INTRODUCTION

Most contemporary computer-based instruction (CBI) authoring systems are of the "fixed content" variety; that is, they require authors 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.

This chapter is concerned with the application of recent advances in the cognitive sciences, instructional systems, and microcomputer technology that make it possible to develop Intelligent CBI (ICBI) authoring systems which are generative in nature. 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 investigated from several perspectives. The predominant approach derives from programming techniques associated with artificial intelligence.

This approach is typically characterized by use of the programming languages LISP and, to a growing but lesser extent, Prolog. These languages are especially good for rapid prototyping. More uniquely, their very nature lends themselves 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 who are working in this tradition actually equate the word intelligent with the ability to give reasons.

Educational applications have tended to parallel this approach. 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 is 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 idiosyncracies and reasoning logically on the basis of available knowledge. Nonetheless, judging from the sparsity 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 sometimes 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 personally 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 would have made development prohibitive without generous federal support. Although 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 content and pedagogical experts be able to participate directly in such development.

In this chapter we describe a microcomputer-based ICBI authoring system that will allow instructional designers and content experts who are not skilled 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, no one has succeeded in developing such an ICBI system, much less an easy-to-use authoring system for developing such systems. However, there

---

14. A STRUCTURED APPROACH TO INTELLIGENT TUTORING

are conceptual, pragmatic, methodological, and technological reasons for believing that such systems can be developed: (a) a well-researched theory (Structural Learning Theory) in which content/tutorial 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, disciplined and cumulative 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 makes it possible to represent content in precisely the form needed for use by any of the planned modular ICBI tutorial systems.

The system being developed by Intelligent Micro Systems, Inc. (IMS) is represented schematically as shown in Fig. 14.1. ICBI systems are depicted as having two parts: an intelligent RuleTutor system (including the learner & tutor models) 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 particular, we shall detail requirements for an intelligent RuleTutor designed to provide optimal diagnosis and remediation with respect to cognitive procedural tasks (i.e., single rules). In addition, we show how this RuleTutor might be extended in principle to accommodate any type of content.

Figure 14.1 also shows that IMS has developed PRODOC (solid line) and intends to develop intelligent RuleTutors based on the Structural Learning Theory (dashes). 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
As described by Scandura (e.g., 1977, Scandura & Scandura, in press), domains and ranges of rules are structures in which some of the specific values of 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 more general because 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 (see Scandura, 1987). 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 same structure may be represented in terms of ordered sets as shown below.

\[
\begin{align*}
\text{REPEAT} & \quad \text{SELECTION} \quad \text{\textless{}X,} \\
& \quad \quad \quad \quad \quad \quad A. \\
& \quad \quad \quad \quad \quad \quad \text{SEQUENCE} \quad \text{\textless{}B,} \\
& \quad \quad \quad \quad \quad \quad C. \\
& \quad \quad \quad \quad \quad \quad \text{D} \\
& \quad \quad \quad \quad \quad \quad \text{\textgreater{}}. \\
& \quad \quad \quad \quad \quad \quad \text{Y}
\end{align*}
\]

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

INTRODUCTION TO STRUCTURAL LEARNING THEORY

In the Structural Learning Theory (SLT) 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, 1977a, 1980, 1981a). As described by Scandura (e.g., 1980, 1981a, 1981b, 1984a, 1984b; Scandura & Dumin, 1977, Scandura et al., 1971), 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.

Once an analysis has been completed, designing an effective instructional strategy follows directly from the theory (e.g., see Scandura, 1981b). Specifically, once an 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 . . . (See Scandura & Scandura, 1987 in press, for directly related discussion. Further details may be found in the literature on the SLT approach to instruction (e.g., Scandura, 1977b), the role of determination in the SLT (e.g., Scandura, 1971, 1977a, 1977c, 1978), the role of higher order rules in learning and creative behavior (e.g., Scandura, 1973b, 1974; Wulf & Scandura, 1977) and the SLT generally (e.g., Scandura, 1971, 1973a, 1977a, 1980, 1985).)

MICROTUTOR II ARITHMETIC TUTOR

Between 1980 and 1982, Intelligent Micro Systems, Inc., implemented a reasonably 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 that 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 users.

Despite these generalized diagnostic and tutorial capabilities, the system 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.

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 conditional 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 change 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 restrict 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 (minor but often 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).

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 (see the section on related research.)

3. The designs ensure 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 AI-based systems.

All structural learning theories (SLT) (e.g., Scandura, 1971, 1977a, 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.

FIG. 14.2. Overview of structural learning theory.

In their simplest form, SLT’s involve one individual interacting with environment. In this case an observer (e.g., psychologist) attempts to predict and/or control the individual’s behavior with respect to his proscribed domain of discourse (characterized in terms of problems in the problem domain and the rules, including higher-order rules, associated with them).

The teaching-learning process involves a variation of the above in which the participants is an idealized teacher and the other, a learner. This interaction is represented schematically in Fig. 14.2.

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, & Wulfek, 1974; Scandura, 1977a; 1984a, 1984b). Today, SA has reached the point in its evolution where critical aspects of the process no longer reasonably be automated.

The current ICBI research is concerned primarily with those portions of the teaching-learning process pertaining to the “idealized” teacher and a 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 has built into it all of the theoretical and optimal machinery for diagnosing learner difficulties and for providing optimal 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. For more extensive discussion of the learner model may be found in a variety of publications (e.g., Scandura, 1971, 1973a, 1977a (esp. Chapter 2), 1980).

This idealized teacher is characterized at a very high level in the FLOWform in Fig. 14.3. Notice that no constraints are placed on the content to be taught.
Curriculum Tutor = provides optimal diagnosis and instruction needed to produce learner mastery on all rules in rule derivation hierarchy. learner = characterized by an universal control mechanism and processing capacity; and a rule set (list of mastered rules plus current rules may be sufficient but, in general, 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.).

FIG. 14.3. Curriculum tutor.

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). At the beginning of instruction, the Curriculum_Tutor does not know which parts of which rules the learner knows. Its task is both to determine that information and to assure mastery in a theoretically optimal manner (cf. Wulfeck & Scandura, 1977).

Refinement of the CURRICULUM_TUTOR into components involves a REPEAT...UNTIL loop: a GENERALIZED_RULE_TUTOR, 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. The basic gist 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 perspective 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 failure, the Intelligent Curriculum_Tutor determines an unmastered rule near the bottom of the hierarchy and provides instruction on that rule.

The fact that the RULETUTOR is an essential component of the CURRICULUM_TUTOR is especially important given our stated interest in fitting generalization. At a high level, notice that the RuleTutor (see Fig. 14.4) 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 (see Fig. 14.5).

Although reduced in scope, the RuleTutor also is designed to provide highly efficient testing and teaching. In general terms, an expanded version of the RuleTutor tests the learner to determine which parts of the to-be-learned target rule have been mastered and which have not. At appropriate points in the testing process (e.g., when all prerequisites to a failed problem type are known to 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. An expanded version of the RuleTutor is given in Fig. 14.5. More details on the Curriculum and Rule tutors can be found in Scandura & Scandura (in press).

SAMPLE ARITHMETIC AND LIBRARY RULES

As already emphasized, the RuleTutor works in conjunction with individual rules. Consequently, to construct a working ICBI RuleTutor system, one must first create rule specifications for the content (i.e., some cognitive procedural task).

Our work with potential ICBI authors (e.g., Scandura, 1984a, 1984b) show that by applying the method of structural analysis systematically, they typically are quite able to construct FLOWforms representing procedures for teaching in their areas of expertise — so 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 formalization of the method of structural analysis (e.g., Scandura, 1984a, 1984b). Given content and pedagogical competence, and guidance in the use of structural analysis, potential ICBI authors were able to create rule representations functionally equivalent to those constructed by expert analysts.

The FLOWform in Fig. 14.6 is illustrative. This FLOWform can be used to

FIG. 14.5. An expanded version of the RuleTutor.

FIG. 14.6. Subtraction: Old FLOWform.
solve any given column subtraction problem. By carrying out successive steps of the FLOWform, one effectively simulates the process of column subtraction. (In a similar manner, one can simulate essentially any process.)

Although authors can be taught how to perform a structural analysis, they do not normally do so—nor for that matter is it absolutely essential in identifying rules in their areas of expertise. Potential authors with some programming experience will often prefer to construct FLOWforms directly without following any particular systematic method of analysis. (Indeed, as powerful as it is, we believe that structural analysis will come into widespread use only when users have access to computer based systems which perform the many time consuming but necessary steps automatically.)

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.

In structural analysis, users are required first to represent data (problem) structures and 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. The clarity of FLOWforms makes it easier to detect errors, if not avoid them altogether. In addition to being easy to read, FLOWforms make excellent 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 ICBT RuleTutor.

In the case of arithmetic, for example, we would need an atomic rule library which provides an adequate basis for constructing rules associated with cognitive procedural tasks in arithmetic—tasks such as column subtraction, addition of fractions, etc.

Fortunately, the atomic library rules currently available to PRODOC provide an adequate foundation for this purpose. One can use these library rules to create a FLOWform (rule) representing essentially ANY cognitive procedural task. For example, the subtraction FLOWform below is constructed entirely from atomic rules in a current PRODOC library (see Fig. 14.7).

---

**FIG. 14.7. Library: Subtraction.**
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 LIBRUBST FLOWform use the current names of library rules and are not necessarily optimal for use by educators.

In addition, PRODOC can be used to create all of the needed data structures, such as current.top_digit, current.bottom_digit, current_answer_digit, and last_nonzero_column (used to determine how far to go back in borrowing across zeros).

To maximize ease-of-use, at the minimum, certain rules in the existing library would have to be modified so that they perform many of the auxiliary operations automatically (and transparently from the perspective of the user). For example, the display rule might be enhanced to operate on location parameters which are more directly meaningful. Thus, the position on the screen where an answer_digit is to be displayed might be represented at a level higher than the top and left margins associated with the normal text screen (which contains 25 x 80 characters). Thus, location might be defined more naturally in terms of rows and columns in the subtraction problem. In this case a sample display rule might take the form:

\[
\text{display (answer_digit, location (answer_row, column))}
\]

Notice that the above display operation has a parameter location which is itself an operation with two parameters: "answer_row" and "column."

This slight of hand, of course, would require that the display rule have considerably more intelligence than it currently has. Given the row and column specifying where something is to be displayed, the (new) display rule would have to calculate the corresponding position on the screen.

Ideally, one might go beyond this minimum, and make the required display operations totally transparent to the user. That is, once a bottom digit has been subtracted from a top digit, the subtract rule might not only subtract the digits but also display the difference on the screen in the proper location. The basic question, in effect, is how easy it will be to construct a given rule from an available library.

Making rule construction as easy as possible for potential ICBI authors will involve the availability both of atomic rules that are especially well suited for constructing rules associated with particular content domains; the availability of appropriate data structures also will be highly desirable.

The overall goal, of course, is to come up with a set of atomic library rules which are adequate, not only for constructing rules associated with whole number arithmetic, but others as well. Hence, we included in our sample analysis a variety of tasks associated with fractions and decimals.

In algorithms for adding and subtracting mixed number fractions, for example, certain components 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., fraction) contexts.

Higher level (but not higher order) rules of this type can be accommodated using the PRODOC system without requiring the user to specify details of these higher level component rules. This can be accomplished as follows: Once a new set (of set of new rules) has been constructed using atomic rules in a given "library" rule (or rules) can be added to the library using the PRODOC19 (library generation) capability. The result in this case would be a new PRODOC prototyping environment with an extended library.

In the fullest sense, of course, teaching arithmetic may involve the meaning of the operations, and potentially even verbal descriptions of arithmetic problems and solution procedures as in verbal problem solving.

Characterization of arithmetic in terms of rules in this broader sense is beyond the scope of the current 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.
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 group. 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 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, they gradually acquire an informal awareness of such rules by solving a variety of specific concrete problems, with concrete objects and/or pictorial representation 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 RuleTutor can use) could require extending the above library by adding to the atomic rules corresponding to the above components (e.g., regroup, etc.).

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 respect are computer memory and/or addressing capacity of the operating system.

**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 even a 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 imperfect 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 former 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..."
memory under MS-DOS. Each of these environments makes use of Scanlan FLOWforms.

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 structural environment. In addition to supporting simulation in English text, the availability of high level library rules makes PRODOCea ideal for rapid prototyping. The availability of graphical support for input and output data structures also makes it possible to directly reflect arbitrary semantic properties.

A recent version of PRODOCea employs a general purpose but relatively low level set of library rules (see Table 14.1).

In conjunction with PRODOC's Library Generation facilities (see (4) below) custom versions of PRODOCea (and PRODOCip) can quickly be created to accommodate library rules having arbitrary semantic properties, and thereby facilitate rapid prototyping in desired application areas.

An unique feature of PRODOCea is its ability to immediately execute not only interpretable library rules but to simulate execution statements using library syntax and/or written in ordinary English. Among other things, this adds discipline to the traditionally unsystematic task of programming. It also makes the difficult and expensive process of developing performance aids or expert assistants almost trivial. Once an (nonprogrammer) expert knows exactly what a human/computer assistant is to do, for example, it is a simple task to develop a computerized performance aid to assist less qualified personnel in performing the required tasks.

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 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.

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

In addition, full pseudo code support currently is available as an option for Pascal, C, and Ada. These options combine the clarity and ease of use of high level fourth generation languages with the flexibility of third generation languages, and include full syntax checking, consistency checking, automatic declarations generation, and full 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.
[SORT] : sort

Sort up to 500 numbers; print result

write ('How many numbers (1 to 500) to be sorted? ')
readln (n)

writeln ('Enter below numbers to be sorted. Press <Return> after each.')
FOR i:=1 to n
  DO  readln (a[i])
FOR i:= 1 to n-1
  DO  FOR j:= 1 to n - i
    DO  IF a[j] > a[j+1]
      THEN  temp := a[j]
            a[j] := a[j+1]
            a[j+1] := temp
  writeln
writeln ('The resulting order is:')
FOR i:= 1 to n
  DO  writeln (a[i]:2)
TABLE 14.1
Rule Library Catalog

<table>
<thead>
<tr>
<th>Input/Output</th>
<th>Standard Rule Name</th>
<th>Application-Specific Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPSSTAT.BRL</td>
<td>display_state</td>
<td>display_state</td>
</tr>
<tr>
<td>DISPELT.BRL</td>
<td>(ROOT_ELEMENT)</td>
<td>display</td>
</tr>
<tr>
<td>LOAD.BRL</td>
<td>load</td>
<td>load</td>
</tr>
<tr>
<td>SAVE.BRL</td>
<td>save</td>
<td>save</td>
</tr>
<tr>
<td>GETINPUT.BRL</td>
<td>get_input</td>
<td>get_input</td>
</tr>
<tr>
<td>LOADPIC.BRL</td>
<td>load_picture</td>
<td>load_picture</td>
</tr>
<tr>
<td>CLRVIDEO.BRL</td>
<td>clear_video</td>
<td>clear_video</td>
</tr>
</tbody>
</table>

Operations

<table>
<thead>
<tr>
<th>BRL</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSRTCPT.BRL</td>
<td>insert_component_after</td>
<td>insert_component_after</td>
</tr>
<tr>
<td>DELETCPT.BRL</td>
<td>delete_component</td>
<td>delete_component</td>
</tr>
<tr>
<td>SHRCTAP.BRL</td>
<td>share_component_after</td>
<td>share_component_after</td>
</tr>
<tr>
<td>DELAY.BRL</td>
<td>delay</td>
<td>delay</td>
</tr>
</tbody>
</table>

Parameter Functions (Arithmetic)

<table>
<thead>
<tr>
<th>BRL</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.BRL</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>SUBTRACT.BRL</td>
<td>subtract</td>
<td>subtract</td>
</tr>
<tr>
<td>MULTIPLY.BRL</td>
<td>multiply</td>
<td>multiply</td>
</tr>
<tr>
<td>DIVIDE.BRL</td>
<td>divide</td>
<td>divide</td>
</tr>
<tr>
<td>POWER.BRL</td>
<td>power</td>
<td>power</td>
</tr>
<tr>
<td>GRTSTINT.BRL</td>
<td>greatest_integer</td>
<td>greatest_integer</td>
</tr>
<tr>
<td>MODULO.BRL</td>
<td>modulo</td>
<td>modulo</td>
</tr>
<tr>
<td>ABSVALUE.BRL</td>
<td>absolute_value</td>
<td>absolute_value</td>
</tr>
<tr>
<td>ROUND.BRL</td>
<td>round</td>
<td>round</td>
</tr>
</tbody>
</table>

Parameter Functions (Relationships)

<table>
<thead>
<tr>
<th>BRL</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND.BRL</td>
<td>find</td>
<td>find_component_with_value</td>
</tr>
<tr>
<td>NEXTCPT.BRL</td>
<td>next_component</td>
<td>next_component</td>
</tr>
<tr>
<td>CONNCPT.BRL</td>
<td>common_component</td>
<td>common_component</td>
</tr>
</tbody>
</table>

Conditions

<table>
<thead>
<tr>
<th>BRL</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQUAL.BRL</td>
<td>lequal</td>
<td>match</td>
</tr>
<tr>
<td>NUMEQUAL.BRL</td>
<td>numerically_equal</td>
<td>equal</td>
</tr>
<tr>
<td>NUMUNEQUAL.BRL</td>
<td>numerically_unequal</td>
<td>unequal</td>
</tr>
</tbody>
</table>

(Logical Connectives)

<table>
<thead>
<tr>
<th>BRL</th>
<th>Expression</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGCLAND.BRL</td>
<td>logical_and</td>
<td>and</td>
</tr>
<tr>
<td>LOGCLROR.BRL</td>
<td>logical_or</td>
<td>or</td>
</tr>
<tr>
<td>LOGCLNOT.BRL</td>
<td>logical_not</td>
<td>not</td>
</tr>
</tbody>
</table>

Assignment

<table>
<thead>
<tr>
<th>BRL</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN.BRL</td>
<td>assign</td>
<td>assign</td>
</tr>
</tbody>
</table>

[NOTE]: sort
Sort up to 500 numbers, print result

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

FIG. 14.9a. Sort Flowform.
Sort up to 500 numbers; print result

1. Specify the number of numbers to be sorted.

```plaintext
write ('How many numbers (1 to 500) to be sorted? ')
readln (n)
```

2. Prompt user, then get the numbers.

```plaintext
writeln ('Enter below numbers to be sorted. Press (Return) after each. ')
```

3. Get the numbers.

```plaintext
FOR i := 1 to n
  DO readln (a[i])
```

4. Sort them.

```plaintext
FOR i := 1 to n - 1
  DO
    . Scan thru items and swap if necessary.
    FOR j := 1 to n - i
      DO
        . Compare and swap if necessary.
        IF a[j] > a[j + 1]
          THEN
            . Swap
            temp := a[j]
            a[j] := a[j + 1]
            a[j + 1] := temp
```

5. Display description, then print the ordered set.

```plaintext
writeln
writeln ('The resulting order is: ')
```

6. Print the ordered set.

```plaintext
FOR i := 1 to n
  DO writeln (a[i]:2)
```
The relationship between Pascal pseudo code in the SORT FLOWform (Fig. 14.9a) and the corresponding full source code (Fig. 14.9b) 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 (see Fig. 14.9c).

4. Library Generation (PRODOC) (see Fig. 14.10)—makes it possible to integrate available rule libraries and new library rules into either PRODOC prototyping environment, thereby creating customized versions of PRODOC for particular uses. 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. It may be ported to other operating systems as need dictates.
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. Scandra 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., Scandra, 1971, 1977a, 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., Scandra, 1970, 1971, 1973a, 1973b, 1980).

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

d. the conceptualization and design of IMS’s PRODOC programming environment (Scandra, 1977a).

While independently derived, these conceptual formalisms share certain features with other work in artificial intelligence and the cognitive sciences. 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), 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 system level. The instructional theory also shares certain elements in common with other algorithmic formulations, such as that by Landau (1978). It does, however, provide a more rigorous base for computer implementation.

Also, as mentioned previously, the method of structural analysis (e.g., Scandra, 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 1950s & Gagné in 1962) can be viewed as a more limited 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.

More specifically in artificial intelligence, 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., Scandra, 1971, 1977a) 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, 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 Scandra (e.g., 1977a, 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. (1982) 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. 68–72). As demonstrated by Dunnin and Scandura (1973a), the behavior of an individual student will be more or less compatible with any given prototype—in this case, the behavior of most American 4th 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 (particularly European) students is more compatible with equal additions, just as others may be more consistent with error, or “buggy” rules. See Scandura (1977a, Ch. 10) for a discussion of considerations involved in determining which of two or more rule prototypes provides the best account of overall performance.

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, as confirmed in recent research, calling attention to incorrect skills can lead to later confusion. According to the Structural Learning Theory (e.g., Scandura, 1971, pp. 41–44; 1973a, Ch. 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 of John Anderson and his associates (unpublished research proposal) on intelligent tutors. 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 (Anderson, 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., Hilke, Kemper & Scandura, 1977c; Scandura, 1971, 1973a, 1977a). Rather than talking about “productions” being available with some probability, as Anderson does, “rules” in SLT's are either 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, 1973a, 1977a, 1977b). As in our research, prototypes are used to characterize what is to be learned. According to Anderson et al., however, they have not been able to completely modularize tutorial and control portions of the system.

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 involve 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), the ICH RuleTutor provides for the automatic generation of content. It also allows for future extensions including provision for “bugs,” alternative perspectives, and logical inference.

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, 1973a, 1977a, 1980). In combination with the capability to serve (used as an authoring system) this modularity will facilitate future development. Instead of taking 250 hours or so to produce an hours worth of intelligent tutors (e.g., Anderson, unpublished talk), it should take barely more time than it takes to specify the rules to be learned—by the most conservative estimate a five fold
improvement. Equally important, it may be feasible for the first time for instructional designers (who are 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 chapter we have focused on how to utilize current understanding of cognitive and instructional processes as a basis for creating intelligent tutoring systems. Among other things we have determined the feasibility of implementing a new class of ICBI authoring/development systems having two distinct but complementary parts: First is a general-purpose, intelligent (ICBI) RuleTutor (or CurriculumTutor) 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 would consist of IMS's PRODOC software development system. To facilitate its use by educators, PRODOC may be supplemented with a rule library specifically designed to simplify authoring of the desired content. Such a system would provide an easy-to-use medium for creating arbitrary cognitive procedures (rules) in the intended content area. Each such rule, in turn, would be interpreted by the ICBI RuleTutor, resulting in a fully operational ICBI RuleTutor system for the given cognitive procedure.

More specifically, we have outlined a general-purpose ICBI RuleTutor which can be used in conjunction with ANY cognitive procedure formulated as a rule (i.e., as defined in the Structural Learning Theory—e.g., Scandura 1970, 1977a, 1984a). (See Scandura & Scandura, in press, for more detailed specifications.) We also have described the kinds of rule components (atomic rules) needed in a rule library optimized for use in arithmetic. The atomic rules in such a library provide a natural basis for formulating arbitrary rules (corresponding to to-be-learned cognitive procedures) in arithmetic.

Taken together, the RuleTutor and PRODOC should provide a sound basis for developing intelligent tutorial systems for any cognitive procedural task. Moreover, when coupled with an optimized library (e.g., in arithmetic) such an authoring system would make it possible for subject matter and/or curriculum experts to automatically generate ICBI systems in arithmetic with little actual programming.

Of perhaps even greater importance, such an authoring system could easily be customized to accommodate ANY new content domain. Since PRODOC already includes a general purpose library of relatively low-level rules it can, in principle, be used with any content. Nonetheless, customization for various particular

domains could be desirable to make it even easier to use by subject matter experts who are not skilled programmers.

The diagnostic/tutorial (RuleTutor) system currently under development is best suited for cognitive procedural tasks. Nonetheless, this system is based on an extensible design, and thereby opens up the possibility of eventually extending the system to include more general curricular content involving higher-order rules (e.g., problem solving—Scandura, 1977a, heuristics—Scandura, Dunn & Wulfek, 1974, inference—Scandura, 1973a, learning—Scandura, 1974a) and alternative perspectives on that content (e.g., Durin & Scandura, 1973; Scandura, 1977a, Ch. 10). Also on our long-term agenda are plans to automate a method of structural analysis (Scandura, 1984a). Combined with the proper ICBI authoring system, authors would not even have to use PRODOC to create rule FLOWforms. Automated structural analysis would do this induction given only samples of solved problems.

AUTHOR NOTE

This project is being 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. A more detailed account of many of the issues raised in this chapter is given in Scandura, J. M. and Scandura, B. The intelligent RuleTutor: a structured approach to intelligent tutoring phase II. Journal of Structural Learning, 1987, 9, in press.

REFERENCES


Scandura and Scandura

14. A STRUCTURED APPROACH TO INTELLIGENT TUTORING


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


