on the world, one which has little directly to do with cognition or instruction.

The goals these investigators have set for themselves have a certain attractiveness. One can hardly question the desirability of generating problems and solutions dynamically as needed, allowing learners to investigate subjects from alternative perspectives, dealing with individual idiosyncrasies and reasoning logically on the basis of available knowledge. Nonetheless, judging from the paucity of concrete results after so many years of generous funding, one can seriously question whether traditional AI provides the best or even a good way of producing practical (much less commercially viable) products.

There are two basic issues here. One has to do with the ICBI systems themselves (or "intelligent tutors" as they are often called); the other has to do with development strategy. Granting that ICBI systems ideally should include (but not be limited to) the above characteristics, we believe there are more efficient means of achieving these goals. Clearly, just developing large numbers and varieties of ICBI systems will not do it. Questions pertaining to quality aside, cost alone makes development prohibitive without generous federal support. While experience can reduce such costs to a degree, order of magnitude improvements are needed if we are to produce (and properly maintain) the needed systems. Equally important, it is essential that computer-literate content and pedagogical experts be able to participate directly in such development.

In this report we describe a microcomputer-based RuleTutor authoring system which will allow instructional designers and content experts who are not programmers to create ICBI systems in their areas of expertise. Toward this end a highly structured, cumulative approach to ICBI development is described. Central to this approach is a sharp conceptual distinction between content and the tutorial aspects of ICBI systems. Making such a distinction is increasingly recognized as crucial in making ICBI development more efficient.

To date, however, no one has succeeded in developing such an ICBI system, much less an easy-to-use authoring system for developing such systems. Some have publicly expressed the opinion that it cannot be done. Why do we feel confident that it CAN be done? As it turns out there are conceptual, pragmatic, technological and methodological reasons: (a) a well researched theory (Structural Learning Theory) in which such distinctions are central, (b) the commercial availability of an ICBI type system (the MicroTutor II intelligent arithmetic tutor) which approaches (but does not

fully achieve) complete modularity, (c) a carefully phased and cummulative approach to system development (as opposed to the more idiosyncratic, less cognitively and/or instructionally based approaches characteristic of AI-based development) and (d) the current availability of a software development system, called PRODOC, which represents content in precisely the form needed for use by any of the planned modular ICBI tutorial systems.

The type of system Intelligent Micro Systems, Inc. (IMS) has developed can be represented schematically as shown in the figure below. ICBI systems are depicted as having two parts: an intelligent RuleTutor system and a set of rules representing the content to be taught. Although not represented explicitly, one can envision intelligent RuleTutor systems ordered according to complexity of the content (e.g, the numbers and types of rules) they can handle. In this report, we describe an intelligent RuleTutor prototype which is designed to provide optimal diagnosis and remediation with respect to cognitive procedural tasks (i.e., single rules). Incidentally, we have shown how this RuleTutor might be extended at some future time to accommodate any type of content.

The solid line indicates that PRODOC was completed before the project was started. Indeed, PRODOC was used in the development of the intelligent RuleTutor prototype (dashed line). More important, PRODOC, in turn, can be used by subject matter and pedagogical experts (e.g., instructional designers) to represent the rules to be learned. In contrast to traditional AI-based approaches to ICBI development, the IMS approach is highly structured and based on the Structural Learning Theory. As noted by Scandura (e.g., 1971, 1973), it provides a systematically structured approach to the unbounded and/or unanticipated.

Given its centrality in both PRODOC and the planned intelligent RuleTutors, we begin our report with discussion of the rule and related constructs. After this come the following sections: (a) Introduction to the Structural Learning Theory (as it pertains to simple ICBI systems and authoring); (b) The MicroTutor II arithmetic tutor, a description and analysis; (c) An overview of the Structural Learning Theory and the kinds of intelligent tutor systems that might be developed based on such theory; (d) A description of the RuleTutor prototype itself; (e) PRODOC, with emphasis on those aspects to be used in ICBI authoring to create rules for use by the intelligent RuleTutor; (f) Arithmetic rules (constructed using PRODOC) to be used in conjunction with the intelligent RuleTutor. Following the above is a section which deals with relationships with other research, followed by a summary of major points.

RULES AND RELATED CONSTRUCTS

The problem and rule constructs serve as the key underlying cognitive constructs in all structural learning theories (Scandura, 1977, 1981a). However, it became increasingly apparent that the precise syntax chosen to represent rules and associated constructs would have a direct effect on the generality and efficiency of any ICBI systems based

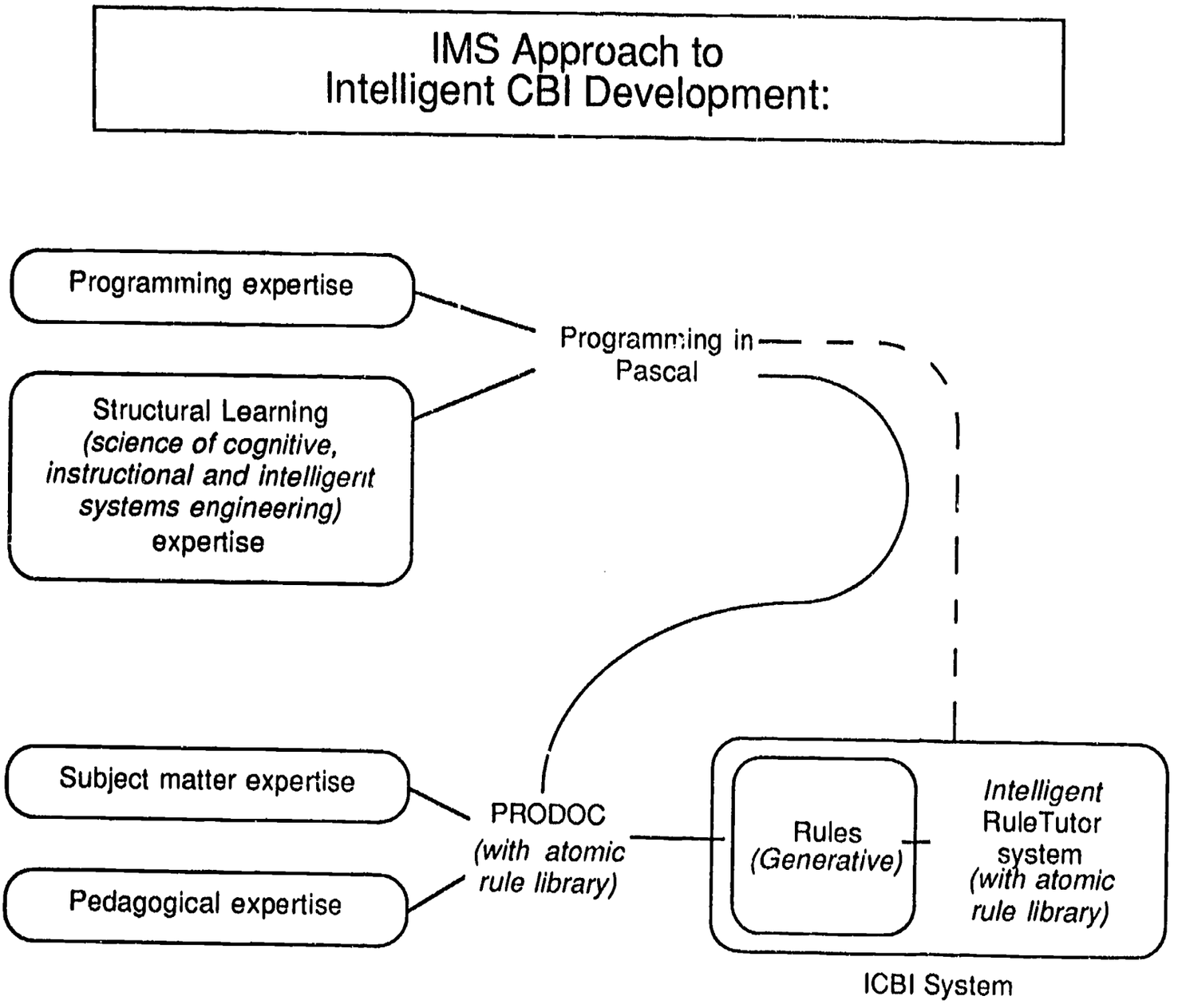


Figure 1. --- Schematic depicting IMS approach to intelligent ICBI (tutor) development and high level structure of ICBI system.

thereon. Consequently, the considerations which led to specifying the form of the rule-oriented language will be discussed first.

Individual rules are characterized as triples, consisting of a domain, a range and a restricted type of procedure. Problems as well as the domains and ranges of rules, in turn, are defined in terms of structures.

In structural learning theories, structures have been defined as n-tuples, consisting of finite (ordered) sets of psychologically meaningful (atomic or indivisible) elements, relations defined on the elements and higher-order relations defined on the relations.

Problems, then, have been defined simply as structures in which those elements corresponding to the problem givens are distinguished (i.e., labeled as givens) and those corresponding to the problem solution are replaced by goal variables. Similarly, the domains/ranges of rules are problem structures in which the givens also are replaced by (different) variables.

The procedures of rules are restricted in the sense that they must be structured. That is, each element in successive decompositions of rule procedures must consist of one of three types of elements: a sequence of simple operations, an iteration (or loop), or a decision (or branch). Rule procedures are further restricted in the sense that recursive operations are explicitly disallowed. (Although higher-order rules do not play a role in the proposed research, the role of recursion is taken over in structural learning theories by a content-independent control mechanism together with higher-order rules which operate on given rules (e.g., see Scandura, 1981a, p. 141.)

These definitions have served well as a basis for characterizing rules, structures, et al., for a wide variety of purposes, ranging from the design of basic research concerned with cognitive processes (e.g., Scandura, 1977) to computer implementation of the MicroTutor II Arithmetic tutor (e.g., Scandura, 1981b, Scandura et al, 1986).

Unlike the MicroTutor II Arithmetic tutor, however, the proposed ICBI systems would accommodate arbitrary (but previously defined) rules characterizing what is to be learned. More specifically, we have implemented an intelligent RuleTutor, which not only is vastly improved and more general, but which also is extensible -- extensible in the sense that additional aspects of the Structural Learning Theory may be added later. In future extensions of the planned implementation, for example, it would be highly desirable to: (a) accommodate arbitrarily complex content (e.g., Scandura,1971,1973,1977) and (b) automate the process of structural analysis (e.g., Scandura, 1982, 1984a, 1984b) by which the to-be-learned rules may be identified.

In this context, certain limitations of the above rule characterization became apparent. Specifically, structures had been defined in a way which could make future implementation of structural analysis more complicated than originally expected.

The original approach to generating the "structure" construct was from the BOTTOM-UP (e.g., Scandura, 1977). That is, basic atomic elements were introduced and relations were defined on them and previously defined relations. However, the process of structural analysis is essentially a TOP-DOWN process -- a successive refinement of elements (e.g., relations, atomic rules).

Given this observation, the solution to the problem became obvious: Reverse the form of the representation used for structures. Rather than thinking of structures as being built up from (sets of) atomic elements, they were recursively defined (from the top down) in terms of ordered sets whose elements themselves might be ordered sets. Since the basic elements at any level might be redefined (as an ordered set), this approach allowed for arbitrary levels of refinement as required in structural analysis. Note that an ordered set whose elements can be ordered sets is equivalent to a partial ordering. (NOTE: A partial ordering is an inverted tree-like structure in which elements may belong to more than one set. Since the following examples are all simple trees, the more familiar term "tree" is used in following discussion.)

Although the BOTTOM-UP and TOP-DOWN representations of structures are formally equivalent, the latter provides a highly efficient basis for computer implementation -- something desirable in all programming and often essential in working with microcomputers.

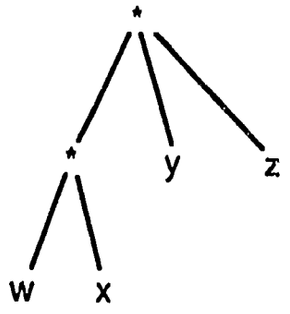
Similarly, a rule procedure generated by a top-down, successive refinement process can naturally be represented in terms of an ordered set whose elements may be ordered sets -- or equivalently as a tree (partial ordering). In structural learning theories, rule procedures (at all levels of refinement) are necessarily structured. Hence, a tree representation of a rule procedure would have three kinds of non-terminal nodes: (a) a "sequence" node would normally have two or more "children" nodes (i.e., immediate descendants in the tree), which would be the components of the sequence into which the "parent" node had been refined; (b) a "selection" node would have two or more children nodes, the first one being a (terminal) condition node and the others being the alternatives; and (c) an iteration or "loop" would have two nodes, a condition node and the body of the loop. (Note: Such a tree representation of a procedure might be executed by a recursive interpreter and/or used in generating compilable source code, a fact used directly in IMS's PRODOC software development system.)

To accommodate the top-down nature of the proposed RuleTutor Authoring system, therefore, the ORDERED SET was chosen as our basic building block. This choice has the added advantage of uniformity: procedures, structures, problems, domains, ranges, et al., can be represented as ordered sets.

Although of only incidental interest as regards the current research (which deals exclusively with cognitive procedures -- individual rules), the uniform use of ordered sets also will facilitate future implementations involving higher-order rules. Thus, since

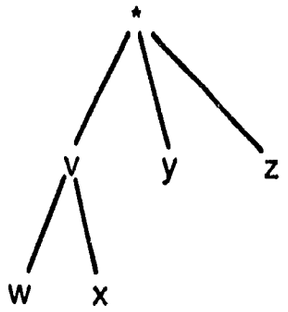
all components (domain, range and procedure) of rules must be represented as ordered sets, rules can easily be included as components of other (higher-order) rules (e.g., Scandura, 1971, 1974).

Angle bracket notation may be used to express ordered sets and trees. For example, <x,y,z> is an ordered set with three elements; x is the first, y is the second, and z is the third. Similarly, <<w,x>,y,z> is an ordered set of three elements, the first of which is itself an ordered set of two elements. Alternatively, this set can be viewed as a tree, where the symbol "*" is used for non-terminal nodes:

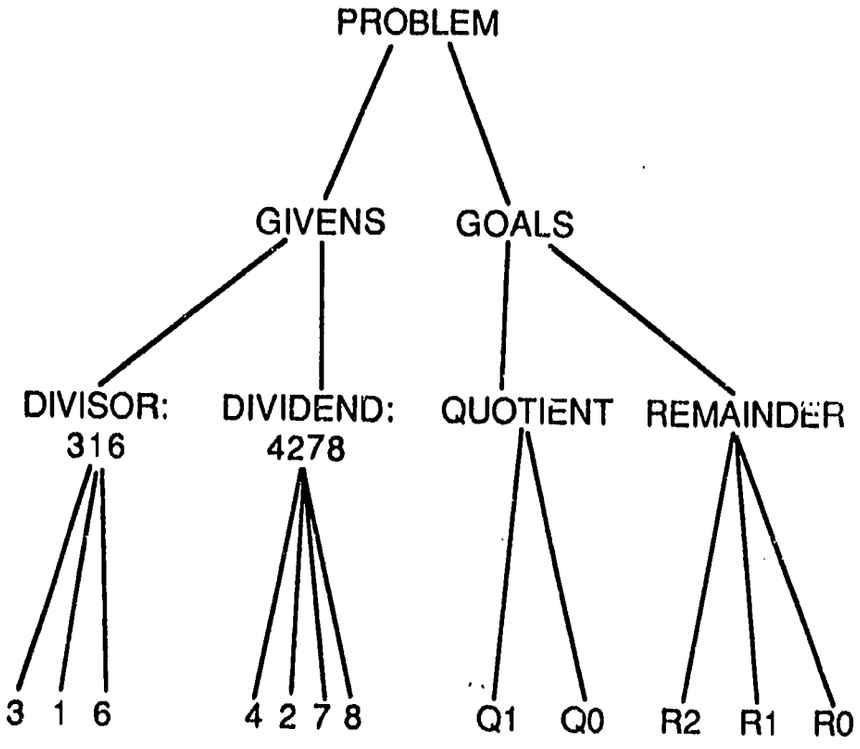


Relatively standard terminology is used to refer to relationships in trees. For example, the nodes corresponding to the elements of a set will be referred to as the "children" of the node corresponding to that set (which will be referred to as their "parent"). The highest node in the tree is called the "root"; it has no parent node. By analogy with biological relationships, the immediate descendants of a node are its children, grandchildren, etc.; ancestors may be parents, grandparents, etc. The root node plays a special role in a tree: every other node in the tree is its descendant.

Where one wants to store textual information in a node of the tree, whether or not it is a terminal node, the above notation may be extended as follows: textual information (e.g., "v") written just prior to specification of an ordered set (without an intervening comma) is associated with the node (e.g., <w,x>) whose children constitute the ordered set. Thus, <v<w,x>,y,z> would be drawn as follows:



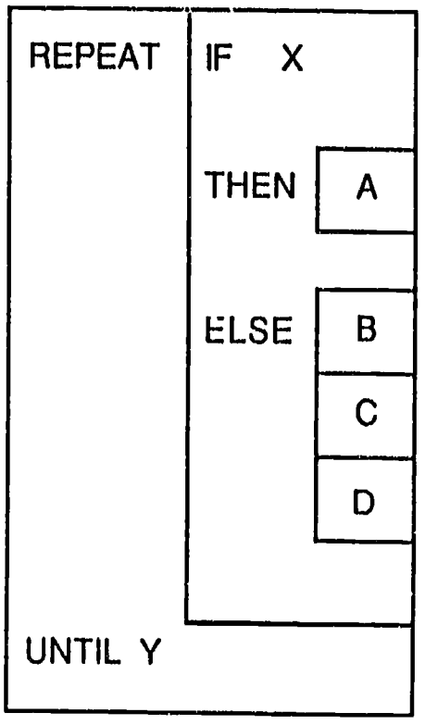
To see how trees can be used to represent previously discussed constructs, consider the long division problem, 4278 divided by 316, for which there are two components in the answer, the (integer) quotient and the remainder. It would have the following tree representation:



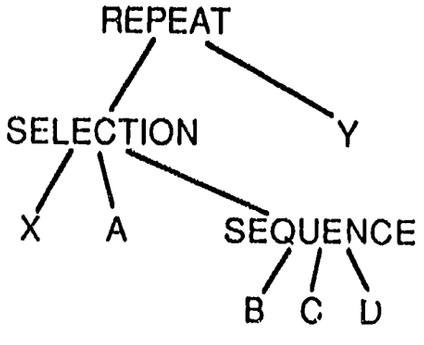
As mentioned earlier, domains and ranges of rules are structures in which some of the specific values or relations are replaced with variables (e.g., 8 by D1). Thus, we can easily derive a domain representation from the GIVENS portion of the above tree and a range representation from the GOALS portion of the tree.

In these trees, the terminal nodes are variables which designate elements from the set of decimal digits (with appropriate place values). The actual domain and range representations for long division would be somewhat complicated by the fact that the number of digits in the various elements can vary. Higher level variation of this type is accommodated naturally by allowing variable numbers of elements (terminal nodes) in the higher level (e.g., divisor) nodes.

We represent rule procedures, in terms of Scandura FLOWforms (an improved and extended form of Nassi-Shneiderman flow charts). In FLOWforms, a sequence of operations is represented by a vertical sequence of adjacent rectangles (e.g., the sequence B, C, D in the following diagram). The alternatives in a selection construct (e.g., A and (B, C, D)) and the body of a WHILE or UNTIL loop (e.g., If X, then A, else (B, C, D)) are rectangles inset within the rectangle representing the structure of which they are a part.



The tree representation of this procedure is as follows:



In the above picture, X and Y are conditions and the operations A, B, C and D are atomic rules. The equivalent ordered set representation of this procedure is as follows:

```
REPEAT < SELECTION <   X,
                       A,
                       SEQUENCE <  B,
                                   C,
                                   D
                                   >
                       >,
         Y
       >
```

Given the central role of problems and rules in both IMS's PRODOC developmental system and the proposed RuleTutor, computer implementation of trees (partial orderings) or ordered sets in the latter (RuleTutor) will directly parallel that used in PRODOC.

INTRODUCTION TO STRUCTURAL LEARNING THEORY

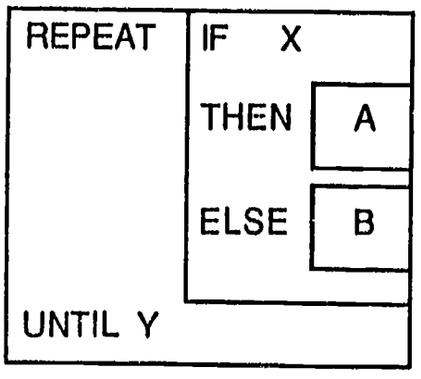
In the Structural Learning Theory (STL) a sharp distinction is made between general diagnostic testing and instructional functions, on the one hand, and the content being taught on the other (e.g., Scandura, 1971, 1977, 1980, 1981a). As detailed by Scandura (e.g., 1970, 1980, 1981a) all content in this theory is represented in terms of rules. In turn, all diagnosis (testing) and instruction is based on such rules -- rules which are identified via prior structural analysis of some body of subject matter content (Scandura, 1977, 1984a, 1984b).

Once an analysis has been completed, designing an effective instructional strategy follows directly from the theory (e.g., see Scandura 1981b, Scandura, Stone & Scandura, 1986). Specifically, once analysis has been completed, one knows: (a) what kinds of things the student is to be able to do after learning and (b) what the student must learn in order to be able to do that.

Given this information, the first thing one must do in designing an effective instructional strategy is to determine what each student already knows, specifically, which parts of what the student knows are relevant to what one wants the student to learn. This is accomplished by a highly efficient process of diagnostic testing, which makes use of the rule-based representation of content.

A basic principle in structural learning theories is that rules, including higher order rules, must be represented in terms of atomic components (i.e., atomic operations and conditions). These components are assumed to be either totally available or totally unavailable to every learner in the target population.

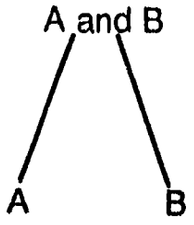
In general, different sequences of components of a rule procedure will be required in order to solve different problems in the domain of the rule, and each such sequence of procedure components is called a path. For example, consider the rule procedure represented by the following FLOWform:



In any execution (application) of this FLOWform, condition X is tested first. If X is true, operation A is carried out; otherwise B is executed. Next, condition Y is tested. If true, the process terminates. If Y is false, the above process (the body of the "loop") is repeated until Y becomes true.

Now, some problems can be solved using only the operation (rule component) labeled A (where X is true during all repetitions of the loop body); some can be solved using only B; and some require both A and B. Thus, there are three distinct paths. The paths of a rule procedure partition the associated problem domain into a set of equivalence classes; each class consists of problems whose solutions utilize the same path. In general, given atomicity assumptions, a student will at a given point in time be able to solve either ALL problems in an equivalent class or none of them (e.g., Scandura, 1977).

Consequently, by testing on as few as one problem from each equivalence class, it is possible to identify precisely and unambiguously which parts of a rule any given student knows and which parts he or she does not know. Testing efficiency can be further enhanced because the paths of a rule are hierarchically related. Higher-level paths are superordinate to lower-level paths in the sense that a higher level path includes all procedure components in its lower-level paths (identically sequenced), as well as some additional ones. Such hierarchies provide a theoretically derived and empirically verified (e.g., Durnin & Scandura, 1973) partial ordering of selected test items, according to difficulty. For the rule procedure illustrated above, the hierarchy is



This type of difficulty hierarchy can be utilized to provide unusually efficient assessment. If a student fails a problem in a given equivalence class, then the student can be presumed unable to solve problems not only in that equivalence class but also in all other equivalence classes for which the given one is a prerequisite. Similarly, if a student solves a problem in a given equivalence class, he can be presumed able to solve problems not only in the tested equivalence class but also in all classes prerequisite to it. Thus, for example, success (or failure) on one or more problems from a class near the "middle" of a hierarchy generally will allow a wide variety of other equivalence classes to be marked as known or not-yet-known (as opposed to undetermined).

In general, the diagnostic testing continues until every class is marked as known or not-yet-known by the learner. At this point, the rule components which the student does not know have been identified. Instruction on problems whose solution includes those unknown portions of the rule can then be prescribed. Structural Learning Theory is neutral on how this information is actually presented (e.g., by exposition or discovery). The important consideration is that the information is in fact learned. From a structural learning perspective: deciding on an appropriate method of presentation depends on secondary (and often higher-order) objectives that the instructional designer may or may not have in mind.

The above description implicitly assumes that there is only one cognitive procedural task to be learned. Simple structural learning theories of this type provide a sufficient basis for the ICBI systems currently under development. They do not, however allow for sets of rules, possibly including higher-order rules. In structural learning theories such phenomena as alternative perspectives, erroneous (or "buggy") rules and logical inference are accommodated in terms of rule sets. Related issues are mentioned below in the context of future extensions of the intelligent RuleTutor currently under development. For further discussion of these and related issues, see Scandura (e.g., 1977, 1980, 1985).

MICROTUTOR II ARITHMETIC TUTOR

Between 1980 and 1982, Intelligent Micro Systems, Inc., implemented an intelligent diagnostic and instructional system, called the MicroTutor II Arithmetic tutor, on the Apple II computer (e.g., see Scandura, Stone & Scandura, 1986). This system has been available commercially to schools since 1982 with the latest version released in

1984. The MicroTutor II Arithmetic tutor is based generally on the Structural Learning Theory and, consequently, incorporates a considerable amount of intelligence concerning both diagnostic testing and instruction. First, diagnostic testing is completed in a conditional and highly efficient manner. Then, instruction is provided on those paths of the rule which the student has not yet mastered.

More specifically, the Apple-based Arithmetic tutor can determine in a highly efficient manner exactly what a learner does and does not know about the task in question. It also infers what is needed to overcome inadequacies and presents that information to the learner in an optimal sequence. As currently implemented, the Arithmetic tutor deals not only with procedural skills per se, but with underlying meaning, "metacognition" (or verbal awareness of what one knows) and short-cuts commonly achieved by experts.

By itself, however, the Arithmetic tutor is useless. Despite its generalized capabilities, it needs content for its completion. This content takes the form of software for generating problems (tasks) and for solving whole number arithmetic problems. The Arithmetic tutor then utilizes these capabilities in deciding which problems to present during testing and which instruction to provide during training.

Let us consider in more detail how the MicroTutor II Arithmetic tutor functions. Given the content-specific information, the diagnostic testing portion of the system efficiently determines a student's entering level, as described above. More specifically, it stores a "checklist" for the current student (in the student records file) of the known and not-yet-known paths. This checklist is read and updated by the instructional portion of the system as it teaches the student in turn each of the not-yet-known paths.

The instruction on each path includes a number of instructional levels:

(1) teaching the meaning of the process,
(2) teaching the relationship between this meaning and the process itself,
(3) teaching the process itself, providing help where necessary,
(4) helping the student to verbalize the cognitive processes learned by having the student name the processes used or observed, and
(5) helping the student to automate the process (once the rule has been learned), thereby increasing his degree of skill.

Within each of the above instructional levels, the system also can vary the difficulty of the material and can adapt to the student by increasing problem difficulty (or type) at a rate depending on the student's prior learning efficiency. For students who are currently learning very efficiently, the difficulty level will increase relatively rapidly. Conversely, difficulty level will increase relatively slowly for students who are currently not learning as efficiently as they might. Furthermore, learners may skip some of the instructional levels mentioned above if warranted by their learning efficiency on

previous paths in the domain.

The instructional portion of the Arithmetic tutor stores detailed information about a student's performance in the student records disk file. This information can be accessed via the management portion of the system, and the parameters (e.g.,) representing, the rate at which problem difficulty is increased for a given student or the paths on which that student should be given instruction can be explicitly altered by an instructor.

In spite of these positive features, the MicroTutor II Arithmetic tutor has a number of important limitations. For one thing, the Arithmetic tutor was implemented in Applesoft (a version of BASIC developed for the Apple II computer) and 6502 assembler. Consequently, it is not easily transportable. Moreover, use of the BASIC language made it difficult to achieve the modularity we strived for.

For another thing, implementation in many cases did not reflect the underlying theory as accurately as possible. For example, meaning, metacognition and automation were treated in an ad hoc fashion (c.f. Scandura, 1973, 1977 & Scandura et al, 1985). Whereas accurate implementation called for introduction of higher-order rules (rules which operate on rules), this was not even attempted for technical reasons (e.g., the limited capacity of the Apple II computer).

Among the major conceptual limitations of the MicroTutor II Arithmetic tutor are the following: First, rule diagnosis and rule instruction in the current RuleTutor are totally independent activities. Thus, all diagnostic testing is completed (albeit in a sequential and highly efficient manner) before any instruction is provided. In fact, however, testing and teaching are highly interrelated both in practice and in principle. Thus, partial information from testing may provide a sufficient basis for (some) instruction. Conversely, instruction on a portion of a rule may influence test performance on other items and, hence, reduce the amount of instruction that otherwise might be prescribed.

Second, design limitations of the Arithmetic tutor fundamentally restricted instruction to individual rules (i.e., cognitive procedures). Consequently, the design used could not be extended to deal with sets of lower- and higher-order rules, even in principle.

Third, the basic design of the system reflected the Structural Learning Theory in only general terms. Consequently, many features of the Arithmetic tutor were fortuitous and opportunistic. In effect, the ability to deal with such things as rule meaning and verbal awareness was bought at the price of significant loss of extensibility.

Fourth, even though modularity and structured programming were at the forefront of the MicroTutor II development effort, the use of BASIC (because of its broad availability on microcomputers) and memory limitations of the Apple II computer resulted in unavoidable compromises along these lines. Thus, for example, it was not always possible to maintain modularity between rule content, on the one hand, and the

diagnostic and remedial components, on the other. Adding new content, even in whole number arithmetic, typically required (sometimes subtle and hard to identify) changes in basic diagnostic and instructional aspects of the system.

In summary, design limitations imposed restrictions on efficiency as well as on both immediate generality (limiting the variety of different content rules that could be accommodated) and future generalizability to more complex content (involving sets of rules including higher-order rules).

STRUCTURAL LEARNING AND INTELLIGENT TUTORING SYSTEMS: DESIGNS AND METHODOLOGY

In this section we describe an approach to ICBI development which not only improves on the MicroTutor II design, but is more general, more transportable, and in future work more extensible. Specifically:

(1) The design specifications not only optimize testing and instruction independently with respect to individual rules but optimize testing and instruction collectively.

(2) These specifications can naturally be extended in future work to encompass arbitrary curricular content involving any number of higher- as well as lower-order rules. This includes the possibility of alternative perspectives, along with "error" (i.e., idiosyncratic or "buggy") rules. (However, see the section on related research.)

(3) The designs ensure that the current implementation will accurately and, to the extent practicable, fully reflect the underlying theory. In the arithmetic tutor for example, basic constructs such as that of "rule" and "problem", were formulated in terms designed more to facilitate implementation in BASIC than to reflect underlying theory. Even where a concept or construct is not proposed for current implementation, every attempt was made to allow for its addition at a later time.

(4) All specifications were made as modular as humanly possible. Every effort was made to define all major ideas rigorously in a form independent of any computer language. Adherence to structured techniques, of course, necessarily biased our designs toward computer languages such as Pascal, Modula 2 and Ada, which readily lend themselves to structured programming. This difference, as much as any other, differentiates our work from traditional A1-based systems

In short, the new design is a major advance over the one used with the MicroTutor II system. Among other things, this should help ensure compatibility with the Structural Learning Theory (SLT) -- and, specifically, generalizability to arbitrary cognitive procedures (rules) and future extensibility to more general curricular content.

While space limitations make it impossible to present here most design details, the discussion which follows will hopefully provide a good sense of the basic approach. To set the context, let us begin by considering the process of instruction more generally.

All structural learning theories (SLT) (e.g., Scandura, 1971, 1977, 1980) include two major components: (1) a problem domain or domain of discourse (e.g., what is to be learned), and a set of (cognitive) rules derived by the process of Structural Analysis, and (2) the individuals (e.g., a teacher and/or learner) participating in the discourse. At this very high level, SLT's are analogous to Pask's (1975) Conversation Theory. In SLT's, however, individual knowledge is represented quite differently (e.g., rules are strictly modular) and explicit attention is given to basic psychological characteristics of the learner.

In their simplest form, SLT's involve one individual interacting with its environment, via (relative to) some proscribed domain of discourse (characterized in terms of problems in the problem domain and the rules, including higher-order rules, associated with them). The goal-directed individual is viewed as attempting to solve "problems" that are somehow presented to him or to achieve desired results. The individual's responses are generated via its available rules of knowledge and the cognitive universals governing their use. (New rules generated via higher-order rules are said to be "learned").

The SLT also addresses the basic question of what an individual knows (that is relevant) to begin with. The individual's knowledge is operationally defined in terms of the rules characterizing the problem domain. As described briefly above, and more fully by Scandura (e.g., 1977), the individual's responses to specific problems are used to indicate which parts of which prototypic rules associated with the problem domain that the individual knows.

A more general form, represents a dialogue between two (or more) individuals (e.g., a teacher and a learner). It should be noted that in general neither individual has perfect knowledge of the domain nor perfect diagnostic and/or teaching knowledge. If one of the participants did, "perfect" communication would theoretically be possible (but only with respect to the domain of discourse). In any case, there is no a priori guarantee that an individual participant can either accurately assess what the other participant knows (or can do) or influence that participant in theoretically optimal ways. In general, these inferencing and influencing capabilities will be partial.

The teaching-learning process involves a variation of the above in which one of the participants is an "idealized" teacher and the other, a learner. This relationship is represented schematically in Figure 2.

Various aspects of this characterization of the teaching-learning process have been discussed in detail in previous publications. The process of structural analysis (SA), for example, has evolved over a period of many years (e.g., Scandura, Durnin & Wulfeck,

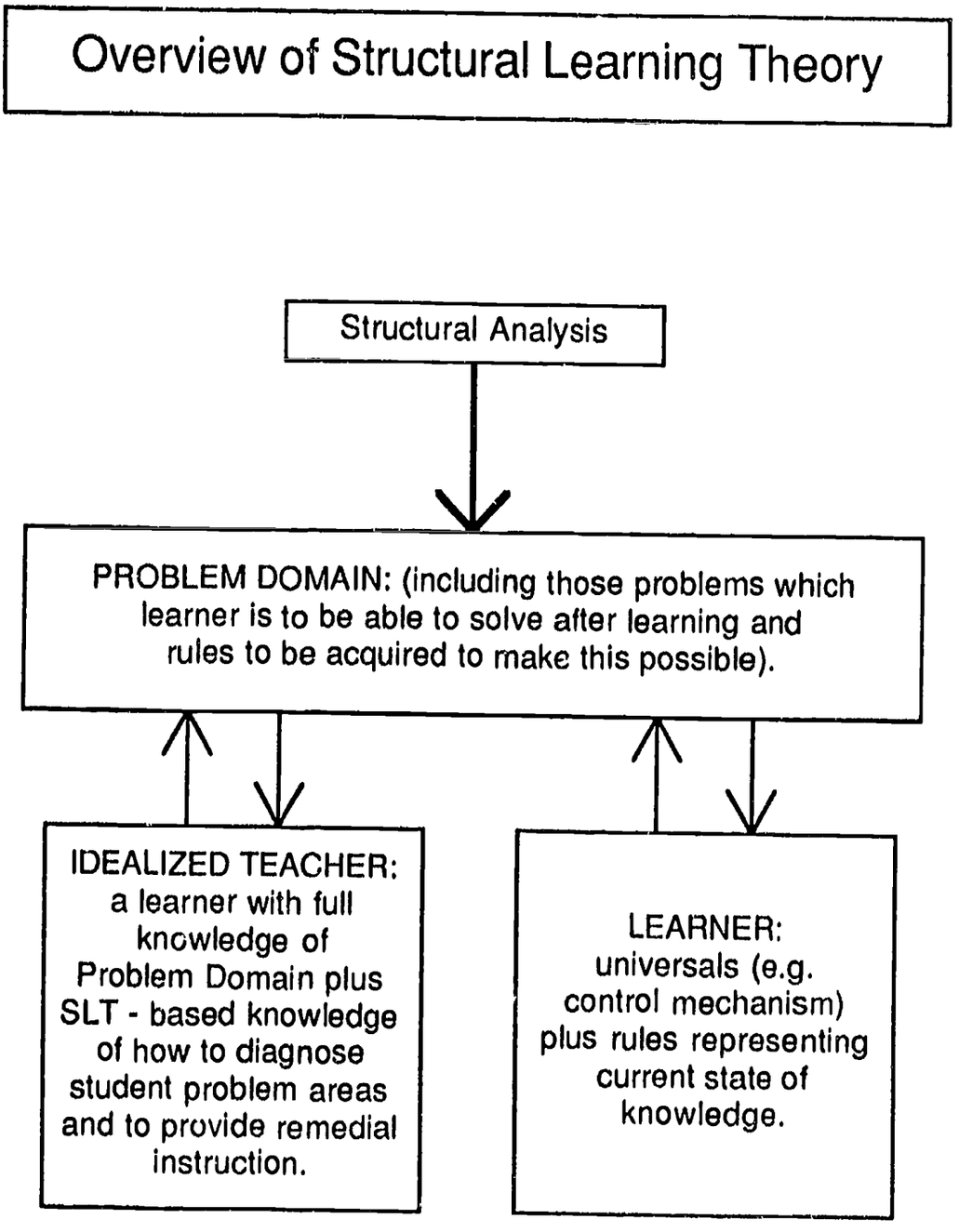


Figure 2. --- Overview of Structural Learning Theory

1974; Scandura, 1977; 1984a,b). Today, SA has reached the point in its evolution where critical aspects of the process might reasonably be automated. A high level summary of this process is shown in the procedure FLOWform of Figure 3.

(Procedure FLOWforms are a convenient means of representing the procedural component of rules in a rectangular area, such as a video screen.) In this FLOWform, the name used by the Disk Operating System (DOS) to access the FLOWform is printed on the top line in brackets with the full name "Structural_Analysis" following the colon. Immediately below is a top-level, symbolic representation of Structural Analysis.

rule_derivation_hierarchy := STRUCTURAL_ANALYSIS (problem_domain)

where "problem_domain" is input into the operation "STRUCTURAL ANALYSIS." When carried out, this operation generates (i.e., results in) a rule_derivation_hierarchy (and indirectly those rules which can be derived directly or indirectly from them, e.g., Scandura, 1971, 1973b, 1977). Next, in braces are more complete descriptions of the new terms in this symbolic representation. This same pattern is used in each of the following FLOWforms.

More details on SA in its current state are given in Scandura (1984a). In the present context, it must be emphasized that there is NO limit on the number of different perspectives from which a problem domain may be analyzed (e.g., see Durnin and Scandura, 1973). We also note that, as with all potential knowledge, logical inferencing capabilities are represented in terms of rules (e.g., Scandura 1973, 1977a). (Where the inferencing involves other rules, the inference rules may be of a higher order.) "Bugs" similarly are represented as (error) rules or perturbations on a given rule. See the section on related research for further discussion.

(Note: Higher "level" rules are not to be confused with higher "order" rules. The former are more encompassing rules that are relatively higher in a rule hierarchy. The latter are rules which operate on other, relatively lower order rules. See Scandura (1973b) for further discussion of this point.)

Although SA has a major place in our long range plans, in the present context the rules to be learned must be identified and represented directly by the instructional designer (or subject matter expert). However, as we shall see in a later section, availability of the PRODOC development system makes this task much less onerous than it otherwise might be.

The current ICBI research is concerned primarily with those portions of the teaching-learning process pertaining to the Idealized Teacher and learner. Notice, in particular, that in this model the idealized teacher knows (i.e., has direct access to) all of the prototype rules (representing what is to be learned), and can recognize and/or generate arbitrary problems in the Problem Domain. In addition, this idealized teacher assumably has built into it all of the theoretically optimal machinery for diagnosing

A.RUL]:Structural_Analysis Copyright 1987 Scandura

le_derivation_hierarchy := STRUCTURAL_ANALYSIS (problem_domain)

peration: STRUCTURAL_ANALYSIS = process described in Scandura, J. M. Structural (Cognitive Task) Analysis: II. Systematization of the Method. Journal of Structural Learning, 1984, 8, 1-28.

nput: problem_domain = generalized specifications for class of problems corresponding to capabilities student is to have after learning.

utput: rule_derivation_hierarchy = hierarchy of rules obtained as a result of structural analysis, performed iteratively, with base rules at bottom and those corresponding to sample problems/curriculum goals at the top; has two parts: hierarchy of rules by name and individual rules.}

prototypic_problems := SELECT_PROBLEMS (problem_domain)

{SELECT_PROBLEMS = selects sample of problems characteristic of problem domain; in curriculum planning these problems may correspond to specific curriculum goals.
prototypic_problems = in education, set of problems illustrative of curriculum goals.}

. rule_derivation_hierarchy := RULE_HIERARCHY_ANALYSIS (prototypic_problems)

REPEAT

. Derivation_set := NEXT_LEVEL_OF_ANALYSIS (rule_hierarchy, prototypic_problems)

rule_derivation_hierarchy {at next level} := RULE_ANALYSIS (prototypic_problems, rule_derivation_hierarchy)

{rule_derivation_hierarchy = hierarchy of rules where set of rules at any level is sufficient to generate both solutions to prototypic problems and underlying rules at higher levels.}

derivation_set := GET_DERIVATION_SET (rule_derivation_hierarchy)

{derivation_set = set of terminal rules at base of rule_derivation_hierarchy.}

UNTIL SUFFICIENTLY_POWERFUL (problem_domain, derivation_set)

{SUFFICIENTLY_POWERFUL = derivation_set provides a sufficient basis for solving all problems in problem_domain (either directly or indirectly in terms of rules derivable from the underlying rules) or they are otherwise judged sufficiently powerful by curriculum designer.}

Figure 3. --- Structural Analysis.

learner difficulties and for providing optimally efficient remediation. This may or may not include idealized inferencing capabilities of the sort used in expert systems.

The learner, in turn, is characterized in terms of some subset of the idealized knowledge plus universals. See the FLOWform for more details. Far more extensive discussion of the learner model may be found in a variety of publications (e.g., Scandura, 1971, 1973, 1977a (esp. Chapter 2), 1980).

This idealized teacher is characterized at a very high level in the FLOWform below. Notice that no constraints are placed on the content to be taught; hence the term "Curriculum-Tutor". Provision is made in the FLOWform for arbitrary problem domains, involving sets of higher- and lower-order rules, albeit at a rather high level. If fully implemented, such a system might be used to provide instruction on learning strategies, including logical inference (higher-order rules), lower-order rules (cognitive procedural tasks) and interactions among them. It also provides for alternative perspectives (including error rules) and perturbations on prototypes.

The Curriculum_Tutor takes a formal characterization of the learner as input and provides optimal diagnosis and instruction needed to produce learner mastery on all rules in the rule derivation hierarchy. The learner, formally speaking, is characterized by a universal control mechanism, a processing capacity and a set of rules (possibly including higher-order rules) representing that portion of available knowledge that is relevant to the problem domain (i. e., curriculum).

Refinement of the CURRICULUM_TUTOR into components involves a REPEAT...UNTIL loop: a GENERALIZED_RULETUTOR, which would work with arbitrary rule_derivation_hierarchies, constitutes the body of the loop and MATCH constitutes the terminating condition.

In our research, the body of the main loop has undergone further refinement but the details are not important here. The basic jist of this design (refinement) is to determine tasks which maximally "stretch", but still lie within, the learner's capabilities (at each point of time). Normally, this will require the learner to mobilize a variety of higher- and lower-order rules, and may involve attacking the problem from any of the perspectives considered during structural analysis of the content.

After testing on each such task, the learner's status is updated. This essentially involves keeping current the list of known and unknown rules. The basic process (loop body) is repeated whenever the learner is successful (until the learner's status matches the rule-derivation hierarchy). Before looping on failure, the Intelligent Curriculum-Tutor determines an unmastered rule nearest the bottom of the hierarchy and provides instruction on that rule.

It is relatively easy to understand the individual components of the refined design -- especially if one is familiar with the underlying structural learning theory.

[CURRTUTR.RUL]:curriculum_tutor

learner {with all rules in rule_derivation_hierarchy mastered} :=
CURRICULUM_TUTOR (rule_derivation_hierarchy, learner)

CURRICULUM_TUTOR = provides optimal diagnosis and instruction needed to produce learner mastery on all rules in rule hierarchy.

learner = characterized by an universal control mechanism and processsing capacity, and a rule set (list of mastered rules in hierarchy, including portions thereof, may be sufficient but, for purposes of efficiency, complete characterization of rule derivation hierarchy, path hierarchy or problem types hierarchy and path components for all rules in rule_derivation_hierarchy might be needed).}

REPEAT

learner {with all paths of teachable target_rule mastered} :=
GENERALIZED_RULE_TUTOR (rule_derivation_hierarchy, learner)

{GENERALIZED_RULE_TUTOR = determines initial target_rule, tests to update undetermined (target and prerequisite) rules, reassigns target_rule if necessary, and provides instruction on teachable target_rules.}

target_rule := GET_MAXIMAL_TARGET_RULE (rule_derivation_hierarchy, learner)

{GET_MAXIMAL_TARGET_RULE = determines rules in rule_derivation_hierarchy requiring maximal use of learner's available rules and processing capacity, then gives learner option of selecting one of these rules (where higher-order selection rules are involved, there will only be one target_rule at each stage) or, if not, chooses the first of these rules.
target_rule = rule in hierarchy to serve as target for diagnosis.}

proposed_solution := RULE_TEST (target_rule, learner)

{RULE_TEST = identify and administer problem.}

learner {with status updated} := EVALUATE_SOLUTION_UPDATE_STATUS (proposed_solution, target_rule, learner, rule_derivation_hierarchy)

{EVALUATE_SOLUTION_UPDATE_STATUS = evaluates and uses proposed solution to update learner status (i.e., components and paths of rules and derivation sets of rules).}

target_rule := GET_MINIMAL_TARGET_RULE (rule_derivation_hierarchy, learner)

{GET_MINIMAL_TARGET_RULE = selects lowest level rule having at least one failed path.
NOTE: On success during RULE.TEST, derivation sets are marked passed so base tends to move toward target_rule; on failure, some paths may be marked failed; we always select a lowest level rule in a derivation set with failed types as target.}

target_rule {with all paths mastered} := TEACHABLE_RULE_TUTOR (target_rule, learner)

IF TEACHABLE (target_rule, learner)

{TEACHABLE = determines whether target_rule has failed paths and is at base of rule_derivation_hierarchy or all rules in one of its derivation sets have been mastered; in latter case, we could teach non-used prerequisites.}

THEN {*****}
target_rule {with all paths mastered} := RULE_TUTOR

```
        [RULE_TUTOR = provides optimal diagnosis and instruction on
                      target_rule.}
UNTIL MATCH (learner, rule_derivation_hierarchy)
      {MATCH = determines whether or not learner mastered rules are
               equivalent to those in rule_derivation_hierarchy.}
```

Figure 4. --- Curriculum_Tutor.

Implementation of many of the components in a computer environment, however, would be something less than trivial. One of the components, GET_MAXIMAL_TARGET_RULE for example, would require implementation of the universal control mechanism assumed to govern interaction among all rules of knowledge. To date, the only serious attempt to accomplish this was in a dissertation by Wulfeck (see Wulfeck & Scandura, 1977). Even here, the control mechanism was not completely independent of the higher order rules. While subsequent work (e.g., Scandura, 1981) provides a potential solution to the problem, implementation would constitute a major research project in its own right.

In effect, while desirable, implementation of a CURRICULUM_TUTOR of this sort is beyond the scope of the present implementation. Our reasons for including discussion here are primarily to provide a broader perspective and to show how the current research could be extended at a later time.

It is rarely (if ever) possible to fully implement any non-trivial psychological theory as an operational computer program. The Structural Learning Theory (SLT) is no exception to this rule. More generally, the SLT makes specific provision for learning strategies (higher-order rules), the process of learning itself (e.g., via rule interactions determined by a universal control switching mechanism), skill acquisition, and in general, instruction on arbitrary content.

THE INTELLIGENT RuleTutor PROTOTYPE

Rather than attempt to implement the SLT in its entirety, the intelligent RuleTutor prototype is concerned exclusively with the teaching and learning of cognitive procedural tasks (i.e., problem domains solvable via a single rule: one domain, one range and one procedure).

In developing the RuleTutor, we introduced simplifying assumptions which made implementation of the prototype feasible. Nonetheless, the fact that the RULETUTOR is an essential component of the CURRICULUM_TUTOR is especially important given our stated interest in future generalizability. (Although greatly improved, and more general than the earlier Apple II implementation, we retain the name "RuleTutor" since the emphasis still is on cognitive procedural tasks.) At a high level notice that the RuleTutor FLOWform (see Figure 5) also has the same general form as the CURRICULUM_TUTOR. Thus, the latter repeats the GENERALIZED_RULE_TUTOR until the rule_derivation_hierarchy characterizing the entire curriculum has been mastered. Analogously, the RuleTutor repeats the PATH-TUTOR until the targeted rule (cognitive procedural task) has been mastered.

Implementation of the RuleTutor, of course, required progressive refinement of the basic design. In the process, we used PRODOC's high level simulation capabilities to systematically test the design at each stage of refinement. Successive refinement

```
DOS name: RULETUTR.NRL          Language: LIBRARY
PROGRAM RuleTutor (learner, target_rule, problem_type_hierarchy, problem_type, ma
ry_criteria, P(k), k, P1, P2, P3, learner);

     learner : status;
     target_rule : ;
     problem_type_hierarchy : ·
     problem_type : ;
     mastery_criteria : ;
     P(k) : ;
     k : 0;
     P1 : 80;
     P2 : 20;
     P3 : 3

VAR learner : status

RULETUTR.NRL]:RuleTutor                                    Copyright 1987 Scandura
OMAIN: learner{}, target_rule{}, problem_type_hierarchy{}, mastery_criteria{}
ANGE: learner{}

earner {with rule mastered} := RULETUTOR (target_rule, learner {with
                                          current rule status as
                                          problem_type_hierarchy,
                                          and mastery_criteria})

RULETUTOR = determines learner status on target_rule,
            then provides optimal diagnosis and instruction on
            target_rule until mastery of all paths (problem_types
            in hierarchy) is achieved.
roblem_type_hierarchy = hierarchy of problem types corresponding
                        to path hierarchy of target_rule.
astery_criteria = criteria used to determine mastery on paths of
                  target_rule.}

learner {with current status of knowledge of target_rule}
        := INITIALIZE_LEARNER (target_rule, learner)

{INITIALIZE_LEARNER = loads in learner file (if any) with current
                      status on target_rule, else makes a copy of
                      target_rule to characterize knowledge with
                      status of all paths undetermined.}

REPEAT  learner {with teachable problem_type mastered} :=
                DIAGNOSTIC_PATH_TUTOR (learner {current status
                                       and mastery_criteria
                                       specified})

        {DIAGNOSTIC_PATH_TUTOR = provides optimal testing and
                                 instruction on a path given
                                 current state of learner.}
         problem_type = equivalence class of problems defined by
                        a path of the target_rule but stored under
                        learner's knowledge.}

UNTIL RULE_MASTERED_OR_END_LESSON (learner)

      {RULE_MASTERED = determines whether learner has mastered all problem
                       types for target_rule.}
```

Figure 5. --- Overview of RuleTutor.

levels of a more detailed Ruletutor design are shown in the second RuleTutor FLOWform.

In general terms, after determining the learner's status at the beginning of a lesson, the RuleTutor tests the learner to determine which parts of the to-be-learned target_rule have been mastered (or retained) and which have not. At appropriate points in the testing process (e.g., when all prerequisites to a failed problem type have been mastered), instruction is provided on missing information. This process is continued until the entire rule has been mastered. Specifically, the high-level REPEAT construct in the FLOWform indicates that the testing/teaching process is repeated until problem types associated with ALL paths of the given rule are mastered by the student. (Note: Although no machinery has been included to allow for rule derivation, control mechanisms and the like, the importance of the fact that they could be added at a later time, without having to change the RuleTutor itself, cannot be overemphasized.)

The first major component (a sequence of steps) within the REPEAT...UNTIL loop involves determining which problem_type (or associated path of the target_rule) in the path hierarchy will provide the maximum amount of information about the learner's (relevant) knowledge. This will always be a problem_type which is as yet "undetermined" (i.e., not yet marked as "mastered" or "failed").

The second major component is an (embedded) REPEAT...UNTIL loop which generates a test problem of the given problem_type, gets the learner's solution, evaluates or grades the solution, and then computes the probability that the learner knows the path corresponding to that problem_type. If the learner's probability of knowing is between two predetermined values (say 20% and 80%), another problem of the same problem_type would immediately be presented.

Otherwise, diagnosis on the problem_type is complete so the RuleTutor updates its information on the learner's knowledge: If the learner's probability of knowing is greater than the pre-set upper limit (e.g., 80%), the RuleTutor will mark that problem_type and its prerequisites, if any, as "mastered"; otherwise, it will mark as "failed" both that problem type and those types, if any, for which it is a prerequisite.

The next step involves determining which failed problem_types or path at the lowest level in the path_hierarchy.

Finally, the last major component of the loop (an IF...THEN construct) determines whether all of the prerequisites of any of the minimal level problem_types/paths have been mastered. If so, instruction is provided on that path. Such a path is an ideal candidate for instruction. Not only can instruction proceed from a solid base (of prerequisites) but its position near the base, relatively speaking, provides optimal potential for transfer to other paths. Transfer of this sort will be detected the next time through the loop making instruction unnecessary on the affected paths.

[RULETUTR.NRL]:RuleTutor
DOMAIN: learner{}, target_rule{}, problem_type_hierarchy{}, mastery_criteria{}
RANGE: learner{}

learner {with rule mastered} := RULETUTOR (target_rule, learner {with current rule status as problem_type_hierarchy, and mastery_criteria})

{RULETUTOR = determines learner status on target_rule, then provides optimal diagnosis and instruction on target_rule until mastery of all paths (problem_types in hierarchy) is achieved.
problem_type_hierarchy = hierarchy of problem types corresponding to path_hierarchy of target_rule.
mastery_criteria = criteria used to determine mastery on paths of target_rule.}

learner {with current status of knowledge of target_rule} := INITIALIZE_LEARNER (target_rule, learner)

{INITIALIZE_LEARNER = loads in learner file (if any) with current status on target_rule, else makes a copy of target_rule to characterize knowledge with status of all paths undetermined.}

learner {with rule mastered} := DIAGNOSTIC_RULETUTOR (learner {with current status and mastery_criteria specified}, problem_type_hierarchy)

{DIAGNOSTIC_RULETUTOR = provides optimal diagnosis and instruction until lesson ends or mastery of all paths is achieved.}

REPEAT

learner {with teachable problem_type mastered} := DIAGNOSTIC_PATH_TUTOR (learner {current status and mastery_criteria specified})

{DIAGNOSTIC_PATH_TUTOR = provides optimal testing and instruction on a path given current state of learner.}
problem_type = equivalence class of problems defined by a path of the target_rule but stored under learner's knowledge.}

problem_type := GET_PROBLEM_TYPE (learner)

{GET_PROBLEM_TYPE = determines undetermined problem_type (equivalence class) for testing, given learner's current state.}

path_level := SET_LEVEL (learner)

{SET_LEVEL = Reset test level so as to minimize expected number of levels that need to be tested -- e.g., determines highest and lowest level paths whose status is still undetermined, then computes the average,
[(highest - lowest) / 2] + 1 .}

problem_type := GET_PROBLEM_TYPE_AT_LEVEL (path_level)

{GET_PROBLEM_TYPE_AT_LEVEL = Find next undetermined problem_type at level unless learner has specified a problem_type or a specific problem, k := zero (k = problem_counter).}

k := "0"

{k = number of presented problems of the problem_type}

P(k) := PROBLEM_TYPE_TEST (problem_type)

```
.        problem_type = corresponds to path of target_rule.
.               P(k) = probability of mastery based on
.                      learner's responses to k test items.}

REPEAT
    P(k) := PROBLEM_TEST (problem_type, P1, P2, P3)

    {PROBLEM_TEST = tests on a specified problem_type.
                P1 = minimum probability P(k) above which
                     indicates mastery.
                P2 = maximum probability P(k) below which
                     indicates failure.
                P3 = maximum_number_test_items.}

        test_problem := PROBLEM_GENERATOR (problem_type)

        {PROBLEM_GENERATOR = generates a test_problem.
        test_ problem = test item for specified
                        problem_type.}

        k := add (k, "1")

        problem_solution := GET_SOLUTION (test_problem, k)

        {GET_SOLUTION = gets learner's responses and displays
                        them on screen.
         problem_solution = learner's sequence of responses to
                            current problem.}

        P(k) := GRADE_SOLUTION (problem_solution, learner,
                                test_problem, k)

        {GRADE_SOLUTION = compares problem_solution with
                          solution to test_problem
                          generated by target_rule
                          (idealized learner) and
                          computes new probability of
                          mastery, P(k);
                          this computation may be based
                          on answers to steps in
                          problem_solution and/or may
                          include partial credit for a
                          partially correct answer.}

UNTIL GREATER_THAN (P(k),P1) OR
      LESS_THAN (P(k),P2)    OR
      GREATER_THAN (k, P3)

      {P(k) > P1 (e.g., 80%) or
      P(k) < P2 (e.g., 20%) or
      k > P3 (e.g., 4); i.e., problem_type success or failure is
      known with measured certainty or a pre-specified number of
      problems have been tested.}

learner {with status updated} :=
        UPDATE_LEARNER_STATUS (P(k), learner, P1)

{UPDATE_LEARNER_STATUS = evaluates proposed_solution and
                         updates learner's status: If
                         probability of mastery, P(k),
                         is greater than given minimum,
                         P1, then mark problem_type and
                         all of its descendants/
                         prerequisites mastered,
                         else mark problem_type and its
                         ancestors failed.
mastery_criteria: P1 = success %.
                  P2 = failure %.
                  P3 = maximum_number_of_test_items.}

problem_type := GET_MINIMAL_PROBLEM_TYPE
                (learner, problem_type_hierarchy)

{GET_MINIMAL_PROBLEM_TYPE = selects lowest level
                            problem_type/path
                            failed by learner.}

.  problem_type := TEACHABLE_PATH_TUTOR (problem_type,
.                                        target_rule, learner)
.
.  {NOTE:  The target_rule includes verbal instruction
.  pertaining to the target_rule whereas the learner's
.  knowledge (rule) does not; conversely, the learner's rule
```

```
        ** PREREQUISITES_PASSED (problem_type, learner)
            {PREREQUISITES_PASSED = determines whether all paths
                                    prerequisite to problem_type
                                    have been mastered.}
        THEN {*****}
             learner {with path mastered}
                 := PATH_COMPONENT_TUTOR (learner, target_rule,
                                          problem_type)
             {PATH_COMPONENT_TUTOR = provides instruction on missing
                                     components of solution path (if all
                                     prerequisites are mastered).}
UNTIL RULE_MASTERED_OR_END_LESSON (learner)
      {RULE_MASTERED = determines whether learner has mastered all problem
                       types for target_rule.}
```

Figure 6. --- Expanded RuleTutor.

The step pertaining to the PATH_TUTOR (marked with "*" in the FLOWform) plays a central role in the RuleTutor prototype and deserves elaboration. Specifically, when a learner has failed a problem_type and is to receive instruction, he or she may very well know some components of the rule path corresponding to that problem type. Consequently, in an optimal system, instruction should focus on those components which the learner does not know. In general terms, this can be inferred from the status of the steps of paths which are known to be mastered or failed.

The RuleTutor also allows the learner some discretion as regards selecting the kind of instruction to be provided. This option recognizes the fact that a given learner may need or wish to receive a particular type of instruction.

The PATH_TUTOR FLOWform shown below describes the above process in more detail.

It is important to observe that receiving instruction is conditional on whether or not the component in question is already known. Also, whereas the instruction will be tailored to both specific path components and individual tastes, the PATH_TUTOR does not assume mastery. Rather, mastery is determined the next time the RULETUTOR recycles through its diagnostic phase (i.e., diagnostic steps).

Just before the instruction begins, GENERATE_PROBLEM is used to generate a test_problem corresponding to the failed path. If the current component is failed, INSTRUCT provides instruction on the component as the learner desires. On mastered steps, the PATH_TUTOR will enter the answer automatically (for purposes of instructional efficiency) and proceed to the next component. The process repeats until all components of the path have been covered.

Notice that in addition to being able to generate needed problems, actual operation of the RULETUTOR and PATH_TUTOR requires interpretation of arbitrary portions of the rule being taught. For example, at various points these tutors must generate answers to particular steps either for display purposes or for comparison with learner answers.

In this regard, all rule components are represented in terms of trees involving only structured components. The RuleTutor prototype incorporates an interpreter which can be used with any rule in which the terminal elements correspond to PRODOC's library rules. Specifically, interpretation of a node in a procedure tree (i.e., FLOWform) will be a function both of the structure type of the node (sequence, selection, etc.) and of the children of the node.

If the node is a sequence node, for example, its children are interpreted successively. If it is a selection node, its condition child will select one of the two other children nodes to be interpreted. Those children nodes which happen to be terminal nodes correspond to previously compiled procedures (psychologically relevant units, or atomic rules) which are executed directly.

```
[PATHTUTR.RUL]:Path_Component_Tutor                    Copyright 1987  Scandura
problem_type {mastered} :=
            PATH_COMPONENT_TUTOR (learner, target_rule, problem_type)
{PATH_COMPONENT_TUTOR = provides instruction on missing components of solution
                        path associated with current problem type.}

solution_path {with status set at first component} :=
              COMPONENT_KNOWLEDGE (learner, target_rule, problem_type)
{COMPONENT_KNOWLEDGE = determines whether components of path associated with
                       the current problem_type are mastered, failed or
                       undetermined.}

test_problem := GENERATE_PROBLEM (problem_type)

. . . . . .
. learner {with all components of solution_path mastered} :=          .
.           COMPONENT_TUTOR (test_problem, solution_path, learner)    .
.                                                                     .
. {COMPONENT_TUTOR = testing and/or instruction on unknown components .
.                    of current problem_type.}                        .
. . . . . .

REPEAT
    . . . . . .
    . learner := CONDITIONAL_TEACH_COMPONENT                          .
    .                 (test_problem, solution_path, learner)          .
    . . . . . .
    IF COMPONENT_FAILED (solution_path)
    THEN
        . . . . . .
        . problem_solution {updated}, solution_path {updated} :=      .
        .             TEACH_COMPONENT (test_problem, solution_path,   .
        .                              learner)                       .
        .                                                             .
        . {TEACH_COMPONENT = provide instruction on current           .
        .                    component.}                              .
        . . . . . .
        trial_solution :=INSTRUCT (test_problem, solution_path)
        {INSTRUCT = give learner option of responding with knowledge
                    of results, being shown how or being given
                    verbal instruction; allow repeats.}
        problem_solution {updated}, solution_path {updated}:=
                  NEXT_COMPONENT = (test_problem, solution_path)
        {NEXT_COMPONENT = generates answer to current component;
                          advances to the next component of the
                          solution_path for testing and/or
                          instruction.}
    ELSE
        problem_solution {updated}, solution_path {updated}:=
                  NEXT_COMPONENT = (test_problem, solution_path,
                                    learner)
        {NEXT_COMPONENT = generates answer to current component;
                          advances to the next component of the
                          solution_path for testing and/or
                          instruction.}
UNTIL FINISHED (solution_path)
```

Figure 7. --- PathTutor FLOWform.

IMS'S PRODOC System

In its most basic sense software development involves describing the tasks to be solved -- including the given objects and the operations to be performed on those objects. Moreover, such descriptions must be precise in order for a computer (or human) to perform as desired. Unfortunately, the way people describe objects and operations typically bears little resemblance to source code in most contemporary computer languages.

There are two potential ways around this problem. One is to allow users to describe what they want the computer to do in everyday, typically imprecise English (or to choose from a necessarily limited menu of choices). This approach has some obvious advantages and a considerable amount of research is underway in the area. The approach, however, also has some very significant limitations: (a) it currently is impossible to deal with unrestricted English, and this situation is unlikely to change in the foreseeable future, and (b) even if the foregoing limitation is eventually overcome, the approach would still require the addition of complex, memory intensive "front ends". These "front ends" interact with the user's typically imprecise English statements and effectively "try to figure out" what the user intends. The result invariably is a system which is both sluggish in performance and limited in applicability.

PRODOC is based on a second, arguably more flexible approach. The terminology used in PRODOC may be customized so as to match the way human experts in any given application area naturally describe the relevant data and operations. This customized terminology is all based on a uniform, very simple syntax that might easily be learned by an intelligent human (in a few minutes time). The approach taken with PRODOC is absolutely general, as well as far more efficient and easy to use.

The PRODOC system provides support for the entire systems software development process, including requirements definition, system design, testing, prototyping, code generation and system maintenance. It consists of four distinct but complimentary and fully compatible software productivity and quality assurance environments running in 640K of memory under MS-DOS. Each of these environments makes use of Scandura FLOWforms. (FLOWforms look similar to Nassi-Shneiderman flow charts, but they make better use of the rectangular screen and allow simultaneous display of as many (or as few) levels of representation as may be desired.)

(1) Applications Prototyping Environment (with interpreter and expert assistant generator) (PRODOCea) - is suitable for use by nonprogrammers as well as programmers for designing, documenting, implementing, and maintaining software systems in an integrated, graphically supported, top-down structured environment. In addition to English text, the availability of greatly simplified, very high level library rules makes PRODOCea ideal for rapid prototyping. Support for input and output data

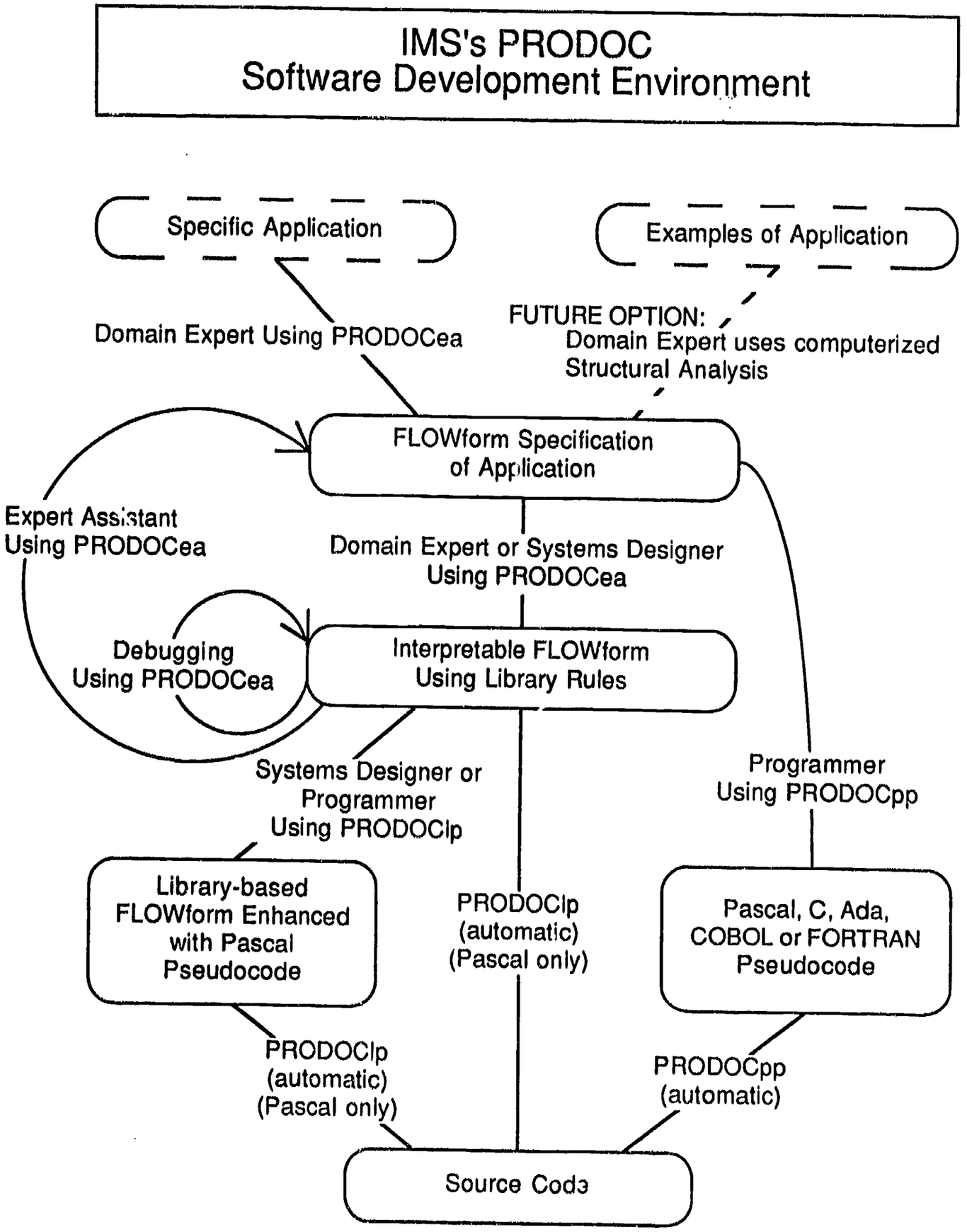


Figure 8. --- Overview of PRODOC software development system.

structures also makes it possible to directly reflect arbitrary semantic properties.

The current version of PRODOCea employs a general set of library rules especially designed for prototyping. Table 1 shows the complete list of library rules and their parameters.

CATALOG OF LIBRARY RULES

- - INPUT/OUTPUT - -

display_structure
(ROOT_ELEMENT,DISPLAY_PARAMETERS,DISPLAY_NON_TERMINAL)
display
(ELEMENT,DISPLAY_PARAMETERS)
load
(ROOT_ELEMENT,DOS_NAME,DRIVE,FILE_TYPE)
save
(ROOT_ELEMENT,DOS_NAME,DRIVE,FILE_TYPE)
get_input
(ELEMENT,DISPLAY_PARAMETERS)
clear_video

- - OPERATIONS - -

insert_component
(SET,PREVIOUS_COMPONENT,DISPLAY_PARAM_OR_VALUE,NODE_TYPE,NAME)
delete_component
(COMPONENT,SET)
share_component
(COMPONENT,SET,PREVIOUS_COMPONENT)
delay
(SECONDS)
insert
(SOURCE_ELEMENT,DESTINATION_ELEMENT,INSERT_POSITION)
delete
(ELEMENT,START_POSITION,LENGTH)
delete_display_parameters
(ELEMENT)
move_component
(COMPONENT,SOURCE_SET,TARGET_SET,PREVIOUS_COMPONENT)

- - PARAMETER FUNCTIONS (Character / Arithmetic) - -

concatenate
(FIRST_ELEMENT,SECOND_ELEMENT)
extract
(SOURCE_ELEMENT,START_POSITION,LENGTH)
add
(ADDEND1,ADDEND2)
subtract
(TOP,BOTTOM)
multiply
(FACTOR1,FACTOR2)

divide
(DIVIDEND,DIVISOR)
power
(BASE,EXPONENT)
greatest_integer
(X)
modulo
(X,BASE)
absolute_value
(X)
round
(X,PRECISION)
component_with_value
(SET,VALUE)
next_component
(SET,PREVIOUS_COMPONENT)
common_component
(SET1,SET2,Nth_ONE)
previous_component
(SET,COMPONENT)

- - CONDITIONS - -

match
(STRING1,STRING2)
equal
(X,Y)
unequal
(X,Y)
less_than
(X,Y)
less_than_or_equal
(X,Y)
greater_than
(X,Y)
greater_than_or_equal
(X,Y)
next_component_exists
(SET,COMPONENT)
same
(COMPONENT1,COMPONENT2)

- - LOGICAL CONNECTIVES - -

and
(EXPRESSION1,EXPRESSION2)
or
(EXPRESSION1,EXPRESSION2)
not
(EXPRESSION)

- - ASSIGNMENT - -

assign_role
(ROLE,ELEMENT)
or
ROLE = ELEMENT
assign
(ELEMENT,VALUE)
or
ELEMENT := VALUE

Table 1. --- Latest library rules.

Considerable effort was expended to insure that rule construction is easy as possible for potential ICBI authors. Given our initial emphasis on arithmetic, we undertook a structural analysis of the whole number algorithms for addition, subtraction, multiplication and division. In this case, structural analysis involved:

1. selecting prototypic problems, including the goal variables.

2. identifying critical components which can vary while still requiring the same solution method. This effectively defines the domain of the rule.

3. solving those problems step by step as the learner is to solve them (after learning).

4. identifying operations to be performed and the conditions underlying the decisions to be made in carrying out each step.

5. determining whether some succession of operations after a condition ever generates a state, satisfying the same condition. If so, the condition defines an iteration or "loop"; otherwise it defines a selection.

In general, the result of carrying out the above steps is only a partially defined rule procedure. Specifically, it was clear only what happens under conditions (e.g., A) associated with the selected problem. To determine what happens under alternative conditions (e.g., not A), the above steps had to be repeated with new problems.

Once a rule procedure at one level was fully defined, the various components of the procedure are further refined in the same way. The process continues until all components are atomic. (For more details on the process of structural analysis see Scandura, 1982, 1984a, 1984b).

The domains and ranges of the resulting rules are all very similar to the tree

representations used as illustrations in the first section on the problem and rule constructs. The main point to stress in this regard is that essentially any domain (or range) structure can be represented as a hierarchy of ordered sets.

The problems on which the RuleTutor operates are closely related and derived directly from rule domains and ranges. The data FLOWform in Figure 9 shows how data structures (problems) are represented using PRODOC. In particular, notice that some elements (e.g. marked) may have two (or more) parents (i.e., be an element of two sets). From a functional standpoint perhaps the most important feature of PRODOC is that all elements of any such data structure may be directly accessed by operations and conditions in the associated rule.

Component operations and conditions in the procedural portions of procedure FLOWforms correspond to the kinds of atomic rules needed in the proposed arithmetic library.

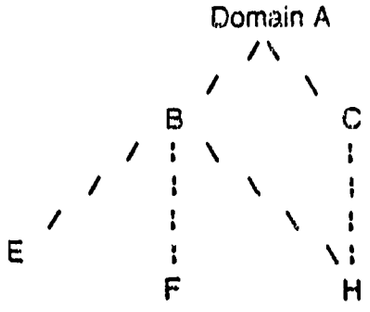
Constructing a library whose constituent atomic rules correspond to these cperations and conditions was primarily a matter of defining atomic rules (corresponding to the arithmetic steps) which both have an easily understood syntax and are meaningful to domain experts.

The "trick" in formulating the component operations and conditions of the various FLOWforms was to do so in a way that maximized generality (i.e., utility) of the components, and hence minimized their number, without losing their heuristic relevance (to the content in question).

It is this PRODOC environment that is used as the "authoring" system. An unique feature of PRODOCea is its ability to immediately execute interpretable library rules. This makes it relatively easy for authors to construct interpretable specifications for cognitive procedural tasks. Once a subject matter expert knows exactly what a human/computer assistant is to do, it is a relatively simple task to develop a rule which captures the competence necessary for performing the required tasks. Example rules constructed for arithmetic are shown in the following section.

Other PRODOC environments are:

(2) Applications Prototyping Environment (for use with a Pascal compiler) (PRODOCIp) - is identical to PRODOCea in so far as prototype design and the use of library rules in rapid prototyping is concerned. Instead of an interpreter, however, PRODOCIp includes a generalized code generator which makes it possible to arbitrarily mix Pascal code with library rules, thereby gaining the prototyping advantages of any number of customized, arbitrarily high level languages, along with the flexibility of Pascal. This feature makes it possible, for example, for a programmer to speed up or otherwise add finishing touches to a working prototype created by a nonprogrammer.



[DOMAIN]:

[A]:

[B]:

[E]:

[F]:

[H]:

[C]:

[H]:

(3) Programming Productivity Environment (PRODOCpp) - has all of the design, etc. features of PRODOCea. PRODOCpp comes in standard form which supports text and source code in any programming language.

In addition, pseudo code support currently is available as an option for Pascal, C, and Ada, combining the clarity and ease of use of high-level fourth generation languages with the flexibility of third generation languages. These options include syntax checking, consistency checking, declarations generation and source code generation. Pseudocode support is totally data driven so similar support for other third and fourth generation languages may be added without modifying PRODOC itself.

The relationship between Pascal pseudo code in the SORT FLOWform and the corresponding full source code is shown below.

Note: This illustration shows only terminal elements of the FLOWform. All design levels of the sort routine are displayed in the second FLOWform.

(4) Library Generation (PRODOClg) - is for "in house" use only and is used to integrate available rule libraries and new library rules into either PRODOC prototyping environment, thereby creating customized versions of PRODOC for particular application areas. Since this requires access to PRODOC source code, customized versions of PRODOC will normally involve collaboration between users and IMS.

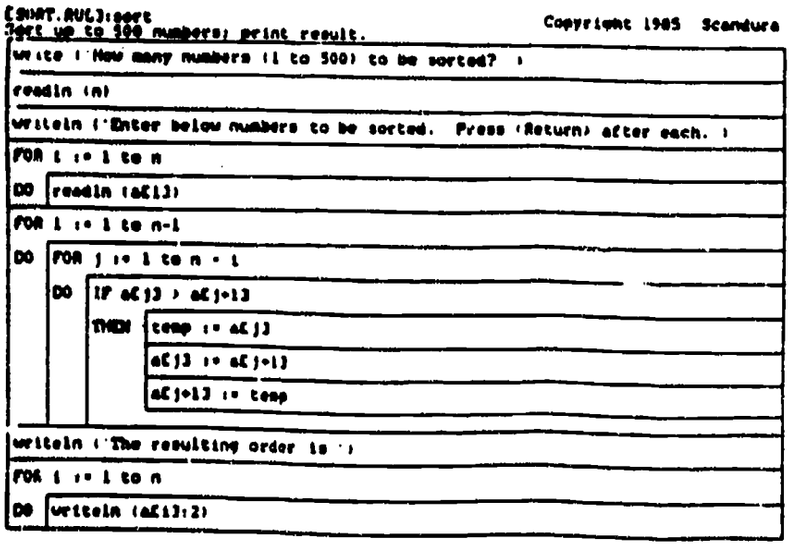
The PRODOC series has been implemented in Pascal and currently runs under MS-DOS.

SAMPLE ARITHMETIC AND LIBRARY RULES

As emphasized above, the RuleTutor is designed to work in conjunction with rules representing content to be taught. Consequently, to construct a working ICBI RuleTutor system, one must first create rule specifications for the content (e.g., some cognitive procedural task).

Our work with potential ICBI authors (e.g., Scandura, 1984a, 1984b) shows that by applying the method of structural analysis systematically, they typically are quite able to construct FLOWforms representing procedures for solving tasks in their areas of expertise -- as long as they can express the components of those procedures in terms with which they are familiar. In a similar manner, they also are able to identify (and hence represent) critical features of the tasks themselves.

Note: These abilities have been demonstrated empirically using a recent formulation of the method of structural analysis (e.g., Scandura, 1984a, 1984b). Given content and pedagogical competence, and guidance in the use of structural analysis, educators




```
PROGRAM sort (input,output);

VAR   n: INTEGER;
      i: INTEGER;
      a: ARRAY [1..500] OF INTEGER;
      j: INTEGER;
      temp: INTEGER;

BEGIN
  write ('How many numbers (1 to 500) to be sorted?  );
  readln (n);
  BEGIN
    writeln ('Enter below numbers to be sorted.  Press (Return) after each.');
    FOR i := 1 to n DO
      readln (a[i])
  END;
  FOR i := 1 to n-1 DO
    FOR j := 1 to n - i DO
      BEGIN
        IF a[j] > a[j+1] THEN
          BEGIN
            temp := a[j];
            a[j] := a[j+1];
            a[j+1] := temp
          END
      END;
  BEGIN
    writeln ('The resulting order is  );
    FOR i := 1 to n DO
      writeln (a[i]:2)
  END
END.
```

(with backgrounds similar to potential ICBI authors) were able to create rule representations functionally equivalent to those constructed by expert analysts.

The FLOWform below *Whole Number Subtraction* is illustrative. This FLOWform is written in English and can be used to solve any given column subtraction problem -- to accomplish this, simply carry out successive steps of the FLOWform in turn.

Although authors can be taught how to perform structural analysis, they do not normally do so -- nor for that matter is it absolutely essential in order to identify rules in their areas of expertise. Potential authors with some programming experience will often prefer to construct flow charts directly without following any particular systematic method of analysis.

The availability of PRODOC greatly facilitates the task of specifying rules in this manner (i.e., directly). Instead of having to draft rule specifications (usually on paper) and converting these to increasingly precise designs, all of this can be done using PRODOC (Scandura, 1987) in an integrated, graphically supported top down structured development environment.

Users are required first to represent rule procedures at a very high level by describing what they do in very general terms. Then, these high level descriptions are refined step-by-step until each component step (of the procedure) is atomic (elementary) -- in the sense that it is either already available to the members of the targeted school population, or is so simple that it would be impossible to teach only part of the step (to members of the population) without their mastering the entire step (i.e., the step is all-or-none as defined by Scandura. 1971, 1973a).

PRODOC makes this possible by allowing the user to view, create, modify and revise graphical representations of rules directly on the IBM PC AT (XT) screen. For this purpose, PRODOC uses a three dimensional variant of the Nassi-Shneiderman representation, called the Scandura FLOWform. The clarity of FLOWforms alone makes it much easier to detect errors, if not avoid them altogether. In addition to being even easier to read, however, FLOWforms make better use of available screen space, and allow the simultaneous representation of as many (or as few) levels of refinement as may be desired.

Not just any FLOWform will do, however. In order for a rule/FLOWform to be usable by the RuleTutor the terminal (atomic) operations and conditions of the rule must be interpretable. Analogous to the above requirements for human atomicity, these atomic operations and conditions must correspond to subroutines in the atomic rule library available to the ICBI RuleTutor.

We have found the library of rules given in the previous section on PRODOC to be more than adequate for arithmetic -- tasks such as column subtraction, addition of fractions, etc. Indeed, one can use these library rules to create a FLOWform (rule)

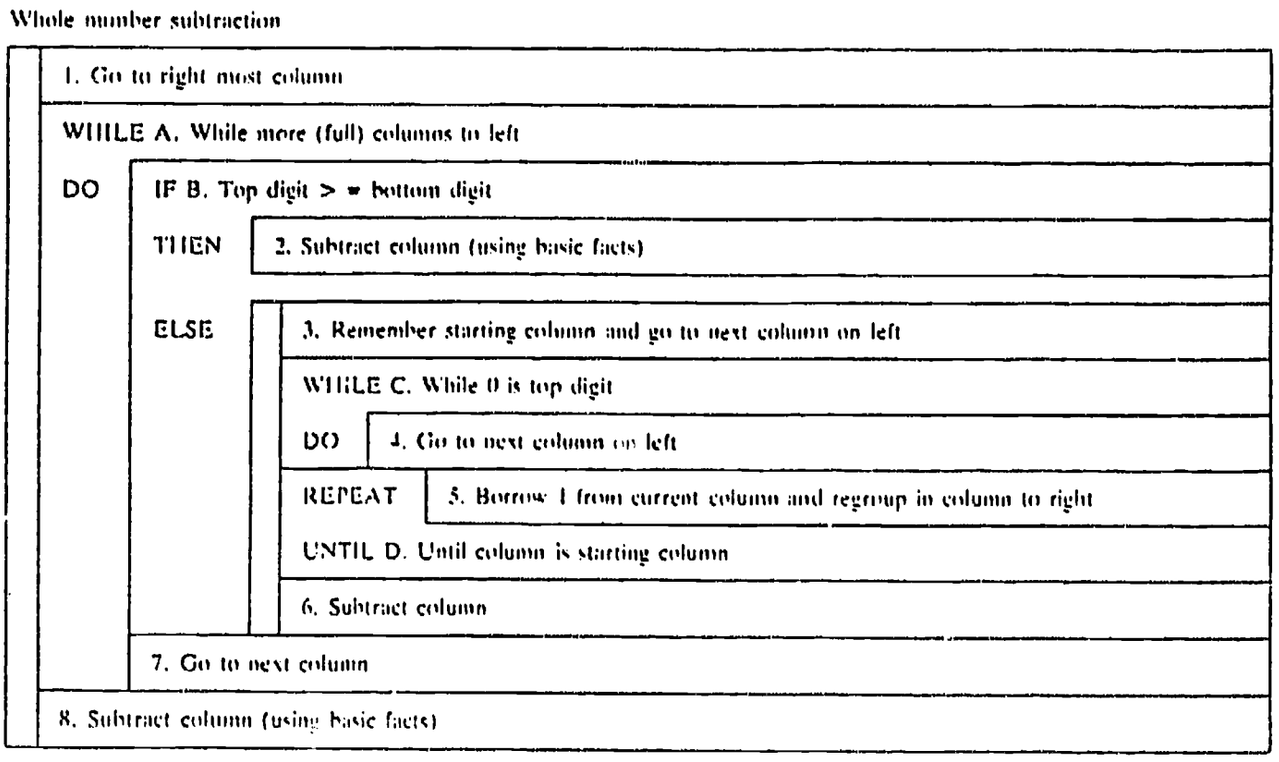


Figure 12. --- Old FLOWform for subtraction.

```
[SUBTRACT.EVL]:subtract numbers by columns                Copyright 1987  Scandura
DOMAIN: MINUEND(), SUBTRAHEND(), COLUMNS(), OPERATION_DESIGNATORS()
RANGE: DIFFERENCE()

DIFFERENCE := whole_number_subtraction (MINUEND, SUBTRAHEND)
```

	display_structure (givens)
	current_column = ones
REPEAT	IF greater_than_or_equal (top_digit, bottom_digit)
	THEN 2. difference_digit := subtract (top_digit, bottom_digit)
	display (difference_digit)
	ELSE 3. borrow_digit = top_digit
	REPEAT 4. borrow_digit = next_component (MINUEND, borrow_digit)
	UNTIL unequal (borrow_digit, '0')
	REPEAT 5. borrow_digit := subtract (borrow_digit, '1')
	display (borrow_digit, 'c7')
	borrow_digit = previous_component (MINUEND, borrow_digit)
	borrow_digit := add (borrow_digit, '10')
	display (borrow_digit, 'm-1,c7')
	borrow_row := subtract (borrow_row, '1')
	UNTIL same (borrow_digit, top_digit)
	6. difference_digit := subtract (top_digit, bottom_digit)
	display (difference_digit)
7.	last_subtracted_column = current_column
	current_column = next_component (COLUMNS, current_column)
UNTIL not (next_column_exists)	

Figure 13A. --- Subtraction represented in terms of PRODOC's library rules.

[SUBTRACT]:subtract_numbers_by_columns;[LIBRARY:procedure];

[DOMAIN]:

[MINUEND]:@L10,c3

@:1

:2

:0

@:4

[SUBTRAHEND]:@L11,c3

@:4

@:3

:3

:1

[COLUMNS]:

[ONES]:@m40

:1

:4

:0

[TENS]:@m37

@:2

@:3

@:0

[HUNDREDS]:@m34

@:0

@:3

@:0

[THOUSANDS]:@m31

@:4

@:1

@:0

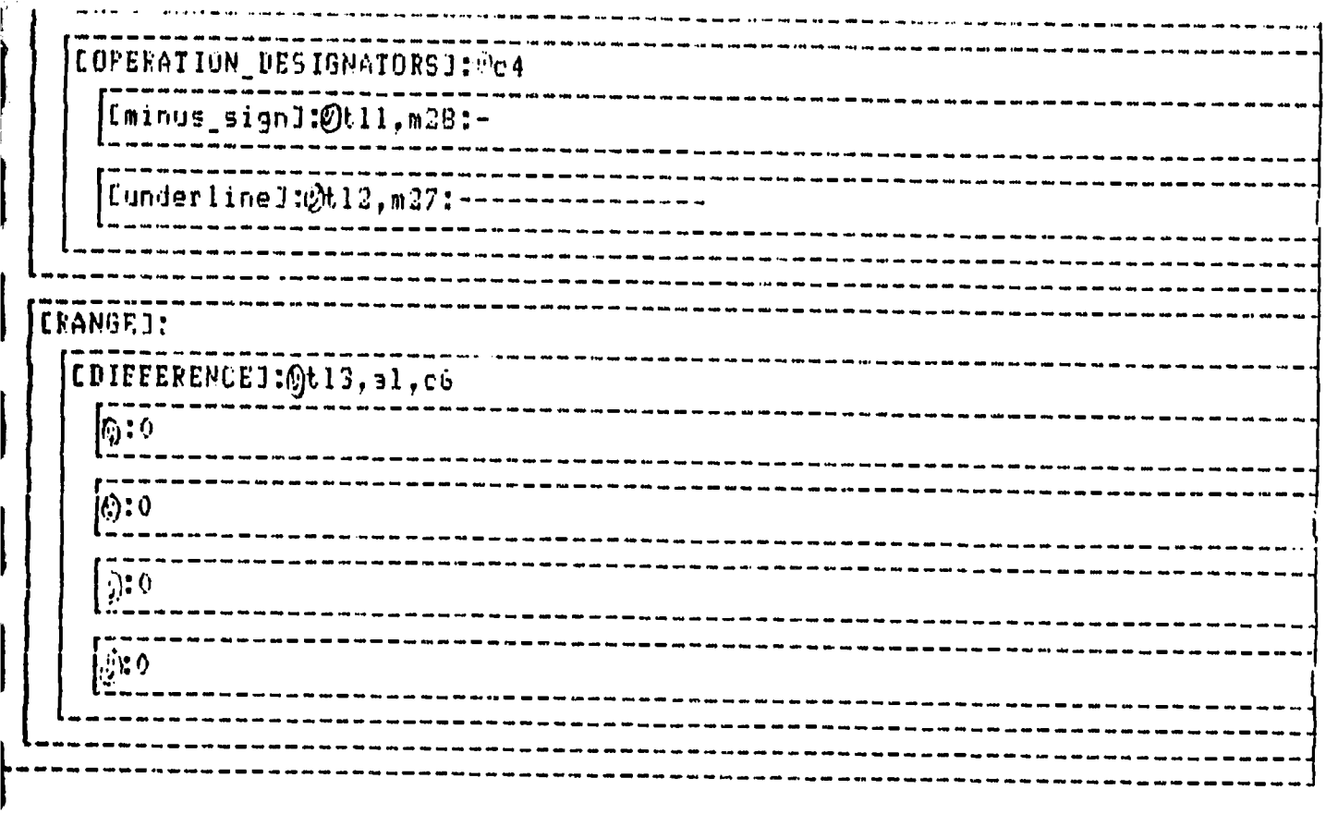


Figure 13B. --- Subtraction data FLOWform.

representing essentially ANY cognitive procedural task. For example, the subtraction FLOWform below is constructed entirely from atomic rules in PRODOC's current library (see Figure 13A).

The numbers to the left of the various structures (higher level steps) of the FLOWform correspond to the numbered conditions and operations in the previously discussed SUBTRACT.OLD FLOWform. The detailed (terminal) steps in the SUBTRACT.EVL FLOWform correspond to library rules.

This rule actually executes under PRODOC[ea]. It simulates the process of subtraction as a child might if he or she were to actually perform the steps. While memory limitations of 640K under DOS preclude the use of fancy graphics, the actual display is in color and quite realistic. (It should be emphasized, however, that the system is set up so that it would be a relatively easy task to "hook" a graphics package into the system when more memory becomes widely available -- e.g., with the OS 2 operating system.)

In addition, PRODOC was used to create all of the needed data structures. The structure of the problem domain and range are shown in Figure 13B.

To maximize ease-of-use, certain rules in PRODOC's library have been modified so that they perform many of the auxiliary operations automatically (and transparently from the perspective of the user). For example, the "display" rule has been enhanced so it can display an entire data structure with one statement. It does this by allowing lower level data elements to "inherit" higher level attributes pertaining to position on the screen, color, intensity, etc. For example, consider the GIVENS data structure in Fig 14. In this case the numerals corresponding to the top and bottom digits will automatically be displayed as specified by the structure as a whole. The parameters t (top margin) and m (left margin) are used to specify location on the screen. Thus the top row might be specified at row (with top margin) 10 and the bottom row at row 11. Rather than having to give the precise coordinates of each digit separately, inheritance makes it possible to specify these margins where they naturally belong -- with the top and bottom rows. The left margin is specified similarly with the various columns (ones, tens, etc.).

In this case routine display operations are largely transparent to the user. Thus, in subtraction the difference digits are automatically displayed in the proper location because they automatically inherit the appropriate coordinates from the DIFFERENCE and COLUMN structures.

The basic question, of course, is how easy it is to construct a given rule from the available library. In general, the basic elements in most programming languages are chosen as they are to allow the programmer as much flexibility as possible while maximizing computational efficiency (i.e., during execution). Typically, such languages are not particularly user friendly, nor do the basic statements in the language, or the

```
[SUBTRACT]:subtract_numbers_by_columns:[]LIBRARY;[procedure]:
  [GIVENS]:
    [MINUEND]:@t10,c3
      <INTEGER_ELEMENT>:@:1
      <INTEGER_ELEMENT>:@:2
      <INTEGER_ELEMENT>:@:0
      <INTEGER_ELEMENT>:@:4
    [SUBTRAHEND]:@t11,c3
      <INTEGER_ELEMENT>:@:4
      <INTEGER_ELEMENT>:@:3
      <INTEGER_ELEMENT>:@:3
      <INTEGER_ELEMENT>:@:1
    [COLUMNS]:
      [ONES]:@m40
        <INTEGER_ELEMENT>:@:1
        <INTEGER_ELEMENT>:@:4
        <INTEGER_ELEMENT>:@:0
      [TENS]:@m37
        <INTEGER_ELEMENT>:@:2
        <INTEGER_ELEMENT>:@:3
        <INTEGER_ELEMENT>:@:0
      [HUNDREDS]:@m34
        <INTEGER_ELEMENT>:@:0
        <INTEGER_ELEMENT>:@:3
        <INTEGER_ELEMENT>:@:0
      [THOUSANDS]:@m31
        <INTEGER_ELEMENT>:@:4
        <INTEGER_ELEMENT>:@:1
        <INTEGER_ELEMENT>:@:3
```

GIVENS

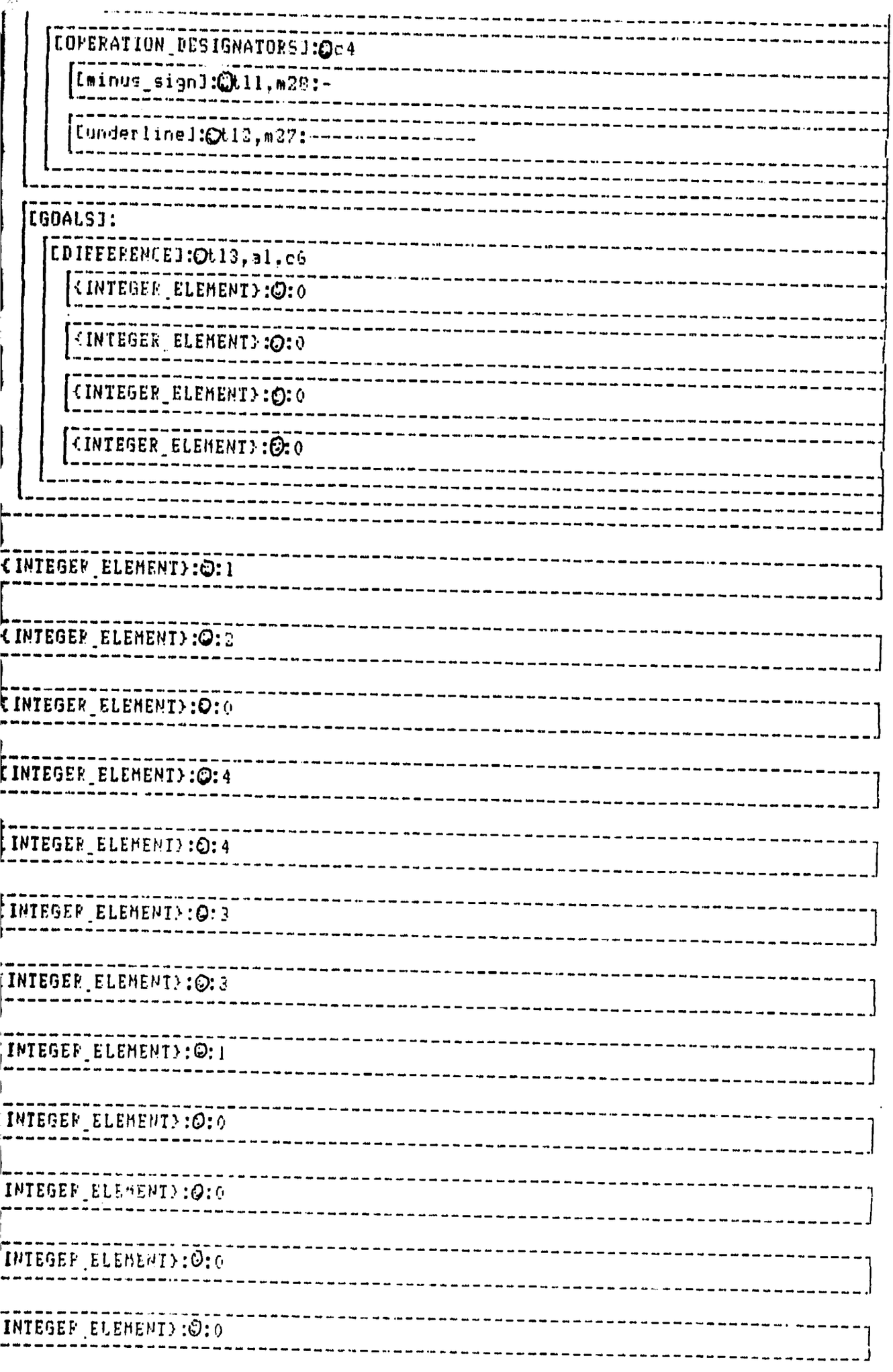


Figure 14. --- Subtraction data FLOWform showing GIVENS and GOALS.

predefined data structures (e.g., real numbers, arrays) have any special relevance to particular applications.

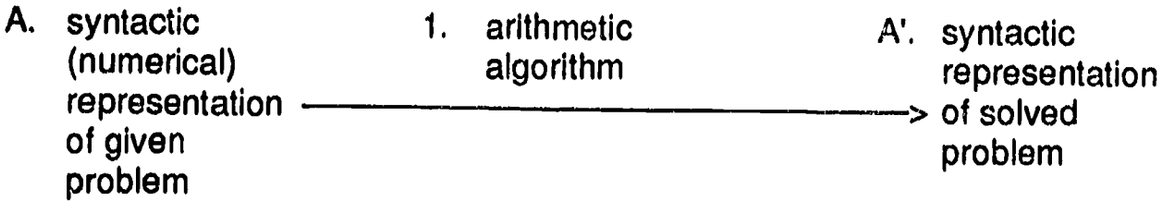
As emphasized above, although the atomic rules in the current PRODOC library are quite easy to use, relatively speaking, educators with no programming experience whatever would probably require some training.

A complete set of FLOWforms for simulating the other arithmetic operations is given as an appendix.

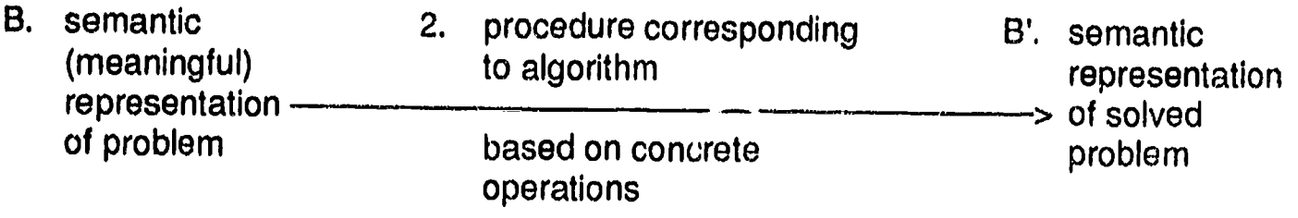
It is important to notice that certain components of some operations (e.g., the division algorithm of Figure 15) correspond to the whole number algorithms. In effect, what are high level rules in one (the whole number) context are atomic ingredients in other (e.g., division) contexts.

In the fullest sense, of course, teaching arithmetic involves much more than simple algorithms: Teaching meaning of the operations and verbal problem solving are just two such goals.

Characterization of arithmetic in terms of rules in this broader sense is beyond the scope of the RuleTutor since it clearly involves sets of lower and higher order rules (e.g., Scandura, 1971, 1973a, 1973b). The general nature of these relationships may be summarized as shown below. See Scandura (1971; 1973a, ch. 5) for a more general discussion of the relationships between syntactic and semantic knowledge in mathematics.



Higher order rules for generating syntactic rules from concrete rules, and vice versa



In this case, problems of the type A would include column subtraction, multiplication of

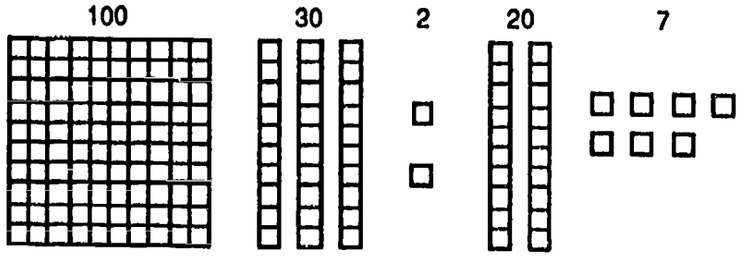
```
[DIVISION]:division                                    Copyright 1987  Scandura
DOMAIN: DIVIDEND{}, DIVISOR{}, OPERATION_DESIGNATORS{}, COLUMNS{}
RANGE: QUOTIENT{}, REMAINDER{}

Divide divisor into dividend.

display_structure (GIVENS)
    TRIAL_DIVISOR = TEN
    display(TRIAL_DIVISOR)
    RUNNER = ONE
    TRIAL_QUOTIENT_POSITION = THOUSANDS
    CURRENT_DIVIDEND := common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND)
    TRIAL_DIVIDEND := common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND)
    display(common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND),' a1:')
    IF greater_than(TRIAL_DIVISOR,TRIAL_DIVIDEND)
    THEN
        TRIAL_DIVIDEND := add( multiply('10',TRIAL_DIVIDEND),
                common_component(previous_component(COLUMNS,
                                        TRIAL_QUOTIENT_POSITION),
        DIVIDEND))
        display(common_component(previous_component(COLUMNS,
                        TRIAL_QUOTIENT_POSITION),DIVIDEND),' a1:')
    WHILE next_component_exists(DIVISOR,RUNNER)
          { No_more_digits }
    DO
        TRIAL_QUOTIENT_POSITION =
        previous_component(COLUMNS,TRIAL_QUOTIENT_POSITION)
        RUNNER = next_component( DIVISOR, RUNNER)
        COL := add(COL,'1')
        CURRENT_DIVIDEND := add( multiply(CURRENT_DIVIDEND,'10'),
              common_component(DIVIDEND,TRIAL_QUOTIENT_POSITION))
    IF less_than(CURRENT_DIVIDEND,DIVISOR)
    THEN
        TRIAL_QUOTIENT_POSITION =
        previous_component(COLUMNS,TRIAL_QUOTIENT_POSITION)
        COL := add(COL,'1')
        CURRENT_DIVIDEND := add( multiply(CURRENT_DIVIDEND,'10'),
              common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND))
REPEAT
    CURRENT_QUOTIENT = common_component(TRIAL_QUOTIENT_POSITION,
                                        QUOTIENT)
    REPEAT
        IF greater_than(CURRENT_QUOTIENT,'0')
        THEN
            display(CURRENT_QUOTIENT,' b0,c7,t-1:')
            CURRENT_QUOTIENT :=
                        subtract(CURRENT_QUOTIENT,'1')
            display(CURRENT_QUOTIENT,' b0,c6,a1:')
        ELSE
            CURRENT_QUOTIENT := greatest_integer(
              divide(TRIAL_DIVIDEND,TRIAL_DIVISOR))
            display(CURRENT_QUOTIENT,' b0,c6,a1:')
```


Figure 15. --- Division showing operations (add, multiply, etc.) which correspond to

fractions, etc. Those of type B might include Dienes blocks, packets of dowels grouped by powers of ten, pie charts (for fractions), etc. Similarly, procedures of type 1 would include the subtraction algorithm, the algorithm for multiplying fractions, etc. and procedures of type 2, concrete manipulations on Dienes blocks, pie charts, etc. The double arrow represents two higher-order rules, one of which can generate concrete rules (e.g., concrete manipulations on Dienes blocks) from the corresponding syntactic rules (e.g., whole number algorithms). The other higher-order rule does the reverse.

Clearly, the proposed RuleTutor would not (simultaneously) accommodate the sets of lower and higher order rules implied by such a broad conception of arithmetic. What it could do, however, is provide diagnostic testing and instruction with respect to any individual (lower or higher order) rule associated with arithmetic.

In the case of subtraction, for example, the meaning of subtraction as taking away and the place value concept in representing numbers are crucial. More particularly, the meaning of a rule (e.g., for subtraction) can be represented in terms of manipulations on concrete objects represented in a standard place value format. Thus, for example, consider the pair of numbers, 132 and 27, represented concretely as



In this case, the student might be shown how to take away the amount represented by the smaller quantity from the larger by taking away from the larger quantity as many groups of each size as there are in the smaller quantity. Where the number of groups of a particular size (e.g., ones) in the larger quantity is smaller than that in the smaller quantity, the student learns to first convert the next larger grouping (e.g., tens) in the larger quantity into a smaller grouping. For example, one ten's group would be converted into ten ones so that the seven ones in the smaller quantity might be taken away. Implicitly, the student also learns to begin work with the smaller place values (groupings) and to work toward larger ones.

Normally, students are not taught general rules (procedures) for performing arithmetic operations on concrete objects in a systematic way. Rather, students gradually acquire an informal awareness of such rules by solving a variety of specific concrete problems, with concrete objects and/or pictorial representations of such objects. Dienes blocks (e.g., Dienes, 1960) are commonly used for this purpose. Nonetheless, manipulative rules CAN be taught explicitly.

Allowing a prospective author to specify manipulation rules (in a form the RuleTutor can use) would require extending the above library by adding (to it) atomic rules corresponding to the above components (e.g., take away, etc.). The "taking away" atomic rule, for example, would have three parameters, one referring to the kind of object (e.g., block, dowel), and the other two to the respective numbers of those objects. Thus,

```
resulting_no_objects :=
  take_away (object_type, original_objects, objects_to_take_away)
```

It is important to emphasize in this regard that new atomic rules identified as a result of analyzing the arithmetic domain can be used to supplement the general purpose library that is currently available. In turn, still additional atomic rules may be added as new domains are analyzed. The only limitations in this regard are computer memory and/or addressing capacity of the operating system.

Indeed, in the course of normal use,the PRODOC library has been used to build a wide variety of prototypes, ranging from arithmetic to creating invoices.

RELATIONSHIPS TO OTHER RESEARCH

Much of the most directly related research and development work has been cited and/or referenced in the body of this proposal. By way of summary, Prof. Joseph M. Scandura and his group at the University of Pennsylvania have been primarily responsible for:

(a) the Structural learning Theory generally, and particularly the theory of diagnostic testing and instruction on which the intelligent RuleTutor is based (e.g., Scandura, 1971, 1977, 1980).

(b) the concept of a rule, including both structural and procedural aspects, which provides the basic theoretical construct on which this work is based (e.g., Scandura, 1970, 1973a, 1973b, 1980).

(c) the method of Structural Analysis (e.g., Scandura, Durnin & Wulfeck, 1974; Scandura & Durnin, 1977; Scandura, 1982, 1984a, 1984b).

(d) the conceptualization and design of IMS's PRODOC programming environment (Scandura, in 1977).

While independently derived, these conceptual formalisms share certain features with other work in artificial intelligence and the cognitive sciences generally. The essential equivalence of structural and procedural representations of knowledge, for example, is

well recognized (e.g., Anderson, 1976). Rule domains (or structure schemas) are similar to "frames" (Minsky, 1975), "schema" theory (Ausubel, 1963; Rumelhart, 1982), etc., although as far as can be determined the particular characteristics of rule domains, range and procedures are original.

The structural-learning-based instructional theory parallels Pask's (1975) Conversation Theory at a general systems level. The instructional theory also shares certain elements in common with other algorithmic formulations, such as that by Landa (1976). It does, however, provide a more rigorous base for computer implementation.

Also, as mentioned previously, the method of structural analysis (e.g., Scandura, 1982, 1984a, 1984b) is potentially compatible with other methods of task analysis (cognitive and otherwise) and automatic programming. Thus, task analysis (proposed by R. B. Miller during the 1950's & Gagne in 1962) can be viewed as a more restricted case of structural analysis where the emphasis is on task requirements rather than cognitive processes. Work by Paul Merrill and that by Lauren Resnick (1984) on cognitive task analysis also shares some features in common with structural analysis. What distinguishes structural analysis is its degree of systematization and rigor, and the fact that it makes provision for higher- as well as lower-order rules, thereby accommodating arbitrarily complex content.

Relationship to research in artificial intelligence and the cognitive sciences

In the cognitive sciences, two ways of using computers to improve learning are generally recognized -- using the computer: (a) as an environment for learning use of "microworlds" and (b) as an intelligent tutor which diagnoses and/or guides student learning (cf. Papert, 1982; Brown & Burton, 1978; Anderson, 1984; Resnick, et al, 1984; Swets, Bruce & Feurzeig, 1984). This research is concerned primarily with the latter. Nonetheless, it should be emphasized that if one were to extract the diagnostic and tutorial aspects of the proposed Curriculum_Tutor (leaving the "idealized" content model and provision for the student interacting with it), one would effectively have a "microworld" (cf. Breuer, 1985). More specifically, the ICBI RuleTutor is an intelligent tutorial system that has various features in common with J. S. Brown's (e.g., Brown & Burton, 1978) systems for identifying procedural "bugs" (i.e., rules which generate incorrect outputs).

Some confusion exists in the literature concerning the question of diagnosis based on the structural learning theory (e.g., Scandura, 1971, 1977) and its relationship to "error patterns" based on the "bug" concept in programming (e.g., Brown & Burton, 1978). In the former case, emphasis has been given to identifying which parts (subrules or subskills) of a to-be-learned rule have and have not been mastered. In the latter case, the emphasis is on the kinds of bugs (e.g., misconceptions) students may have -- even bugs "which have no vestiges in the correct skill (Burton, 1982, p. 177)."

In terms of structural learning theories, "bugs" correspond to what Scandura (e.g., 1977, pp. 75-77) has called "error" rules -- rules which generate incorrect responses but which nonetheless are prototypic of the way certain classes of students behave. Such "bugs" may be characterized either as "perturbations" on some standard (e.g., correct rule) or as *distinct* rules. In the former case, the concept of a prototype "rule procedure" must be generalized to allow nondeterminism (e.g., see Scandura, 1973a, Ch. 8). Individual steps in the prototype may allow for more than one possible response; deterministic procedures will be assigned to individuals (e.g., students) based on diagnosis. This approach parallels that proposed by Brown et al (1984) and has the apparent advantage of simplicity. However, as one of us (Scandura, 1973a, Ch.8) has shown, resorting to nondeterminism "camouflages" the issue of rule selection.

In the latter case recall that a given structural learning theory may involve any (finite) number of alternative prototypes or perspectives (rules or sets of rules) (e.g., Scandura, 1977a., pp.68-76). As demonstrated by Durnin and Scandura (1973), the behavior of an individual student will be more or less compatible with any given prototype -- in this case, the behavior of most American fourth graders was shown to be more compatible with borrowing in column subtraction than with equal additions. That is, they tended to be either consistently successful or unsuccessful on problems associated with various kinds of borrowing. This was NOT true for those students in the case of equal additions.

To be sure, the behavior of some students was more compatible with equal additions, just as others may be more consistent with error, or "buggy" rules. For considerations involved in distinguishing among two or more rule prototypes to see which provides the best account of overall performance, see Scandura (1977, Chapter 10).

Irrespective of the additional information that may be provided when a variety of prototypic rules (including "error" or "buggy" rules) is used in knowledge assessment, explicit verbal attention to such defects may NOT be desirable from an instructional point of view. In particular, calling attention to incorrect skills can lead to later confusion. According to the Structural Learning Theory (e.g., 1971, pp. 41-44; 1973a, Chapter 8) this is because students must choose between or among the two or more rules which may be used in the situation -- the error rule originally learned and the correct one. This ambiguity must be resolved via higher-order selection rules and is a frequent source of difficulty for students. In effect, it is almost always better to learn new skills correctly the first time. The proposed ICBI RuleTutor deals with this problem by combining testing with teaching at the lowest meaningful levels. As soon as a problem is established, remedial instruction is provided immediately, thereby avoiding debilitating misconceptions which otherwise would inevitably surface.

Our research is also paralleled in many ways by the recent work on intelligent tutors by John Anderson (unpublished research proposal) and others (e.g., Larkin, 1983) at Carnegie-Mellon. Both approaches start with a model of the learner (albeit quite

different ones). In Anderson's case, the learner is modeled by his ACT theory (1976). Originally inspired by S-R association principles, this theory currently is based on productions (condition-action pairs). Even today, however, it retains such S-R constructs as "strength," "spreading activation" and "probability."

In the Structural Learning Theory (SLT) the learner is characterized exclusively in terms of lower and higher-order rules, plus universals such as processing capacity and speed and a common control mechanism. Equally fundamental, knowledge in the SLT is treated deterministically (e.g., Scandura, 1971, 1973a, 1977; Hilke, Kempf & Scandura, 1977c). Rather than talking about "productions" being available with some probability, as Anderson does, "rules" in SLT's are either in an "undetermined" state, or available or unavailable.

In the more explicitly instructional aspects of their research, Anderson et al have adopted the structural learning concept of idealized (or prototypic) knowledge (cf. Scandura, 1971, 1973, 1977a,b). As in our research, prototypes are used to characterize what is to be learned. They also have introduced modules incorporating various teaching principles and for coordinating input and output. Each of these has parallels in our research.

Inference based instructional systems such as those developed by A. Collins and his colleagues (e.g., Collins et al., 1975) also deserve mention because of their relevance to the proposed curriculum.Tutor. Inference in structural-learning-based tutors can take the form of higher order rules (to be learned by students) as well as "idealized" inferencing on the part of the computer tutor.

In short, like other intelligent tutors and some CAI systems (e.g., Resnick, 1984; Resnick & Omanson, 1985; Tennyson & Christensen, 1986; Breur, 1985), the ICBI RuleTutor provides for the automatic generation of content. It also allows for future extensions including provision for "bugs", alternative perspectives and logical inference (e.g., Brown et al 1982; Lesh et al,1986; Collins et al, 1975).

Unlike other intelligent tutors, however, there is a sharp distinction in the RuleTutor between the diagnostic/tutorial system, on the one hand, and content (e.g., arithmetic) on the other. The desirability of this type of modularity has never been fully achieved but also has been recognized by others (e.g., Clancey, 1982; Brown, Burton & de Kleer, 1982, p. 280). Like our original RuleTutor, for example, GUIDON (Clancey, 1982) is a multiple-domain tutorial program. However, in neither case is the conceptual distinction between content and instruction fully reflected in modular code. Some, indeed, have voiced the opinion that it may not be possible.

What makes modularity feasible and is unique about the proposed RuleTutor and Curriculum-Tutor systems is an explicit theoretical foundation which has been demonstrated empirically to have the desired modularity and universality (e.g., Scandura 1971, 1973, 1977, 1980). In combination with PRODOC (used as an

authorizing system) this modularity will facilitate future development. In any case, it should take considerably less than the accepted 250 hours or so to produce an hours worth of intelligent tutoring (e.g., Anderson, 1986). Equally important, it may be feasible for the first time for instructional designers (who are computer literate but not skilled programmers) to develop their own intelligent tutoring systems. Only content will have to be dealt with directly since all of the necessary diagnostic and tutorial intelligence will already have been built in.

SUMMARY

In this report we have first describe how the Structural Learning Theory might be used as a basis for creating intelligent tutoring systems. Among other things, we described a new class of ICBI authoring/development systems having two distinct but complementary parts: The first part of each such system is a general purpose, intelligent tutor which is able to perform both diagnostic testing and instruction -but which does not contain content specific knowledge, either of the problem/tasks to be generated or the cognitive procedures (rules) to be taught. The second part consists of IMS's PRODOC software development system. PRODOC provides an easy-to-use medium for specifying arbitrary classes of rules characterizing the desired content area(s). Each such rule, in turn, is interpretable by a general purpose tutor, resulting in a fully operational intelligent tutoring system.

More specifically we have described a general-purpose ICBI RuleTutor which can be used in conjunction with ANY cognitive procedural task formulated as a single rule (i.e. as defined in the Structural Learning Theory - e.g., Scandura 1970,1977,1984. Independently, we also found that PRODOC's rule library might reasonably accommodate essentially any content area. The atomic rules in this library have been shown to provide a natural basis for formulating arbitrary rules (corresponding to to-be-learned cognitive procedures) not only in arithmetic but other areas as well.

REFERENCES:

Anderson, J.R. Language, Memory and Thought. Hillsdale, NJ: Erlbaum Associates, 1976.

Anderson, J.R. *Proposal for the development of intelligent computer-based tutors for the high school mathematics.* (unpublished proposal) Washington, D.C. : NSF, 1984.

Ausubel, D.P. The Psychology of Meaningful Verbal Learning. New York: Grune & Stratton, 1963.

Bork, A. *Advantages of computer-based learning.* Journal of Structural Learning, 1986, 9, No. 1, 63-76.

Breuer, K. *Computer Simulations and Cognitive Development.* In K. Duncan and D. Harris (Eds.) Computers in Education, Amsterdam: (North-Holland)

Brown, J.S. & Burton, R.R. *Diagnostic models for procedural bugs in basic mathematical skills.* Cognitive Science, 1978, 155-192.

Brown, J.S. & Lenat, D.B. Artificial Intelligence, 1984.

Brown, J.S., Burton, R.R. & de Kleer, J. *Pedagogical, natural language and knowledge engineering techniques in SOPHI I, II, and III.* In D. Sleeman & J.S. Brown (Eds.) Intelligent Tutoring Systems. New York: Academic Press, 1982.

Burton, R.R. *Diagnosing bugs.* In D. Sleeman & J.S. Brown (Eds.), Intelligent Tutoring Systems. New York: Academic Press, 1982.

Carbonell, J.R. *Learning by analogy: formulating and generalizing plans from past experience.* In R.S. Michalski, J.R. Carbonell & T.M. Mitchell (Eds.) Machine Learning, an Artificial Intelligence Approach. Palo Alto: Tioga Press, 1983.

Carpenter, T.P. *Conceptual knowledge as a foundation for procedural knowledge: Implications from research on the initial learning of arithmetic.* In J. Heibert (Ed.) Conceptual and Procedural Knowledge: The Case of Mathematics. Hillsdale, N.J.: Lawrence Elbraum Associates, 1986.

Clancey, W.J. *Tutoring rules for guiding a case method dialogue.* In D.Sleeman & J.S. Brown (Eds.) Intelligent Tutoring Systems. New York: Academic Press, 1982.

Collins, A., et al. *Reasoning from incomplete knowledge.* In D.G. Bobrow & A. Collins (Eds.) Representation and Understanding: Studies in Cognitive Science. New York: Academic Press, Inc. 1975, 383-414.

Dale, N. & Lilly, S.C. Pascal Plus Data Structures, Algorithms and Advanced Programming. Lexington, MA: D.C. Heath, 1985.

Davis, R. B. *Conceptual and procedural knowledge in mathematics: A summary analysis.* In J. Heibert (Ed.) Conceptual and Procedural Knowledge: The Case of Mathematics. Hillsdale, N.J. : Lawrence Elbraum Associates, 1986.

Davis, R. & Lenat, D.B. Knowledge-based Systems in Artificial Intelligence. New York: McGraw-Hill International Book Company, 1982.

Dienes, Z.P. Building Up Mathematics. London: Hutchinson, 1960.

Durnin, J.H., & J.M. Scandura, *An algorithmic approach to assessing behavior potential: Comparison with item forms and hierarchical analysis.* Journal of Educational Psychology, 1973, 65, 262-273.

Gagne, R.M. *The acquisition of knowledge.* Psychological Review, 1962, 69, 355-365.

Hiebert, J. (Ed.) Conceptual and Procedural Knowledge: The Case of Mathematics. Hillsdale, N.J. : Lawrence Elbraum Associates, 1986.

Hilke, R., Kempf, W. F., & Scandura, J. M. *Deterministic and probabilistic theorizing in structural learning.* In H. Spada & W. F. Kemp (Eds.), Structural Models of Thinking and Learning. Bern: Huber, 1977.

Hoare, C.A.R. & Shepherdson, J.C. (Eds.) Mathematical Logic and Programming Languages. Englewood Cliffs, NJ: Prentice-Hall International, 1985.

Kieras, D.E. & Polson, P.G. *An approach to the formal analysis of user complexity.* Project on User Complexity of Devices and Systems. Working Paper No. 2. Oct. 1, 1982.

Kinnucan, P. *Computers that think like experts.* New York: High Technology, 1984, Jan. 30-42.

Kleiman, G. & Humphrey, K. *Writing your own software: authoring tools make it easy.* Electronic Learning, 1982, Vol. 1(5), 37-44.

Knuth, D.E. The Art of Computer Programming Vol. 1: Fundamental Algorithms. Reading, Mass.: Addison-Wesley, 1968.

Landa, L.N. Algorithmization in Learning and Instruction. Englewood Cliffs, NJ: Educational Technology, 1976.

Larkin, J.H. *The role of problem representation in physics.* In D. Genter and A.L. Stevens (Eds.) Mental models. Hillsdale, N.J. Lawrence Elbraum Associates, 1983.

Lenat, D.B. *The role of heuristics in learning by discovery: three case studies.* In R.S. Michalski, J.R. Carbonell & T.M. Mitchell (Eds.) Machine Learning, an Artificial Intelligence Approach. Palo Alto: Tioga Press, 1983.

Lesh, R. Paper presentation of Fairwether, P., Lesh, R. and O' Neal, A.F. *What do instructional authoring systems need for automated reasoning* ? American Educational Research Association, San Francisco, April 1986.

Martin, J. System Design from Provably Correct Constructs. The Beginnings of True Software Engineering. Englewood Cliffs, NJ: Prentice-Hall, 1985.

Martin, J. and McClure , C. Diagramming Techniques for Analysts and Programmers. Englewood Cliffs, N.J. : Prentice Hall International, 1985.

Michalski, R.S., Carbonell, J.R. & Mitchell, T.M. (Eds.) Machine Learning, an Artificial Intelligence Approach. Palo Alto: Tioga Press, 1983.

Minsky, M. *A framework for representing knowledge.* In P.H. Winston (Ed.) The Psychology of Computer Vision. New York: McGraw-Hill, 1975.

Papert, S. Mindstorms: Computers, Children and Powerful Ideas. N.Y. : Basic Books, 1980.

Pask, G. Conversation, Cognition and Learning. Amsterdam: Elsevier, 1975.

Quinlan, J.R. *Learning efficiency, classification procedures and applications to chess and games.* In R.S. Michalski, J.R. Carbonell & T.M. Mitchell (Eds.) Machine Learning, an Artificial Intelligence Approach. Palo Alto: Tioga Press, 1983.

Reif, F. and Schoenfeld, A.H. *Principled teaching of scientific and mathematical concepts.* (unpublished proposal) Washington, D.C. : NSF, 1980.

Resnick. L.B. *Beyond error analysis: the role of understanding in elementary school arithmetic.* In H.N. Cheek (Ed.) Diagnostic and prescriptive mathematics: issues, ideas and insights. Kent, Ohio: Research Council for Diagnostic and Prescriptive Mathematics (Research Monograph), 1984.

Resnick, L.B. *Intelligent tutors for elementary and middle school mathematics: A proposal to develop instructional materials based on cognitive research.* (unpublished proposal) Washington, D.C. : NSF, 1984.

Resnick, L.B. and Omanson, S. *Learning to understand arithmetic.* In R. Glaser (Ed.) Advances in Instructional Psychology. Hillsdale, N.J. Lawrence Elbraum Associates, 1985.

Rumelhart, D.E. *Schemata: The building blocks of cognition.* In R. Spiro &W. Brewer. (Eds.) Theoretical Issues in Reading Comprehension. Englewood Cliffs, N.J.: Erlbaum, 1982.

Scandura, J.M. *The role of rules in behavior: Toward an operational definition of what (rule) is learned..* Psychological Review, 1970, 77, 516-533.

Scandura, J.M. *Deterministic theorizing: Three levels of empiricism.* Journal of Structural Learning, 1971, 3, 21-53.

Scandura, J.M. Structural Learning I: Theory and Research. London: Gordon & Breach Sci. Pub., 1973a.

Scandura, J.M. *On higher-order rules.* Educational Psychologist, 1973b, 10, 159-160.

Scandura, J.M. *The role of higher-order rules in problem solving.* Journal of Experimental Psychology, 1974, 120, 984-991.

Scandura, J.M. (with the collaboration of others). Problem Solving: A Structural/Process Approach with Instructional Implications. New York: Academic Press, 1977.

Scandura, J.M. *Structural approach to instructional problems..* American Psychologist, 1977, 32, 33-53.(b)

Scandura, J.M. A *deterministic approach to research in instructional science.* Educational Psychologist, 1977, 12, 118-127.(c)

Scandura, J.M. *Theoretical foundations of instruction: A systems alternative to cognitive psychology.* Journal of Structural Learning, 1980, 6, 347-394.

Scandura, J.M. *Problem solving in schools and beyond: Transitions from the naive to the neophyte to the master.* Educational Psychologist, 1981a, 16,139-150.

Scandura, J.M. *Microcomputer-based system for authoring, diagnosis and instruction in algorithmic content.* Educational Technology, 1981b, 13-19.

Scandura, J.M. *Structural (cognitive task) analysis: A method for analyzing content; Part I: Background and empirical research.* Journal of Structural Learning, 1982, 7, 101-114.

Scandura, J.M. *Structural (cognitive task) analysis: A method for analyzing content; Part II: Toward precision, objectivity and systematization.* Journal of Structural Learning, 1984a, 8, 1-28.

Scandura, J.M. *Structural (cognitive task) analysis: A method for analyzing content; Part III: Validity and reliability.* Journal of Structural Learning,1984b, 8, 173-193.

Scandura, J.M. *Theory-driven expert systems.* Educational Technology, 1984c, 24 (11) 47-48.

Scandura, J.M. *System issues in problem solving research.* Journal of Structural Learning, 1985, 9, 49-62.

Scandura, J.M. A *cognitive approach to software development: the PRODOC system and associated methodology.* Journal of Pascal, Ada and Modula 2. 1987, in press.

Scandura, J.M. & Brainerd, C.J. (Eds.) Structural/Process Models of Complex Human Behavior. Alphen ann den Ryn, The Netherlands: Sijthoff & Noordhoff, 1978.

Scandura, J.M. & J.H. Durnin. *Algorithmic analysis of algebraic proofs.* In J.M. Scandura, Problem Solving: A Structural/Process Approach with Instructional Implications. New York: Academic Press, 1977.

Scandura, J.M., Durnin, J.H. et al. An Algorithmic Approach to Mathematics: Concrete Behavioral Foundations. New York: Harper & Row, 1971.

Scandura, J.M., Durnin, J.H. & Wulfeck, W.H. II. *Higher-order rule characterization of heuristics for compass and straight-edge construction in geometry.* Artificial Intelligence, 1974, 5, 149-183.

Scandura, J.M. & Durnin, J.H. *Algorithmic analysis of algebraic proofs.* In J.M. Scandura, Problem Solving: a structural process approach with instructional implications. New York: Academic Press, 1977.

Scandura, J.M. & Scandura, A.B. Structural Learning and Concrete Operations: An Approach to Piagetian Conservation. New York: Praeger Sci. Pub., 1980.

Scandura, J.M., Stone, D.C. & Scandura, A.B. *The "RuleTutor": an intelligent CBI system for diagnostic testing and instruction.* Journal of Structural Learning, 1986, in press.

Swets, J.A., Bruce, B. and Feurzeig, W. *Cognition, computers and curricula: Software tools for curriculum design* . (unpublished proposal) Washington, D.C. : NSF, 1984.

Tennyson, R.D. and D.L. Christensen, *Artificial intelligence in computer-based instruction.* Paper presented at American Educational Research Association, San Francisco, April 1986.

Thorndyke, P.W. & Wescourt, K.T. *Representation and diagnosis of skills for training time-stressed planning.* Paper presented at American Educational Research Association, New Orleans, April, 1984.

VanLehn, K. *Arithmetic procedures are induced from examples.* In J. Hiebert (Ed.) Conceptual and Procedural Knowledge : The Case for Mathematics. Hillsdale, N.J. : Lawrence Elbraum Associates, 1986.

Wulfeck, W.H. An Algorithmic Approach to Curriculum Development, Computer Implementation, and Evaluation of a Method for Identifying Instructional Content Sequences. Doctoral Dissertation, Philadelphia, PA: University of Pennsylvania, 1975.

Wulfeck, W.H. & Scandura, J.M. *Theory of adaptive instruction with application to sequencing in teaching problem solving.* In J.M. Scandura et al, Problem Solving: A Structural/process Approach with Instructional Implications. New York: Academic Press, 1977.

APPENDIX

FLOWform for simulating addition.

01-31-89

```
[ADDITION.EVL]:add_numbers_by_columns;[]LIBRARY;[procedure];
  display_structure (givens)
  current_column = ONES
WHILE next_column_exists
DO  current_addend = next_component (ADDENDS)
    REPEAT  IF not (same (next_digit, nill))
            THEN current_number := add (current_number, next_digit)
            last_added_addend = current_addend
            current_addend = next_component (ADDENDS, current_addend)
    UNTIL not (next_addend_exists)
    current_column = next_component (COLUMNS, current_column)
    IF greater_than_or_equal (current_number, '10')
    THEN  sum_digit := modulo (current_number, '10')
          display (sum_digit, ' t16, mVresult_col,c6,a1')
          carry := divide (current_number, '10')
          carry := round (subtract (carry, '0.5'), '0')
          IF next_column_exists
          THEN display (carry, ' tVcarry_row,mVcarry_col')
    ELSE  sum_digit := current_number
          display (sum_digit, ' t16, mVresult_col,c6,a1')
          carry := '0'
    carry_col := subtract (carry_col, '3')
    result_col := subtract (result_col, '3')
    current_number := carry
  current_addend = next_component (ADDENDS)
  REPEAT  IF not (same (next_digit, nill))
          THEN current_number := add (current_number, next_digit)
          last_added_addend = current_addend
          current_addend = next_component (ADDENDS, current_addend)
  UNTIL not (next_addend_exists)
  current_column = next_component (COLUMNS, current_column)
  IF greater_than_or_equal (current_number, '10')
  THEN  sum_digit := modulo (current_number, '10')
        display (sum_digit, ' t16, mVresult_col,c6,a1')
        carry := divide (current_number, '10')
        carry := round (subtract (carry, '0.5'), '0')
```

```
        THEN | display (carry, ' tVcarry_row,mVcarry_col')

    ELSE | sum_digit := current_number
         | display (sum_digit, ' t16, mVresult_col,c6,a1')
         | carry := '0'

  carry_col := subtract (carry_col, '3')
  result_col := subtract (result_col, '3')
  current_number := carry
IF unequal (carry, '0')
THEN | sum_digit := carry
     | display (sum_digit,' t16, mVresult_col,c6,a1')
```

```
[DOMAIN]:
  [ADDENDS]: c3
    [ADDEND1]: t10
      {INTEGER_ELEMENT}: :8
      {INTEGER_ELEMENT}: :7
      {INTEGER_ELEMENT}: :6
      {INTEGER_ELEMENT}: :5
    [ADDEND2]: t11
      {INTEGER_ELEMENT}: :5
      {INTEGER_ELEMENT}: :6
      {INTEGER_ELEMENT}: :7
      {INTEGER_ELEMENT}: :8
    [ADDEND3]: t12
      {INTEGER_ELEMENT}: :3
      {INTEGER_ELEMENT}: :2
    [ADDEND4]: t13
      {INTEGER_ELEMENT}: :9
    [ADDEND5]: t14
      {INTEGER_ELEMENT}: :2
      {INTEGER_ELEMENT}: :0
      {INTEGER_ELEMENT}: :9
```

[COLUMNS]:

[ONES]: m40

{INTEGER_ELEMENT}: :8

{INTEGER_ELEMENT}: :5

{INTEGER_ELEMENT}: :3

{INTEGER_ELEMENT}: :9

{INTEGER_ELEMENT}: :2

{INTEGER_ELEMENT}: :0

[TENS]: m37

{INTEGER_ELEMENT}: :7

{INTEGER_ELEMENT}: :6

{INTEGER_ELEMENT}: :2

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[HUNDREDS]: m34

{INTEGER_ELEMENT}: :6

{INTEGER_ELEMENT}: :7

{INTEGER_ELEMENT}: :9

{INTEGER_ELEMENT}: :0

[THOUSANDS]: m31

{INTEGER_ELEMENT}: :5

{INTEGER_ELEMENT}: :8

{INTEGER_ELEMENT}: :0

[TEN_THOUSAND]: m28

{INTEGER_ELEMENT}: :0

[OPERATION_DESIGNATORS]: c4

[plus_sign]: t14,m25:+

[underline]: t15,m24:----------

[E]:

```
    [INTEGER_ELEMENT]: :0
    [INTEGER_ELEMENT]: :0
    [INTEGER_ELEMENT]: :0
    [INTEGER_ELEMENT]: :0
    [INTEGER_ELEMENT]: :0
[intermediate]:
    [next_addend_exists]next_component_exists (ADDENDS, last_added_addend):
    [next_column_exists]next_component_exists (COLUMNS, current_column):
    [carry_row][CHAR_ELEMENT]:'9'
    [carry_col][CHAR_ELEMENT]:'37'
    [result_col][CHAR_ELEMENT]:'40'
    [current_addend][CHAR_ELEMENT]:
    [carry][CHAR_ELEMENT]:
    [current_number][CHAR_ELEMENT]:
    [last_added_addend][CHAR_ELEMENT]:
    [next_digit]common_component (current_addend, current_column):
    [sum_digit] common_component (SUM, current_column): c12

INTEGER_ELEMENT]: :8
INTEGER_ELEMENT]: :7
INTEGER_ELEMENT]: :6
INTEGER_ELEMENT]: :5
INTEGER_ELEMENT]: :5
INTEGER_ELEMENT]: :6
INTEGER_ELEMENT]: :7
INTEGER_ELEMENT]: :8
```

[INTEGER_ELEMENT]: :2

[INTEGER_ELEMENT]: :9

[INTEGER_ELEMENT]: :2

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :9

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

FLOWform for simulating subtraction.

01-31-89

```
[SUBTRACT.NVL]:subtract_numbers_by_columns;[]LIBRARY;[procedure];
display_structure (givens)
current_column = ones
WHILE next_column_exists
DO  IF greater_than_or_equal (top_digit, bottom_digit)
    THEN  difference_digit := subtract (top_digit, bottom_digit)
          display (difference_digit)
    ELSE  borrow_digit = top_digit
          borrow_digit = next_component (MINUEND, borrow_digit)
          WHILE equal (borrow_digit, '0')
          DO  borrow_digit = next_component (MINUEND, borrow_digit)
          REPEAT  borrow_digit := subtract (borrow_digit, '1')
                  {Borrow from current column}
                  display (borrow_digit, 'c7')
                  borrow_digit = previous_component (MINUEND,
                                                     borrow_digit)
                  borrow_digit := add (borrow_digit, '10')
                  {Compute new value for column}
                  display (borrow_digit, 'm-1,c7')
                  borrow_row := subtract (borrow_row, '1')
          UNTIL same (borrow_digit, top_digit)
                difference_digit := subtract (top_digit, bottom_digit)
                display (difference_digit)
    current_column = next_component (COLUMNS, current_column)
difference_digit := subtract (top_digit, bottom_digit)
display (difference_digit)

[DOMAIN]:
  [MINUEND]: t10,c3
    [INTEGER_ELEMENT]: :1
    [INTEGER_ELEMENT]: :2
    [INTEGER_ELEMENT]: :0
    [INTEGER_ELEMENT]: :4
  [SUBTRAHEND]: t11,c3
    [INTEGER_ELEMENT]: :4
    [INTEGER_ELEMENT]: :3
```

[INTEGER_ELEMENT]: :1

[COLUMNS]:

[ONES]: m40

[INTEGER_ELEMENT]: :1

[INTEGER_ELEMENT]: :4

[INTEGER_ELEMENT]: :0

[TENS]: m37

[INTEGER_ELEMENT]: :2

[INTEGER_ELEMENT]: :3

[INTEGER_ELEMENT]: :0

[HUNDREDS]: m34

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :3

[INTEGER_ELEMENT]: :0

[THOUSANDS]: m31

[INTEGER_ELEMENT]: :4

[INTEGER_ELEMENT]: :1

[INTEGER_ELEMENT]: :0

[OPERATION_DESIGNATORS]: c4

[minus_sign]: t11,m28:-

[underline]: t12,m27:----------

[RANGE]:

[DIFFERENCE]: t13,a1,c6

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[Intermediate]:

```
        [borrow_row]{CHAR_ELEMENT}:-1
          [borrow_digit]:
    [next_column_exists]next_component_exists (COLUMNS, current_column):
    [current_column]{CHAR_ELEMENT}:
    [top_digit]=common_component (MINUEND, current_column):
    [bottom_digit]=common_component (SUBTRAHEND, current_column):
    [difference_digit]=common_component (DIFFERENCE, current_column):

INTEGER_ELEMENT}: :1

INTEGER_ELEMENT}: :2

INTEGER_ELEMENT}: :0

INTEGER_ELEMENT}: :4

INTEGER_ELEMENT}: :4

INTEGER_ELEMENT}: :3

INTEGER_ELEMENT}: :3

INTEGER_ELEMENT}: :1

INTEGER_ELEMENT}: :0

INTEGER_ELEMENT}: :0

INTEGER_ELEMENT}: :0

INTEGER_ELEMENT}: :0
```

FLOWform for simulating multiplication.

01-31-89

```
[MULTIPLI]:multiply_numbers_by_columns;[]LIBRARY;[procedure];
    display_structure (MULTIPLICAND)
    display_structure (MULTIPLIER)
    display (time_sign)
    display (problem_underline)
    current_partial_product = PARTIAL_PRODUCT1
    bottom_digit = next_component (MULTIPLIER)
REPEAT
    carry := '0'
    partial_product_digit = next_component (current_partial_produc
    top_digit = next_component (MULTIPLICAND)
    REPEAT
        partial_product_digit := multiply (top_digit,
        bottom_digit)
        partial_product_digit := add (partial_product_digit,
        carry)
        carry := divide (partial_product_digit, '10')
        carry := round (subtract (carry, '0.5'), '0')
        partial_product_digit := subtract
        (partial_product_digit, multiply (carry, '10'))
        display (partial_product_digit)
        partial_product_digit = next_component
        (current_partial_product, partial_product_digit)
        last_top_digit = top_digit
        top_digit = next_component (MULTIPLICAND, top_digit)
    UNTIL NOT (next_component_exists (MULTIPLICAND, last_top_digit))
    IF greater_than (carry, '0')
    THEN
        partial_product_digit := carry
        display (partial_product_digit)
    current_partial_product = next_component (PARTIAL_PRODUCTS,
    current_partial_product)
    last_bottom_digit = bottom_digit
    bottom_digit = next_component (MULTIPLIER, bottom_digit)
UNTIL not (next_component_exists (MULTIPLIER, last_bottom_digit))
display (partial_product_underline)
    current_partial_product = PARTIAL_PRODUCT1
    product_result := '0'
    factor2 := '1'
    REPEAT
        partial_product_digit = next_component
        (current_partial_product)
        factor1 := '1'
        current_partial_product := '0'
        REPEAT
            partial_product_digit := multiply
```

```
                    current_partial_product := add
                    (current_partial_product, partial_product_digit)
                    factor1 := multiply (factor1, '10')
                    last_partial_product_digit = partial_product_digit
                    partial_product_digit = next_component
                    (current_partial_product, partial_product_digit)
                UNTIL not (next_component_exists (current_partial_product,
                      last_partial_product_digit))
                current_partial_product := multiply (current_partial_product,
                factor2)
                factor2 := multiply (factor2, '10')
                product_result := add (product_result,
                current_partial_product)
                last_partial_product = current_partial_product
                current_partial_product = next_component (PARTIAL_PRODUCTS,
                current_partial_product)
        UNTIL not (next_component_exists (PARTIAL_PRODUCTS,
              last_partial_product))
    product_digit = next_component (PRODUCT)
    WHILE greater_than (product_result, '0')
    DO  product_value := product_result
        product_result := divide (product_result, '10')
        product_result := round (subtract (product_result, '0.5'), '0')
        product_digit := subtract (product_value, multiply (product_result,
        '10'))
        display (product_digit)
        product_digit = next_component (PRODUCT, product_digit)
```

```
[DOMAIN]:
  [MULTIPLICAND]: t9,c3
    {INTEGER_ELEMENT}: :9
    {INTEGER_ELEMENT}: :2
    {INTEGER_ELEMENT}: :3
  [MULTIPLIER]: t10,c3
    {INTEGER_ELEMENT}: :9
    {INTEGER_ELEMENT}: :7
    {INTEGER_ELEMENT}: :8
  [COLUMNS]:
    [ONES]: m40
      {INTEGER_ELEMENT}: :9
      {INTEGER_ELEMENT}: :9
      {INTEGER_ELEMENT}: :0
```

[TENS]: m37

{INTEGER_ELEMENT}: :2

{INTEGER_ELEMENT}: :7

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[HUNDREDS]: m34

{INTEGER_ELEMENT}: :3

{INTEGER_ELEMENT}: :8

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[THOUSANDS]: m31

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[TEN_THOUSAND]: m28

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[HUNDRED_THOUSAND]: m25

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[OPERATION_DESIGNATORS]: c4

[time_sign]: t10,m28:*

[problem_underline]: t11,m28:----------

[partial product underline]: t15,m22:----------

[RANGE]:

[PARTIAL_PRODUCTS]:

[PARTIAL_PRODUCT1]: t12

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[PARTIAL_PRODUCT2]: t13

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[PARTIAL_PRODUCT3]: t14

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[PRODUCT]: t16,a1

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

{INTEGER_ELEMENT}: :0

[intermediate]:

[current_column]:

[last_bottom_digit]:

[top_digit]:

[bottom_digit]:

[product_digit]: c6

[partial_product_digit]:

[carry]:

[current_partial_product]:

[next_digit]:

[current_number]:

[product_result]:

[factor1]:

[factor2]:

[last_partial_product]:

[product_value]:

[last_partial_product_digit]:

[GLOBAL]:

[nill]:

[display_structure]:[INCLUDED FILE]

INTEGER_ELEMENT]: :9

INTEGER_ELEMENT]: :2

INTEGER_ELEMENT]: :3

INTEGER_ELEMENT]: :9

INTEGER_ELEMENT]: :7

INTEGER_ELEMENT]: :8

INTEGER_ELEMENT]: :0

[1 TEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

[INTEGER_ELEMENT]: :0

FLOWform for simulating division.

```
[DIVISION]:division;[]library;[procedure];                                01-31-89
  display_structure (GIVENS)
    TRIAL_DIVISOR = TEN
    display(TRIAL_DIVISOR)
    RUNNER = ONE
    TRIAL_QUOTIENT_POSITION = THOUSANDS
    CURRENT_DIVIDEND := common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND)
    TRIAL_DIVIDEND := common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND)
    display(common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND),' a1:')
    IF greater_than(TRIAL_DIVISOR,TRIAL_DIVIDEND)
    THEN
      TRIAL_DIVIDEND := add( multiply('10',TRIAL_DIVIDEND),
            common_component(previous_component(COLUMNS,
                                    TRIAL_QUOTIENT_POSITION),
      DIVIDEND))
      display(common_component(previous_component(COLUMNS,
                    TRIAL_QUOTIENT_POSITION),DIVIDEND),' a1:')
    WHILE next_component_exists(DIVISOR,RUNNER)
          { No_more_digits }
    DO
      TRIAL_QUOTIENT_POSITION =
      previous_component(COLUMNS,TRIAL_QUOTIENT_POSITION)
      RUNNER = next_component( DIVISOR, RUNNER)
      COL := add(COL,'1')
      CURRENT_DIVIDEND := add( multiply(CURRENT_DIVIDEND,'10'),
            common_component(DIVIDEND,TRIAL_QUOTIENT_POSITION))
    IF less_than(CURRENT_DIVIDEND,DIVISOR)
    THEN
      TRIAL_QUOTIENT_POSITION =
      previous_component(COLUMNS,TRIAL_QUOTIENT_POSITION)
      COL := add(COL,'1')
      CURRENT_DIVIDEND := add( multiply(CURRENT_DIVIDEND,'10'),
            common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND))
  REPEAT
    CURRENT_QUOTIENT = common_component(TRIAL_QUOTIENT_POSITION,
                                        QUOTIENT)
    REPEAT
      IF greater_than(CURRENT_QUOTIENT,'0')
      THEN
        display(CURRENT_QUOTIENT,' b0,c7,t-1:')
        CURRENT_QUOTIENT :=
                    subtract(CURRENT_QUOTIENT,'1')
        display(CURRENT_QUOTIENT,' b0,c6,a1:')
      ELSE
        CURRENT_QUOTIENT := greatest_integer(
            divide(TRIAL_DIVIDEND,TRIAL_DIVISOR))
        display(CURRENT_QUOTIENT,' b0,c6,a1:')
      PRODUCT := multiply(DIVISOR,CURRENT_QUOTIENT)
      IF greater_than(PRODUCT,'9')
```

```
                    display(PRODUCT,' b0,c7,tvROW,mvCOL:
                    ')
                    COL := add(COL,'2')
                ELSE
                    COL := subtract(COL,'1')
                    display(PRODUCT,' b0,c7,tvROW,mvCOL:
                    ')
                    COL := add(COL,'1')
            ELSE display(PRODUCT,' b0,c7,tvROW,mvCOL:')
UNTIL greater_than_or_equal(CURRENT_DIVIDEND ,PRODUCT)
      { current_quotient_not_too_large }
    DIFFERENCE := subtract(CURRENT_DIVIDEND,PRODUCT)
    ROW := add(ROW,'1')
        COL := subtract(COL,'2')
        display(' dt,tvROW,mvCOL:---')
        COL := add(COL,'2')
    ROW := add(ROW,'1')
    IF greater_than(DIFFERENCE,'9')
    THEN IF greater_than(DIFFERENCE,'99')
         THEN
             COL := subtract(COL,'2')
             display(DIFFERENCE,' b0,c7,tvROW,mvCOL:')
             COL := add(COL,'2')
         ELSE
             COL := subtract(COL,'1')
             display(DIFFERENCE,' b0,c7,tvROW,mvCOL:')
             COL := add(COL,'1')
    ELSE display(DIFFERENCE,' b0,c7,tvROW,mvCOL:')
CURRENT_DIVIDEND := DIFFERENCE
IF not (same (TRIAL_QUOTIENT_POSITION,ONES))
THEN
    TRIAL_QUOTIENT_POSITION =
    previous_component(COLUMNS,TRIAL_QUOTIENT_POSITION)
    COL := add(COL,'1')
    CURRENT_DIVIDEND := add(
    multiply(CURRENT_DIVIDEND,'10'),
    common_component(TRIAL_QUOTIENT_POSITION,DIVIDEND))
    IF greater_than(CURRENT_DIVIDEND,'9')
    THEN IF greater_than(CURRENT_DIVIDEND,'99')
         THEN
             COL := subtract(COL,'2')
             display(CURRENT_DIVIDEND,' b0,c7,tvROW,m
             vCOL:')
             COL := add(COL,'2')
         ELSE
             COL := subtract(COL,'1')
             display(CURRENT_DIVIDEND,' b0,c7,tvROW,m
             vCOL:')
```

```
                ELSE    display(CURRENT_DIVIDEND,' b0,c7,tvROW,mvCOL:')

WHILE and (not (same (TRIAL_QUOTIENT_POSITION,ONES)),
           greater_than(DIVISOR,CURRENT_DIVIDEND))
DO      CURRENT_QUOTIENT =
        common_component(TRIAL_QUOTIENT_POSITION,
                                        QUOTIENT)
        display(CURRENT_QUOTIENT,' c6,a1:')
        IF not (same (TRIAL_QUOTIENT_POSITION,ONES))
        THEN    TRIAL_QUOTIENT_POSITION =
                previous_component(COLUMNS,TRIAL_QUOTIENT_POSITI
                ON)
                COL := add(COL,'1')
                CURRENT_DIVIDEND := add(
                multiply(CURRENT_DIVIDEND,'10'),
                common_component(TRIAL_QUOTIENT_POSITION,DIVIDEN
                D))
                IF greater_than(CURRENT_DIVIDEND,'9')
                THEN    IF greater_than(CURRENT_DIVIDEND,'99')
                        THEN    COL := subtract(COL,'2')
                                display(CURRENT_DIVIDEND,' b0,c7,t
                                vROW,mvCOL:')
                                COL := add(COL,'2')
                        ELSE    COL := subtract(COL,'1')
                                display(CURRENT_DIVIDEND,' b0,c7,t
                                vROW,mvCOL:')
                                COL := add(COL,'1')
                ELSE    display(CURRENT_DIVIDEND,' b0,c7,tvROW,mvC
                        OL:')
        COL := add(COL,'1')
    TRIAL_DIVIDEND := '0'
    TRIAL_DIVIDEND := extract(CURRENT_DIVIDEND,'1','1')
    IF greater_than(CURRENT_DIVIDEND,'9')
    THEN    IF greater_than(CURRENT_DIVIDEND,'99')
            THEN    COL := subtract(COL,'2')
                    display(TRIAL_DIVIDEND,' b0,c7,a1,tvROW,mvCOL:
                    ')
                    COL := add(COL,'2')
            ELSE    COL := subtract(COL,'1')
                    display(TRIAL_DIVIDEND,' b0,c7,a1,tvROW,mvCOL:
                    ')
                    COL := add(COL,'1')
    ELSE    display(TRIAL_DIVIDEND,' b0,a1,c7,tvROW,mvCOL:')
```

```
                THEN    temp := extract(CURRENT_DIVIDEND,'2','1')
                        insert(TEMP,TRIAL_DIVIDEND,'2')
                        IF greater_than(CURRENT_DIVIDEND,'9')
                        THEN    IF greater_than(CURRENT_DIVIDEND,'99')
                                THEN    COL := subtract(COL,'2')
                                        display(TRIAL_DIVIDEND,' b0,a1,c7,tvROW,mvCOL:')
                                        COL := add(COL,'2')
                                ELSE    COL := subtract(COL,'1')
                                        display(TRIAL_DIVIDEND,' b0,a1,c7,tvROW,mvCOL:')
                                        COL := add(COL,'1')
                        ELSE    display(TRIAL_DIVIDEND,' b0,a1,c7,tvROW,mvCOL:')
                ROW := add(ROW,'1')
        UNTIL   greater_than(DIVISOR,CURRENT_DIVIDEND) {
                divisor_is_greater_than_current_dividend }
            display(CURRENT_QUOTIENT,' b0,a1,c6:')
            REMAINDER := DIFFERENCE
            display(' t3,m36,c4:R')
            display(REMAINDER,' m38,c6,a1:')
[DOMAIN]:
  [DIVIDEND]: c3,t5:1895
    :5
    :9
    :8
    :1
  [DIVISOR]: t5,c3:53
    [ONE]: m26:3
    [TEN]: m25:5
  [OPERATION_DESIGNATORS]: c4
    [DIVISION_BAR]: t4,m28:_____
    [DIV_CURLY]: t5,m28:)
  [COLUMNS]:
    [ONES]: m33:
      :5
```

[TENS]: m32:

:9

: c0:0

[HUNDREDS]: m31:

:8

: c0:0

[THOUSANDS]: m30:

:1

: c0:0

[RANGE]: t3:

[QUOTIENT]:

: c0:0

: c0:0

: c0:0

: c0:0

[REMAINDER]:

[intermediate]:

[TRIAL_DIVISOR]: c1,a1:

[CURRENT_DIVIDEND]:

[TRIAL_DIVIDEND]:

[DIFFERENCE]:

[PRODUCT]:

[CURRENT_QUOTIENT]:

[TRIAL_QUOTIENT_POSITION]:

[RUNNER]:

[TEMP]:

[ROW]:6

[COL]:30

5

9

8

1

c0:0

c0:0

c0:0

c0:0