Problem-solving Transfer among Programming Languages

Quanfeng Wu and John R. Anderson
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

June 4, 1990

For submission to: *Human-Computer Interaction*

Problem-solving Transfer among Programming Languages

Quanfeng Wu and **John R. Anderson**
Carnegie Mellon University

ABSTRACT

This study investigated knowledge transfer among three programming languages -- namely, LISP, PROLOG, and PASCAL. Three experiments were conducted: the first two on transfer between LISP and PROLOG, while the third on transfer between LISP and PASCAL. The results from all the experiments showed evidence of positive transfer; the transfer was positive in terms of time savings from programming in one language to that in another for the same problem. Furthermore, it was found that transfer in programming was largely localized in writing the first drafts of programs. Sometimes this transfer was manifested even though different algorithms were used in programming in different languages for the same problem. To account for the results on transfer across programming languages, a three-level transfer -- namely, transfer at the syntactic, algorithmic, and problem levels -- theory was proposed in this paper.

Author's present address: Quanfeng Wu and John R. Anderson, Department of Psychology, Carnegie Mellon University, PA 15213.

CONTENTS

1. INTRODUCTION

Most people have the experience that if they write a program in one language then it will be much easier to write the program in a second language. This paper will demonstrate the accuracy of that experience and explore the basis for this transfer. Theoretically, this is an interesting phenomenon because it reflects on the mental representation of a computer program. Since the codes in the two languages may be completely different, then what must be transferring is the understanding of the program. But what is really meant by "the understanding of the program"? In this paper we will offer some proposals as to what is being transferred.

This theoretical question is of practical significant as well. Not only is it the case in the real world that we have to convert programs from one language to another, but we also have to transfer skill in programming generally from one language we learn to a later language. It is often thought that when someone learns how to program one is learning problem solving skills which are more general than programming in any particular language (e.g., Mandinach & Linn, 1987). This again raises the question of the nature of the knowledge representation that is involved. Perhaps if we knew that was in common between two languages we would be in a better position to teach for transfer. This is in fact the issue that led us to this research project. We are also building a computer tutoring system to teach multiple programming languages with the goal that there be substantial transfer among the languages (Anderson, Corbett, Finchman, Hoffman, & Pelletier, in preparation). We felt we needed to know better what could be transferred in order to pursue this goal.

There are two kinds of transfer possible among programming languages. One is transfer from learning one language to learning another. This type of transfer can be called learning transfer; it is the goal in our tutoring work. The transfer studied in Scholtz &

Wiedenbeck (1989) is also of this type; they studied transfer from knowing PASCAL and C to learning a new language ICON. The other kind of transfer is what we will study here. This kind of transfer may be called problem solving transfer;[1] it involves people who already know both languages and have to translate a solution from one language to the other. It is our belief that both types of transfer depend on commonalities in the representation of the problem solution; however, other relationships are also logically possible. As we only explore the second kind of transfer in this paper we will not be able to establish anything definite about this issue of whether the two types of transfer both depend on the same kinds of commonalities in program representation.

There is a long history of research on transfer (e.g., Thorndike & Woodworth, 1901; Kotovsky & Fallside, 1988; Singley & Anderson,1985, 1988, 1989). Particularly relevant to the present study is Katz (1988) which demonstrated substantial transfer between PASCAL and LISP programming. In the present study, three experiments were conducted -- the first two on transfer between LISP and PROLOG and the last on transfer between LISP and PASCAL. Basically, the first experiment demonstrated the existence of transfer between writing LISP and writing PROLOG programs; the second attempted a detailed analysis of the locus of this transfer; while the third was an extension of the second experiment to transfer between LISP and PASCAL. The last experiment also replicated some of Katz's results on transfer between these two languages.

The rest of the paper is structured as follows: In the next section a brief comparison of the three programming languages involved in this study will be presented; this is purported to demonstrate that there are indeed commonalities shared by them. Then the three experiments will be reported and discussed; also, Katz's results will be contrasted with ours in the third experiment. Finally, some further discussions on transfer among

[1] Generally, analogical problem solving is also such a kind of transfer; namely, transfer among isomorphic problems. For examples, Simon & Hayes (1976), Smith (1986), and Kotovsky & Fallside (1988) called it transfer; while keane (1988) called it analogical problem solving.

programming languages will be presented; specifically, a three-level theory of transfer will be proposed to account for the results from the experiments.

2. A COMPARISON OF THE THREE LANGUAGES

As stated earlier, it is our belief that commonalities in knowledge representation constitute the basis for transfer. Therefore, here we will present a brief comparison of LISP, PROLOG, and PASCAL to illustrate some commonalities shared by them. This will supply a basis for understanding why transfer across them exists at all. Moreover, two examples of problems will be shown to further illustrate the similarities and differences among the three languages.

2.1. A Comparison of LISP, PROLOG, and PASCAL

Generally speaking, PASCAL, LISP, and PROLOG are three very different programming languages; they fall into three different categories of programming languages -- namely, procedural, functional, and logical languages (Figure 1). However, there are also a number of commonalities among them. More specifically, we can take a continuum view -- that is, viewing programming languages as constituting a continuum from purely prescriptive languages to purely descriptive languages. Taken that view, PASCAL, LISP, and PROLOG are seen as occupying overlapping ranges along the continuum with LISP standing in the middle. Following this view, it is also plausible that there are more commonalities between PASCAL and LISP and between LISP and PROLOG than between PASCAL and PROLOG. However, as there is neither purely prescriptive language nor purely descriptive language, even between PASCAL and PROLOG there are still a good number of commonalities. As the present study was only concerned with transfer between LISP and PROLOG and between LISP and PASCAL, we will only further explicate some commonalities and differences of these two pairs.

Insert Figure 1 about here

An outstanding difference between LISP and PROLOG is their control-flows. LISP is mainly a prescriptive language while PROLOG is more towards to descriptive. The control-flow of a LISP program is transparent seen from the static organization of the code; but this is not the case for PROLOG. The control-flow of a PROLOG program is embedded in the unification procedure, or in other words, the matching and backtracking mechanism, of the language implementation. However, in LISP programming recursion is usually a major form of control-flow; correspondingly, in PROLOG the mechanism of matching and backtracking also entails that programs are actually executed in a recursive way. Thus, recursiveness turns out to be an important commonality between LISP and PROLOG. As to data structures and facilities operating on them, LISP and PROLOG also have a great deal of commonalities. In both languages, the primary constructive data structure is the list. Although PROLOG offers far fewer built-in operators for list processing than LISP does, the basic operations facilitated by bother languages have strikingly correspondences. Another prominent commonality between LISP and PROLOG is their elementary constituents -- namely, the function in LISP and the predicate in PROLOG. Mathematically, there is a correspondence between the two; however, novice PROLOG programmers may also be very confused with the notion of predicate in programming.

A major commonality shared by LISP and PASCAL is their control-flows. That is, programs written in these two languages are executed sequentially according to the static order of their codes and can be organized hierarchically in that a procedure/function (in PASCAL) or a function (in LISP) can call itself and/or other procedures/functions (in

PASCAL) or functions (in LISP). However, usually, there is also a big difference between the emphases put on the repetition structures facilitated by these two languages. In general, there are two types of repetition control-flows -- namely, iteration and recursion. Although both iteration and recursion can be implemented in both languages, iteration is usually emphasized in PASCAL whereas recursion is usually stressed in LISP. To most programmers, this difference is simply incurred by the instructional materials which introduce PASCAL and LISP. However, this difference is also associated the compiling mechanisms normally used for implementing the two languages. That is, generally, recursion is less efficient in PASCAL than in LISP. Some striking differences existing between PASCAL and LISP are their facilities for manipulating data structures. In PASCAL, the basic types of constructive data structures are sets, arrays, records, and files of records. However, in constructing any complex data structures the base type of structures usually have to be the same. While in LISP, the primary type of constructive data structures is the list; however, this type of data structures is very powerful and versatile in constructing any complex data structures.

2.2. Two Examples of Programming Problems

Two examples of programming problems which were actually used as testing materials in the experiments are here chosen to further illustrate commonalities and differences among the three languages. The first problem is to evaluate the Fibonacci function; the problem and some modal programs are shown in Figure 2. Generally, there are two kinds of solutions to the problem -- that is, iterative and recursive solutions. Both solutions can be implemented in the three languages, though generally it is more difficult to write the iterative solution than the recursive solution, especially in PROLOG. [2] Nevertheless, to a great extent, this example shows that there may be numerous

[2] However, the iterative solution is more efficient for execution than the recursive one. The computational complexity of the iterative algorithm is only linear: $C_{iteration} = O(n)$; whereas, the complexity of the recursive algorithm is exponential: $C_{recursion} = O(5^{n/2})$.

commonalities among the programs written in the three languages for the same problem.

Insert Figure 2 about here

The second problem is to construct a simple database as to contain entries representing books and then to search the database to count the books by a given author. The problem and some modal programs are shown in Figure 3. Contrary to the previous example, here we see that the programs written in the three languages for the same problem may appear to be quite different from one another. That is, each of the implementations can utilize the specific facilities provided by the language and conform to the styles of programming in that language. However, although these programs are very different in appearance, there still may be some transfer in the understanding phase of programming for the problem.

Insert Figure 3 about here

---

3. EXPERIMENT 1: TRANSFER BETWEEN LISP AND PROLOG

Basically, this experiment was to demonstrate the existence of transfer between programming in LISP and in PROLOG. As we were only investigating problem solving transfer between programming in the these languages, our subjects knew both LISP and PROLOG beforehand. Also, here transfer was studied bidirectionally; that is, either from LISP to PROLOG or from PROLOG to LISP.

3.1. Method

Subjects. The six subjects involved this experiment were from the CMU (Carnegie Mellon University) community; among them 1 undergraduate, 2 graduate students, and 3 research assistants. They were reimbursed for participating in the experiment. From the information taken from Questionnaire, the subject's mean GRE/SAT scores were 750; the mean self-ratings of LISP skill was 3.83, while that for PROLOG skill was only 2.50. Thus, in average, the subjects were relatively more proficient in LISP than in PROLOG.

Design. Figure 4 shows the design of the experiment. The 6 subjects were partitioned into two groups. There were four sessions of the experiment; in each session, the subjects were asked to solve one problem. For each problem, the subjects had to write two programs, one in LISP and the other in PROLOG. The order of programming in LISP and PROLOG was alternated from one problem to the next and switched between the two groups, as shown in the figure. This experimental design is a mixture of the within-subject and between-subject designs; one advantage of this design is that it would allow us to analyze the transfer bidirectionally.

Insert Figure 4 About Here

Materials. The four problems used in the experiment were Fibonacci Function, Powerset, Simple Database Searching, and Simple Expression Parsing; among them, two were already shown in Figure 2 & 3. [3]

Procedure. Before the experiment actually begun, the subjects were requested to fill out a questionnaire form. The relevant information gathered from the questionnaire was SAT/GRE math scores and self-ratings of their proficiencies in various programming

[3] All the problems and modal solutions to them can be obtained by writing to the authors.

languages.

The actual procedure of the experiment is shown in Figure 5. The subjects did programming on the ANDREW system, a campus network at CMU. Two windows were created: an editing window in which programs were typed in and modified and a debugging window in which LISP and PROLOG programs were debugged and tested. For each problem, the subject went through the cycle of editing, debugging, if errors found then editing and debugging again, until the programs worked correctly. Various drafts of the program under writing were saved into different files. The versions of the languages actually used in the experiment were COMMONLISP and CPROLOG; the editor was EMACS.

Insert Figure 5 about here

Throughout the whole experimental sessions, the experimenter was with the subjects and helped them as necessary with the syntax of the programming languages and the computer system. Test cases for the problems were stated on the problem statement sheets, but the final correctness of the programs was determined by the experimenter.

3.2. Results

Two kinds of quantitative measures were used here for analyzing the results of the experiment: the time measures and the numbers of program drafts produced by subjects. For the time measures, we only collected the total problem solving time and the first draft time in this experiment. The first draft time included the time spent on thinking, typing, and debugging; so was the total problem solving time. The difference between the total time and the first-draft time is the rest-draft time -- namely, the time spending on revisions of the initial draft of problem; this rest-draft time was also used for data analysis. Furthermore, we calculated the mean per-draft time for the subsequent drafts; that is,

Per-draft time = Rest-draft time / (No. of drafts - 1).

In other words, this time measure was used to gauge the time spent on each rest draft in average.

The data for the mean total problem solving time are shown in Figure 6; the data averaged across the subjects within one group for the four separate problems are in the graphical form and the data averaged across the two groups and the four problems are in the tabular form. From the graph we can see that there were time savings manifested from solving a problem in one language to solving it in the other language, a kind of positive transfer in terms of cognitive expenditure. The only exception was the first problem for Group 1; there was more time spent on the second PROLOG program than the initial LISP program (one factor for this result is that one in this group had a relatively weaker skill in PROLOG). A three-way transfer X language X problem (transfer: first/second program; language: LISP/PROLOG; problem: 1 to 4; that is, 2 X 2 X 4) pattern ANOVA was used to test the statistical significance of the data. Since each subject only realized half of the conditions in this design, for purposes of statistical convenience we treated each pair of subjects as a macro subject; the subjects in the two groups were paired according to their programming skills as measured by their total problem solving times. The results revealed significant main effect due to transfer: $F(1, 2) = 367.11$, $p < 0.01$. As can be seen, despite the small numbers of degrees of freedom, the effects were typically highly significant.

Insert Figure 6 About Here

Figure 7 shows the data for the mean first-draft time. Here again, the data averaged across the subjects within one group for individual problems are shown in graphical form whereas the data averaged across both two groups and four problems are shown in the tabular form. The data pattern shown in the graph was consistent with the transfer pattern we would expect; this indicated a substantial portion of transfer was manifested on first drafts. A second three-way ANOVA was used to analyze these data of the first-draft time; the results indicated significant main effect due to transfer: $F(1, 2) = 230.69$, $p > 0.01$. As mentioned above, we also calculated the rest-draft time; Figure 8 presents the results of this time measure. Again, a third ANOVA was used to analyze these results; the analysis indicated significant main effect due to transfer: $F(1, 2) = 187.03$, $p > 0.01$.

Insert Figure 7 About Here

Insert Figure 8 About Here

The second measure used for data analysis was the number of program drafts made by subjects in the course of programming; roughly speaking, this measure also corresponds to the numbers of mistakes subjects made in programming. Because the experimenter had assisted the subjects in the syntactic checking of their programs, these mistakes were largely semantic errors, in other words, algorithmic errors. Figure 9 presents the data in terms of this measure; another three-way ANOVA was performed and the test showed significant main effect due to transfer: $F(1, 2) = 25.41$, $p < 0.05$. Figure 9 also presents the mean per-draft time for subsequent drafts averaged across both groups and problems. From the results, we see that this per-draft time was fairly stable from programming in the first language to in the second. In other words, the transfer on subsequent drafts was mainly manifested as fewer drafts rather than less time per draft. The subjects usually attempted to fix one bug each draft, although on some occasions some might trace down several small bugs in one pass or one big bug in several passes.

Thus, we can grossly identify the number of drafts with the number of semantic bugs in the program.

Insert Figure 9 About Here

3.3. Discussion

From the quantitative results shown above, we clearly see that there is substantial transfer from solving a problem in the first language (LISP or PROLOG) to solving it in the second language (PROLOG or LISP) in terms of time. The time saving is consistent for both total problem solving time and the time for finishing the first drafts of the programs. Although in this experiment we did not have a very accurate measure of the planning time, the pattern of the first draft times in fact seems to indicate that subjects did transfer a lot of the algorithmic knowledge gained from programming in the first language to in the second. We will refer this type of transfer as algorithmic transfer.

It is also conceivable that subjects might be just mechanically translating the code of one language into the code of the other without reference to a deeper algorithmic representation. However, this seems unlikely as our subjects usually did not access to the their own solution in the first language. Only one subject seemed to make such mechanical translations of her solution for Powerset problem in LISP to PROLOG and her solution for Parse-Expression from PROLOG to LISP. She was the weakest programmer in both LISP and PROLOG among all the subjects. She asked for print-outs of her solutions and since she found a great deal of difficulty in these two problem we allowed her access to her solutions in the first language; she then worked on paper to break the solutions into components and translate them into the second language before actually typing in the codes on the computer.

Another possibility of transfer other than the above two is a kind of transfer which may occur before algorithmic planning; in other words, its level is higher than the above two types of transfer. For this type of transfer, subjects may use rather different algorithms in the two languages and yet they may still transfer something which is the result of problem understanding. In fact, two subjects in this experiment changed their algorithms on four problems. In later experiments, we will again witness such cases of using different algorithms in different languages and yet still of positive transfer; and we will refer transfer occurring at this level as problem-level or pre-algorithmic transfer.

At the syntactic level, due to switching back and forth between programming in LISP and in PROLOG, it appeared that subjects did get some syntactic interference between programming in these two languages, in both coding and debugging processes. For example, they would forget to use the correct form of list structure representation -- that is, instead of (a b c) in LISP, they might use (a, b, c). (The correct list structure representation is [a, b, c] in PROLOG.) However, it appeared that the more attentive subjects were, the less interference seemed to occur; when some of them were getting bored in later problems, they were more likely to produce such syntactic interference. Overall, such minor syntactic interference was overwhelmed by the positive transfer at the more semantic level.

4. EXPERIMENT 2: TRANSFER BETWEEN LISP AND PROLOG -- A FURTHER STUDY

This is a further experiment on transfer between LISP and PROLOG. In the first experiment we collected only rather gross measures of time saving in writing a program. In this second experiment we attempted to decompose the data down into smaller units to allow greater localization of the transfer. Also, two additional problems were incorporated in this experiment to give further observations of transfer.

4.1. Method

Subjects. 8 subjects were involved in this experiment. They were all from CMU; among them 3 were undergraduates and 5 were graduates. They were reimbursed for taking part in the experiment. From the information gathered from Questionnaire, the subject's mean GRE/SAT scores were 760; the mean self-ratings of LISP skill was 4.14, while that for PROLOG skill was only 3.14. Thus, again as in Experiment 1, on average, the subjects were relatively more proficient in LISP than in PROLOG.

Design. Basically, the design was the same as that in Experiment 1 (see Figure 4) except that two additional problems were used here.

Materials. The six problems used in this experiment were Fibonacci-Function, Powerset, Database-Search, Add-Fractions, Print-Number, and Parse-Expressions (again in the order of actual presentation).

Procedure. Almost the same procedure as used in Experiment 1 was followed in this experiment (see Figure 5). However, here instead of using the standard EMACS editor we designed a special editor based on EMACS; this editor was used to time-stamp each step of interactions between the subject and the computer. Each time when the subject begun to edit or modify a program he was requested to turn on the timing editor; and each time when he quitted the editing window to go to the debugging window he was asked to turn off the timing editor (see the two shadowed boxes in Figure 5). This editor enabled us to collect fine-grained measures of time saving and to attempt a detailed analysis of the locus of transfer.

4.2. Results

Two types of measures were used in analyzing the data from this experiment -- the time measures and the numbers of program drafts made by the subjects in programming.

For the time measures, 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 (1986) and Katz (1988), we further decomposed the programming time into thinking time (planning time) and keystriking time (execution time). The keystriking time was defined as follows:

$$\text{Keystriking time} = \sum_{\text{over all keystrokes}} T$$

Where T is either the interval between two consecutive keystrokes itself if it is less than or equal to 2 second, 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 - Keystriking time.

In addition, we also could further decompose the thinking, keystriking 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. Moreover, a particular component of the thinking time was used for further analysis -- that is, the time spent on thinking before any typing, including problem reading and understanding; we will refer this component as pre-programming time. This measure is only the first portion of the first-draft thinking time; it largely reflects the process of problem understanding, although some components of algorithm selecting and planning may also be involved within it.

Figure 10 presents the mean times at various levels of localization -- namely, the various levels of decomposition of the problem solving times. The data presented were averaged over both the problems and the two groups. From the figure we can see that substantial transfer was again exhibited both from PROLOG to LISP and from LISP to PROLOG. For the first-draft times, the major part of transfer occurred in the first-draft thinking and debugging; but no significant transfer manifested in the first-draft keystriking.

Insert Figure 10 about here

As in Experiment 1, transfer X language X problem (2 X 2 X 6) pattern ANOVAs were performed on the separated data for the six individual problems. For the total problem solving time, the main effect due to transfer was significant: $F(1, 3) = 144.3$, $p < 0.01$; for the first-draft thinking time, the main effect due to transfer was also significant: $F(1, 3) = 347.12$, $p < 0.01$; however, for the first-draft keystriking time, the main effect due to transfer was not significant: $F(1, 3) = 5.57$, $p > 0.05$; for the first-draft debugging time, the main effect due to transfer was marginally significant: $F(1, 3) = 10.21$, $p < 0.05$. For all these time measures, the main effects due to language were not significant. Figure 10 also presents the data for the pre-programming time; as seen from the results, this time only occupied a small portion of the first-draft thinking time. However, for this small portion of thinking time -- namely, the time for problem reading and understanding -- substantial transfer was also demonstrated. Another the same pattern ANOVA as the above was performed on the individual data of this time measure; the results revealed significant main effect due to transfer: $F(1, 3) = 21.03$, $p < 0.02$. While the absolute times were small, in terms of relative gains these pre-programming times showed the greatest transfer benefit with subjects only spending 23% as much time on the second program.

For the rest-draft times, there was also substantial transfer manifested; this transfer was distributed over the thinking, keystriking, and debugging times. This distribution was related to the numbers of drafts subjects made in the course of programming; that is, the more drafts made the more rest-draft time. Figure 11 presents the results of this measure. Again, a three-way transfer X language X problem ANOVA was performed on the data of this measure; the results revealed significant main effects due to transfer: $F(1, 3) = 35.33$, $p < 0.01$. As in Experiment 1, we also calculated the mean per-draft time for the rest drafts; the results are also presented in Figure 11. The results indicated there was no transfer for this per-draft time; that is, the transfer that occurred on the rest drafts was mainly the result of the fewer total drafts.

Insert Figure 11 about here

4.3. Discussion

The first experiment demonstrated transfer between LISP and PROLOG programming; this experiment further indicated this transfer was positive and substantial in terms of time saving (again about 37% total time saving). Furthermore, this experiment indicated that the effect is in the planning of the first draft and in the number of subsequent revisions that are necessary. This is consistent with our identification of the transfer with the algorithm design phase. Subjects took longer in designing the algorithm (the first draft time) and had a less complete algorithm (the number of drafts) in the first language than in the second.

Also, as in Experiment 1, there were several cases of the same subjects using different algorithms and/or styles in LISP and PROLOG programming while for the same problems. Again, these cases demonstrated the high-level transfer as discussed in Experiment 1. We will discuss these cases in detail in GENERAL DISCUSSION section.

5. EXPERIMENT 3: TRANSFER BETWEEN LISP AND PASCAL

This experiment was basically an extension of the second experiment to transfer between LISP and PASCAL. Some of Katz's (1988) results on transfer between these two language were replicated.

5.1. Method

Subjects. Again, 8 CMU students -- 3 undergraduates and 5 graduates -- were involved in this experiment. They were reimbursed for participating in the experiment. From the information collected in Questionnaire, the mean GRE/SAT math scores of the subjects were 730; the mean self-ratings of LISP proficiency was 3.29, while that for PASCAL was 3.71.

Design. The design of this experiment was almost the same as that in Experiment 2 except that PROLOG was substituted by PASCAL and that only five problems were used in this experiment.

Materials. The five problems used in this experiment were Fibonacci-Function, Add-Fractions, Print-Number, Powerset, and Database-Search. (Since it is very difficult to program Parse-Expression in PASCAL, we did not include this problem in the present experiment.)

Procedure. Almost the same as in Experiment 2, the procedure of this experiment is also illustrated in Figure 5 where in the debugging window LISP and PASCAL (instead of PROLOG) programs were debugged and tested. The actual versions of the languages used on the ANDREW system were COMMONLISP and UNIX-PASCAL.

5.2. Results

The same measures as in Experiment 2 were also used in the present experiment for data analysis. The averaged times at various levels of localization are presented in Figure 12. As in the situation of transfer between LISP and PROLOG, here we also see that transfer between LISP and PASCAL was demonstrated. Furthermore, a large portion of transfer seemed to occur in the first draft programming and debugging; also, some transfer was distributed among the rest drafts. Within the first-draft thing time, the pre-programming time was again a small fraction but showed great proportionate transfer (72% saving).

Insert Figure 12 about here

As in the previous experiment, transfer X language X problem (2 X 2 X 5) pattern ANOVAs were performed on the separated data for the five individual problems. For the total problem solving time, the main effect due to transfer was significant: $F(1, 3) = 74.04$, $p < 0.01$; for the first-draft thinking time, the main effect due to transfer was also significant: $F(1, 3) = 124.09$, $p < 0.01$; however, for the first-draft keystriking time, the main effect due to transfer was marginally significant: $F(1, 3) = 8.84$, $p < 0.06$; for the first-draft debugging time, the main effect due to transfer was marginally significant: $F(1, 3) = 20.15$, $p < 0.05$; for the pre-programming time, the main effect due to transfer was significant: $F(1, 3) = 43. 76$, $p < 0.01$.

Also as in Experiment 2, we would expect that the transfer occurred on rest-drafts was related to the savings of the numbers of program drafts made in transfer programming. Figure 13 shows the results of this measure on the five individual problems. Generally, this data pattern conformed to the transfer occurred on rest-drafts. However, as PASCAL is a compiling language whereas LISP is interpretative (that relatively more facilitates program debugging), there were usually more drafts of PASCAL programs than LISP programs for the same problems. Another transfer X language X problem ANOVA was performed on the results of this measure; the analysis revealed significant main effect due to transfer: $F(1, 3) = 99.81$, $p < 0.01$.

Insert Figure 13 about here

As mentioned earlier, Katz (1988) also investigated transfer between LISP and PASCAL programming. The first experiment of Katz (1988) employed the same design as we used here; but he used only two problems -- namely, Add-Fractions and Print-Number -- as testing materials. (Actually, the problem Print-Number used in Katz's was different from ours: he provided two procedures which could be called by the to-be-written program, whereas in our experiment we required subjects to start from scratch.) His results also demonstrated comparable transfer between in programming in these two languages.

6. GENERAL DISCUSSION

In this section, we will present some further discussions on the results from the three experiments; especially, we will propose that transfer across programming language may occur on three different levels. Furthermore, we will offer some speculations on the nature of program representation involved in programming where transfer would exhibit.

6.1. Three Levels of Transfer in Programming

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 two domains of knowledge that enables the knowledge acquired in one domain to transfer to the other. In our earlier analyses on the similarities and differences among PASCAL, LISP, and PROLOG, we have seen that there are indeed a number of commonalities shared by the three languages. Therefore, in the light of the identical-element theory, it is not surprising that in the three experiments we witnessed a great deal of transfer between PASCAL and LISP and between LISP and PROLOG.

However, as noted earlier, although sometimes the programs written in different languages by the same subjects appeared to be very different in algorithms, data structures, and programming styles, there was still quite a lot transfer manifested in programming from one language to another. For instance, in Experiment 3 all the subjects used different algorithms to implement Database-Search problem (see Figure 3) in the two languages; namely, all used recursive solutions in LISP and iterative solutions in PASCAL. Nevertheless, on this problem the subjects still displayed a 27% time savings. A transfer X language ANOVA (2 X 2) was performed on the data of total problem solving time for this problem; the analysis revealed significant main effect due to transfer: $F(1, 3) = 43.10$, $p < 0.01$.

To account for the above results, we propose that there are three levels of transfer across programming in different languages. These three levels may be called the syntactic level, the algorithmic level, and the problem level. The syntactic level is the lowest level and played a relatively minor role, positive or negative, in our transfer results. However, there was some transfer at this level. We have already noted the interference between list conventions in PROLOG and LISP. Subjects also transferred variable names and function names from one language to another; Katz (1988) also documents a number of instances of transfer at this level.

The algorithmic level refers to the knowledge that is semi-language-dependent in that the particular algorithm chosen may be just appropriate for only certain languages (e.g., choosing recursive algorithms for LISP and PROLOG whereas iterative algorithms for PASCAL). However, we suspect much of our transfer was at this level. Most of the occasions subjects used highly similar algorithms in the two languages; that is, subjects copied algorithms (not code) from one language to another. This is why the transfer was mainly located in the first-draft planning time and number of drafts. Subjects were able to use the algorithm they had perfected in the first language to cut down the planning in the second language, and, since they already had a worked out algorithm in one language, there was less need for revision.

However, as we noted all transfer could not be at the algorithmic level since sometimes transfer was displayed even when a different algorithm was used in the two languages. We believe this involves understanding the problem itself and the knowledge responsible for this transfer is language-independent but problem-related. The dramatic decreases in pre-programming time are one manifestation of transfer at this problem level. Clearly, subjects had to spend little time rereading the problem statement. However, transfer at this level goes beyond reading the problem statement. For instance, in the Database-Search problem (Figure 3) subjects would realize in the first program that they would have to have a system that represented multiple books by the same author but not multiple authors of the same book. This problem constraint was important to designing the algorithm for LISP or PASCAL; and this constraint could be transferred from one language and algorithm to a different language and algorithm.

6.2. Declarative versus Procedural Transfer

ACT* is a general theory of human cognition embedded in a computer simulation model (Anderson, 1983). The theory proposes that there are generally two kinds of knowledge involved in human cognitive activities -- namely, declarative and procedural knowledge. The theory has been successful in explaining a large set of empirical data on memory processes, problem solving, and learning. Singley & Anderson (1989) have recently extended the theory to encompass knowledge transfer as well. Essentially, the ACT* explanation of transfer is an elaboration of the identical-elements theory; in fact, it instantiates the common elements shared by two skills in the form of production rules -- a general form of representing procedural knowledge. This explanation of transfer has been successfully applied to transfer in the domain of computer-based text-editing (Singley & Anderson, 1985; 1987; 1989).

However, since our subjects in this study were already relatively competent programmers in the three languages, it is unlikely that the locus of transfer can be in procedural form of programming skill. Thus, the explanation of transfer in terms of identical productions does not seem to be applicable here. Rather, the knowledge transferred seems declarative. Subjects are transferring their mental representation of one solution to the next solution. This hypothesis is consistent with the transfer being largely in the initial draft time and number of revisions but not in the execution time.

7. SUMMARY

The following conclusions concerning knowledge transfer across programming in different languages can be drawn from this study:

1. There was substantial transfer between LISP and PROLOG and between LISP and PASCAL. We hypothesize that this can be generalized to transfer between any two programming languages as there usually would be some commonalities shared by them; however, the amount of transfer might be related to the closeness of the two languages.

2. The positive transfer among programming languages was mainly localized in the amount of time for constructing an initial draft of program and the number of revisions required for finalizing that draft.

3. We propose that there are three levels of transfer between programming in two different languages -- namely, the syntactic, algorithmic, and problem levels. It is also proposed that most of the transfer we observed occurred at the algorithmic and problem levels.

Acknowledgement. This research was supported by grand MDA 90 38K K0190 from ARI. We are grateful to Drs. Herbert Simon and Kurt VanLehn, and to many graduate students, in the Department of Psychology at CMU, for their comments on various portions of this work.

REFERENCES

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

Anderson, J. R., Corbett, A., Finchman, J., Hoffman, D., & Pelletier, R. (in preparation). General Principles for an intelligent tutoring architecture.

Katz, I. R. (1988). *Transfer of Knowledge in Programming.* Ph.D. Dissertation, Department of Psychology, Carnegie Mellon University, Pittsburgh.

Keane, M. T. (1988). *Analogical Problem Solving.* New York: John Wiley & Sons.

Kessler, C. M. & Anderson, J. R. (1986). Learning flow of control: recursive and iterative procedures. *Human-Computer Interaction,* 2, 135-166.

Kotovsky, K. & Fallside, D. (1988). Representation and Transfer in Problem Solving. In Klahr, D., and Kotovsky, K. (Eds.), *Complex Information Processing: The Contributions of Herbert Simon.* Hillsdale, N. J.: Erlbaum.

Lewis, C., Carroll, J., & Bever, T. (1985). An exploratory study of problem formulation in PROLOG, LISP and PASCAL. IBM Technical Report.

Mandinach, E. B. and Linn, M. C. (1987). Cognitive consequences of programming: Achievements of experienced and talented programmers. *Journal of Educational Computing Research*, Vol. 3, pp. 53-57.

Scholtz, J. and Wiedenbeck, S. (1989). Learning second and subsequent programming languages: a problem of transfer. *Technical Report #80 of Department of Computer Science and Engineering.* University of Nebraska.

Simon, H. A., & Hayes, J. R. (1976). The understanding process: problem isomorphs. In *Model of Thought.* Vol. I. Cambridge, MA: Harvard University Press.

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies,* Vol. 22, pp. 403-423.

Singley, M. K., & Anderson, J. R. (1987). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction,* Vol. 3, pp. 223-274.

Singley, M.K., & Anderson, J. R. (1989). *The Transfer of Cognitive Skill.* Cambridge, MA:

Harvard University Press.

Smith, S. B. (1986). *An Analysis of Transfer between Tower of Hanoi Isomorphs.* Ph.D. Dissertation, Department of Psychology, Carnegie Mellon University.

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, pp. 247-261.

Figure 1. A comparsion of PROLOG, LISP, and PASCAL -- major commonalities and differences among them.

	PROLOG	LISP	PASCAL
Control-flow Difference	Matching and back-tracking.	Function calling.	Sequential execution + function/procedure calling.
Similarity	Recursion can be implemented by backtracking.	Recursion as the basic flow of control.	ITERATION as the major type of repetition control-flow, but RECURSION can also be implemented.
Constitutents: Difference	Predicates.	Functions.	Statements + Procedures/Functions.
Similarity	f(N1, N2, •••, Nn, Result).	(f N1 N2 ••• Nn).	
List processing: Similarity	Contructive, can be nested; Head-Tail recursion; [X|Y], [X, Y]. length(X).	Contructive, can be nested; car-cdr recursion; (cons X Y), (list X Y); (length X).	Sets, arrays, records, and files of records; they can be used to implement list structures, but the entities in one list must be of the same type.
Difference	Denoted as [a, b, c d]; fewer built-in predicates.	Denoted as (a b c d); more built-in functions.	
		e.g.: Allowed: x=(R, G, Blue); Allowed: x1=(R, G, (R, G)).	e.g.: Allowed: x:=[R, G, Blue]; No allowed: x1:=[R, G, [R, G]].
		list x: (nth x) list x: (cdr x) list x, y: (union x) list x, y: (intersect x)	array x: x[n] record x: x^ (next) set x, y: x + y set x , y: x * y
Arithmetic expression: Similarity	Procedural evaluation.	Procedural evaluation.	Procedural evaluation.
Difference	Infix representation.	Prefix representation.	Infix representation.

Figure 2. First example of programming problems for illustrating the similarities and differences of the three languages -- programs in PASCAL, LISP, and PROLOG for evaluating the Fibonacci function.

The Fibonacci function is defined as follows:

$$f(n) = \begin{cases} 1, & \text{if } n = 0; \\ 1, & \text{if } n = 1; \\ f(n-1) + f(n-2), & \text{if } n > 1. \end{cases}$$

Model PASCAL programs:

Recursive solution --

```
function f(n: integer): integer;
begin
  if n <= 1 then f :=1
  else f :=f(n-1)+f(n-2)
end;
```

Iterative solution --

```
function f(n: integer): integer;
begin
  if n <= 1  then  f :=1
  else
    begin
      f1:=1;  f2:=1;
      for i :=2 to n do
        begin
          f := f1 + f2;
          f1:= f2;
          f2 := f
        end;
    end;
end;
```

Model LISP programs:

Recursive solution --

```
(defun f(n)
  (if (<= n 1) 1
      (+ (f (- n 1)) (f (- n 2)))
  )
)
```

Iterative solution --

```
(defun f(n)
  (if (<= n 1) 1
      (do ((f1 1 (+ f1 f2))
           (f2 1 f1)
           (i 1 (+ i 1))
          ((= i n) f1)
      )
  )
)
```

Model PROLOG programs:

Recursive solution --

```
f(N, 1) :- N<=1;
f(N, F) :- N1 is N-1,
           N2 is N-2,
           f(N1, F1),
           f(N2, F2),
           F is F1 + F2.
```

Iterative solution --

```
f(N, 1) :- N <=1;
f(N, R) :- f(N, 1, 1, 1, F).
f(N, I, F1, F2, F) :-
           I < N,
           T is F1 + F2,
           I1 is I + 1,
           f(N, I1, T, F1, F).
f(N, N, F1, F2, F1).
```

Figure 3. Second example of programming problems for illustrating the similarities and differences of the three languages -- programs in PASCAL, LISP and PROLOG for counting the number of books by a certain author in a database of books.

Suppose there is the following database:

Anderson, The Architecture of Cognitive, 1983.
Anderson, Cognitive Psychology, 1985.
MeClelland, Parallel Distributed Processing, 1986.
Simon, The Sciences of the Artificial, 1969.
Simon, Human Problem Solving, 1972.
Simon, Protocal Analysis, 1987.

Write programs in PASCAL, LISP and PROLOG to count the number of books by a given author.

Model PASCAL programs:
Iterative solution --

```
type
  string=array[1..10] of char;
  book_type=record
              name: string;
              title: string;
              year: integer
            end;
var
  database: file of book_type;

procedure
  count(author: string);
var
  a_book: book_type;
  n: integer;
begin
  n := 0;
  reset(database);
  while not eof(database) do
    begin
      read(database, a_book);
      if a_book.name=author
      then n:=n+1
    end;
  writeln('No. of books:', n)
end;
```

Model LISP programs:
Recursive solution --
(Using association list)

```
(setq database
   '((anderson arc 1983)
     (anderson cog 1985)
     (mccleland pdp 1986)
     (simon art 1969)
     (simon hps 1972)
     (simon pro 1987)))

(defun count-books (author)
  (count-help
       author database))

(defun count-help (auth db)
  (cond
    ((null db) 0)
    (t (+ (if (eq auth
                (caar db)) 1 0)
          (count-help auth
                  (cdr db))))))
```

NOTE: in some COMMONLISP versions, this function can be written more compactedly:

```
(defun count -books (author
  (count author database
    :test #'eq :key #'car))
```

Model PROLOG programs:
Declarative solution:

```
book(anderson, arc, 1983).
book(anderson, cog, 1983).
book(mccleland, pdp, 1986).
book(simon, art, 1969).
book(simon, hps, 1972).
book(simon, pro, 1987).

books_by(Author, L):-
      setof(X,
      book(Author, X, _), L).
no_of_books(Author, N):-
      books_by(Author, L),
      length(L, N).
```

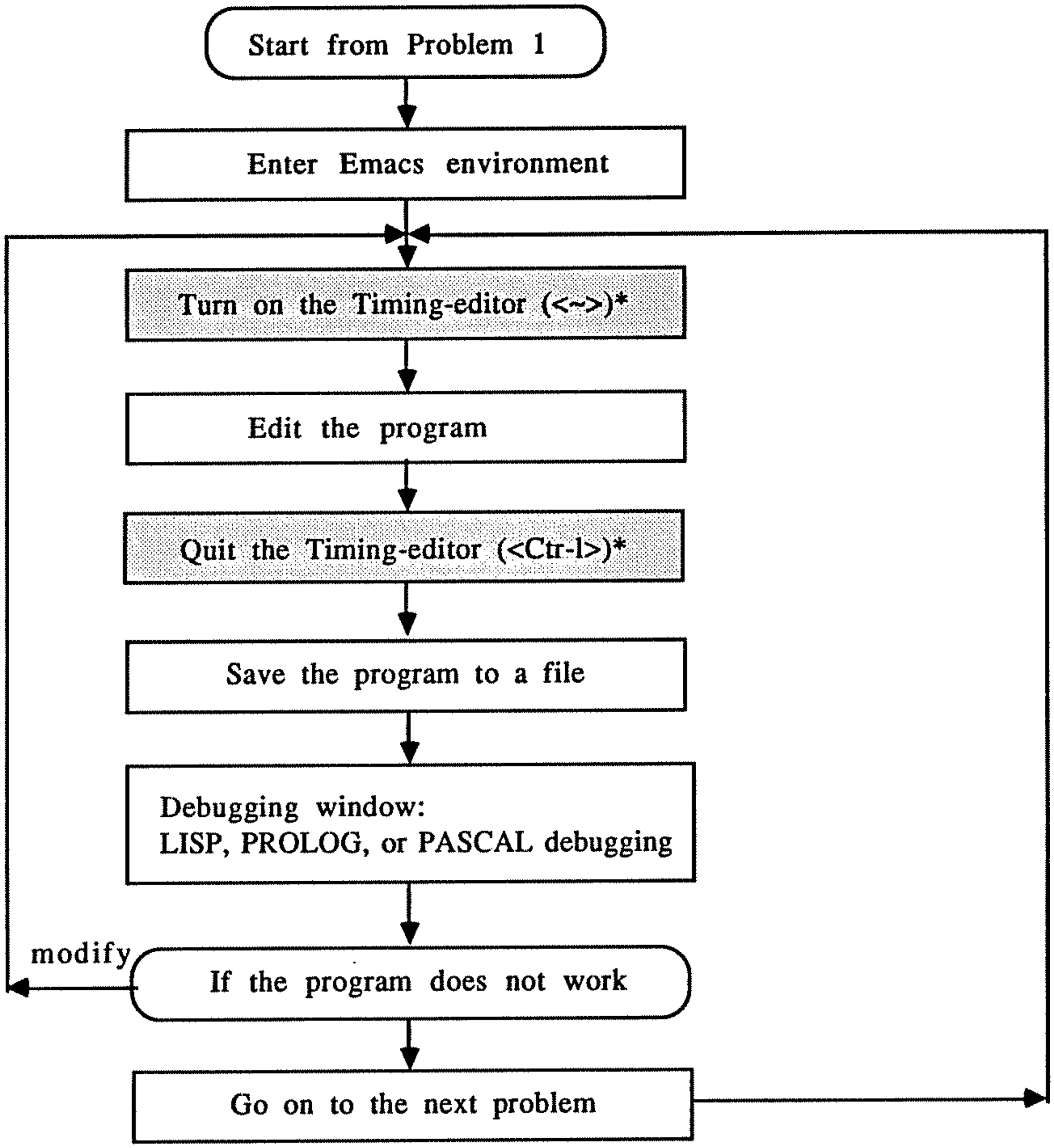
Figure 4. The design of the experiments -- transfer between LISP and PROLOG as for Experiment 1 and 2 and transfer between LISP and PASCAL as for Experiment 3.

	Prob-1	Prob-2	Prob-3	Prob-4	Prob-5*	Prob-6*
Group-1	LISP-->PROLOG**	P-->L	L-->P	P-->L	L-->P	P-->L
Group-2	PROLOG-->LISP**	L-->P	P-->L	L-->P	P-->L	L-->P

NOTE *: This figure presents the design which was common to the three experiments. However, 4 problems were used in Experiment 1; 6 in Experiment 2; and 5 in Experiment 3.

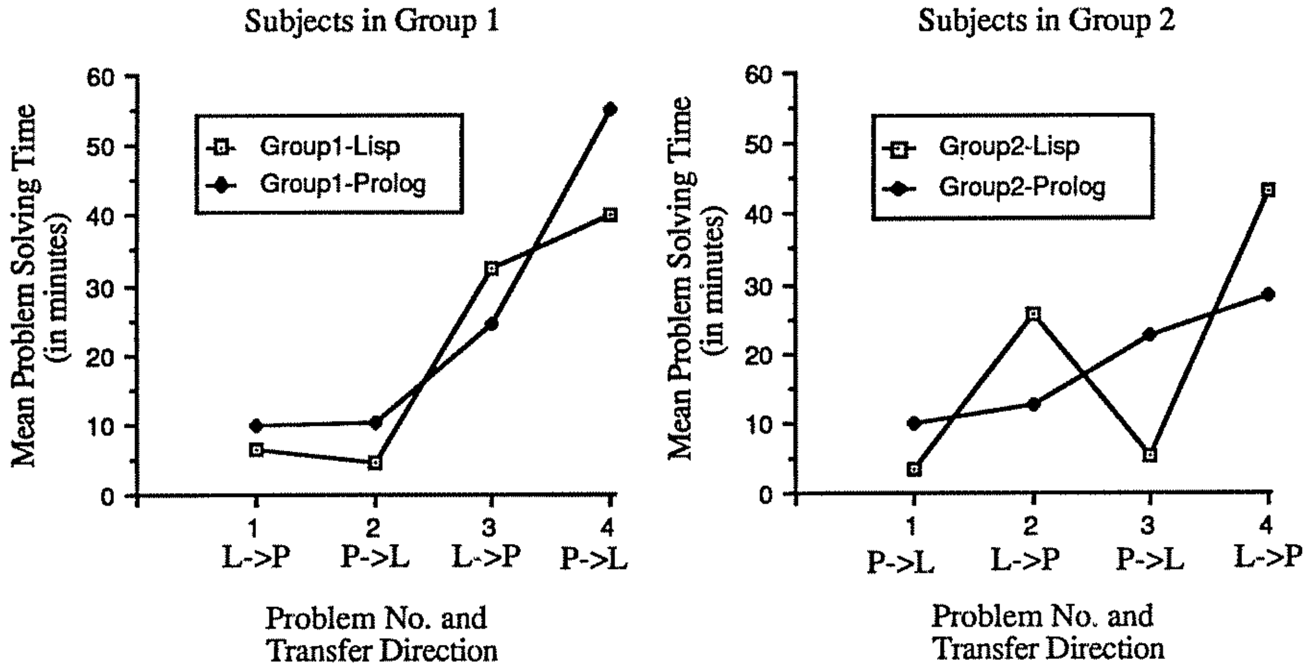
**: Instead of PROLOG, it was PASCAL in Experiment 3.

Figure 5. The experimental procedure used in the experiments -- transfer between LISP and PROLOG as for Experiment 1 and 2 and transfer between LISP and PASCAL as for Experiment 3.



NOTE *: This figure presents the procedure which was common to all the three experiments. However, the timing-editor (as shown in the two shadowed boxes) was only used in Experiment 2 and 3; in Experiment 1, standard EMACS editor was used.

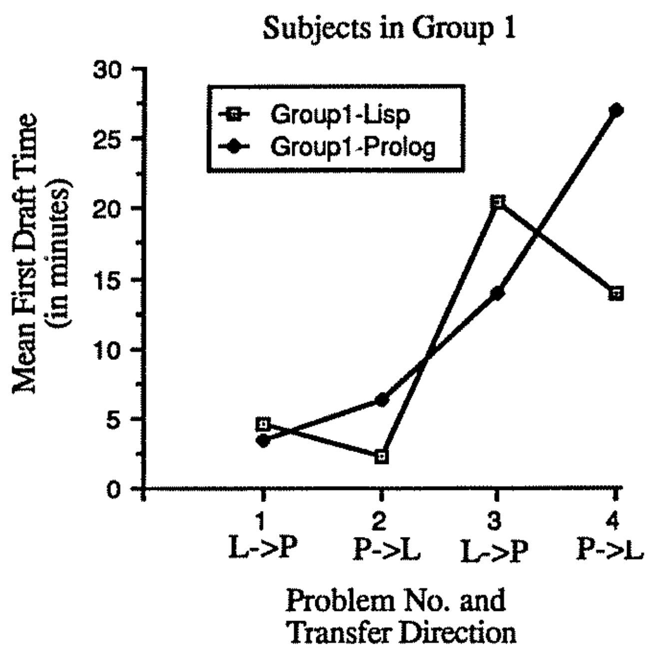
Figure 6. The mean problem solving time by two groups of subjects in Experimen 1, averaged across subjects within groups and across two groups and four problems.

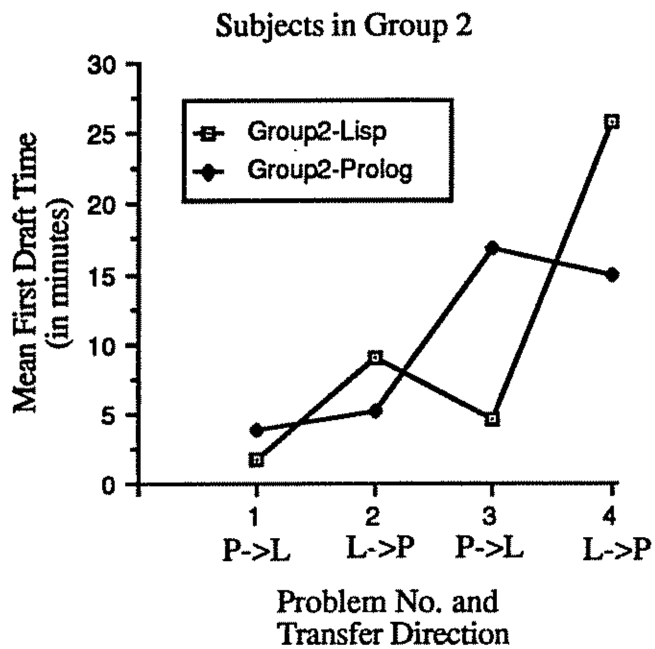


Mean Total Problem Solving Time (in minutes)
Across Two Groups and Four problems

	First Program	Second Program
Lisp programming	26.7	13.4
Prolog programming	24.4	18.8

Figure 7. The mean first-draft time by two groups of subjects in Experiment 1, averaged across subjects within group and across two groups and four problems.

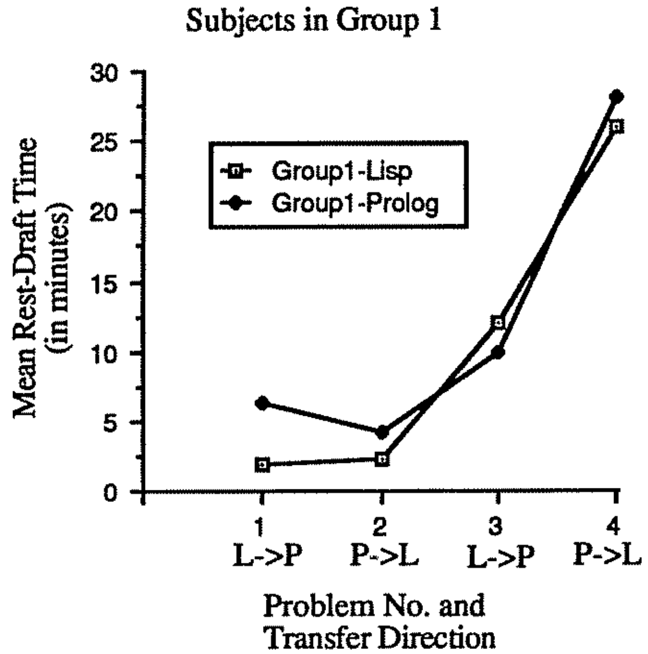


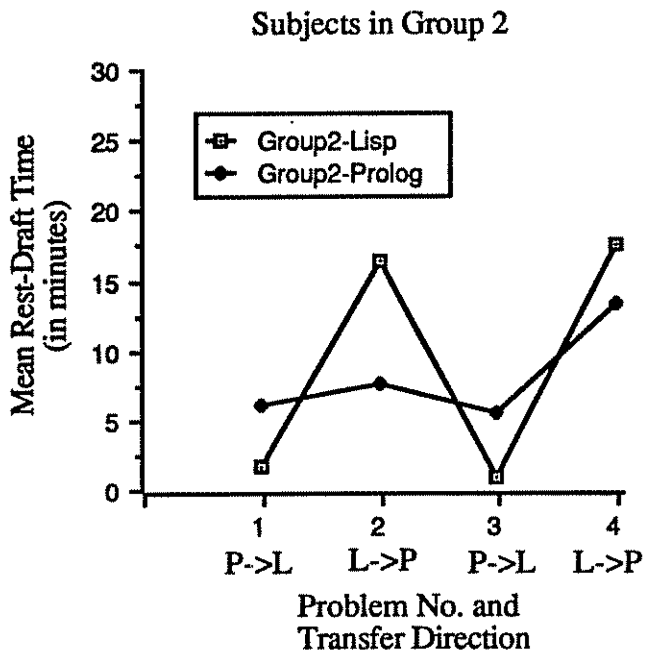


Mean First Draft Time (in minutes) Averaged across Two Groups and Four Problems

	First Program	Second Program
Lisp programming	14.4	5.7
Prolog programming	13.5	9.4

Figure 8. The mean rest-darft time, as the difference between the total time and the first-draft time, by two groups of subjects in Experiment 1, averaged across subjects within group and across two groups and four problems.



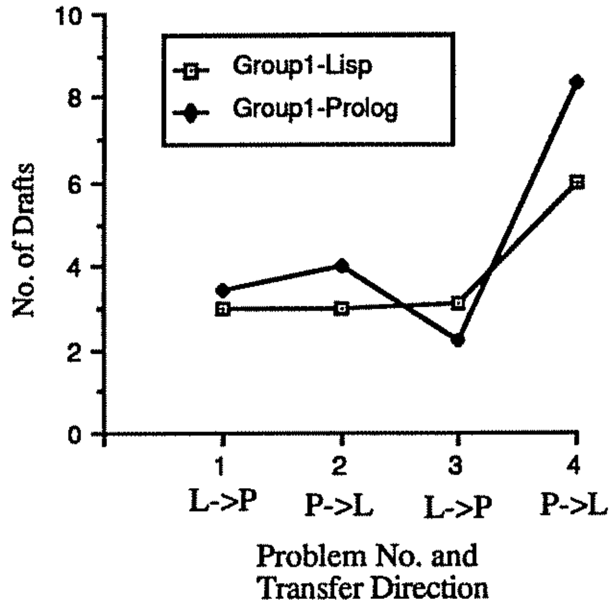


Mean First Draft Time (in minutes) Averaged across Two Groups and Four Problems

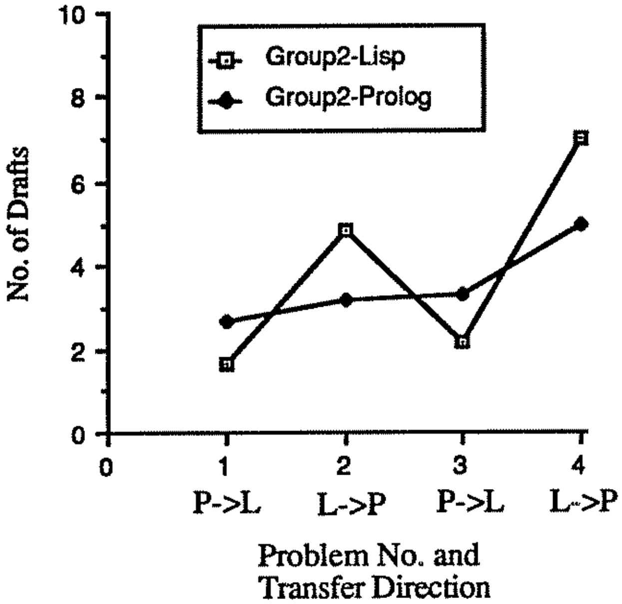
	First Program	Second Program
Lisp programming	12.3	7.7
Prolog programming	13.5	9.4

Figure 9. Results of Experiment 1 on transfer between programming in LISP and in PROLOG -- No. of drafts, roughly corresponding to semantic mistakes made in programmin

Subjects in Group 1.



Subjects in Group 2.



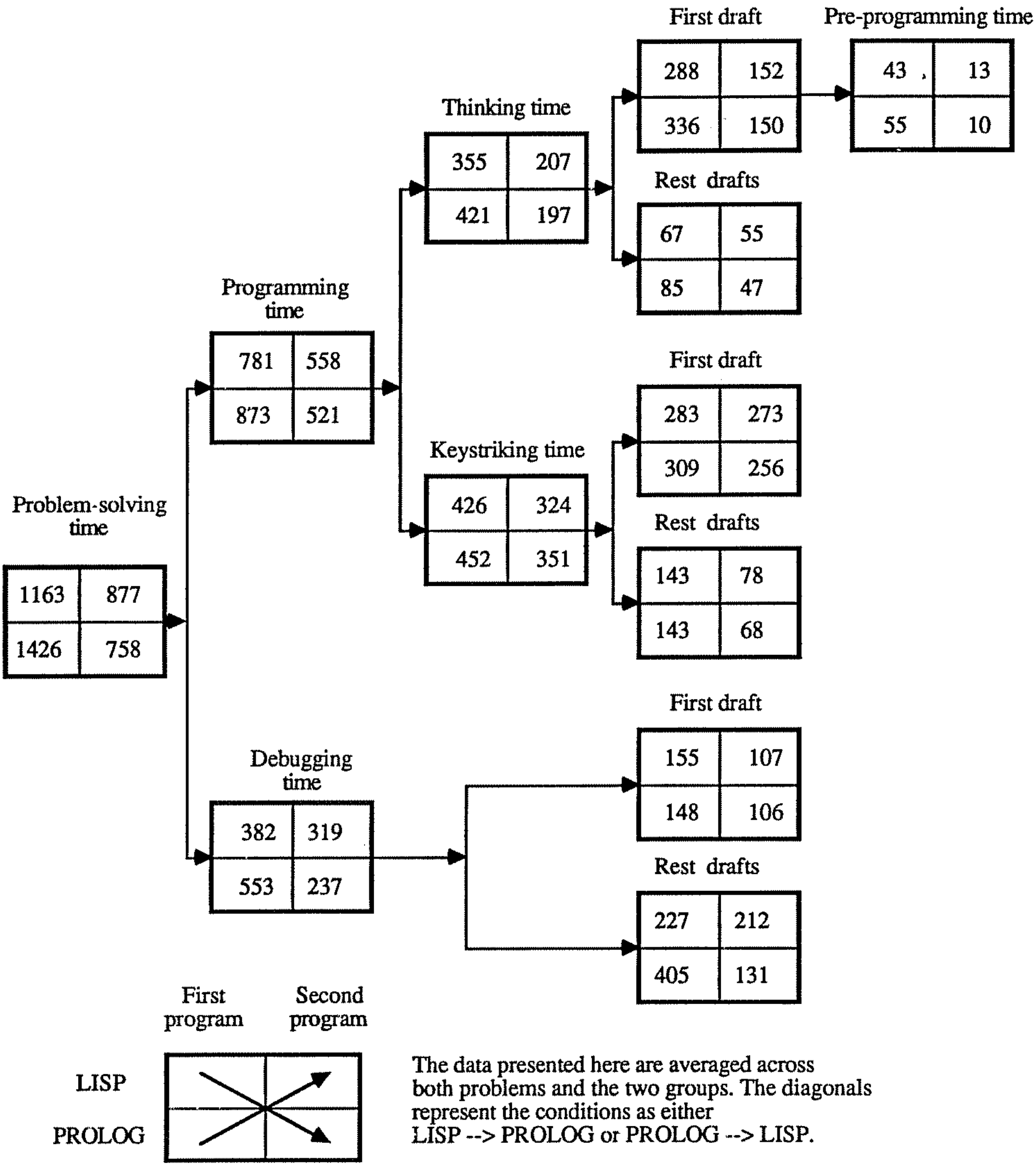
Mean Number of Drafts Averaged Across Two Groups and Four problems

	First Program	Second Program
Lisp programming	4.49	3.20
Prolog programming	4.58	3.44

Mean Per-Draft Time (in minutes) for Rest Drafts Averaged Across Two Groups and Four problems

	First Program	Second Program
Lisp programming	3.52	3.48
Prolog programming	3.77	3.85

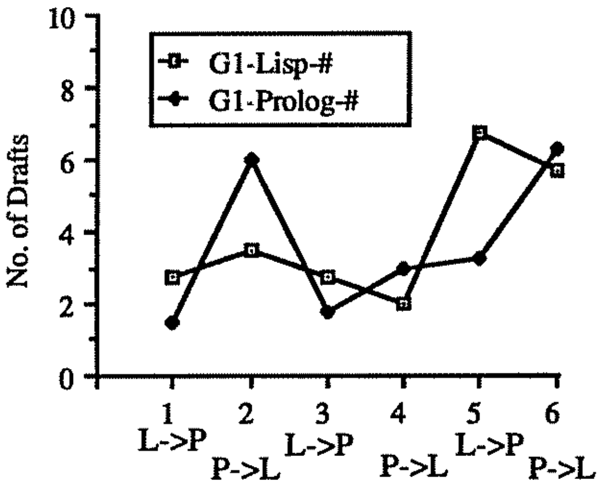
Figure 10. Results of Experiment 2 -- time measures (in seconds) as various levels of localization in transfer between programming in LISP and in PROLOG.



The data presented here are averaged across both problems and the two groups. The diagonals represent the conditions as either LISP --> PROLOG or PROLOG --> LISP.

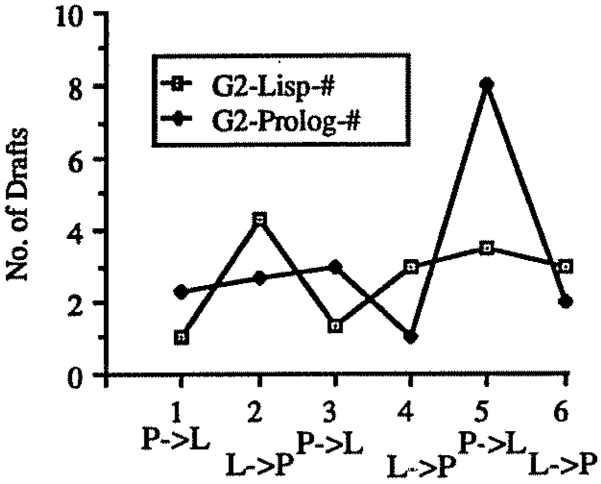
Figure 11. Results of Experiment 2 on transfer between programming in LISP and in PROLOG -- No. of drafts, roughly corresponding to semantic mistakes made in programming

Subjects in Group 1.



Problem No. and Transfer Direction

Subjects in Group 2.

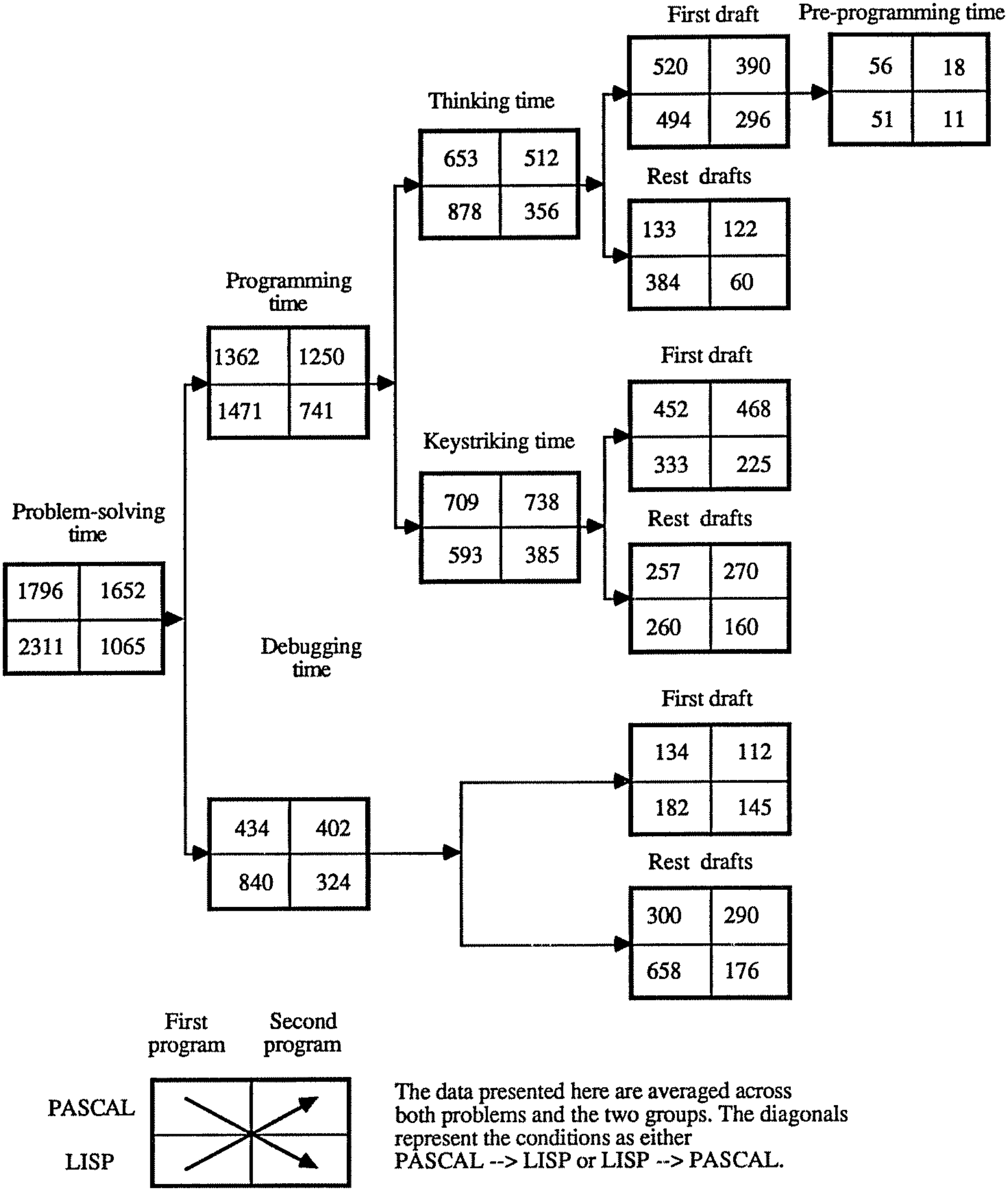


Problem No. and Transfer Direction

Mean Per-Draft Time (in seconds) for Rest Drafts
Averaged Across Two Groups and Four problems

	Thinking Time		Keystroking Time	
	First Program	Second Program	First Program	Second Program
LISP	19	22	41	31
PROLOG	18	26	30	38

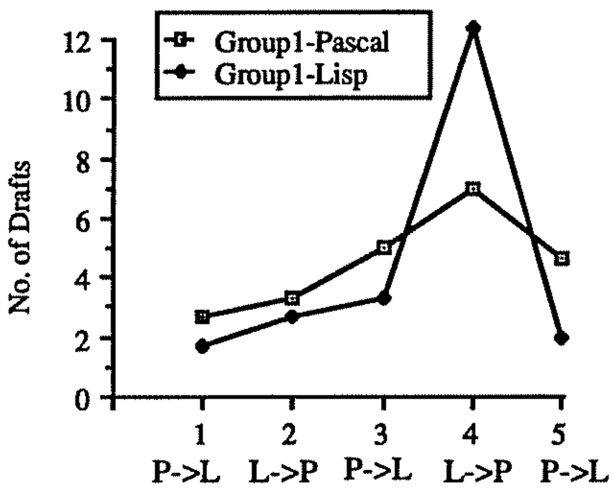
Figure 12. Results of Experiment 3 -- time measures (in seconds) as various levels of localization in transfer between programming in PASCAL and in LISP.



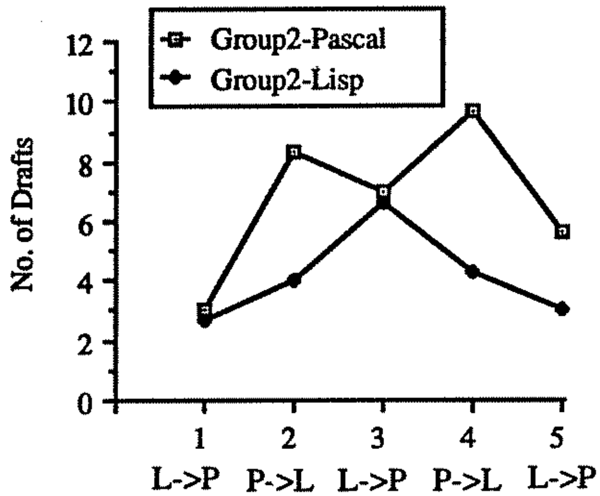
The data presented here are averaged across both problems and the two groups. The diagonals represent the conditions as either PASCAL --> LISP or LISP --> PASCAL.

Figure 13. Results of Experiment 3 on transfer between programming in PASCAL and in LISP -- No. of drafts, roughly corresponding to semantic mistakes made in programming.

Subjects in Group 1.



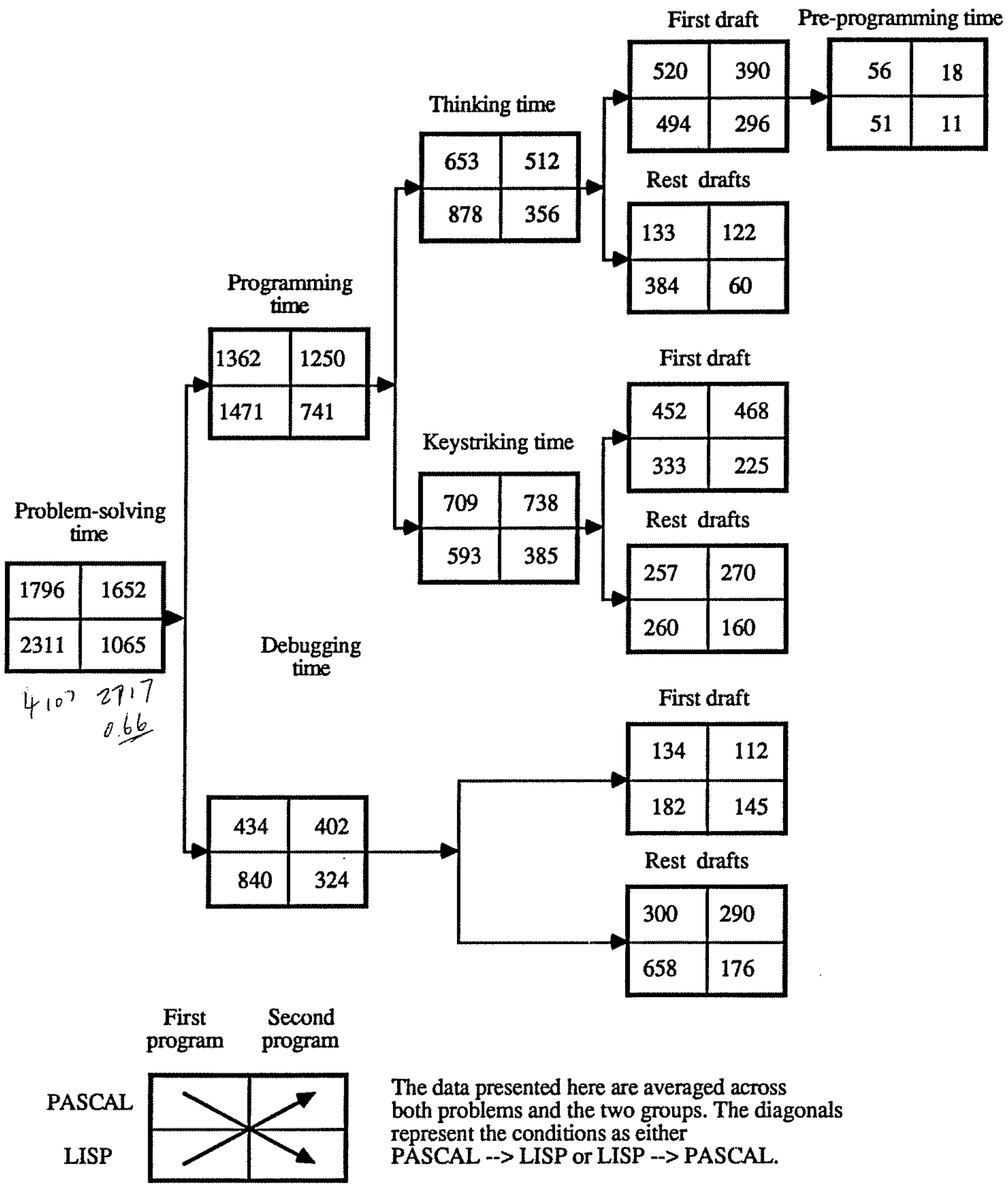
Subjects in Group 2.



Mean Per-Draft Time (in seconds) for Rest Drafts
Averaged Across Two Groups and Four problems

	Thinking Time		Keystroking Time	
	First Program	Second Program	First Program	Second Program
PASCAL	22.0	25.0	43.0	56.0
LISP	73.0	20.0	50.0	53.0

Figure 12. Results of Experiment 3 -- time measures (in seconds) as various levels of localization in transfer between programming in PASCAL and in LISP.



The data presented here are averaged across both problems and the two groups. The diagonals represent the conditions as either PASCAL --> LISP or LISP --> PASCAL.