Knowledge Transfer among Programming Languages

Quanfeng Wu & John R. Anderson
Department of Psychology, Carnegie Mellon University
Pittsburgh, PA 15213
wu@psy.cmu.edu, anderson@psy.cmu.edu

Abstract

Two experiments were conducted to investigate knowledge transfer from learned programming languages to learning new ones. The first experiment concerned transfer from knowing LISP to learning PROLOG; the results showed that subjects who knew LISP had significant advantages over subjects who did not. Moreover, among the subjects who knew LISP those who knew LISP better seemed to learn PROLOG faster. The second experiment studied transfer from knowing either PASCAL or PROLOG to learning LISP; attention was specifically focused on transfer of knowledge of writing recursive and iterative programs in these languages. The results indicated that PROLOG programmers, who were usually more knowledgeable on recursion, were more ready to learn the recursive part of the LISP language. Some general theoretical discussion about knowledge transfer among programming languages is also presented in the paper.

Introduction

Two kinds of knowledge transfer among programming languages are possible and can be separately studied. One is problem solving transfer; it involves people who already know both languages and transfer knowledge from solving a certain problem in one language to that in the other. Katz (1988) and Wu & Anderson (in preparation) have reported some evidence of transfer of this type among LISP, PROLOG, and PASCAL; these studies demonstrated significant positive transfer among the three languages. The other type is learning transfer; i.e., transfer from knowing one or more programming languages to learning a new one. The present paper concerns this second type.

This article will report two experimental studies of such transfer, independent from the tutoring work. The first experiment investigated learning transfer from LISP to PROLOG. The results showed significant positive transfer. Moreover, the first experiment also revealed that subjects who knew LISP better seemed to learn PROLOG faster. The second experiment studied learning transfer from either PASCAL or PROLOG to LISP. This experiment reinforced the evidence of positive transfer among the three languages. Also, in this experiment attention was specifically focused on transfer of knowledge of writing two types of repetition programs -- namely, recursive and iterative programs -- among these languages. The results indicated that PROLOG programmers, who were usually more knowledgeable on recursion, were more ready to master the recurive part of the LISP language.

It is our conviction that commonalities in knowledge representation form the basis for transfer. That is, it is the common knowledge shared by two domains that makes transfer possible from one domain to the other. PASCAL, LISP, and PROLOG are generally classified into three different categories of programming languages -- namely, imperative (or procedural), functional, and logical languages. However, these three languages actually have a great deal in common. These commonalities include similar basic arithmetic operations and predicates, and similar basic data types and operations on them. Some comparative analyses about commonalities shared by the three languages are given in Wu & Anderson (in preparation). Following the analyses there, it is reasonable to expect that learning transfer would occur among them, just as problem solving transfer occurred among them.

Experiment 1: Transfer from LISP to PROLOG

As this experiment was to investigate learning transfer from knowing LISP to learning PROLOG, the experimental design was basically a between-subject type. By comparing the learning performance of Know-LISP and Not-Know-LISP subjects we can study whether and to what extent LISP experience would facilitate learning PROLOG.

Method

Subjects. 24 subjects participated in the experiment; they were recruited from CMU (Carnegie Mellon University), either graduates or undergraduates. Among them, sixteen knew LISP beforehand and the rest did not know LISP. From the information gathered from questionnaire, subject's averaged GRE/SAT math scores were 708. For those who knew LISP, their averaged self-ratings of LISP proficiency were 3.44 (along a scale from 0 to 5) and their self-ratings of PASCAL proficiency were 3.63 (all of them knew PASCAL). Among those who did not know LISP only 6 knew PASCAL and their self-rating of PASCAL proficiency were 2.67. The GRE/SAT math scores for the Know-LISP condition and the Not-Know-LISP condition were comparable.

Design. Table 1 shows the design of the experiment. There were three groups of subjects with eight in each group. The subjects who knew LISP were divided into Group 1 and Group 2; the subjects who did not know LISP constituted Group 3. There were four experimental sessions, roughly about two hours for each.

Group 1 and Group 2 were treated differently only in Session 4: one group was shown LISP solutions only on odd-number problems while the other was shown LISP solutions only on even-number problems. The manipulation of Know-LISP versus Not-Know-LISP was aimed to get a global measure of how LISP knowledge facilitates learning PROLOG; the manipulation of shown versus not-shown LISP solutions was intended to get some specific understanding of how LISP knowledge helps PROLOG programming on some particular problems. These two manipulations can be seen at the two different levels of transfer between LISP and PROLOG -- that is, leaning transfer and problem solving transfer.

Table 1. The design of Experiment for transfer from knowing LISP to learning PROLOG.

	Know-LISP conditions		Not-Know-LISP condition
	Group 1	Group 2	Group 3
Session 1	Questionnaire; Two LISP programming problems for pretesting LISP knowledge; Read introductory material which is based on LISP knowledge.		Questionnaire; Read introductory material to PROLOG which is independent from LISP knowledge.
Sesssion 2 & Session 3	Use the PROLOG tutoring system to go through a series of problems covering basic concepts and syntax in PROLOG.		The same as the knew-LISP conditions.
Session 4	Use real PROLOG environment to do eight problems, mostly dealing with list processing;		The same as the knew-LISP conditions;
	For odd number problems, shown LISP solutions; for even number problems, not shown LISP solutions.	For even number problems, shown LISP solutions; for odd number problems, not shown LISP solutions.	No LISP solutions shown at all.

Materials and Procedure. In Session 1, two LISP programming problems were used as a pretest of subject's LISP knowledge for Know-LISP subjects. Subjects did LISP programming on the Andrew system, a campus network at CMU; the particular language version used was CommonLisp. Also in Session 1, all the subjects were asked to read an introductory material to PROLOG. For most of the subjects participating in the experiment, that was their first time exposed to PROLOG language; few of them had already learned some concepts about PROLOG, but none had ever actually programmed in PROLOG. Two different kinds of instructional materials were used for Know-LISP subjects and Not-Know-LISP subjects. For Know-LISP subjects, the material was written by the authors based on a comparison of LISP and PROLOG; for Not-Know-LISP subjects, the instructional material was a chapter taken Mayer (1988) which assumes no prerequisite knowledge in LISP.

In Session 2 & 3, subjects used a PROLOG tutoring system developed here in the Psychology Department at CMU to go through a series of problems with the purpose of reinforcing their knowledge about the basic concepts and syntax of PROLOG they had studied in Session 1. In Session 4, subjects again used the Andrew system to solve eight problems in a real PROLOG environment; the particular language version used was CPROLOG. Session 4 was particularly intended as a post-test of subject's PROLOG knowledge learned in the first three sessions. The problems were mostly dealing with list processing as that is a prominent feature of both LISP and PROLOG. These eight problems were: Membership, Append, Attach, Reverse, Powerset, Binary-Code, Flatten, and Skeleton; structurally, the odd-number problems are isomorphic to the following-up even-number problems (e.g., Powerset to Binary-Code).[1]

When subjects were using real LISP or PROLOG programming environments on the Andrew system, they had two windows -- one was EMACS editor window to edit the program and the other was used to debug and test the program. Each time when they made revisions to the program they save the program into different files so that the experimenter could keep track the significant errors made by subjects in the process.

Results and Discussion

Here two measures were used for the comparison made between the two groups of Know-LISP subjects: the total problem solving time and the algorithmic similarity between the shown LISP solutions and the first drafts of subject's PROLOG programs.

[1] The problems and instructuctional materials used in the experiment can be obtained by writing to the authors.

Figure 1 shows the mean total problem solving times subjects spent on the last six problems in Session 4. Note that there are some dramatic contrasts between these two groups for odd-number problems, but less dramatic contrasts for even-number problems; this is because the even-number problems are isomorphic to the just-previous odd-number problems. Thus in this experiment, we not only got the results of transfer from studying LISP solution to PROLOG programming, we also got the manifestation of transfer between isomorphic problems (However, since this transfer is not the primary goal of the present study, it is not investigated any further here). To test the statistical significance of the data, a three-way 2 X 2 X 3 ANOVA (Shown/Not-shown LISP solutions, Odd/Even problems, 3 pairs of problems) was performed. The results showed significant main effects due to whether or not LISP solutions were shown: $F(1, 7) = 48.4$, $p < 0.001$. Averaging the data over the two groups and the six problems, there was 42% saving of the total problem solving time for shown LISP solutions than not shown LISP solutions.

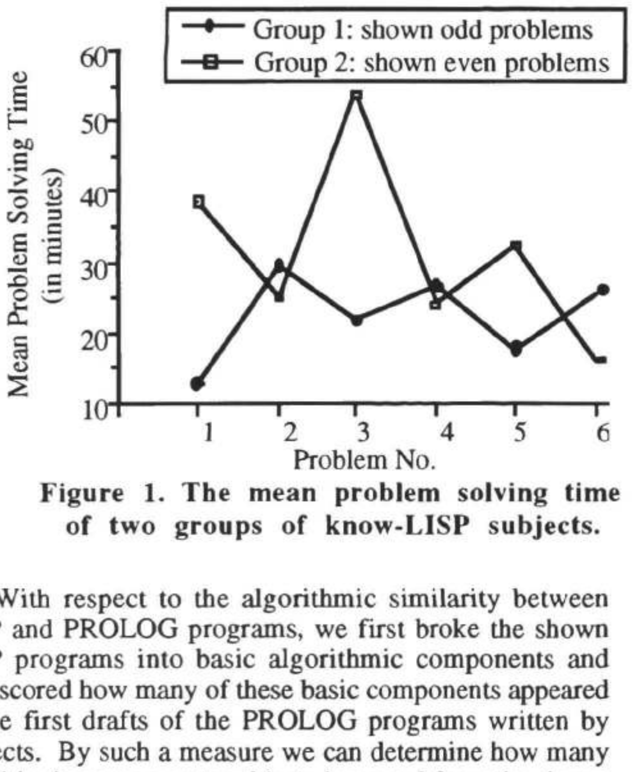

Figure 1. The mean problem solving time of two groups of know-LISP subjects.

With respect to the algorithmic similarity between LISP and PROLOG programs, we first broke the shown LISP programs into basic algorithmic components and then scored how many of these basic components appeared in the first drafts of the PROLOG programs written by subjects. By such a measure we can determine how many algorithmic components subjects imported from the shown LISP solutions to PROLOG programming. The data for this measure are presented in Figure 2. The transfer pattern of the results is the same as the patterns appearing in the first measure; this again demonstrated significant transfer from the shown LISP solutions to PROLOG programming. Again, a three-way ANOVA was performed on the data; the test results once more showed significant main effect due to whether or not LISP solutions were shown: $F(1, 7) = 47.53$, $p < 0.001$.

The above two measures largely contribute to our understanding of problem solving transfer from particular

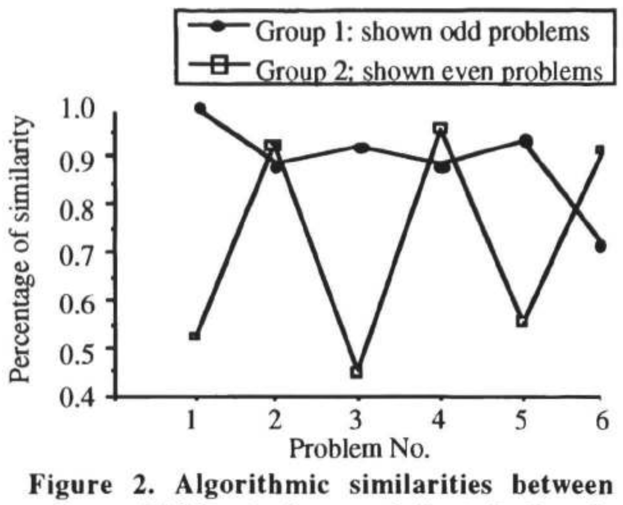

Figure 2. Algorithmic similarities between shown LISP solutions and first drafts of PROLOG programs by Know-LISP subjects

LISP programs to PROLOG programming. To see the effect of learning transfer of LISP knowledge upon PROLOG learning, we used the problem solving time for two comparisons: one between Know-LISP subjects versus Not-Know-LISP subjects, and the other between those who had stronger LISP background versus those who had weaker LISP background among Know-LISP subjects. However, the Not-Know-LISP subjects only finished on average 3.5 problems in this session. Thus, here we only used the problem solving time for the first three problems. Table 2 show the data for these two comparisons; as only six Not-Know-LISP subjects finished three or more problems, the data presented for this condition were only averaged over them. For Know-LISP subjects, the two classes of Strong-LISP and Weak-LISP were determined by the mean problem solving time spent by them on the two LISP pretest problems. As seen from the data, Know-LISP subjects spent less time on solving all the three problems; that is, they were learning faster than Not-Know-LISP subjects. The results also indicated that stronger LISP background helped subjects to learn PROLOG faster. Overall, these results tend to indicate that LISP knowledge indeed facilitates PROLOG learning. However, as the questionnaire information indicated, those who were good in LISP were usually also good in PASCAL or other programming languages; therefore, in general, we may say it is the knowledge of previously learned languages which helps learning a new programming language.

Although there were some syntactic interferences from LISP knowledge to PROLOG programming such as confusing list conventions in the two languages, these interferences seemed to be insignificant and could be avoided by paying more attention in both programming and debugging. Stylistically, LISP knowledge also seemed to make subjects come up with PROLOG solutions which were more procedural that they might have otherwise, and led to other confusions such as the notions of a predicate in

PROLOG and a function in LISP. However, these negative effects could be easily overcome by conscious practice. To some extent, since our introduction to PROLOG based on LISP has emphasized the differences between the two languages, subjects were prepared to avoid difficulties.

Table 2. The comparison between Know-LISP and Not-Know-LISP subjects and the comparison between Strong-LISP and Weak-LISP subjects.

		GRE/SAT Math scores	Lisp Experience	Time for problem 1	Time for Problem 2	Time for Problem 3	Mean
Know-LISP	Strong-LISP	714	41.0	16.3	19.5	30.5	22.1
	Weak-LISP	740	79.8	35.7	34.5	45.5	38.6
Not-Know-LISP		710		53.3	36.8	55.2	48.4

Experiment 2: Transfer from PASCAL/PROLOG to LISP

The purpose of this second experiment was to investigate learning transfer from either PASCAL or PROLOG to LISP. Here we focused our attention specifically on transfer of knowledge of writing repetition programs in these languages. In each of these languages there are two ways of implementing repetition, i.e., iteration and recursion. However, in PASCAL iteration is usually stressed while in PROLOG recursion is usually emphasized.[2] We would expect that PASCAL programmers might be more ready to transfer knowledge of iteration to LISP programming while PROLOG programmers might be more ready to transfer knowledge of recursion to LISP programming. The experimental design used here was specifically aimed at testing this hypothesis.

Methods

Subjects. 9 subjects participated in this experiment. They were all from CMU; among them 2 were undergraduates and 7 were graduates. Eight of them knew PASCAL before taking part in the experiment, while only three knew PROLOG beforehand (i.e., two knew both PASCAL and PROLOG). From the information collected from questionnaire, subject's averaged GRE/SAT math scores were 734. For those who knew PROLOG, their averaged self-ratings of PROLOG skill were 3.29; for those who knew PASCAL, their averaged self-ratings of PASCAL skill were 3.33; besides, all but one subject knew C, their averaged self-ratings of C skill were 3.00. The GRE/SAT math scores of the Know-PASCAL condition and that of the Know-PROLOG condition were comparable.

Design. The experimental design is shown in Table 3. Basically, six subjects who knew PASCAL but did not know PROLOG were used as the Know-PASCAL condition; while the other three, all of whom knew PROLOG but two of them also knew PASCAL, consisted the Know-PROLOG condition (the small number of subjects in this condition was the result of our difficulty of recruiting PROLOG programmers as subjects from the CMU community). There were three experimental sessions, each approximately one and half hours long. The manipulation of Know-PASCAL versus Know-PROLOG, together with the manipulation of presenting recursive vs. iterative problems in Session 2 & 3, were aimed to test the hypothesis mentioned above.

Materials. In Session 1, three problems were used to pretest subject's knowledge of either PASCAL or PROLOG programming; these problems are Summorial, Read-month, and Fibonacci. Also in Session 1, subjects were asked to read an introduction to LISP based on either PASCAL or PROLOG according to their conditions. The materials covered in these introductions included basic concepts and syntax, arithmetic expressions, flow-of-control, and some intermediate data types in LISP. Subjects normally spent about one hour to cover the major portion of the material in Session, and spent about twenty minutes in Session 2 & 3 to cover the rest. There were 12 problems used in Session 2 & 3 for LISP programming. These problems were: Simple-Expression, Circle-Area, Line-Slope, Add-Fraction, Least-Common-Multiplier, (the next four on recursion) Summorial, Integer-Power, Add-Together, Ackerman, (the next three concerning iteration) Summorial, Add-Together, Exponential.[3]

Procedure. As in Experiment 1, subjects were asked to fill out the questionnaire form. For programming in the three languages, almost the same procedure as used in Session 4 of Experiment 1 was followed in all the sessions of this experiment. However, here instead of using the standard EMACS editor we adopted a specifically designed editor based on EMACS; this editor was used to time-stamp each step of interaction subjects made with the computer. Each time when the subject began to edit or modify a program he was requested to turn on the timing editor; and each time when he quit the editing window to

[2] The PROLOG interpreter employs a back-tracking mechanism which inherently implies recursive programming is natural for most of problems which involve repetition.

[3] Again, the problems and instructional materials used in this experiments can be obtained by writing to the authors.

go to the debugging window he was asked to turn off the timing editor. This editor enabled us to collect some fine-grained measures of the time course of programming.

Table 3. The design of Experiment 2 on transfer from PASCAL/PROLOG to LISP.

	Know-PASCAL condition	Know-PROLOG condition
Session 1	Questionnaire; Three PASCAL problems for pretesting PASCAL knowledge; Read an introduction to LISP which is based on PASCAL knowledge.	Questionnaire; Three PROLOG problems for pretesting PROLOG knowledge; Read an introduction to LISP which is based on PROLOG knowledge.
Session 2	Five LISP problems for basic syntax and simple arithmetic expressions; Two LISP problem for recursion.	The same as the know-PASCAL conditions.
Session 3	Two LISP problem for recursion; Three LISP problems for iteration.	The same as the know-PASCAL conditions.

Results and Discussion

For the time measure, we dissected the problem solving time for each problem into programming (coding) time and debugging time; this could be easily done by counting the time spent either in the editing window or in the debugging window. Taking the method from Singley & Anderson (1990) and Katz (1988), we further decomposed the programming time into thinking time (planning time) and keystroking time (execution time). The keystroking time was defined as follows:

Keystroking time = Σ over all keystrokes T

Where T is either the interval between two consecutive keystrokes if that is less than or equal to 2 seconds, or just 2 seconds if the interval is greater than 2 seconds. The thinking time is simply the rest of the programming time; that is,

Thinking time=Programming time- Keystroking time.

We could further decompose the thinking, keystroking and debugging times into first-draft (initial-draft) time and rest-draft (subsequent-draft) time; this could be done by considering the transitions made between the two windows.

In Wu & Anderson (in preparation) we found that time-savings in problem-solving transfer were largely localized in the first-draft coding and debugging. This finding was verified also to be true for our present situation of learning transfer. So in this article we will limit our report of data only to the first-draft time; however, the data pattern also applies to the total problem solving time. [4]

Table 4 presents the data for the first-draft time for a comparison between Know-PASCAL and Know-PROLOG subjects. A two-way ANOVA (recursion/iteration X PASCAL/PROLOG) was performed on the data of PASCAL/PROLOG programming time. The effect due to recursion/iteration was insignificant: $F(1, 8) = 0.51$, $p > 0.1$; the effect due to PASCAL/PROLOG was also insignificant: $F(1, 8) = 0.09$, $p > 0.1$; however, the interaction between them was significant: $F(1, 8) = 8.04$, $p < 0.05$. Another similar two-way ANOVA was performed on the data of LISP programming time for both PASCAL and PROLOG knowers. The effect due to recursion/iteration was significant: $F(1, 32) = 6.72$, $p < 0.05$; the effect due to PASCAL/PROLOG was also significant: $F(1, 32) = 13.08$, $p < 0.05$; however, now the interaction between them was insignificant: $F(1, 32) = 0.21$, $p > 0.1$. Also from the table we see that the time ratio between these two conditions on recursion problems remained almost the same from PASCAL/PROLOG to LISP. Know-PROLOG subjects were better on recursion that Know-PASCAL subjects in the initial test. This advantage was maintained in recursive programming in LISP. That is, PROLOG programmers indeed transferred what they learned about recursion in PROLOG to LISP programming. On the other hand, our Know-PROLOG subjects also did better on iteration in LISP even though they had not in the original test. Concerning iteration, it seems that no general conclusion could be reached about transfer between the three languages from this experiment.

General Discussion and Conclusion

In explaining knowledge transfer, the theory of identical elements had been proposed very early in the century (Thorndike & Woodworth, 1901). Basically, the theory postulates that it is the common elements shared by two domains of knowledge that enables the knowledge acquired in one domain to be transferred to the other. As there are indeed a great many commonalities between LISP and PROLOG, it is not surprising that in the first experiment we witnessed substantial transfer from LISP knowledge to learning PROLOG. With respect to recursion, as PROLOG and LISP have more in common than PASCAL and LISP have, so in the second experiment we demonstrated that PROLOG programmers had relative advantage over PASCAL programmers in learning the recursive part of LISP.

[4] A more detailed report can be obtained by writing to the authors.

Table 4. Data comparison between the Know-PASCAL and the Know-PROLOG conditions in Experiment 2 (time in seconds).

	Total time on 3 PASCAL/ PROLOG problems	Time on one PASCAL/ PROLOG iteration problem	Time on one PASCAL / PROLOG recursion problem	Total time on 12 LISP problems	Total time on 3 iteration LISP problems	Total time on 4 recursion LISP problems
Knew-PASCAL	2173	547	917	8051	2780	2628
Knew-PROLOG	2152	922	352	3778	1313	980
PROLOG/PASCAL	99%	168%	38%	47%	47%	36%

In Wu & Anderson (in preparation), we reported three experiments demonstrating the existence of problem solving transfer among PASCAL, LISP, and PROLOG. To account for the results, there we proposed that there are basically three levels of transfer across programming in different languages -- namely, the syntactic, algorithmic, and problem levels. There we found that at the syntactical level there were minor interferences (a type of negative transfer) among programming in different languages; at the algorithmic level, substantial positive transfer was manifested as on most occasions subjects used the same algorithm for the same problem in different languages and a great deal of time-saving was exhibited in reprogramming. Besides these two levels, there we also found that sometimes, although different algorithms were used for the same problem in different language, there was still positive transfer manifested (as time-saving); this transfer must have resulted from problem understanding and we postulated it as transfer at the problem level.

In the present investigation we have demonstrated transfer both for problem solving and for learning. Here again we showed that syntactic transfer is largely negative but only plays a minor role in learning and problem solving. We also showed that algorithmic transfer constitutes the major portion of transfer, as manifested by saving the total problem solving time, reducing the number of program drafts needed for revising the program, and importing algorithmic components from LISP solutions to PROLOG programming. However, in both experiments we did not see that different algorithms were used for the same problem in different languages, thus transfer at the problem level was not demonstrated in this study.

On the other hand, as this study was focused on learning transfer, we also witnessed a higher level transfer than the syntactic and algorithmic level transfer; and this transfer is also different from the transfer at problem level. This level may be called learning transfer -- the transfer of the most general programming knowledge which one does not need to learn again and again when learning new programming languages. One example of such transfer would be the notion of recursion; after learning how to implement repetition in a recursive way in one language one can easily transfer that knowledge to learning a new language provided that the new language also easily facilitates recursion.

W would conclude from the present study:

1. There was significant transfer from learned LISP knowledge to learning PROLOG; that is, LISP knowledge helped to learn PROLOG faster. Moreover, the more LISP knowledge the more facilitation to learning PROLOG.

2. Although syntactically and stylistically, there were some interferences from LISP to PROLOG programming, these negative effects were greatly overwhelmed by positive transfer at other higher levels. At the algorithmic level, transfer was manifested as substantial time saving, fewer revisions of programs, and importing algorithmic components from shown LISP solutions to PROLOG programming.

3. As PROLOG programmers usually would be more knowledgeable than PASCAL programmers on recursive programming, PROLOG programmers seemed to have relative advantages over PASCAL programmers in learning the recursive part of LISP programming.

4. Aside from the three levels of transfer we proposed for problem solving transfer among programming languages -- the syntactic, algorithmic, and problem levels, one more level of transfer was proposed there -- the learning transfer. The knowledge responsible for this transfer is the most general programming knowledge across several languages and across a large number of problems, such as the knowledge about recursive programming.

References

Anderson, J. R. 1983. *The Architecture of Cognition.* Cambridge, Mass: Harvard University Press.

Katz, I. R. 1988. Transfer of Knowledge in Programming. Ph.D. Diss., Dept. of Psychology, Carnegie Mellon Univ.

Mayer, H. G. 1988. *Programming Languages.* New York: MacMillan Publishing Company.

Singley, M.K.; Anderson, J. R. 1990. *The Transfer of Cognitive Skill.* Cambridge, Mass: Harvard University Press.

Thorndike, E. L.; Woodworth, R. S. 1901. The influence of improvement in one mental function upon the efficiency of the other function. *Psychological Review,* Vol. 8.

Wu, Q.; Anderson, J. R. in preparation. Problem solving transfer among programming languages (submitted to *Human-Computer Interaction*).