Knowledge Decomposition and Subgoal Reification in the ACT Programming Tutor

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

John R. Anderson
Departments of Psychology and Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
412-268-2788

Abstract

A major issue in education is whether a complex skill can be decomposed into components skills and whether the components can be independently learned and assessed. This study examines the issue of knowledge decomposition in the context of the ACT programming Tutor (APT). APT is a cognitive tutor that assists students in learning to write short programs. The tutor is constructed around a production-rule model of programming knowledge that allows the tutor to solve the exercises along with the student and provide help as needed. This paper describes the model's decomposition of early programming exercises into planning steps that are intrinsic to the structure of the task and coding steps that should be isolable. Implications of this decomposition for subgoal scaffolding and mastery-based learning are discussed and an interface that reifies programming subgoals is evaluated empirically. Students are shown to acquire programming knowledge most readily when the isolable coding knowledge is mastered ahead of time and when an overt action is associated with each planning subgoal. This knowledge is shown to generalize to programming performance in the absence of such subgoal scaffolding. Finally, the knowledge acquisition assumptions of the model are shown to fit student performance best when coding rules are mastered ahead of time and planning subgoals are scaffolded.

Keywords

Intelligent Tutoring Systems
Knowledge Representation
Student Modeling
Subgoal Scaffolding
Mastery Learning
Empirical Evaluation

A major issue in education is whether a complex skill can be decomposed into component skills and whether these components can be learned and assessed separately (Resnick & Resnick, 1992; Shepard, 1991). Mastery learning, in particular, assumes that complex knowledge can be decomposed into hierarchy of prerequisites and that prerequisites should be mastered before moving to higher level knowledge. Review articles overwhelmingly confirm that mastery learning yields higher achievement levels (cf. Kulik, Kulik & Bangert-Drowns, 1990), but critics note that these achievement effects fall short of predictions (Resnick, 1977; Slavin, 1987).

The ACT Programming tutor attempts to implement mastery learning in the framework of a detailed cognitive model. The tutor is a practice environment in which students learn to write short programs in Lisp, Prolog and Pascal. Each of these modules is constructed around a language-specific cognitive model of the programming knowledge the student is acquiring. The embedded cognitive model allows the tutor to solve the exercises along with the student and provide step-by-step assistance as needed. The tutor is used to teach programming courses at Carnegie Mellon University and in a local high school and has proven effective. Students complete practice exercises with the tutor in as little as one-third the time required by students working on their own and perform as well or better on tests (Anderson, Corbett, Koedinger & Pelletier, in press). The tutor was developed, in part, to examine the ACT model of procedural knowledge and skill acquisition. Early analyses of student performance with the Lisp tutor largely confirmed assumptions concerning the form of procedural knowledge in the ACT* model, but contributed to revisions in the learning mechanism embodied in the more recent ACT-R model (Anderson, Conrad & Corbett, 1989; 1993). In this study, we further examine the assumptions of the model concerning the decomposition of procedural knowledge and implications both for the scaffolding of planning subgoals and for mastery learning.

The Cognitive Model

ACT-R assumes that procedural knowledge can be represented as a set of independent production rules that associate problem states and problem solving goals with actions. Table 1 presents five example production rules that are acquired in the first two sections of the Lisp curriculum. Rules P1 and P4 fire in sequence in solving the first exercise in the Lisp curriculum, as shown here:

Exercise: Write a lisp function call that returns c from the list (c d e).
Solution: (car '(c d e))

The tutor curriculum is structured around production sets. In each section of the curriculum students read text that introduces a small set of production rules. Then the tutor provides practice problems that exercise those rules. As the student works, the tutor maintains an estimate of the student's knowledge state in a process we call *knowledge tracing*. As each opportunity arises to apply a production rule during practice, the tutor estimates the probability that the student has learned the rule, contingent on the student's action. The learning and performance assumptions and Bayesian computational procedure are described elsewhere (Corbett, Anderson & O'Brien, in press). In knowledge tracing two learning parameters (probabilities) are estimated empirically for each rule in the model, as shown in Table 2. These parameters are employed to assess the decomposition assumptions of the model below.

(P1)	IF the goal is to return the first element of a list, THEN code the function ***car***, and set a goal to code the list as an argument.
(P2)	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.
(P3)	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.
(P4)	IF the goal is to code a literal list, THEN code a single quote, followed by the list.
(P5)	IF the goal is to code a literal atom, THEN code a single quote, followed by the atom.

Table 1. Five production rules introduced early in the Lisp curriculum.

The learning and performance assumptions that underlie knowledge tracing also yield performance predictions that can be evaluated empirically. Several studies have examined the validity of knowledge tracing and found that the model reliably predicts individual differences in test performance across students. (Corbett, Anderson, Carver & Brancolini, 1994; Corbett, Anderson & O'Brien, 1993; in press).

Knowledge tracing is employed to implement mastery learning in the tutor. The tutor continues presenting exercises in each curriculum section until the probability that the student knows each rule reaches a criterion of 0.95. Thus, students master the simple rules exemplified in Table 1 in the early curriculum sections under the assumption that they enter into more complex programming tasks in later sections. An analysis relating these simple skills to a somewhat more complex task is presented in the next section.

Knowledge Decomposition and Planning Productions

Table 3 presents problem descriptions and solutions to four programming exercises. The first two exercises are drawn from the first curriculum section that introduces three extractor functions, ***car***, ***cdr*** and ***reverse***. Each of these operators takes a single list and returns components of, or a transformation of, the list. In the first exercise the student applies the operator ***car*** to the list ***(a b c)*** to return the first element, ***a***. In the second exercise, the student applies the operator ***reverse*** to

$p(L_0)$	The probability the rule is in the learned state after reading the text, prior to the first opportunity to apply the rule.
p(T)	The probability that a student will learn the rule at each opportunity to apply the rule during practice.

Table 2. The learning parameters employed in knowledge tracing.

the list ***(d e f)*** to return the list ***(f e d)***. The third exercise is drawn from the next curriculum section that introduces three constructor functions, ***append***, ***cons*** and ***list***. These operators take two or more arguments and create new lists. In this exercise the student employs the operator ***cons*** to insert the atom ***a*** at the beginning of ***(f e d)***. Note that these three exercises employ the five simple rules displayed in Table 1.

Exercise Description	Solution
(1) Write a function call that takes the list (a b c) and returns a.	**(car '(a b c))**
(2) Write a function call that takes the list (d e f) and returns (f e d).	**(reverse '(d e f))**
(3) Write a function call that takes a and (f e d) and returns (a f e d).	**(cons 'a '(f e d))**
(4) Description: Write a function call that takes the lists (a b c) and (d e f) and returns the list (a f e d). Subgoals: Planning Goal 1: Recognize that a must be extracted from (a b c) Planning Goal 2: Recognize that the list (d e f) must be reversed Planning Goal 3: Recognize that a and (f e d) must be combined in a list Coding Goal 1: Code a call to cons Coding Goals 2&3: Code a call to car with the first given as an argument Coding Goals 4&5: Code a call to reverse with the second given as an argument Solution: **(cons (car '(a b c)) (reverse '(d e f)))**	

Table 3. Three simple exercises and a more complex subsequent exercise.

Now consider the more complex fourth exercise at the bottom of the table, drawn from the fifth curriculum section. Unlike the earlier exercises, the student must apply extractor functions to the given lists *and* combine the results into a new list. The subgoal structure of this exercise is depicted in the table. It begins with three planning goals in which the student must analyze the relationship between the given and goal lists and recognize that (1) the first element ***a*** must be extracted from the first given list, (2) the second list must be reversed to yield ***(f e d)***, and (3) that the atom ***a*** must be combined with ***(f e d)***. The goal list concludes with five coding goals in which the student employs the five production rules displayed in Table 1.

Note that under this analysis, the complex task in exercise four does not simply reduce to the five coding productions that are sufficient for exercises 1-3. However, these five rules are necessary components of the solution to exercise 4. If the simpler rules are mastered in early sections, it should not yield immediate mastery of the more complex exercises, but should speed learning in these complex exercises.

Reifying Planning Subgoals

Consider exercise 4 again. Since the student enters code top-down, the first overt coding action is to code the constructor function ***cons***. This coding action is the culmination of a four-step reasoning chain:

(1) Set a goal to return the atom ***a*** from the list ***(a b c)***.
(2) Set a goal to return the list ***(f e d)*** from the list ***(d e f)***.
(3) Set a goal to write a function call that combines ***a*** and ***(f e d)*** in a list.
(4) Fire coding production P3 from table 1.

If we assume the student has mastered the production P3 in an earlier section, correct performance depends on correct execution of three novel steps. We can simplify the problem solving complexity for the student if we reify two of the three planning steps with overt actions. This planning reification also simplifies the attribution task in modeling the student's knowledge, since we can assume that the accuracy of each of the overt actions now depends primarily on a single new rule.

We developed a variation of the tutor interface for the fifth curriculum section to reify the first two subgoals in the above analysis. The standard coding interface is depicted in Figure 1a and the revised interface is depicted in Figure 1b. In the standard interface, the student simply begins coding the solution beginning with the operator ***cons***, as in other curriculum sections. In the plan reification interface the student is required to fill in the two subgoal nodes to indicate what expressions must be extracted from the givens before entering any code. In this example students must type the expression ***a*** for <subgoal1> and ***(f e d)*** for <subgoal2> before entering the code ***(cons (car '(a b c)) (reverse '(d e f))***.

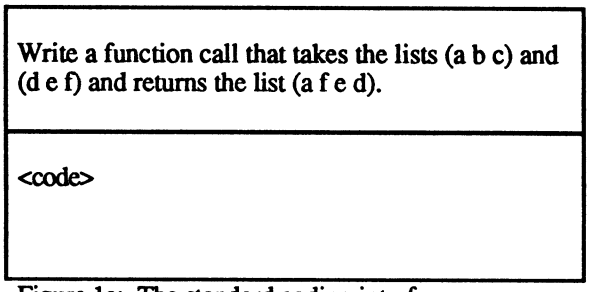

Figure 1a: The standard coding interface.

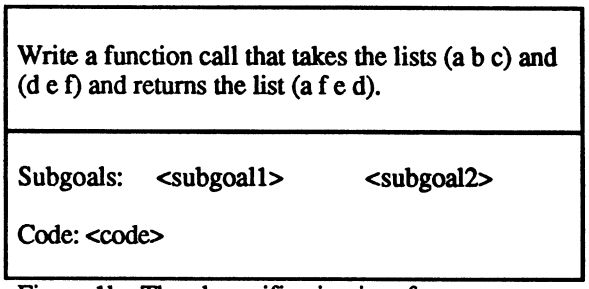

Figure 1b. The plan reification interface

This interface was employed in a study of the modeling assumptions. All students worked with the standard coding interface through the first four curriculum sections. In section 5 one group used the new planning interface while the other group continued to work with the standard interface. Within each of these groups, half the students worked to mastery in each of the curriculum sections, while the other group completed a fixed set of required exercises. Several issues are examined:

(a) Do students learn the complex programming skill more readily if the first two planning subgoals are reified as overt actions?

(b) Does the knowledge acquired with plan scaffolding in the revised interface generalize to a standard coding environment that does not reify planning?

(c) Do students learn the complex programming skill more quickly if they have already mastered the component simple coding rules.

(d) The first coding step (coding the constructor function) is governed by three new productions in the standard coding interface, but only one new production in the plan scaffolding interface. Does the knowledge tracing model fit student performance better in the latter case?

The Design of the Study

Students in this experiment worked through the early sections of the ACT Programming Tutor Lisp curriculum and completed three tests.

Subjects

Forty-six students were recruited to participate in the study for pay. These students had an average Math SAT score of 665 and had completed an average of 1.0 prior programming courses, although none had prior experience with Lisp. Both of these variables were controlled in assigning students to the four groups.

Design

Students worked through five curriculum sections in this study. This curriculum introduces two data structures, *atoms* and *lists*, and introduces *function calls*. The first section introduces three extractor functions, ***car***, ***cdr*** and ***reverse***, that return components of, or a transformation of, a list. The second and third sections introduce three constructor functions, ***append***, ***cons*** and ***list*** that create new lists. The fourth section introduces extractor algorithms - nested function calls that apply successive extractor functions to extract components of lists. This study focuses on the fifth section in which students embed extractors and extractor algorithms as arguments to constructor functions as exemplified exercise 4 above. These five curriculum sections contain 30 required tutor exercises.

One group of twenty-five students worked to mastery in all five curriculum sections under the control of knowledge tracing, while a control group of twenty-one students completed just the 30 required exercises. All students used the standard coding interface through the first four curriculum sections. Approximately half of the students in each group employed the planning interface in section 5 (thirteen mastery students and eleven control students), while the remaining students in each group continued with the standard coding interface.

Procedure

Students worked through the curriculum at their own pace. In each section, students read text describing Lisp, then completed a set of required exercises that cover the programming rules being introduced. Students in the mastery condition then completed remedial exercises as needed to bring all production rules introduced in the section to a mastery criterion (minimum knowledge probability of 0.95). Students completed cumulative tests following the first, fourth and fifth sections. These tests contained six, twelve and eighteen programming exercises respectively. These exercises were similar to those completed with the tutor and the test interface was identical to the tutor interface, except that students could freely edit their code and received no tutorial assistance. No plan scaffolding was provided in the test environment; all students used the standard coding interface.

Results

One student in the mastery-learning/plan-scaffolding condition was atypical in completing three times as many remedial exercises as any other student and is excluded from further analysis. The remaining twenty-four mastery students completed an average of 18 remedial exercises (range = 1 to 53) in addition to the 30 required exercises.

Plan Scaffolding Effectiveness

In the mastery condition the dependent measure of plan scaffolding effectiveness is the number of remedial exercises needed to achieve mastery in the tutor. All

students in this condition should master the programming skills and should perform similarly on the tests. In the control group all students complete a fixed number of exercises and the primary measure in assessing the planning interface is test performance.

Tutor Measure: Remedial Exercises. Within the mastery condition, students working with the planning interface required an average of 8.8 remedial exercises to reach mastery in curriculum section 5. Students in the standard coding condition required 12.2 remedial exercises to reach mastery. This advantage for the plan scaffolding condition is reliable, $t(22) = 2.32$, $p < 0.05$. However, this comparison is valid only if students in the two groups are shown to achieve comparable test performance below.

Test Measures. The third test contained six exercises drawn from section 5. Recall that all students completed these test exercises in the standard coding interface, although without tutorial support. Table 4 displays the average probability of completing these six exercises correctly for the four groups. In a two-way analysis of variance, the main effect of tutor interface (plan reification vs standard coding) is non-significant, $F(1,41) = 1.7$. The main effect of practice (mastery vs fixed control) is reliable, $F(1,41) = 25.39$, $p < .01$ and the interaction is reliable $F(1,41) = 4.22$, $p < .05$. As expected the students who worked to mastery in the tutor performed reliably better on the test than students in the control group who completed the fixed set of required exercises. Within the mastery condition, the tutor interface manipulation had no impact on test performance. This has two implications. First, the interpretation of remedial exercises above is valid. Second, the knowledge that students acquired with the plan scaffolding interface fully transferred to the standard coding interface in the test. Within the fixed practice control condition, the plan-reification group performs better than the standard coding group. This test advantage for plan scaffolding in the tutor is reliable, $t(19) = 2.08$, $p = .05$.

	Tutor Interface	
Practice	Planning	Coding
Mastery	0.74	0.79
Control	0.52	0.27

Table 4. The impact of practice condition and tutor interface on test accuracy (average probability of correct exercise)

Modeling Knowledge Acquisition

Finally, we can assess the decomposition and acquisition assumptions of the model by examining the fit of the model to coding performance in the tutor. More specifically, we examine the fit to the first coding step in the fifth section exercises, in which the student codes a constructor function. Recall that in the mastery-learning/plan-reification condition, performance accuracy is primarily governed by a single new planning production. In the other three conditions, performance accuracy will be determined by multiple productions. In the mastery-learning/standard-coding condition, performance is governed by three new planning productions. In the fixed-practice/plan-reification condition,

performance is largely determined by a new planning production and an old coding production that is not fully learned. Finally, in the fixed-practice/standard-coding condition performance is determined by three new planning productions and an old production that is not fully learned.

We fit the learning and performance model to each group's performance on the first coding step across the six required tutor exercises in section 5. Table 5 displays the best fitting learning parameter estimates for the four conditions and the correlation of actual and predicted accuracy across these six coding goals. The third column in the table displays the best fitting estimate of $p(L_0)$, the probability that the students learned an appropriate rule from reading the text prior to the first practice exercise. The fourth column displays the best fitting estimate of p(T) the probability of acquiring a rule at each opportunity to apply it in practice. The fifth column represents the goodness of fit of the model.

Practice	Tutor Interface	Learn from Text	Learn in Practice	r
Mastery	Planning	0.50	0.27	0.72
Mastery	Coding	0.30	0.14	-0.04
Control	Planning	0.13	0.15	0.22
Control	Coding	0.20	0.10	-0.02

Table 5. Best fitting estimates of the two learning parameters in the model

As can be seen the mastery-learning/plan-reification condition is the only condition that is well fit by the model. The correlation coefficient, 0.72, is reliably different from each of the other three conditions. Not only does the model provide the best fit in this condition, but the best fitting parameter estimates indicate that students are learning most quickly. The probability that students in this condition understand the rule from reading is 0.50 and the probability of acquiring at each opportunity to practice the rule is 0.27. Each of these probabilities is approaches twice the corresponding value of the other conditions.

Conclusion

The model-guided plan scaffolding proved an effective pedagogical device. The model suggests that coding the first Lisp operator in these exercises requires three planning step and a previously learned coding production. When subgoal scaffolding is used to map the three subgoals onto distinct actions, students acquire the skill more quickly. Students learn the skill most quickly if they have also already mastered the fourth coding step. It is important to note that this benefit generalized to the test environment in which no scaffolding was provided. Finally, the model was most successful in the mastery-learning/plan-reification condition in predicting the accuracy of the students' first coding action. This is the only condition in which constructor coding accuracy is primarily governed by a single production rule.

The results support the proposition that a complex skill can be decomposed into underlying knowledge components. The results also provide some insight on why

previous mastery efforts may have only had partial success, as noted in the introduction. Some of the underlying curriculum components are like the planning productions in this research and do not normally have behavioral indicants. This makes it difficult to teach the components and to assess whether they have been mastered. However, we have shown that at least in one case it is possible to design an interface which reifies the hidden components.

References

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

Anderson, J. R., Conrad, F.G. and Corbett, A.T. (1989). The Lisp Tutor and skill acquisition. In J. Anderson (Ed.) *Rules of the mind.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J.R., Corbett, A.T. Koedinger, K.R. and Pelletier, R. (in press). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences.*

Corbett, A.T., Anderson, J.R., Carver, V.H. and Brancolini, S.A. (1994). Individual differences and predictive validity in student modeling. In A. Ram & K. Eiselt (eds.) *The Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Lawrence Erlbaum.

Corbett, A.T., Anderson, J.R. and O'Brien, A. T. (1993) The predictive validity of student modeling in the ACT Programming Tutor. In P. Brna, S. Ohlsson & H. Pain (eds.) *Artificial Intelligence and Education, 1993: The Proceedings of AI-ED 93.* Charlottesville, VA: AACE.

Corbett, A.T. and Anderson, J.R. and O'Brien, A.T. (in press). Student modeling in the ACT Programming Tutor. In P. Nichols, S. Chipman and B. Brennan (eds.) *Alternative Diagnostic Assessment..* Hillsdale, NJ: Erlbaum.

Kulik, C.C., Kulik, J.A. and Bangert-Drowns, R.L. (1990). Effectiveness of mastery learning programs: A meta- analysis. *Review of Educational Research, 60,* 265-299.

Resnick, L.B. (1977). Assuming that everyone can learn everything, will some learn less? *School Review, 85,* 445-452.

Resnick, L.B. and Resnick, D.P. (1992). Assessing the thinking curriculum: New tools for educational reform. In B. Gifford & M. O'Connor (eds.) *Changing assessments: Alternative views of aptitude, achievement and instruction.* Boston: Kluwer.

Shepard, L. A. (1991). Psychometrician's beliefs about learning. *Educational Researcher,* 20, 2-16.

Slavin, R.E. (1987). Mastery learning reconsidered. *Review of educational research, 57,* 175-213.