4

Some techniques for building Mathematical intelligent tutoring systems

TAK-WAI CHAN
National Central University, Taiwan, Republic of China.

4.1 Introduction

This chapter discusses some techniques in building an intelligent tutoring system called Integration-Kid. Integration-Kid is a *learning companion systems* (LCS) (Chan & Baskin, 1988, 1990; Chan, 1991) in the domain of integral calculus. In the learning environment of a learning companion system, there are three agents involved, namely, the *human student*, the *computer learning companion*, and the *computer teacher*. As implied by its name, the role of the computer learning companion is to act as a learning companion for the student. In this chapter, we discuss a production system which is used to simulate different agents' interactions via a common blackboard. Then we discuss how to model domain knowledge; in particular, we describe a term rewriting system which is the basis of the problem solver for both the companion and the teacher. We next discuss how the system handles students' bugs. Then an algorithm which parses mathematical expressions into readable two dimensional expressions is given. Finally, we provide a brief conclusion.

In designing Integration-Kid, it is natural to start by considering the original LCS environment as shown in Figure 1. From the figure, there are two major issues: representation of each agent (denoted by the ellipses) and the interactions among them (denoted by the double arrow lines).

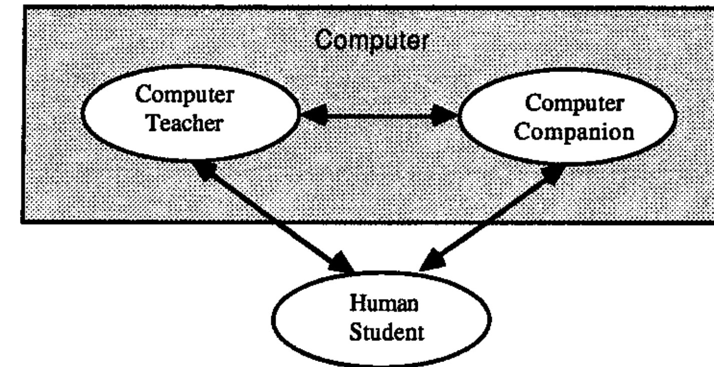


Figure 1 - Learning Companion System Environment

With the above intuition, we represent the three agents separately in the system. Each agent is a set of *rules of behaviour* modelling the behaviour of the agent. The three agents communicate via a *blackboard*. A simple *agent scheduler* controls the system when the agents look at the blackboard and execute. The output of an agent consists of the utterances on its own window in the interface. Since there is no student model in Integration-Kid, the student agent only consists of those rules that interpret the student's input and put it on the blackboard for the other two agents to react.

4.2 Overview of Production and Blackboard Systems

We use production system and blackboard system as a part of our building block for Integration-Kid. They are two common knowledge-based system architectures. We first give an overview of production systems and blackboard systems and their uses in ITS.

Production systems (Forgy, 1981), a type of pattern-directed inference systems, is a class of programming languages used primarily for applications in the areas of artificial intelligence, expert systems, and cognitive psychology. A production system has two kinds of memory: a collection of *production rules* (also known as *productions*) called the *production memory*, and a set of data called the *working memory*. A production rule can be perceived as a condition-action, situation-action, or cause-effect procedural element and the data in the working memory represents a situation at a certain point of time. An inference engine looks at the current situation and tests which production rules are appropriate and decides which action to take. Taking this action will then cause the situation or data in working memory to change and, sometimes, can alter the production memory

satisfied, then the rule's *action* (or right-hand-side part) may be executed, and the rule is said to have *fired*. Production systems resemble closely how an agent looks and reacts to a situation, that is, it exhibits some kind of psychological validity. For those readers interested in detail, consult one of the best-known production system language, OPS5, which is nicely documented in a book by Brownston *et al*. (1985).

The basic flow of control in a production system is called the recognise-act cycle (Figure 2).

1. [Recognise] Evaluate the conditions of rules and determine which are satisfied given the current data in the working memory.
2. [Conflict Resolution] Select one rule with a satisfied condition. If no rules have satisfied conditions, then halt the system.
3. [Act] Execute the action part of the selected rule.
4. Go to step 1.

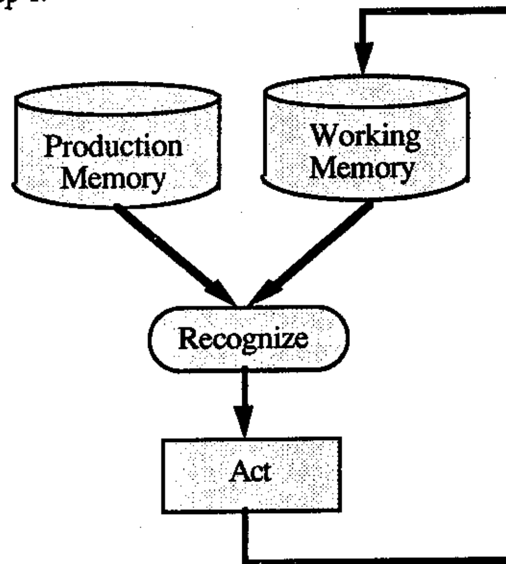


Figure 2 - Recognise-Act Cycle

Each rule's condition is a description of the states in which the rule is applicable; the condition is satisfied when there is information in the working memory that the production can process. When the system performs the recognise process, it is in effect searching for rules that know how to process the data in the working memory. When those rules are found, one rule will be selected (conflict resolution) and its action part executes. Data in the working memory will then be changed and

common strategy is to use some preference rules to determine exactly one rule to fire. After the rule has been fired, the set of eligible rules to be fired is recomputed, so that rules that are no longer eligible to fire are removed from the set, and rules that just become eligible to fire are added to it.

Blackboard architectures, first introduced in the Hearsay-II speech understanding system from 1971 to 1976 (Erman *et al.*, 1980), have become popular for knowledge-based applications. The blackboard paradigm is rather simple to describe (Figure 3). It consists of a set of *knowledge sources*, a global data structure called the *blackboard*, and a control component called the *scheduler*. A knowledge source is usually "larger" than a rule and functions independently from others, thus resembling a single expert. It can be a procedure, a production system or even a blackboard system itself. The scheduler, among the qualified knowledge sources, selects one and executes it, thereby changing the blackboard.

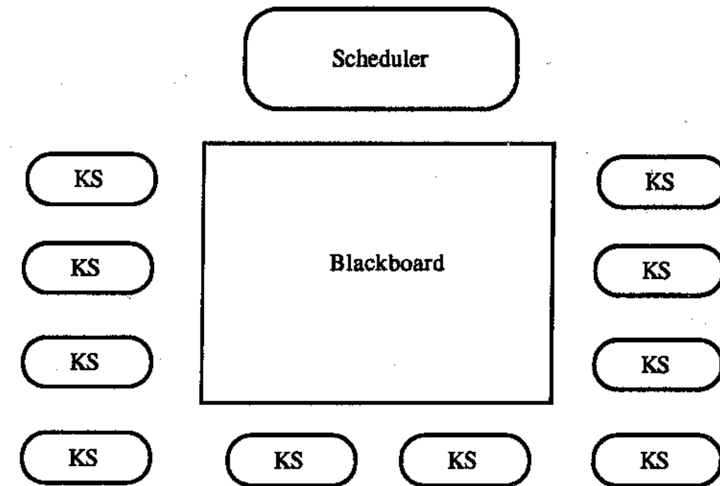


Figure 3 - Schematic of a Blackboard System

In the subsequent cycle, the change in the blackboard triggers another qualified set of knowledge sources and so on. While the blackboard model is reminiscent of the production system, it does *not* specify a methodology for design and implementation (Nii, 1986). Historically, most blackboard-based systems have been built according to the criteria that appeared most appropriate for the particular application (Corkill *et al.*, 1986). The blackboard approach may be viewed as a philosophy or a set of guidelines rather than as a carefully specified process. Some blackboard systems emphasise the control process, e.g., BB1 (Hayes-Roth, 1985), and some have complicated insertion/retrieval strategies for the blackboard data, e.g. Hearsay III,

GUIDON and CMU tutors discussed below are two important ITS researches which adopt production systems as the framework for building ITS. Their experiences show that production rules are an appropriate representation for coding teaching knowledge. GUIDON is the earliest work using production rules for teaching. Its domain and tutorial knowledge are both represented separately by MYCIN type production rules. The tutorial knowledge is stated explicitly in the form of 200 production rules, which include guiding the dialogue, presenting diagnostic rules, constructing a student model, and responding to the student's initiative. GUIDON demonstrates that teaching knowledge can be codified in production rules and built incrementally and that the framework of tutoring rules can be organised into discourse procedures (Clancey, 1979).

CMU Tutors use the production system GRAPES. GRAPES is a modular representation language to encode cognitive processes. Different from other production systems, production rules in GRAPES (Anderson et al., 1984) are strictly interpreted within a hierarchical goal structure (AND/OR graph). In one of their tutors, the LISP tutor, there are 325 correct rules and 475 mal-rules. After numerous problem solving protocols by human students have been analysed, the number of mal-rules has increased to over 1,200 (Anderson & Skwarecki, 1986). Experience of CMU Tutors indicates that the development of ITS can and should be an empirical experiment with a large amount of revisions.

Blackboard-based architectures for instructional planning have been adopted by a few ITS researchers (MacMillan & Sleeman, 1987; Murray, 1989, 1990). They focus on how to generate dynamically, execute, and revise instructional plans. MacMillan *et al.* (1988) point out that the process of understanding instructional planning and the requirements for machine instructional planners are still at an early stage of research. They note that better understanding of how teachers plan and make decisions during instruction will be vital for the complex process of machine-based instructional planning.

4.3 LPS — A Language to Simulate Agents' Behaviour

In Integration-Kid, the blackboard is a collection of data, representing the current situation. When an agent looks at the current situation, rules of behaviour of that agent will be tested to see which one is appropriate to execute. Such rules of behaviour are represented as production rules. Thus an agent is a simple production system. If all the data on the left-hand side (LHS) of a rule is on the blackboard, the sequence of actions on the right-hand side (RHS) will be executed. The protocols of activities discussed in the last chapter are all constructed with such rules of behaviour.

The behaviour of a human student is driven by his/her own intelligence; for the teacher and the companion, their general communicative behaviour in learning is supported by their domain knowledge. Their problem solving abilities, part of their domain knowledge, are modelled by the problem solver and are called by the RHS of the rules of behaviour. This problem-solving ability can be viewed as part of an agent's behaviour. For the student agent, apart from those rules that interpret the student's input, there are rules that allow the student to control the system at his own pace. Some examples are referencing an old problem or the table of basic rules of integration, editing a sequence of mathematical expressions, or exploring further details of certain points from the teacher's or the companion's utterances by some reference facilities provided in the interface. Some sub-programs such as editor facility, reference facility, utterance processor, syntax checker, problem solver etc., support the rules of behaviour of different agents. Organisation of these rules of behaviour through inheritance and the scheduling of a piece of learning activity which we call an *episode* is via a 'curriculum tree' architecture (Chan, 1992a).

The production system implemented in Integration-Kid is called LPS. LPS is object-oriented. It has simple syntax and a simple working mechanism (by adopting the simplest conflict resolution strategy). Besides, it uses different rule types to support changes of memories.

LPS is implemented as object-oriented, a new production system can easily be created as an instance of the object. For example,

```
(setq *scheduler* (make-instance 'lps :name 'SCHEDULER))
```

defines a production system, *scheduler* (a global variable), which is an instance of an LPS object with the name SCHEDULER. Four other production systems used in Integration-Kid are:

```
(setq *bbs* (make-instance 'lps :name 'BBS))
(setq *student* (make-instance 'lps :name 'STUDENT :parent-ps
   *bbs*))
(setq *teacher* (make-instance 'lps :name 'TEACHER :parent-ps
   *bbs*))
(setq *companion* (make-instance 'lps :name 'COMPANION
   :parent-ps *bbs*))
```

The production systems, *student*, *teacher*, and *companion* simulate the behaviour of the agents, the student, the teacher, and the companion, respectively. We shall call such a production system an *agent production system* or just an agent.

A production system may have a parent production system. The agent production systems all have a parent production system *bbs*. If a production system has a parent, running the production system will use both its working memory as well as its parent's, that is, the union of its working memory and its parent's. For example, when running the production system *student*, the actual working memory used will be the union of the working memory of *student* and *bbs*.

A parent will not see any of its child's working memory while a child will see all of its parent's working memory. One way to view this design is: suppose there is a large set of data that a set of rules will be looking at. If a subset of such data will be only interested in a subset of those rules while the whole set of data will be interested in the rest of the rules, then we may factor out the subset of data with its interested rules to be a parent production system. The additional work is to control the decision when to run a parent or its children.

However, Integration-Kid uses a simpler relationship of the set of production systems. The working memory of the agent production systems is empty. The production memory of the *bbs* is also empty. The system will not run *bbs*. Thus the agent production systems are essentially three sets of rules while *bbs* is a common working memory shared by three agents. Thus,

```
(assert! *bbs* (problem ($ (sin (2 * x)) d x)))
```

(discussed further below) will assert the datum (problem ($ (sin (2 * x)) d x)) in the common working memory *bbs* and

```
(rule! *teacher*
     (teacher check student-substitution)
     (student substitution ?stu-subst)
     (teacher substitutions ?tea-substs)
     ->
     code for checking student's chosen substitution
     )
```

is a rule of the agent *teacher*.

4.3.1 Conflict Resolution in LPS

At the recognise part of running LPS, if there is no rule eligible to fire, then the LPS will halt. If only one rule is eligible to fire, then it will fire that rule. When more than one rule is eligible to fire, we could have them all fire in parallel by updating copies of working memory, or we could use some preference strategies to decide which rule to fire. In LPS, we simply choose the first eligible rule in the recognise part to fire. This idea is inspired by the simple matching procedure of the language Prolog. Matching of a pattern in Prolog is a linear search (from top to bottom) of the sequence of Prolog rules and facts until there

and facts in the order decided by the programmer. In order to write a Prolog program, the user has to know such a matching process. Similarly, in order to use LPS, the user has to know its simple conflict resolution strategy. A rule that appears earlier in the production memory will have a higher priority to fire than one that appears later. An immediate advantage is that this strategy essentially needs no effort for conflict resolution. Hence, conflict resolution is determined when the rules are written rather than when the system is running. An immediate problem of this strategy is that after a rule has fired, if the data in the working memory that triggered the rule has not been changed and that rule remains in the production memory after firing, and no new rule is added in the production memory before it, then this rule will fire again and thus an infinite loop occurs. This problem will be discussed further when we describe the different rule types in LPS.

4.3.2 Representation of Data in Working Memory of LPS

Data in the working memory are stored as a linear list. Thus, matching with triggers of rules is a linear search. It could be stored in a discrimination net to enhance the search. However, with the curriculum tree architecture, most of the time the number of data in the working memory is less than 12, a very small number which will not affect the efficiency of the system even using a linear list. Data in the working memory looks like this:

```
(student useful substitution)
(student suspicious)
(companion not useful alternative).
```

This data describes that the student is using a proper substitution in using the substitution method but he is now suspicious about the substitution he has picked. Unfortunately, the companion's suggested alternative of substitution at this point is not a useful one. Notice that such data of the student and the companion form a part of the current learning situation. They are all represented in *bbs*.

While the above data mainly describes the status of agents, there is other data needed for recording and passing to the action part of the rules to process when running. For example:

```
(problem ($ (sin (2 * x)) d x)
(student substitution (2 * x)).
```

Data in the working memory may also be atoms, for example:

```
more
2.
```

Storing data will either call a macro *assert!* or a function *assertf!*.

```
(assert! *bbs* (problem ($ (sin (2 * x)) d x)))
```

will assert the datum (problem ($ (sin (2 * x)) d x)) to the common working memory *bbs* and

```
(assertf! *bbs* `(student substitution, student-substitution))
```

will evaluate the student-substitution which is (2 * x) and assert the datum (student substitution (2 * x)) to the common working memory.

4.3.3 Representation of Rules in LPS

Syntactically, a rule in LPS consists of several components: a *rule type* that states what type of rule is being used. There are four rule types (discussed below); the *LPS instance*, for example, *teacher*; a list of *triggers* that represents the conditions that must all be true in order for the rule to fire; and an *action* that represents what happens when the rule fires. Here is a rule:

```
(rule! *teacher*
     (teacher check student substitution)
     (student substitution ?stu-subst)
      (teacher substitutions ?tea-substs)
     ->
      (delete! *bbs* (teacher check student substitution))
      (if (member ?stu-subst ?tea-substs :test #'equal)
      (assert! *bbs* (student good substitution))
      (assert! *bbs* (student bad substitution))))
```

The list of triggers, which is referred to as the LHS of the rule, are patterns, with variables indicated by symbols whose first character is "?". The action, which is referred to as the RHS of the rule, is a piece of Lisp code that will be executed when the rule fires. In running LPS, rules will not be added to the production memory, though possible to do, but sometimes they will be deleted from the production memory.

4.3.4 Rule Types in LPS

As mentioned above, after a rule has fired, if that rule does not change the working memory or the production memory, then that rule will fire again and an infinite loop occurs. Many production systems use the strategy of *refraction* where the data in the working memory that triggers a rule to fire is recorded by a time tag. LPS does not use this method. Which rule will fire next depends on what has changed in both memories. The trade-off is: refraction is automatic while LPS is more efficient because of deletion of data and rules in the memories. Rule types in LPS is a feature to enhance such change.

The other two reasons to have different rule types in LPS are: (1) it is a short-hand to change the memories; and, (2) because of the change, it increases the efficiency of the matching process in subsequent running.

There are four types of rules in LPS:

1. Rule!: The previous rule (in Section 2.3) example belongs to this type. A rule of this type will remain in the production memory after firing. Rules will not be added to the production memory. Data in the working memory may be changed but have to be specified by the user at the RHS of the rule. In the example, the datum (teacher check student substitution) is specified to be deleted from the working memory when the rule fires so that the same rule will not fire again.

2. Once-Rule!: An example of this rule:

```
(once-rule! *teacher*
    (problem ?p)
       ->
     (assertf! *bbs* `(integrand ,($ ?p))))
```

This rule will fire once and be deleted from the production memory.

(3) Bridge-Rule!: An example of this rule:

```
(bridge-rule! *student*
   (student edit substitution)
   ->
   code for student to edit a substitution expression)
```

This rule remains in the production memory after firing. However, all data that trigger the rule will be deleted from the working memory automatically. In this example, the datum, (student edit substitution), is removed from the working memory after firing.

(4) Once-Bridge-Rule!: An example of this rule:

```
(once-bridge-rule! *teacher*
   more
   2
   ->
   code for the teacher telling both learners something
   (assert! *bbs* 3))
```

This rule will be deleted as well as the triggers more and 2 after firing.

The use of the word 'bridge' is adopted from a Chinese saying:

once-bridge-rules are usually used in situations where the teacher explains a concept or format of activities, or demonstrates an example to the learners. Rules and bridge-rules, on the other hand, are used in situations that may re-occur again — some sort of looping. One may notice that while we would like to use once-rules or once-bridge-rules in certain episodic situations, for example, a rule that tells the students what is the current problem, such rules may be reused again in similar episodes. That problem is solved by adopting the curriculum tree architecture in which rules that will be used in similar episodes are inherited from the upper level of the tree. So those rules will come back again when switching to the next similar episode.

4.4 Modelling of Domain Knowledge

While the companion's conversation with other agents prescribed by the rules of behaviour simulates different protocols of activities, the companion's problem solving behaviour forms a basis of his interaction with others. First, the companion possesses certain background knowledge, such as simplification of algebraic expressions and differentiation. Presumably, he has no trouble with such knowledge. After the teacher introduces the basic concepts and demonstrates some simple examples, the companion acquires a set of basic rules of integration. These basic rules of integration are described as *term rewriting rules* (discuss below). However, this set of rules is imperfect. There are incorrect and missing rules. They will be fine tuned when the companion solves problems independently with the student and reveals problems with the rules. When learning new techniques such as the substitution method, the companion acquires procedures which incorporate the basic rules and his background knowledge such as differentiation. With support from the teacher, both the companion and the student have no trouble applying these procedures. However, successful use of such techniques hinges on the right choice of a sub-expression for the integrand, a choice which is largely heuristic in nature. Now, instead of generating and testing all possible candidate sub-expressions for each problem as other mathematical software does, the companion will use a set of plausible candidates as a base to hold conversation with the student via the protocol of activities. For complex problems such as the miscellaneous problems, such a list of candidates is not sufficient to represent the particulars of a problem and we use a *situation map* (discussed later in this chapter) which lays out such candidates as well as detailed situations in solving the particular problems.

As a summary, the companion first acquires a set of basic rules of integration which is fine-tuned as he uses it. Later, he learns advanced techniques which incorporate the rules and some background

apply, the companion uses an assigned set of candidates for each problem. Finally, for complex problems, a richer representation which includes both candidates and the detail of each situation is set for the companion for each problem.

For teacher, his problem solver is the same as that of the companion's, except at the outset, the set of rules of integration is complete and sound. Also, the teacher adopts the same list of candidates and the situation map used by the companion (Figure 4). This is the case because some knowledge encoded in the the list of candidates or the situation map is for the teacher. Furthermore, the rules of behaviour of the companion and the teacher defining the protocol reason differently on the list of candidates or situation map.

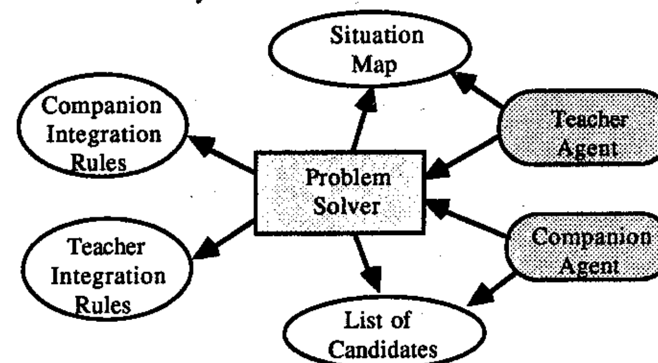


Figure 4 - Modelling Domain Knowledge of Companion and Teacher

It is interesting to note that, at the later stages, the teacher needs to rely on a given set of candidates or the situation map in order to monitor the learning activities. However, having the teacher completely acquire the knowledge is not our purpose. We can write code for the teacher to generate such candidates to get the answer. But that is not enough to support interaction with the learners.

4.4.1 Term Rewriting System as the Basic Problem Solver

We adopt procedures to solve complex problems; for example, we use a procedure that substitutes the substitution expression back into the integral at the last step of applying the substitution method to a problem. However, the basic building blocks of the problem solver can be conceived of as a set of *term rewriting rules*. In solving simple integration problems, it can be viewed as a continuous process of changing the given mathematical expression (rewriting sub-terms of the expression) until there is no integral sign. Apart from solving simple integration problems, background knowledge such as

represented as a term rewriting system. Our term rewriting system basically consists of two components: the *current expression* and *a set of term rewriting rules*. A term rewriting rule for the teacher is shown as follows:

```
(rrule *tea-ele-rs*
    ($ (?x ^ ?n) d ?x)
    :name "Rule 1"
    :var ?int
    :test (numberp ?n) (not (= ?n -1))
    ->
    (add-binding *tea-ele-rs* ?n+1 (1+ ?n))
    (subst! *tea-ele-rs* ((?x ^ ?n+1) / ?n+1) ?int))
```

Similar to a production rule, a term rewriting rule consists of a LHS and a RHS. In the above term rewriting rule, ($ (?x ^ ?n) d ?x) is the sub-expression. ?x and ?n are two variables that match any expressions. There is an attribute of the LHS, :name, for the name of the rule. Another attribute, :var, is a variable which denotes the sub-expression of the current expression that matches ($ (?x ^ ?n) d ?x). After successfully matching the sub-expression of the current expression, the list of predicates will test further constraints on the bindings. In the above rule, it will check whether ?n is a number and whether number is not -1. If the test passes, then, the RHS in the above rule will do two things: (1) (add-binding *tea-ele-rs* ?n+1 (1+ ?n)) assigns variable ?n+1 to be 1 + ?n and finally (2) substitutes the matched sub-expression of the current expression by ((?x ^ ?n+1) / ?n+1). Below is a rule the companion initially acquires. Notice that this is an erroneous rule in which the companion forgets to check the index of ?x (if the index is -1, the rule should not apply), a common mistake by a student.

```
(rrule *com-ele-rs*
    ($ (?x ^ ?n) d ?x)
    :name "Rule 1"
    :var ?int
    :test (numberp ?n)
    ->
    (add-binding *com-ele-rs* ?n+1 (1+ ?n))
    (subst! *com-ele-rs* ((?x ^ ?n+1) / ?n+1) ?int))
```

Running a term rewriting system simply tests every term rewriting rule from the rule base. Whenever a rule is matched, the current sub-expression is rewritten. As can be seen, our term rewriting rules are close production rules. In a term rewriting system, there is only one datum, the current expression, in the working memory. Also, the action of RHS of a rewriting rule is only to change the current expression. Thus, the term rewriting system's function is more particular but simpler than a

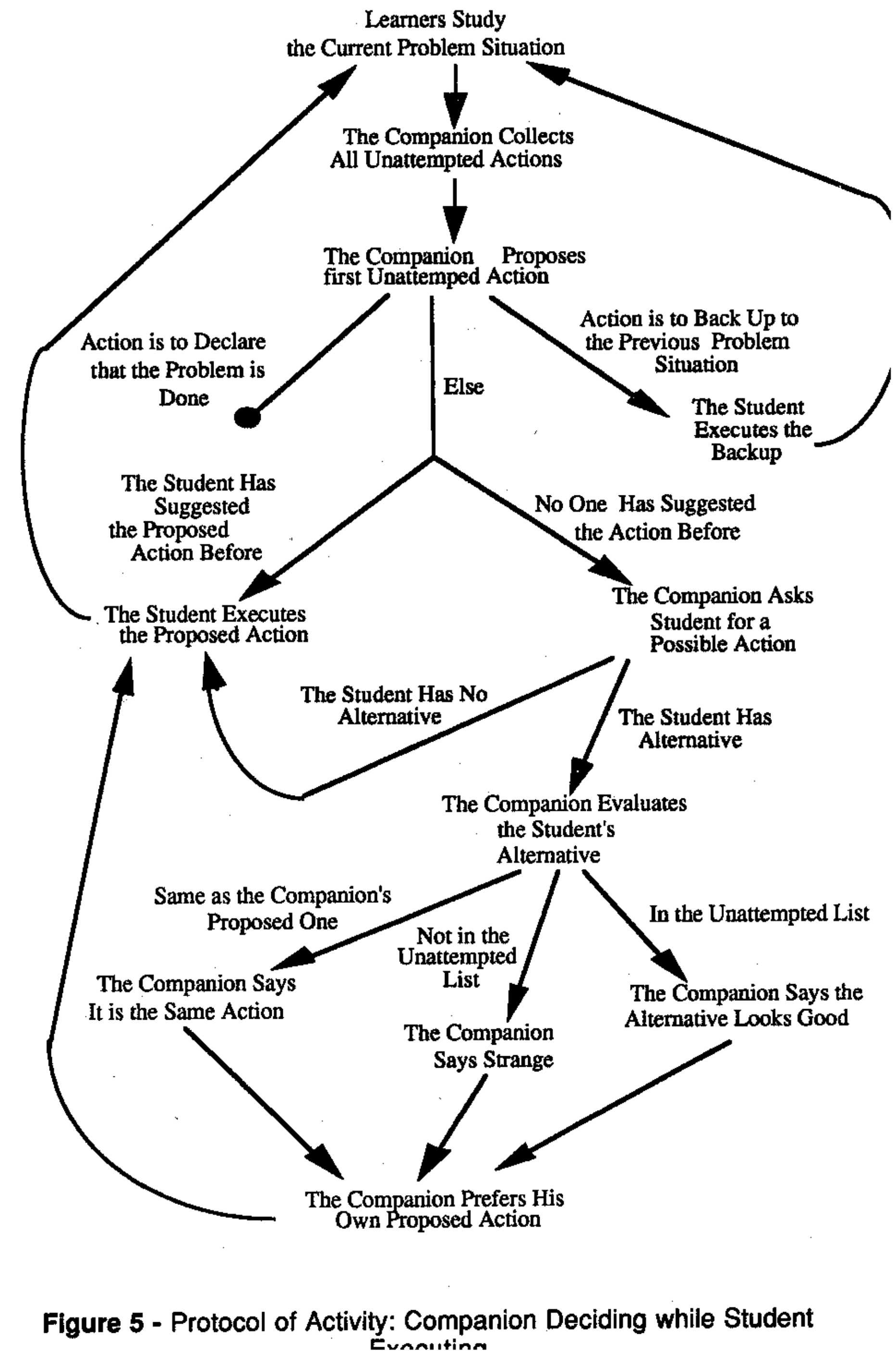


Figure 5 - Protocol of Activity: Companion Deciding while Student Executing

4.5 Situation Map — a Representation of Problem Solving Situations

Some protocols of learning activities among the three agents are defined. These protocols reflect the different learning stages of the learners in the learning process and appropriately restrict their possible unbounded interactions. For example, when the learners have equipped with most techniques, they can then start solving complex miscellaneous problems. Miscellaneous problems are problems that the learners cannot expect to have any similar pattern. This is in contrary with problems that focus on practicing a particular technique (e.g. substitution), and the problems are usually similar with increasing complexity. When the learners work on miscellaneous problems, they have to judge whether or not the current problem situation has improved by the method. If the situation is improving, what will be the next method? Otherwise, should the problem be backed up to the very beginning, or just to the last situation? The protocol at that stage is *one deciding and one executing* (Figure 5) where different situations, including the past stages of solving a problem, are needed to be taken into account for making decisions to apply which method. Plain facts or simple data such as a list of alternatives in the blackboard can neither capture the relations of different situations nor connect the current situation with the past situations in solving the problem.

Another concern of working on complex problems is the addition of social context to the current situation, for example, making a comment like "We are getting close." Such social context is usually situation specific to a particular problem. Moreover, personal subjective feeling, speculation, or even inspiration could possibly contribute to such social texture. Neither encoding such context within the rules of behaviour which simulate the protocol of activities for a general set of problems nor using a simple data structure will suffice to represent such particular knowledge.

Situation map is the representation we attempt to meet the needs mentioned above. Its data structure is defined as follows:

```
(defstruct  sit
 (problem nil)
 (sub-problem nil)
 number
 (visited-time 0)
 (paths nil))

(defstruct path
 from
 to
 (traversed? nil)
```



```
(defstruct action
 type
 (proposed-agent 'no-one)
 (sub-exp nil)
 (simp-steps nil)
 (pre-comment nil)
 (post-comment nil))
```

The structure *sit* stands for situation. The slot *problem* is the current mathematical expression the learners are working on. If the slot *sub-problem* is not nil, then it is a sub-expression to focus on instead of the problem, for example, if $-\cos x \sin 2x + 2\int \cos x \cos 2x dx$ is the problem, then the sub-problem is $\int \cos x \cos 2x dx$. *Number* is the identity slot for the *sit* structure. *Visited-time* records the number of times that the situation has been visited. The most important slot is *paths*. It is a list of outgoing paths to other situations.

For the structure *path*, the slots *from* and *to* point to the situations where the path started from and is going to respectively. *Traversed?* records whether the path has been traversed. *Action* is the actual action which changes the situation pointed to by the 'from' slot to the situation pointed to by the 'to' slot. Each path has an action associated with it and vice versa.

There are five types of actions for the structure *action*. The types *substitution* and *by-parts* are two typical actions to transform the current problem situation by substitution method or by integration by parts method to another problem situation. The type *back-up* backs up the current situation to the one pointed to by the 'to' slot of the path which the action is associated with. Another type, *problem-done*, is just to notify that the problem is done and the corresponding path points to the same situation as it started from. The last type of action, *unexpected*, is an interesting action. This action denotes the action the student decides to take in the current situation. So the corresponding path will go to a particular situation which has only one path out with the action back-up to the previous situation since the unexpected action will not be a useful one. The proposed-agent is the agent who proposed the action, initially, no one. If the action type is substitution, the *sub-exp* slot will be the substitution expression. For by-parts action, the sub-exp slot is the list of u and dv expressions. Otherwise, it is nil. At this stage, we assume that the simplification of mathematical expressions involved in these two types of actions are well mastered by the student and the companion, thus it is not treated as a type of action. The *simp-steps* slot contains simplified steps of the problem after the action. This can be viewed as a utility or as a part of the teacher's support. Finally, the *pre-comment* and *post-comment* are some comments made by the companion and/or the teacher which is the social texture for the action

Figure 6 is a situation map for the problem $\int e^x \sin x dx$ where the companion decides and the student executes.

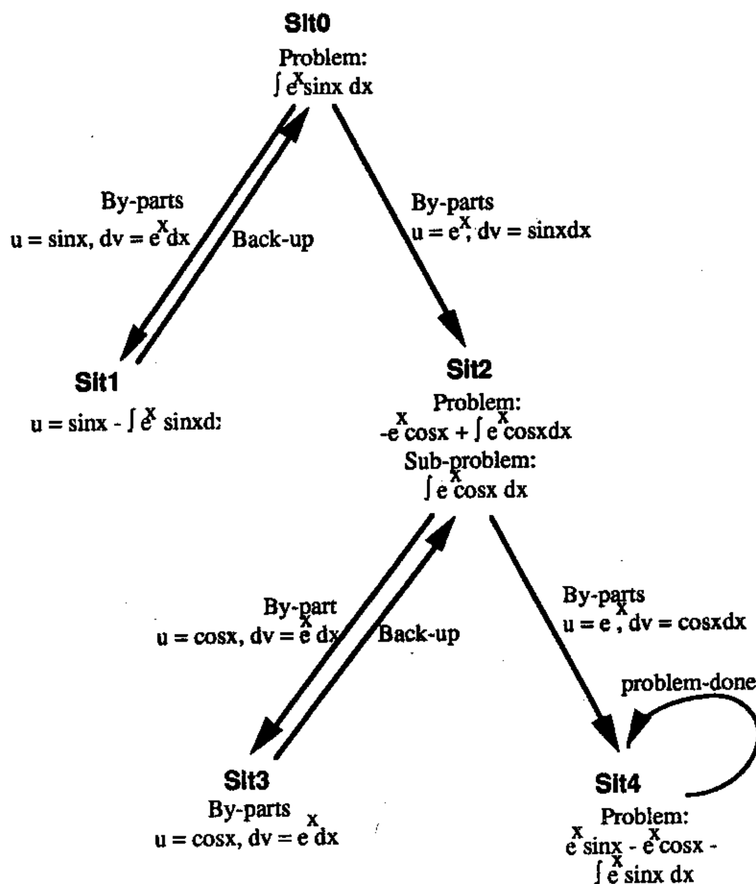
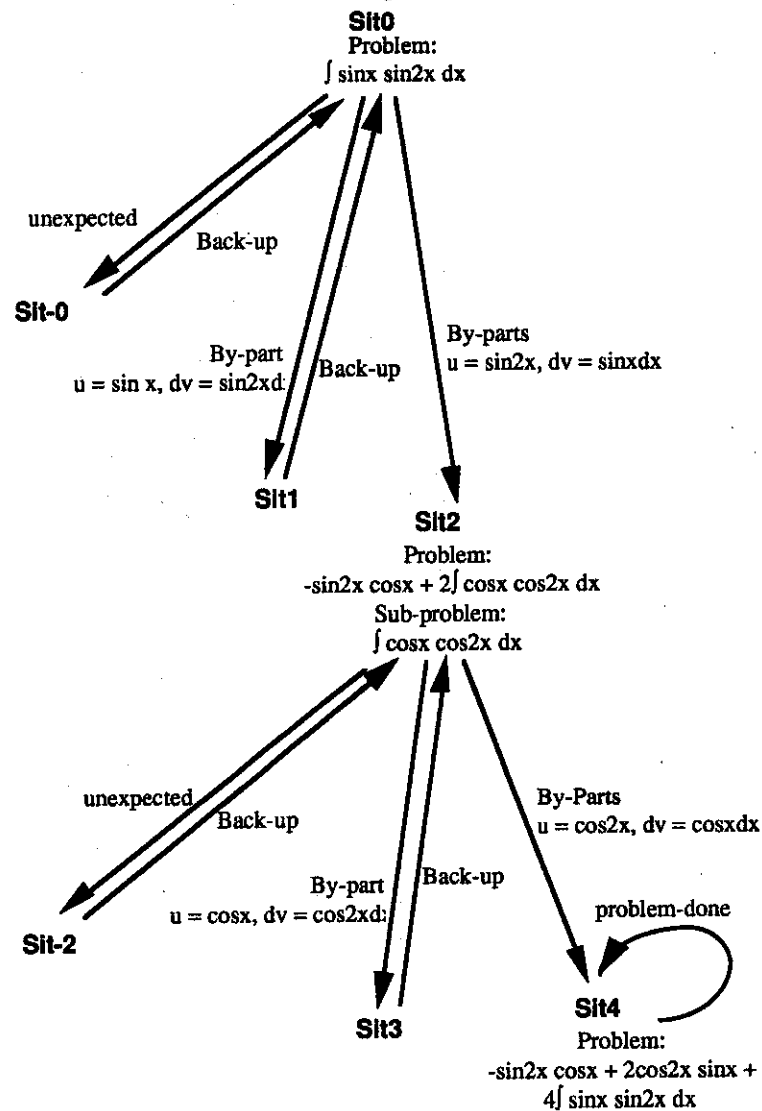


Figure 6 - A Situation Map Where the Companion Decides

At the beginning, the companion uses integration by parts with $u = \sin x$ and $dv = e^x dx$. He finds that there is no improvement. He decides to back up and try integration by parts with a different u and dv, but it turns out to be a similar situation as before. Now the teacher interrupts and asks them to carry on. The companion then tries one of the plausible actions which is surprisingly reduced to the original problem. The companion then decides to back up and tries another action which goes to a situation which is very close to having the problem solved but the learners may not recognise this. Now the teacher jumps in and

some algebraic manipulation, the problem is solved. The final action *problem-done* is to signal to the system that the episode has finished. The timely interrupts and the final wrap up by the teacher are necessary in that difficult problem. Notice that when the companion decides and the student executes, all the paths of the situation map are traversed because the design of the protocol controls the companion to only prefer his own choice. On the other hand, we can say that because of such a companion's character, we do not need to lay out every possible situation and thus the situation map is not very large.



Another example of the situation map (shown in Figure 7) is the case where the student decides and the companion executes, the subsequent problem of the previous one. At the beginning, because of the previous experience, there are two actions that the companion expects the student to choose. Any other action, like the substitution method, is regarded as unexpected. Executing an unexpected action will go to situation **sit 0** which will have to back up. For the other expected actions, taking integration by parts with $u = \sin x$, $dv = \sin 2x$ will need to integrate $\int \sin 2x \, dx$ which is not that simple and the companion will ask to back up. If the student decides the right choice, the problem situation will move to **sit 2**. Now, the focus is the sub-problem $\int \cos x \cos 2x dx$. Again, there is an unexpected action path leading to **sit 2**. If the student takes an expected but not useful choice, at some point of execution, the companion will call for a back up. Finally, with the suggestion by the companion, the student will make the right choice.

Unlike the previous case where the companion is to decide, only part of the situation map may be traversed since the student may decide to take all the right actions and get the answer. Also, in this case, unexpected paths are needed for the student may try unexpected actions. We do not need unexpected paths in the previous case just because of the stubborn character of the companion.

With the use of the situation map, the rules of behaviour for simulating the protocol can focus on the negotiation process and, in fact, becomes an interpreter of a situation map. Also, because the situation map includes particulars of a problem, rules of behaviour for each problem are significantly simplified.

4.6 Student's Bug Handling

In learning, a student's responses always are sometimes correct or sometimes buggy. If the student's responses are always correct, then he does not need to learn; but, if the student's responses are always buggy, it is doubtful that the system is useful to the student. No matter whether buggy or correct, the student's response is either expected or unexpected to the system and has to be dealt with. Thus, the student's input can be viewed by the system as either buggy or correct, expected or unexpected (Figure 8), even though sometimes such a division is not clear. From Figure 8, the smoothest situation is that the student's input is correct and expected. And it is desirable to reduce unexpectancy of the student's response to the system; in particular, the correct unexpected student's response.

In Integration-Kid, when the student and the companion interact intensively such as in learning advanced methods, whether a learner's move is correct or buggy is the subject of their discussion. Thus, not only

only is the computer responsible for discovering buggy moves, the human student is also. However, the partition of Figure 8 is still a valuable framework to discuss how Integration-Kid handles the student's errors; for example, typing mistakes may be frequently made by the student but not the companion.

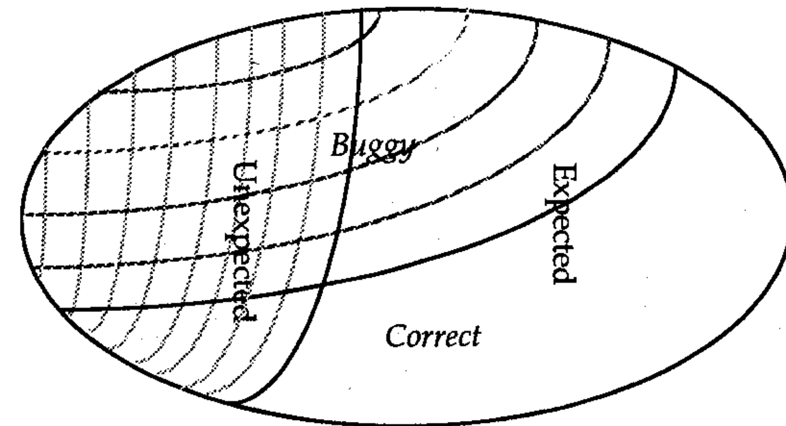


Figure 8 - Student Responses Divided by Correctness and System's Expectancy

A way that the Integration-kid reduces unexpectancy is to limit the student's input to be one of the choices on a request menu. Any choice by the student is then expected by the system assuming that the student understands the meaning of the choices and has clicked the choice he intended. But still, there are cases where the student regrets his choice right after he has clicked on a choice.

In the early stages of learning in Integration-Kid, the student's expected buggy response is modeled as a part of the companion's behaviour e.g. applying a buggy rule. In some ITS, like CMU tutors, expected bugs are modeled by the buggy library. When the student makes such an error, in addition to the teacher's indication and explanation, the student can compare his bug with that of the companion's. Also, there are situations where the student's error is partially expected by the system, although the system cannot tell or explain exactly what is wrong. Typing mistakes in writing mathematical expressions are such an example. In that case, the system locates the sub-expression that it cannot understand by using the expression parser. Another example is to use constraint knowledge to inform the student of his mistake. For example, if the student chooses the parts $u = \cos x$ and $dv = e^x dx$ for the problem $\int e^x \sin x dx$. The system

$e^x \sin x dx$. This is helpful information for reminding the student that his choice is limited under such a constraint. With such information, it is possible that the student may get the right choice of u and dv ($u = \ln x$ and $dv = dx$) for a tricky problem, $\int \ln x dx$.

It is always possible that a student's response is totally out of expectation of the system. Learning under a rich social context such as Integration-Kid, the system can directly tell the student that his response cannot be understood. This makes sense to the student because, based on the history of their learning interaction, the computer agents cannot understand the student's current response. By the same token, we can expect that the student may understand that the system cannot understand his response if the system's ability to understand the student's situation is more or less the same as a teacher's. Thus, it is perfectly fine in learning with a social context, that the system tells the student it cannot understand his response whenever it is out of the system's expectation. However, a student's response that is out of the system's expectation is not necessarily incorrect. Suppose the expected correct response is 4, it is not impossible that the student puts down 'four' or 'IV', an example of unexpected correct student response.

Another way to reduce the unexpectancy is to identify as much as possible the student's expected correct response by using the domain knowledge. But there are cases where this is not a very simple job. In Integration-Kid, when learners are working on simple problems independently, the student writes down a sequence of mathematical expressions as his solution. His solution can be long or short for he may skip steps or even just write down the final answer. The way the Integration-Kid traces the steps of the student's solution is as follows. Suppose the first step is the original problem, then the teacher will use the set of elementary rules to generate a comprehensive sequence of steps as a solution. Each of these steps and its simplified version is matched against the student's next step or its simplified version. That is, the set of all the steps of the teacher's solution and their simplified version are the potential candidates to match the student's next step. If the student's next step does not fall into this expectancy, then the teacher informs the student that he does not understand the step. If the matching is successful, then the current step will move downwards. Thus, the current step becomes the second step and the next step will be the third step. The same mechanism applies again. Finally, if the last step of the student's correct response still has an integral sign, the teacher will ask the student to continue; or, if the last step can be further simplified, the teacher will ask the student to simplify.

In Integration-Kid, the teacher's scaffolding can substantially reduce the student's errors when applying background knowledge. We do not model errors in the companion's background knowledge. If the

For the student, any mistake related to the background knowledge is regarded as a careless mistake and is unexpected to the system. The teacher would say "don't understand". Because of the scaffolding by the teacher and the availability of references of background knowledge, errors on the parts of a task that involve background knowledge are minimised.

4.7 Parsing of Mathematical Expressions

In this last session, we discuss a parsing algorithm which transforms a one dimensional infix expression to a two dimensional expression. For example, when the student types in ($ (1 / (x ^ 2)) d x), the system will echo $\int \frac{1}{x^2} dx$. Such algorithms have been developed in some systems such as Mathematica and Macsyma. But unfortunately, they have not been available in the literature while they are an essence for mathematics ITS. Of course, some systems may provide a better input environment for the user and in order to use the technique described in this session, there may be an additional parsing of the user input expression to the one required by the algorithm.

The best way to describe the parsing algorithm perhaps is by demonstrating a rather complex example with the explanation of a list of parse trees. The input of the example we consider is :

```
(($ (((sin ((x - 1) / 3)) + ((x - 1) ^ 4)) / (cos (((x - 1) /
    3) ^ 5))) d x) + ($ ((1 / x) ^ (1 / 2)) d x)))
```

The intended output will be an array of strings of the same length which when arranged as a rectangle will look like :

```
"   X - 1       4     1 "
" Sin[-----] + (X - 1)       - "
"    3              1 2 "
"∫[--------------------]dX + ∫[-] dX"
"     X - 1 5       X  "
"   Cos[-----]           "
"      3          "
```

The main observation is that the output expression will be more than one line when either there exists a sub-expression which is a fraction or an index. That is, when a sub-expression involves the symbol / or ^. The first phase of parsing is to obtain a parse tree from the input expression. All the largest sub-expressions of the input expression of the form (?numerator / ?denominator) or (?base ^ ?power) are matched.

These sub-expressions are labeled as nonterminals[1] / or ^ respectively. The rest of the input expression is translated into strings as terminals. Thus, a sequence of strings and nonterminals is obtained which is the root node of Figure 9.

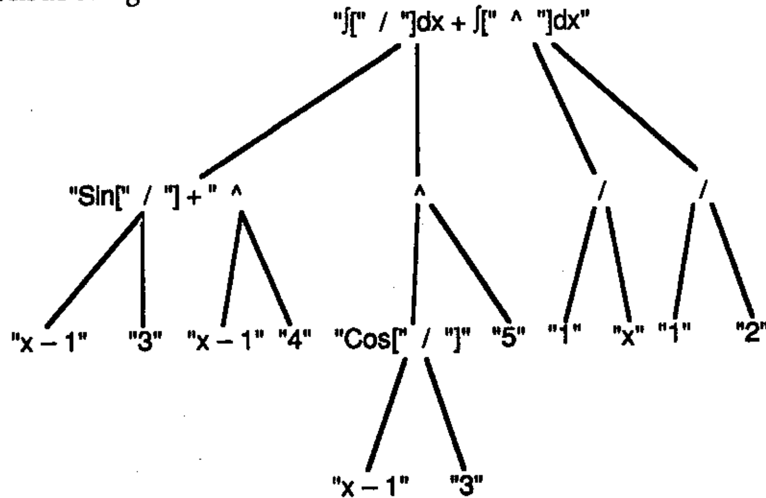


Figure 9 - Parse Tree of the Infix Expression

Those strings will be on the same line in the final form of the two dimensional expression. For the nonterminal /, the corresponding sub-expression is split into two parts, one for the numerator and the other for the denominator, and, similarly, the nonterminal, ^, is split into the base and the power. Each of the split parts of the sub-expression are parsed in the same fashion recursively until a parse tree as in Figure 9 is obtained where all the leaf nodes are strings.

After getting the parse tree, the second phase of parsing begins. In this phase, strings are combined together to form rectangular arrays of strings in a bottom up manner. First, go down to the smallest left-most subtree and collect the leaf nodes which are our first focus (the shaded rectangle of Figure 10). Since their parent is a non-terminal, /, we add a string of bar characters forming a line to separate the numerator and the denominator. For the string "3", we add spaces on both sides so that a rectangle of strings, as shown in Figure 11, representing $\frac{x-1}{3}$ is formed. Now three consecutive rectangles of strings appear on the left bottom of the tree. These consecutive rectangles of strings are then merged to form

---

1 Terminal and non-terminal are common terminology used in compiler design. A

a larger rectangle, $\sin[\frac{x - 1}{3}]+$, as shown in Figure 12. Next to that rectangle is a non-terminal, /. Again, go down to the smallest left most subtree of the non-terminal, ^, and combine the leaf nodes (the shaded rectangle of Figure 12) of that subtree in a similar bottom up fashion. But this time the sub-expression is an index, a rectangle of two strings representing $(x - 1)^4$ is formed (Figure 13). Then two consecutive rectangles appear again and they then merge together to form another larger rectangle (Figure 14). Continue this way until a final rectangle of strings is obtained.

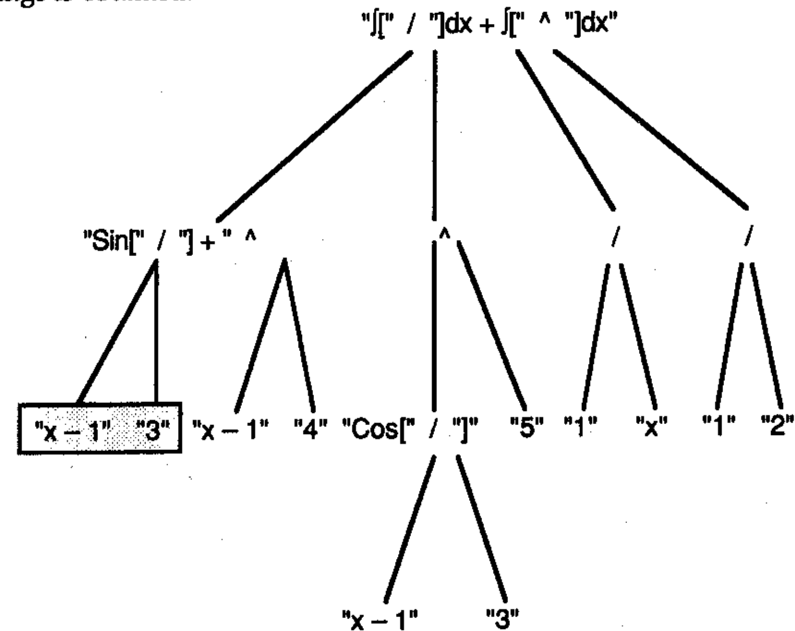


Figure 10 - Focus the Leaf Nodes of the Smallest Left-Most Subtree

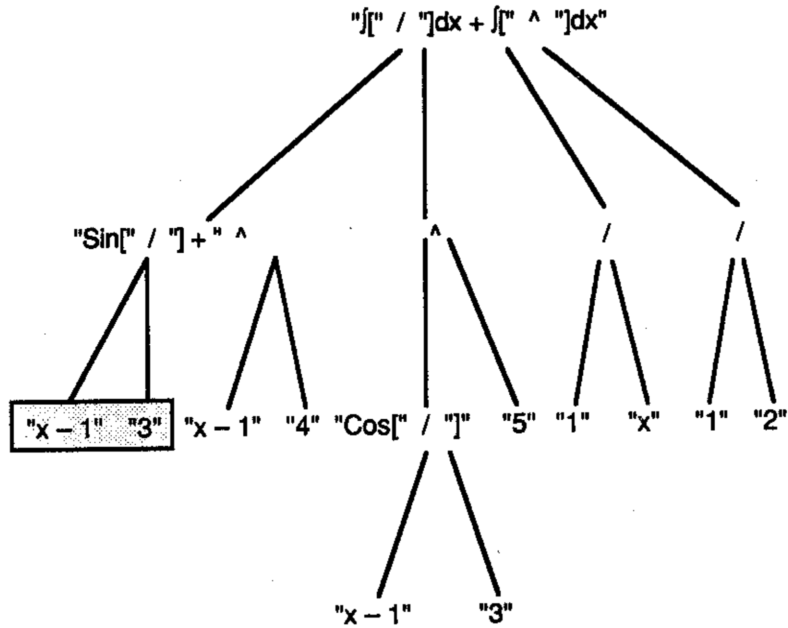


Figure 11 - Three Consecutive Rectangles of Strings are Obtained

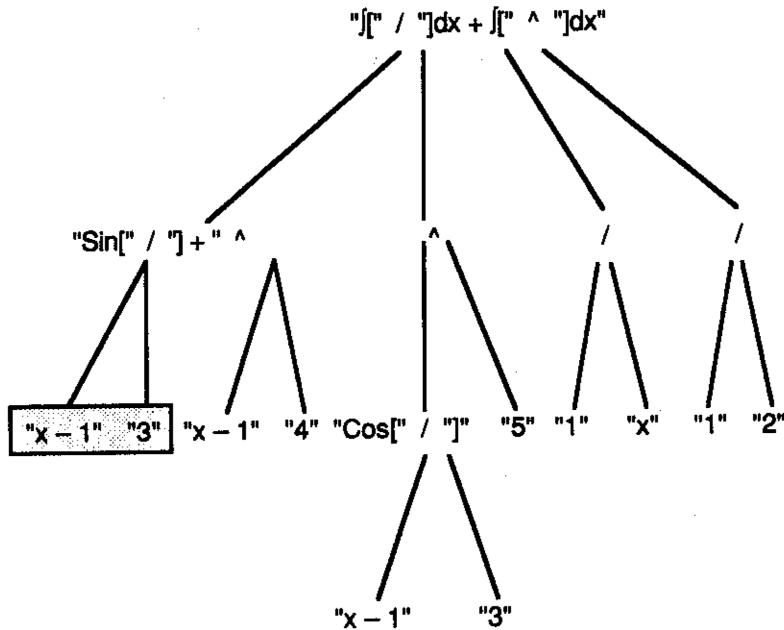


Figure 12 - Merge the Three Rectangles and Focus to the Non-leaf

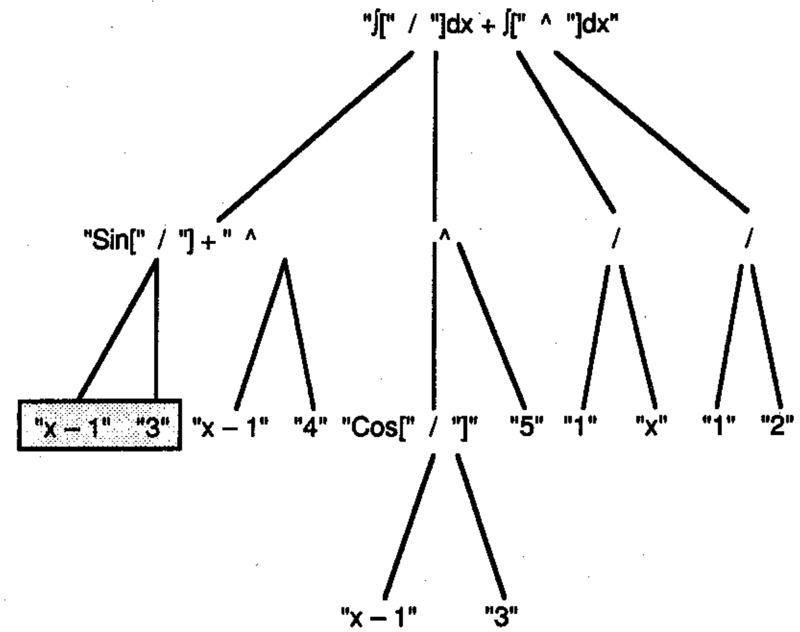


Figure 13 - Two Rectangles are Obtained Again

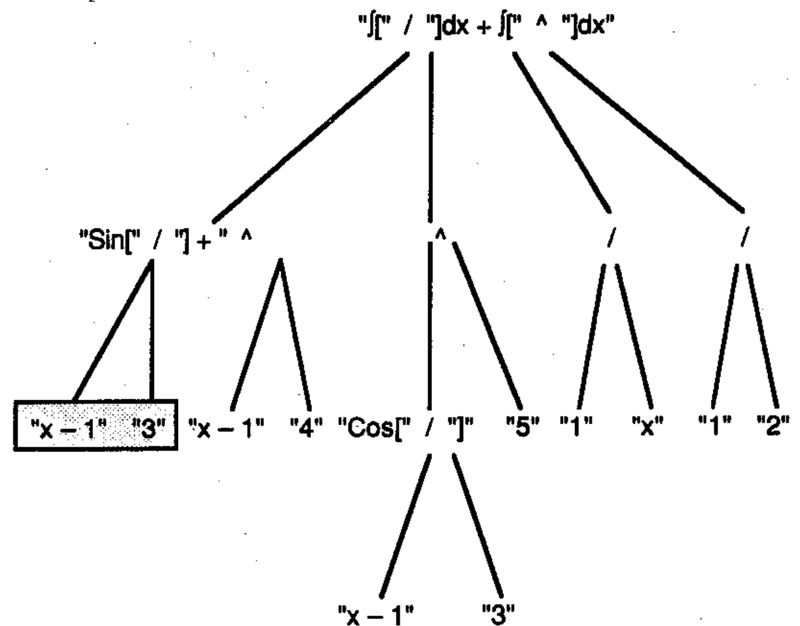


Figure 14 - Merge of the Two Rectangles

In order to merge rectangles correctly, the position of the centre line for each rectangle of strings is recorded. A rectangle with only one string has centre line position 0. A rectangle of a simple fraction of three line has centre line position 1, that is, the middle line separating the numerator and the denominator is the centre line. A rectangle of an index of two lines has centre line position 0, that is, the base is the centre line. When the rectangles merge, all the centre lines of the rectangles line up. This line up is achieved by adding space lines above and/or below some of the rectangles so that all the rectangles have the same height and the same position in the centre line. All these rectangles are then put together forming a resultant rectangle with the centre line updated. Interested reader may request the Lisp code from the author.

4.8 Conclusion

In this paper, we have discussed several critical techniques for building Integration-Kid, an ITS in the domain of indefinite integration. A detailed example use of the system is described in (Chan, 1992b). Integration-Kid has been tested by a small number of individual case studies for verification and revision of the system, but has not been fully tested or evaluated. However, we discover that the human students seem to be very curious about the companion's response and pay more attention to it than the teacher's. Recently we have implemented another LCS (distributed version) by rebuilding the WEST program (Burton & Brown, 1979) for learning binary numbers (Chan *et al.* ,1992). But the companion is another human student, not the computer simulated companion, playing at another connected computer. That is, two human students are companion to each other, using the system via two connected computers. The new system is currently used to evaluate learning effects of ITS environment that are alternative to on-on-one tutoring.

5

The anatomy of FITS: A Mathematic tutor

HYACINTH S. NWANA
Univesity of Keele, U.K.

5.1 Introduction

It seems accurate to remark that most papers on the numerous published ITSs provide enough information to understand it but not reconstruct it. To be fair, this has resulted due to the sensible publication restrictions on length, depth, etc. This has in turn resulted in mainly overviews of systems being published. However, good and progressive research requires others being able to understand fully and even reconstruct our systems so as to be able to evaluate them. Evaluation is still a relative taboo activity in AI research; it is our belief that if others were able to reconstruct the systems then more of them could be appraised. It is currently the case that most systems remain as prototypes which are rarely evaluated. Since these prototypes also seldom go out of the laboratories in which they are developed, their research value is largely lost, unless the researchers continue the work. However, if enough was published of the system to as to be reconstructable, it is likely the research value of these systems may be preserved as *much more* other researchers may base their future research on what others have accomplished in theirs. The benefits to be accrued by reporting more of our systems can not thus be overstated.

This paper follows in Ford's (1987b) example of reporting the anatomy of his TUTOR system. The key intention is to provide the knowledgeable reader with enough details to be able to largely recreate the fractions tutor, FITS.

5.2 Overview of FITS

5.2.1 History

Mathematics has provided a very suitable domain for intelligent