In Turner, T., Szwillus, G., Czerwinski, M. and Paterno, F. (Eds.) Proceedings of ACM CHI'2000 Conference on Human Factors in Computing Systems, 97-104. New York: ACM Press.

Instructional Interventions in Computer-Based Tutoring: Differential Impact on Learning Time and Accuracy

Albert Corbett
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412 268 8808
corbett@cmu.edu

Holly Trask
American Management Systems
12601 Fair Lakes Circle
Fairfax, VA 22030 USA
+1 703 227 4885
holly_trask@amsinc.com

ABSTRACT
We can reliably build "second generation" intelligent computer tutors that are approximately half as effective as human tutors. This paper evaluates two interface enhancements designed to improve the effectiveness of one successful second generation tutor, the ACT Programming Tutor. One enhancement employs animated feedback to make key data structure relationships salient. The second enhancement employs subgoal scaffolding to support students in developing simple programming plans. Both interventions were successful, but had very different impacts on student effort required to achieve mastery in the tutor environment and on subsequent posttest accuracy. These results represent a step forward in closing the gap between computer tutors and human tutors.

Keywords
Intelligent Tutoring Systems, Instructional Interface Design, Animation, Plan Scaffolding, Student Modeling

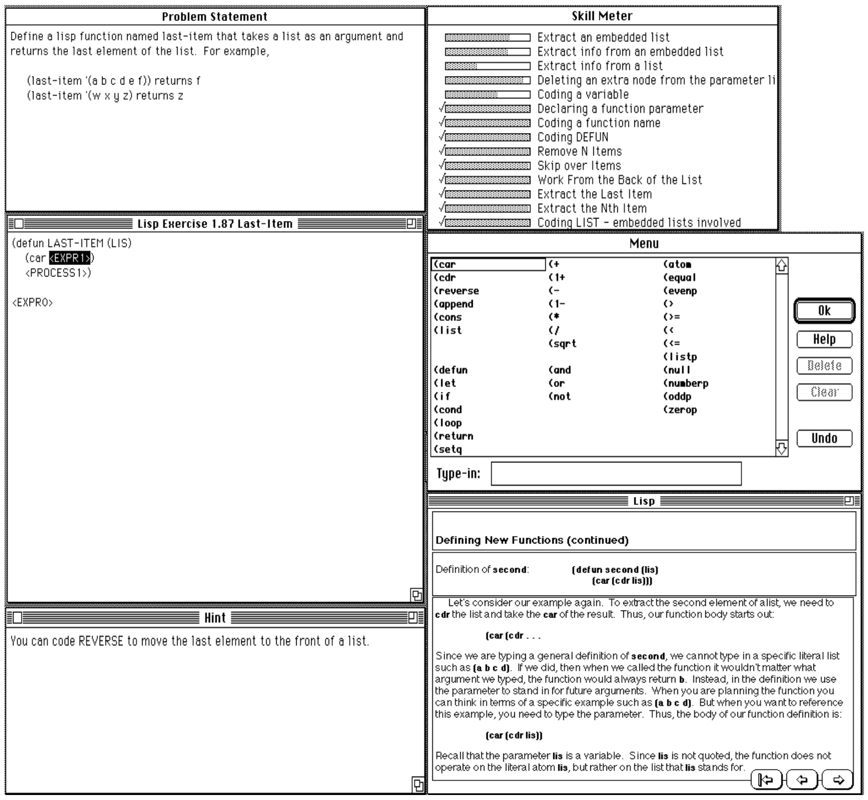
INTRODUCTION
Computer-based learning environments first appeared three decades ago that afford one advantage of human tutors: individualized interactive learning support. Fifteen years ago "second generation" computer tutors began to appear that incorporate artificial intelligence technology and we can reliably build intelligent tutors that are about half as effective as human tutors. How will we develop "third generation" tutors that approach the effectiveness of human tutors? A significant effort to bridge the gap focuses on natural language dialogs between student and tutor [8, 12]. In contrast, this paper evaluates two enhancements embedded directly in the problem solving interface. These enhancements are designed to increase the educational efficiency of *cognitive mastery learning* in the ACT Programming Tutor (APT).

APT is a computer-based problem solving environment in which students learn to write short programs. It is a *cognitive tutor* that employs a detailed cognitive model of the programming knowledge students are acquiring to support students in learning. Cognitive tutors [2] are designed to (a) provide students an authentic problem solving interface, (b) provide assistance if needed on each problem solving action, (c) monitor the students growing knowledge in the course of problem solving and (d) provide sufficient learning opportunities for the student to reach mastery.

As described in the first two main sections of the paper, APT is a highly successful learning environment. However, there is evidence that providing more and more problems in the standard problem solving interface ultimately yields diminishing educational gains and some students fall short of genuine mastery. This paper reports two interface modifications that were developed in response to this evidence. As described in the third and fourth main sections these interventions are introduced early in the Lisp curriculum when students confront the first two substantial learning challenges. The first intervention employs animated feedback to help clarify basic operator functionality. The second intervention employs subgoal scaffolding to support students in planning very simple algorithms. The final sections report an empirical evaluation of these enhancements. The two interface interventions proved to be successful, but had very different impacts on learning.

THE ACT PROGRAMMING TUTOR
APT is a *cognitive tutor* which helps students learn to write short programs in Lisp, Prolog or Pascal. Each of these three programming language is constructed around a language-specific cognitive model of the knowledge the student is acquiring. The cognitive model enables the tutor to trace the student's solution path through a complex problem solving space in a process we call *model tracing*. The tutor provides feedback on each problem solving action and, if requested, provides advice on steps that achieve problem solving goals.

Figure 1 displays the APT Lisp Module midway through an exercise. The student has previously read text presented in the window at the lower right and is completing a sequence of corresponding exercises. The problem description appears in the upper left window and the student's solution appears in the code window immediately



Define a lisp function named last-item that takes a list as an argument and returns the last element of the list. For example,

```
(last-item '(a b c d e f)) returns f
(last-item '(w x y z) returns z
```

✓ Extract an embedded list
Extract info from an embedded list
Extract info from a list
Deleting an extra node from the parameter li
Coding a variable
✓ Declaring a function parameter
✓ Coding a function name
✓ Coding DEFUN
✓ Remove N Items
✓ Skip over Items
✓ Work From the Back of the List
✓ Extract the Last Item
✓ Extract the Nth Item
✓ Coding LIST - embedded lists involved

```
(defun LAST-ITEM (LIS)
  (car <EXPR1>)
  <PROCESS1>)

<EXPR0>
```

You can code REVERSE to move the last element to the front of a list.

Defining New Functions (continued)

Definition of **second**: **(defun second (lis)**
(car (cdr lis)))

Let's consider our example again. To extract the second element of a list, we need to **cdr** the list and take the **car** of the result. Thus, our function body starts out:

(car (cdr . . .

Since we are typing a general definition of **second**, we cannot type in a specific literal list such as **(a b c d)**. If we did, then when we called the function it wouldn't matter what argument we typed, the function would always return **b**. Instead, in the definition we use the parameter to stand in for future arguments. When you are planning the function you can think in terms of a specific example such as **(a b c d)**. But when you want to reference this example, you need to type the parameter. Thus, the body of our function definition is:

(car (cdr lis))

Recall that the parameter **lis** is a variable. Since **lis** is not quoted, the function does not operate on the literal atom **lis**, but rather on the list that **lis** stands for.

Figure 1. The APT Lisp Tutor interface.

below. The student selects operator templates and types constants and identifiers in the user action window in the middle right. In this figure the student has encoded the operator *defun* used to define a new operator, has entered the operator name, declared input variables and begun coding the body of the definition. The three angle-bracket symbols in the figure, <EXPR1>, <PROCESS1> and <EXPR0> are placeholders which the student will either replace with additional Lisp code or delete. Communications from the tutor appear in the Hint window in the lower left. In this figure the student has asked for a hint on how to proceed.

Cognitive Tutor Effectiveness

We have been developing cognitive tutors for mathematics and programming at Carnegie Mellon for more than a decade. APT has been used to teach self-paced programming courses here since 1984. More recently we have developed cognitive tutor-based algebra and geometry courses that are in use in more than 100 schools around the country. We have completed several summative evaluations comparing the effectiveness of cognitive tutors and traditional problem solving activities. In research which compared college students working with APT to those completing the same programming problems in a conventional programming environment, APT was shown to speed learning by as much as a factor of three and yield small but reliable increases in test performance [2]. Evaluations of the cognitive mathematics tutors have compared classroom use of the cognitive tutors to conventional classroom problem solving activities (seatwork and blackboard work). Time on task is constant in these studies and students in the cognitive tutor condition score about one standard deviation higher (or about a letter grade higher) on achievement tests than students in the conventional classroom condition [2, 9]. A one standard deviation effect size is two or three times larger than the average obtained by conventional computer-based instruction [5], and about half as large as the effect size that can be achieved by individual human tutors [4].

KNOWLEDGE TRACING: COGNITIVE MASTERY

Following the ACT-R theory of skill knowledge [3], the ACT Programming Tutor assumes that goal oriented problem-solving knowledge can be represented as a set of independent production rules that associate problem states and goals with problem solving actions and consequent state changes. For example, Table 1 displays two productions that are acquired early in the Lisp curriculum. The tutor curriculum is structured around production sets. In each curriculum section students read text that introduces a small set of production rules. Then the tutor provides problems that exercise those rules. As the student works, the tutor monitors the student's growing knowledge in a process we *call knowledge tracing*. The tutor employs some simple learning and performance assumptions and a Bayesian computational procedure to estimate the probability that the student has learned each of the productions. As each opportunity arises to apply a rule, the tutor updates its estimate of the probability that the student has learned the rule, contingent on the student's action. (See [6] for more details). The Skill Meter in the upper right corner of Figure 1 depicts the tutor's model of the student's knowledge state. Each entry in the Skill meter represents a production rule in the underlying cognitive model and the shading represents the probability that the student knows the rule. A check mark indicates that the student has mastered the rule.

IF the goal is to return the elements of a list in
    reverse order,
  THEN code the function *reverse*, and set a goal to
    code the list as an argument.

  IF the goal is to insert an expression at the beginning
    of a list,
  THEN code the function *cons*, set a goal to code the
    expression as the first argument, and set a goal to
    code the list as the second argument.

Table 1: Two early productions in the Lisp curriculum.

This knowledge tracing process is employed to implement cognitive mastery learning in APT. In each curriculum section the student completes a small fixed set of required problems that introduce all the productions being traced. The tutor then continues providing problems in the section until the student reaches a criterion probability for each rule in the set. This mastery criterion in the tutor is a probability of 0.95.

Cognitive Mastery Learning Successes

Cognitive mastery has proven effective in APT, by several criteria: (a) Average posttest scores are reliably higher for students who work to cognitive mastery than for students who complete a fixed set of required tutor problems [1]; (b) Similarly, twice as many cognitive mastery students reach "A" level performance (at least 90% correct) on posttests in the early Lisp curriculum [6]; (c) Individualized remediation reduces total time invested by a group of students in reaching mastery. (The number of remedial problems varies substantially across students and, in the absence of such individualization, mastery could only be ensured if all students complete the greatest number of problems any student needs); and (d) The knowledge tracing model that guides mastery learning in the tutor reliably predicts individual differences in posttest performance across students [6].

Cognitive Mastery Learning Shortcomings

Nevertheless, there are noteworthy shortcomings in cognitive mastery learning: (a) A substantial proportion of students in the cognitive mastery condition still fall short of "A" level performance in the early Lisp curriculum [6]; (b) Students are investing substantial amounts of time and effort to reach mastery. In a typical study, students in the cognitive mastery condition completed an average about 75% more problems than students who completed the fixed set of required tutor problems [6]; (c) There is an inverse correlation between the number of problems students required to reach mastery and students' test performance. The students who struggle the most and complete the most problems do not do as well on the test as students who perform well in the tutor [6]; and (d) The tutor's knowledge tracing model slightly but systematically overestimates average test performance. Evidence suggests that this occurs because some students are learning sufficient, but suboptimal productions [7].

These shortcomings represent converging evidence that providing more and more remedial problems of the same type yields diminishing educational returns. In this study we explore two forms of augmented support in the problem solving interface designed to speed learning and increase asymptotic test performance in early sections of the Lisp curriculum.

AUGMENTED SUPPORT FOR EARLY CHALLENGES IN THE LISP CURRICULUM

This study focuses on the first four sections in the APT Lisp curriculum. These sections introduce (a) two data types, *atoms* (symbols) and *lists* (hierarchical groupings of symbols), (b) three "extractor" functions, *car*, *cdr* and *reverse*, that take a single list as an argument and return a component or transformation of the argument, (c) three "combiner" functions, *append*, *cons*, and *list*, that take multiple arguments and return new lists, and (d) the syntax and functionality of simple algorithms involving embedded function calls. Table 2 displays sample exercises from each of these sections.

Section 1: Write a lisp function call that takes the list
        (c d e) and returns (d e)
  Solution: (cdr '(c d e))

Section 2: Write a lisp function call that takes the
        arguments (a b) c (d) and returns
        ((a b) c (d))
  Solution: (list '(a b) 'c '(d))

Section 3: Write a function call that takes the list
        (a b c d) and returns the last element, d.
  Solution: (car (reverse '(a b c d)))

Section 4: Write a function call that takes the lists
        (a b c) and (d e f) and returns (a f e d).
  Solution: (cons (car '(a b c)) (reverse '(d e f)))

Table 2. A sample problem in each of four tutor sections.

Students have little difficulty with the three extractor functions, *car*, *cdr* and *reverse* and with extractor algorithms in section 3. However, many students struggle with the differences among the three combiner functions, *append*, *cons* and *list* in section 2 and with the combiner/extractor algorithm problems in section 4. The two instructional interventions described below focus on these two curriculum sections.

Supporting Data Structure Parsing: Animated Feedback

In curriculum section 2 students are learning to distinguish among three combiner functions and the subtlety of this distinction is captured in the text excerpt displayed in Table 3. There are multiple reasons that this discrimination is difficult. First, the terminology is difficult, with semantically related and confusable terms for novices – atoms, elements, arguments, expressions. Second, students have trouble grasping the hierarchical structure of lists. They may not understand that they need to analyze the structural relationship among the input arguments and results and, finally, students have trouble visually parsing the parentheses when they do try to analyze these structural relationships.

Study the following examples to see how **list**, **append** and **cons** are distinguished from each other.

(list '(a b) '(c (d e) f)) returns ((a b) (c (d e) f))

(append '(a b) '(c (d e) f)) returns (a b c (d e) f)

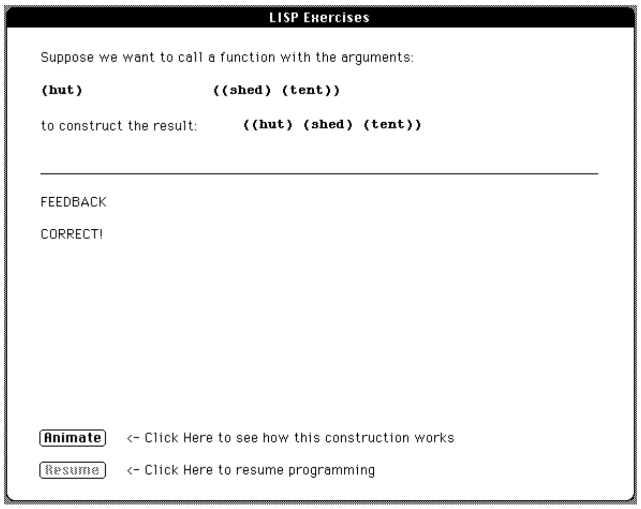
(cons '(a b) '(c (d e) f)) returns ((a b) c (d e) f)

The function **list** can take one or more arguments, and makes a new list by "wrapping parentheses" around its arguments. The function **append** takes one or more lists as arguments, makes a new list by "removing the parentheses" from around each of its arguments and merging all the elements into one long list. The function **cons** always takes two arguments and inserts the first argument at the beginning of the second.

Table 3. An excerpt from the Lisp text describing the operators *append*, *cons* and *list*.

The college students in our samples do eventually learn this discrimination reasonably well. Students' performance is quite good by conventional standards, but falls short of the mastery ideal that all students may become "A" students. There is evidence that students' posttest performance falls short of true mastery because some students encode rules that are correlated with, but different from optimal rules [7]. For example, students may encode a heuristic that if the arguments in a problem are all lists, then *append* is the appropriate function.

Can we help all students become "A" students? The effectiveness of natural language instruction is limited here by terminology confusion and the perceptual challenge of parsing parentheses. Instead, to highlight the key structural relationships among input arguments and results, we developed an augmented interface for combiner function calls that provides animated graphical feedback, as displayed in Figures 2 and 3.



LISP Exercises

Suppose we want to call a function with the arguments:

(hut) **((shed) (tent))**

to construct the result: **((hut) (shed) (tent))**

FEEDBACK

CORRECT!

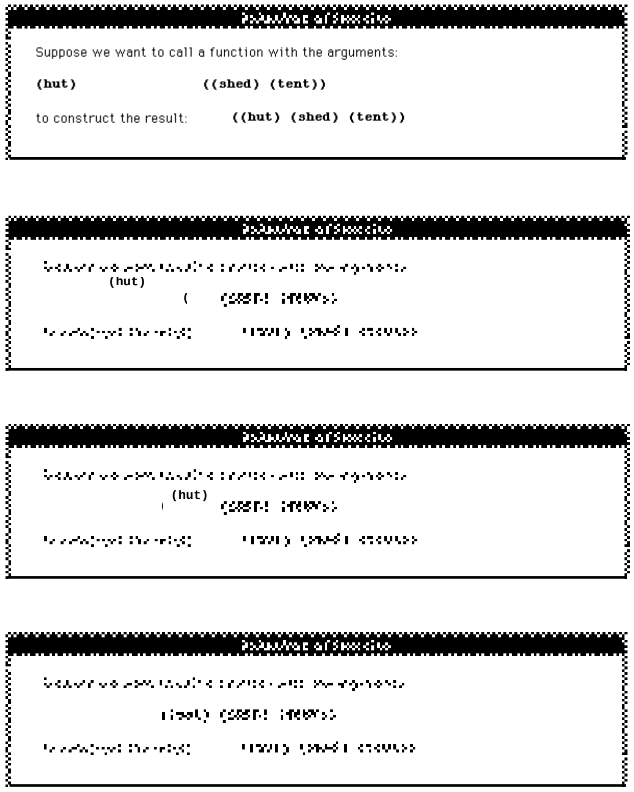
Animate
<- Click Here to see how this construction works
Resume
<- Click Here to resume programming

Figure 2 Animated feedback window for combiner function calls in curriculum section 2.

After the student has selected a combiner function in a tutor problem the feedback window displayed in Figure 2 appears. This window appears whether or not the student selects the correct function, since there is a 1/3 chance of selecting the correct one by guessing. When the student clicks the animate button at the bottom of the window, the structural relation between the arguments and result is animated. Arguments and parentheses move on the screen to display this structural relationship. Figure 3 depicts the animation for the function *cons*, in which the initial parenthesis of the second argument slides over to the left and the first argument slides to the right to become the first element of the second argument. In the case of *list*, the input arguments slide together and an encompassing set of parentheses descends. In the case of *append*, the outer parentheses of each input argument descend off the screen, the arguments slide together on the screen, and two new parentheses encompass the result.

Plan Support: Subgoal Scaffolding

Consider the section 4 problem displayed in Table 4. The tutor's structure editor supports top-down programming so under the ACT-R model, students need to satisfy three planning goals before they begin coding. Under cognitive mastery learning, the students should have mastered the final five coding productions at the bottom of the table in the first three curriculum sections. The unique components of this section are the productions needed to satisfy the three planning goals. Again, students master this task reasonably well, but often require many remedial problems to do so.



Suppose we want to call a function with the arguments:

(hut) ((shed) (tent))

to construct the result: ((hut) (shed) (tent))

Figure 3. The initial state, two intermediate states and the final state in animating the structural relationship between input arguments and the output list for the function *cons*.

To support more efficient learning in this curriculum section, we developed a plan reification interface as displayed in Figure 4. Panel 4a displays the standard coding interface at the beginning of a programming problem. Students simply select the <code> symbol and begin generating Lisp code. Panel 4b displays the plan reification interface which requires the student to post the

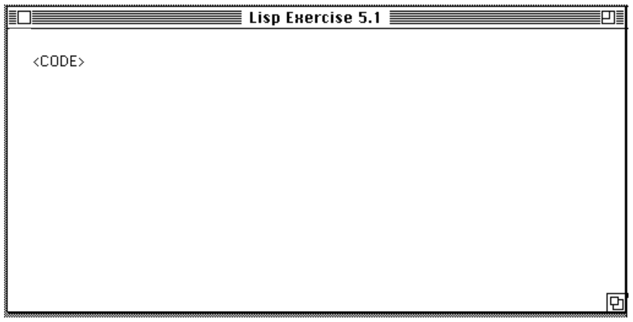
Exercise: Write a function call that takes the lists
(a b c) and (d e f) and returns the list
(a f e d).

Solution: (cons (car ‘(a b c)) (reverse ‘(d e f)))

Recognize that a must be extracted from (a b c).
Recognize that the list (d e f) must be reversed.
Recognize that a and (f e d) must be combined in a list.
Code a call to *cons*
Code a call to *car.*
Code the first given as the argument to *car.*
Code a call to *reverse.*
Code the second given as the argument to *reverse.*

Table 4. Planning and coding goals for a section 4 problem

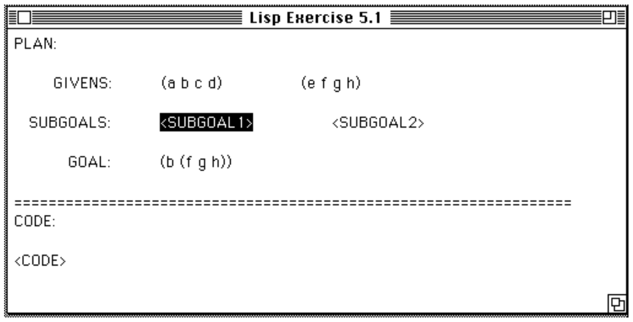
two expressions that must be extracted from the given arguments before entering any code. In this example, students must type b for <subgoal1> and (f g h) for <subgoal2> before entering the code (list (car (cdr ‘(a b c))) (cdr ‘(e f g h))).



Lisp Exercise 5.1

<CODE>

Figure 4a. The standard coding interface.



Lisp Exercise 5.1

```
PLAN:

   GIVENS:     (a b c d)        (e f g h)

 SUBGOALS:     <SUBGOAL1>         <SUBGOAL2>

     GOAL:     (b (f g h))

==============================================================
CODE:

<CODE>
```

Figure 4b. The plan scaffolding interface

DESIGN OF THE STUDY

In this evaluation of animated parenthesis feedback and subgoal scaffolding interface enhancements, students worked through the first four sections in the ACT Programming Tutor Lisp curriculum and completed three programming tests.

Participants

Thirty nine college students were recruited to participate in the study for pay. These students had an average Mathematics SAT score of 648 and had completed an average of 1.3 programming courses previously, although none had prior experience with Lisp. Both these variables were controlled in assigning students to two groups. Eighteen students completed the study with the standard coding interface. Twenty-one completed the study with augmented feedback and subgoal scaffolding as described above.

Procedure

Students in this study completed the first four sections in the APT Lisp curriculum. In each curriculum section, students read text describing Lisp, completed one or two sets of quiz questions on the text, completed a small fixed set of required programming exercises that covers the rules being introduced, then completed remedial exercises as needed to bring all production rules in the section to a mastery criterion (knowledge probability > 0.95). Students in the standard interface condition completed all exercises in the standard APT coding interface. Students in the Augmented Support group received animated parenthesis feedback in the second curriculum section on basic combiner functions and subgoal scaffolding in the fourth section on combiner/extractor algorithms.

Students completed programming tests following the first, third and fourth sections. These tests were cumulative and contained six, twelve and eighteen programming exercises respectively. The test exercises were similar to the tutor exercises and the test interface was identical to the standard tutor interface, except that students could freely edit their code. No tutorial assistance and no augmented support of any kind was available in testing.

RESULTS

Two measures were employed to evaluate the students' learning effort in reaching cognitive mastery: number of problems needed to reach mastery and elapsed time in reaching mastery. The first measure is theoretically relevant, since learning is assumed to occur at opportunities to fire productions. The second measure is of more practical significance. We also employed two measures of learning outcomes, mean test accuracy and proportion of students reaching "A" level performance (at least 90% correct) on the posttests.

Learning Effort: Number of Problems to Mastery

Table 5 displays the mean number of problems required to reach mastery in each curriculum section for students in the two conditions. All students completed 21 required problems in the study. Students in the standard condition needed an additional 37.2 remedial problems to reach mastery, while students in the augmented feedback condition needed 22.8 remedial problems. While this difference is large, the main effect of interface condition is not significant, F(1,37) = 1.47. The main effect of curriculum section is significant, F(3,111) = 18.55, p < .01, confirming that sections 2 and 4 are more challenging than sections 1 and 3. More importantly, the interaction of interface condition and curriculum section is significant, F(3,111) = 3.09, p < .05).

Students in both groups worked with the standard coding interface in sections 1 and 3 and essentially completed the same number of remedial exercises in those sections. Note that while animated parenthesis feedback was provided to the Augmented Support group in curriculum section 2, this augmented feedback had no impact on the number of problems needed to reach mastery learning. Both groups completed 9 required problems and an average of about 15 remedial problems in section 2. But, subgoal scaffolding in section 4 had a large impact on problems needed to reach mastery. Students in the Standard Coding condition averaged 15.44 remedial problems, while students in the Augmented Support condition averaged only 2.05 problems. This difference is marginally significant, t(37) = 1.83, p < 0.08.

	Standard Interface		Augmented Support	
	Problems	Time	Problems	Time
Extractors	6.3	9.0	6.3	6.2
Combiners	24.7	28.3	23.7	32.7
Extractor Algorithms	8.8	10.7	8.7	9.0
Combiner Algorithms	18.4	35.2	5.1	15.2
Total	58.2	83.2	43.8	63.1

Table 5. Average Number of APT Tutor Problems Required to Reach the Mastery Criterion and elapsed time (minutes) in reaching mastery for Students in the Standard Interface and Augmented Support conditions.

Learning Effort: Time to Reach Cognitive Mastery

Table 5 also displays the mean time students spent completing tutor problems in reaching cognitive mastery. Students in the Standard Interface group needed 30% more time to reach mastery than the Augmented Support group, but this difference is not reliable, F(1,37) = 1.2. The main effect of curriculum section is again significant, F(3,111) = 13.46, p < .01, and more importantly, the interaction of interface condition and curriculum section is significant, F(3,111) = 2.9, p < .05).

Note that the two groups spent virtually the same amount of time reaching mastery in the first three curriculum sections (48.0 minutes vs. 47.9 minutes). The entire 20 minute difference in elapsed time occurs in the fourth, combiner algorithm section in which the Augmented Support condition benefited from subgoal scaffolding. However, the difference between the means in curriculum section 4 is not reliable, t(37) = 1.6, ns.

While mean learning time did not vary reliably across groups in section 4, the variance in learning times in that section is much larger for the Standard Interface group (3089 vs 160) and this difference is reliable F(17,20) = 19.3, p < .01. An inspection of the learning time distributions in this section makes the impact of subgoal scaffolding more apparent. In both conditions about 80% of students completed section 4 in 30 minutes or less. The remaining 20% of subgoal scaffolding students finished in less than an hour, while the remaining 20% of standard coding students required between 2 and 3.5 hours. So, the main impact of subgoal scaffolding on learning time is to greatly reduce the time needed for the slowest students to reach cognitive mastery.

To further explore this evidence that subgoal scaffolding primarily helped the slowest students, we divided each group of students in half based on Math SAT scores and reanalyzed the learning effort data for the section 4 combiner algorithms just for the students with lower Math SAT scores. The average Math SAT score for this subset of students was 571. As displayed in Table 6, students with lower SAT scores in the standard interface condition required an average of 33.6 problems and 60.7 minutes to reach mastery while students working in the subgoal scaffolding condition required 6 problems and 19.8 minutes. The difference in number of problems is reliable, t(18) = 2.11, p < .05 and the difference in elapsed time is marginally reliable t(18) = 1.87, p < .08.

	Standard Interface		Augmented Support	
	Lower SATs		Lower SATs	
	Problems	Time	Problems	Time
Combiner Algorithms	33.6	60.7	6.0	19.8

Table 6. Average Number of APT Tutor Problems Required to Reach the Mastery Criterion and elapsed time (minutes) in reaching mastery for Students in the Standard Interface and Augmented Support conditions.

Test Performance

Posttest performance of the two groups is displayed in Table 7. Two performance measures are displayed: (1) mean percent correct and (2) the probability that students reach "A" level performance (90% correct).

	Standard Interface		Augmented Support	
	%C	P> 0.9	%C	P > 0.9
Test 1				
Total	94%	0.72	98%	0.90
Test 2				
Total	86%	0.50	96%	0.90
Extractors	92%	0.83	97%	0.81
Combiners	80%	0.39	95%	0.86
Test 3				
Total	83%	0.40	88%	0.65
Extractors	95%	0.72	95%	0.76
Combiners	81%	0.39	96%	0.76
Combiner Algorithms	71%	0.28	75%	0.24

Table 7. Mean percent correct and probability of reaching "A" level performance on three posttests for the Standard Interface and Augmented Support Groups.

Students in both groups perform very well on the basic extractor problems in Test 1 characteristic of curriculum section 1. Test 2 was administered after curriculum section 3 and contained problems characteristic of the first three sections. Students in the Standard Interface condition perform very well on this test. They average 86% correct and half the students reach "A" level performance. However, students in the Augmented Support condition perform even better. They score 96% correct and 90% of students reach "A" level performance. The difference in average percent correct is reliable, F(1,37) = 8.7, p < .01, as is the difference in proportion of students reaching "A" level (z = 2.8, p < .01). Two subscores are reported for Test 2. The first includes extractor and extractor algorithms questions from sections 1 and 3. As can be seen, there is little difference between the two groups on these questions. However, students in the augmented feedback condition perform substantially better on the combiner problems in testing. The main effect of problem type is reliable F(1,37) = 5.5, p < .05, so the combiner problems are reliably more difficult, and the interaction of interface condition and problem type is marginally reliable, F(1,37) = 2.7, p = 0.10. In pairwise comparisons, the difference between the two interface groups on the extractor problems is non-significant, t(37) = 1.6, while the difference between the two groups on the combiner problems is significant t(37) = 2.7, p < .01. Similarly, the difference in proportion of students reaching "A" level performance in the combiner section is reliable, (0.86 vs. 0.39), z = 3.0, p < .01.

Test 3 was administered following the fourth curriculum section and contained problems characteristic of all four sections. Overall, students again performed quite well on this test. The Augmented Support students performed slightly better than the Standard Interface students, and this overall difference is marginally reliable, F(1,37) = 3.8, p < .06. The proportion of students reaching "A" level performance overall did not vary significantly, z = 1.7. Three test subscales are reported in Table 7, extractor problems including extractor algorithms, basic combiner problems and combiner/extractor algorithm problems. The main effect of problem type is significant, F(2,74) = 326.1, p < .01, and the interaction of interface condition and problem type is reliable, F(2,74) = 5.5, p < .01. Again, there is very little difference between the two groups on the extractor and extractor algorithm problems. The Augmented Support students continue to perform better on the section 2 combiner problems than the Standard Interface students (96% vs 81%). In a pairwise test, this difference is reliable, t(47) = 3.1, p < .01, as is the difference in proportion of students reaching "A" level (76% vs 39%), z = 2.4, p < .05. Finally, performance on the section 4 combiner/extractor algorithm problems is virtually identical for the two groups. Students in the Augmented Support condition completed an average of just 5 tutor problems in section 4, yet reached the same level of test performance on those problems as the Standard Interface students who averaged 18 tutor problems in section 4.

DISCUSSION

The two interface enhancements evaluated in this study led to very different gains in educational efficiency. Animated parenthesis feedback was introduced in curriculum section 2 to make relevant data structures salient in discriminating among three combiner functions. This intervention had no impact on the effort required to satisfy the tutor's cognitive mastery criterion, but this augmented feedback resulted in a substantial gain in test performance, both in mean accuracy and proportion of students reaching “A” level performance. In contrast, the section 4 subgoal scaffolding designed to help students organize productions they mastered in previous lessons, led to a large decrease in number tutor problems required to reach cognitive mastery, but did not lead to increased posttest accuracy. Overall, subgoal scaffolding also sharply reduced the maximum elapsed time that students needed to reach cognitive mastery, but this effect was marginally reliable only for students with lower Math SAT scores.

Although the educational impact of animation in technology enhanced learning has been mixed at best [10, 11], animated parenthesis feedback had a decisive positive impact on test performance in this study. It did not make learning “easier” as measured by time on task or number of tutor problems, but fostered programming knowledge that transferred more successfully to the test environment. This successful transfer implies that students are acquiring a deeper, more optimal encoding of relevant aspects of list structure. Indeed, we believe that animation was successful because it addressed a crucial topic that students find hard to grasp and that does not lend itself well to natural language discussion.

Subgoal scaffolding, in contrast, decreased the average number of problems required to reach mastery in the tutor, but did not reliably reduce average learning time, nor enhance subsequent test performance. This suggests that scaffolding did make it easier for students to organize previously mastered operator knowledge into more complex plans, but did not lead to a substantially deeper knowledge of the operators nor of the plans. Reducing the learning effort needed to reach equivalent performance levels is an important accomplishment, though, and subgoal scaffolding did dramatically reduce section 4 learning time for about 20% of students, from a maximum of 2 to 3.5 hours to a maximum of 1 hour.

This study raises some interesting questions to be pursued. For example, the animated parenthesis feedback was provided *retrospectively*, after the student selected a combiner function, while students posted subgoals *prospectively*, before selecting the subsequent combiner function. Perhaps subgoal scaffolding could lead to deeper understanding if subgoal posting followed rather than preceded the initial combiner selection. Another intriguing question is why the Augmented Support students’ superior understanding of basic combiner functionality demonstrated in section 2 problems did not transfer to higher test accuracy in section 4 combiner/extractor algorithm questions.

Nevertheless, these results are very encouraging. Relatively simple interface enhancements can have a substantial impact on learning rate and asymptotic test performance. These results serve as a reminder that as we continue to develop computer-based learning environments, that we should not limit ourselves to studying the strategies that make human tutors effective, but need to assess domain specific challenges and tailor instructional interventions to meet those challenges.

ACKNOWLEDGMENTS

This research was supported by the Office of Naval Research grant number N00014-95-1-0847 and by NSF grant number 9720359 to CIRCLE: Center for Interdisciplinary Research on Constructive Learning Environments. We thank Dana Heath and Susan Klein for help in data collection and Megan McLaughlin for help in manuscript preparation.

REFERENCES

1. Anderson, J.R., Conrad, F. and Corbett, A.T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, *13*, 467-505.

2. Anderson, J.R., Corbett, A.T., Koedinger, K.R., and Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, *4*, 167-207.

3. Anderson, J.R., and Lebiere, C. (1998*). The atomic components of thought*. Mahwah, NJ: Erlbaum.

4. Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher, 13*, 4-16.

5. Cohen, P. A., Kulik, J. A., & Kulik, C. C. (1982). Educational outcomes of tutoring: A meta-analysis of findings. American *Educational Research Journal*, *19*, 237-248.

6. Corbett, A.T. and Anderson, J.R. (1995). Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction, 4*, 253-278.

7. Corbett, A.T. and Bhatnagar, A. (1997). Student modeling in the ACT Programming Tutor: Adjusting a procedural learning model with declarative knowledge. *Proceedings of the Sixth International Conference on User Modeling.* New York: Springer-Verlag Wein.

8. Graesser, A.C., Person, N.K., & Magliano, J.P. (1995). Collaborative dialogue patterns in naturalistic one-on-one tutoring. *Applied Cognitive Psychology*, 9, 359-387.

9. Koedinger, K.R., Anderson, J.R., Hadley, W.H. & Mark, M.A. (1995). Intelligent tutoring goes to school in the big city. *Proceedings of the 7th World Conference on Artificial Intelligence in Education.*

10. Pane, J.F., Corbett, A.T. and John, B.E. (1996). Assessing dynamics in computer-based instruction. *Proceedings of ACM CHI’96 Conference on Human Factors in Computing Systems*, 197-204.

11. Rieber, L.P., Boyce, M.J., and Assad, C. (1990). The effects of computer animation on adult learning and retrieval tasks. *Journal of Computer-Based Instruction*, 17, 46-52.

12. VanLehn, K., Siler, S., Murray, C. & Bagget, W. (1998). What makes a tutorial event effective? In: M. A. Gernsbacher & S. Derry (Eds*.) Proceedings of the Twenth-first Annual Conference of the Cognitive Science Society,* Hillsdale, NJ: Erlbaum. pp. 1084-1089.