```
AN YOU SUPPLY ?  YES  NO  COND  FUTUREDATE
:ILL: 713981        :Borrower: PMC       :ReqDate: 20041124 :NeedBefore: 20041225
 :Status: IN PROCESS 20041129            :RecDate:          :RenewalReq:
 :OCLC: 8018488     :Source: OCLCILL     :DueDate:          :NewDueDate:
 :Lender: LYU,GDC,KZS,*LAS,UWC
:CALLNO:      Q334.I568g 1983                                    copy chapter
:AUTHOR:  International Joint Conference on Artificial Intelligence.
:TITLE:   IJCAI : proceedings of the conference / (1983)
:IMPRINT: [S.l.] : IJCAI, c1977-
:ARTICLE: Anderson, J. R.: Learning to Program
:VOL:          :NO:          :DATE: 1983                :PAGES: 57-62
:VERIFIED: <TN:96809>OCLC ISSN: 1045-0823 [Format: Serial]
:PATRON:  Ferris, Jennifer
:SHIP TO:  ILL/Hunt Library/Frew Street/Carnegie Mellon
niversity/Pittsburgh/PA/15213-3890
:BILL TO:  Same***FEIN 25-0969-449***
:SHIP VIA: IDS/Lib Rate            :MAXCOST: $35.00IFM
 :COPYRT COMPLIANCE: CCL
:FAX: (412)268-6944
:E-MAIL: hl82@andrew.cmu.edu
:BILLING NOTES: Please include IL# on invoice.
:BORROWING NOTES: ARIEL: hl3bb701.library.cmu.edu or 128.2.20.246
:AFFILIATION: PALCI recip. copy pilot
:LENDING CHARGES:                  :SHIPPED:            :SHIP INSURANCE:
:LENDING RESTRICTIONS:
:LENDING NOTES:
:RETURN TO:
:RETURN VIA:
```

Learning to Program

John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Three aspects of learning to program are described--the organization and compilation of problem-solving operators, the impact of knowledge representation, and the impact of working memory limitations. The GRAPES system simulates the organization and compilation of these operators. The simulation of one problem solving episode is discussed. Also discussed are the impact of different data notations and the impact of working memory load on successful application of LISP functions.

Introduction

This paper is concerned with the cognitive factors that are involved in acquiring a skill like programming in LISP. Among the relevant factors are the following:

1. Understanding sufficient instruction about the system to enable effective problem-solving.
2. Organization and compilation of this knowledge into a procedural form.
3. Tuning of the operators that underlie the procedural form.
4. Representing the problem optimally for the operators possessed.
5. Expanding working memory capacity to keep relevant knowledge available.

The first is the focus of most of the educational effort -- whether in textbooks, user's manuals, or the classroom. The other four are largely ignored. They are things that are learned implicitly in the act of solving problems. In looking at novices learning how to program in LISP, they spend less than 25% of their time actually trying to absorb the relevant knowledge and more than 75% of their time learning how to use that knowledge. We have yet to see a novice understand any instruction of modest complexity such that he could do the task without error. There are always serious holes in the understanding which are only dealt with in the act of solving problems.

This paper will describe what we have learned about topics 2,4, and 5 above. I want to stress, however, that our findings are not unique to LISP or indeed to programming. Similar things are being found in areas such as geometry and physics (e.g. Anderson, 1981; Larkin, 1981).

Skill Organization and Compilation

We have developed a production system, called GRAPES, that simulates the problem-solving involved in programming and the learning that occurs in the course of this problem-solving. GRAPES uses a set of problem-solving operators that decompose goals into subgoals. A programming problem is solved in GRAPES by decomposing an initial goal to program a function into subgoals and these into subgoals, etc., until goals are reached which correspond to LISP code. Given the right operators GRAPES can solve moderately difficult LISP problems, write, test, and debug the code (see Anderson, Farrell & Sauers, 1982). This is a critical sufficiency test because humans eventually are capable of this. However, more interesting is its simulation of the behavior of novices in their first hours of learning LISP. Here I will consider one of the many simulations we have performed and what this simulation says about skill organization and knowledge compilation.

Simulation of ONETWO

The case comes from a subject, SS, who had written just three functions--to take first, second, and third elements of a list. Then she was given the ONETWO problem. The ONETWO problem required the subject to write a function which would take a list as an argument and return a new list consisting of the first two elements of the argument list. The LISP functions that the subject knew at this time included CONS, but the subject had not yet learned about LIST. She knew about CAR and CDR and with these she had defined functions that would return the first, second, and third arguments of a list. These were the only functions that she had written up to this point in time.

Initial Attempt at ONETWO. She flailed at writing the function ONETWO and the experimenter suggested writing a simpler function, ADDTWO, which would take two arguments and make a list out of them. This problem she was able to make some headway on. It is interesting to speculate why ADDTWO was more tractable than ONETWO. As we will see, the basic problem and its solution did not change in going from ONETWO to ADDTWO. However, by reducing the complexity of the task by one level, the burden on working memory was reduced so that the subject was better able to match operators. A later section discusses in more detail the impact of working memory limitations.

Figures 1-6 illustrate the simulation's attempts to solve ONETWO. Given the perfect correspondence between the

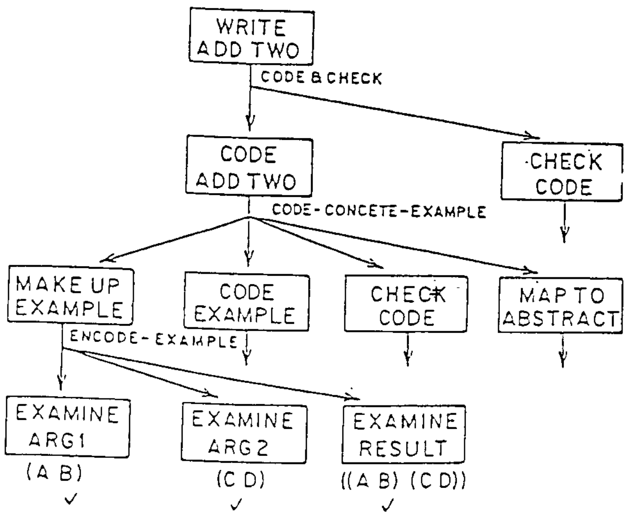

Figure 1

simulation and SS's protocols, we infer that these figures also describe the goal structures that were guiding her problem solutions. Figure 1 illustrates the first work that was done on the ADDTWO subproblem. The first operator decomposes this into the subgoals of coding the function and checking the code. The first operator set subgoals to come up with concrete examples of the input to ADDTWO and what its output should be, to find some code that could be typed at the top level that would convert the concrete input into the concrete output, to check this code, and then to map this code into an abstract function. The inputs she chose to pass to ADDTWO were (A B) and (C D). Why she chose list arguments we are unsure. The result she wanted for these inputs was ((A B) (C D)).

Figure 2 illustrates the processes by which she decided how to create this example at the top level. After deciding on the example, she went through an episode where she explicitly reviewed the definition of all the functions she knew, searching for an appropriate one. She selected CONS. We represented the definition of CONS to GRAPES as

> The first argument of CONS is any S-expression and the second argument is a list. Its result is a list. The first element of the result is the first argument. The rest of the result consists of the second argument.

She and GRAPES chose CONS on the basis of the fact that a list was wanted and CONS makes lists. Having selected CONS the subgoals were now to determine what arguments to pass to CONS in order to get the intended result.

The critical piece of information in selecting the first argument is the definition statement *The first element of the result is the first argument.* GRAPES interfaces this with the desired result, ((A B) (C D)), to determine that the correct argument should be (A B).

Next, SS and GRAPES turn to the second argument. The appropriate part of this definition is *The rest of the result consists of the second argument.* Matching this would retrieve ((C D)) as the second argument. However, our subject retrieved (C D). We assume (see detailed discussions in Anderson, Farrell and Sauers) that the semantic feature of *consists* were partially lost and this statement became *The rest of the result contains the second argument.* We manipulated GRAPES' working memory so

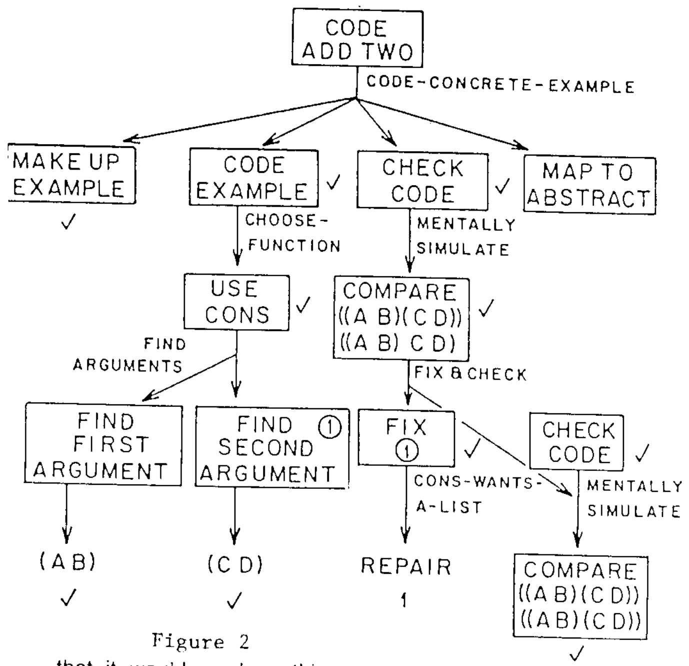

Figure 2

that it would produce this error -- that is, we broke in and changed the contents of working memory.

The subject and GRAPES mentally simulated what the outcome would be of the code (CONS '(A B) '(C D)). This involved retrieving the definition of CONS again. As evidence that her definition of CONS was not in error, she correctly determined that ((A B) C D) would result as a answer. This corresponded to an error she had encountered frequently and we assume she had acquired an operator to repair this which embedded the second argument to CONS in a extra list. In this way, she and GRAPES recover from their error and make up the concrete example (CONS '(A B) '((C D))).

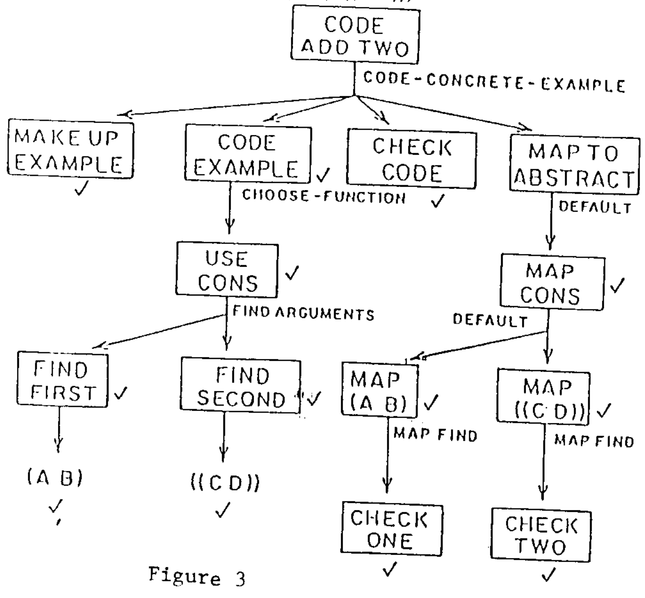

Figure 3

The Mapping. Figure 3 illustrates the simulation of SS's initial attempt to map from the concrete code to an abstract LISP function. First she maps CONS in the concrete code into CONS in the LISP function. At this point the structure of the function is

```
(def addtwo (lambda (one two)
   (cons <?> <?>)))
```

The remaining task is to map the two concrete arguments into abstract arguments. She first focuses on mapping (A B). The following rule applies:

```
        IF the goal is to map a concrete expression of LISP
           and the expression is a data structure involving
              a term
           and the term corresponds to an argument of
              the function
     THEN the abstract expression can be obtained from
              the data structure
          by replacing the term with the argument
```

So, in this case she is trying to map the *concrete expression* (A B) where the *argument* ONE corresponds to the *term* (A B). Therefore, after substituting the argument for the term, the abstract expression becomes simply ONE. This same rule applies to map the second *concrete expression* ((C D)). In this case the *argument* TWO corresponds to the *term* (C D) and the abstract expression after substitution is (TWO). Note this rule has correctly mapped the first expression but incorrectly mapped the second expression. The function definition at this point is

```
(def addtwo
   (lambda (one two)
      (cons one (two))))
```

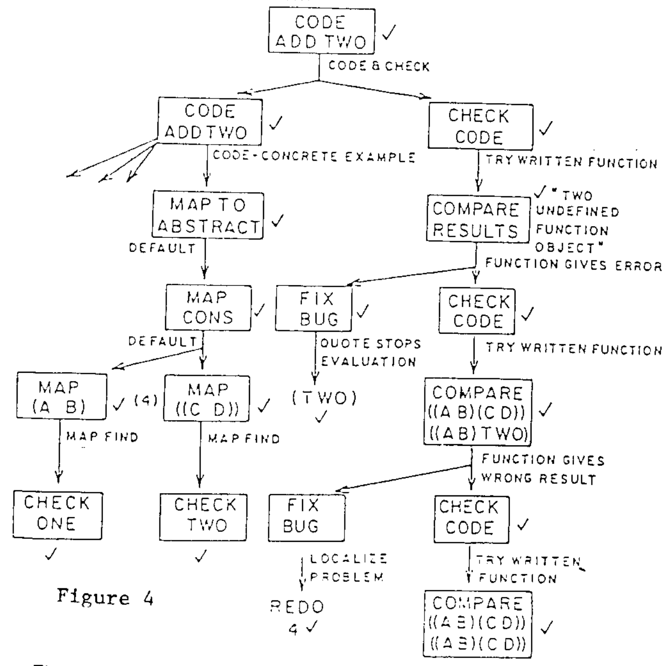

Figure 4

Figure 4 illustrates some of the subsequent evolution of this definition. The coding of ADDTWO had the brother goal of checking that code. Both SS and GRAPES called the LISP interpreter to try the code with the arguments (A B) and (C D). Both received the same error message "TWO undefined function object." This also corresponds to an error that SS had encountered a few times previously in her problem solving. In previous occasions, the cause had been failure to quote an argument. Therefore, we assumed that she had acquired an operator that used quote to stop evaluation. When this operator applied, her LISP code became

```
(def addtwo
   (lambda (one two)
      (cons one '(two))))
```

Again, the code was tried. This time it returned the result ((A B) TWO). Comparing this with her desired result the problem was localized to the second argument given to CONS and GRAPES went back to retrying the goal of mapping ((C D)).

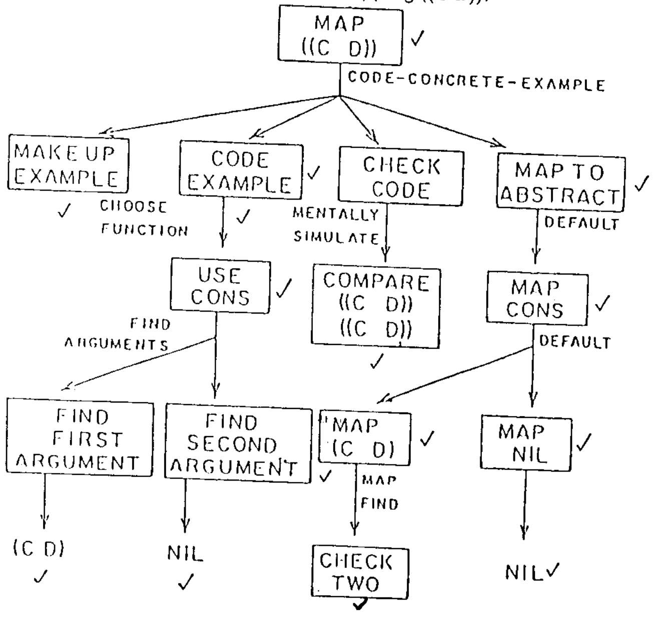

Figure 5

Figure 5 illustrates the simulation of this mapping. Having returned to this goal, the previous MAP-FIND operator will not apply again. Therefore, a default rule applies which creates a new subgoal of coding a list consisting of a single argument. As in the case of coding the full ADDTWO problem, GRAPES falls back on the plan of making up a concrete example, coding it, checking the code, and then mapping the code into an abstract code for the function. The previous concrete example of ((C D)) is used. Again, CONS is chosen because it makes lists and again its definition is used to determine the correct arguments. This time the definition is correctly used and GRAPES plans the concrete code as (CONS '(C D) NIL).

After mentally simulating this, GRAPES turns to the goal of mapping the concrete code (CONS '(C D) NIL) into a LISP function. The process of performing this mapping is quite analogous to the original mapping in Figure 3. Again, CONS is mapped into CONS. The same MAP-FIND operator as before maps (C D) into TWO. An operator for special LISP symbols, like NIL, maps NIL onto itself. So, the final successful code becomes

```
(def addtwo
   (lambda (one two)
      (cons one (cons two nil))))
```

Return to ONETWO. Figure 6 illustrates the behavior of the simulation and the subject when they returned to the original

ONETWO problem. The code they generated is given below:

```
(def onetwo
    (lambda (list)
        (cons (first list)
              (cons (second list) nil))))
```

Whereas the subject had taken an hour to code ADDTWO, she only took ten minutes to solve ONETWO and most of that time was spent confirming what the functions FIRST and SECOND did. ONETWO is solved by the same method that ADDTWO is solved, but without any rehearsal of the ONETWO method. Our assumption is that operators were compiled from this problem that summarized the planning steps that went into the problem solution.

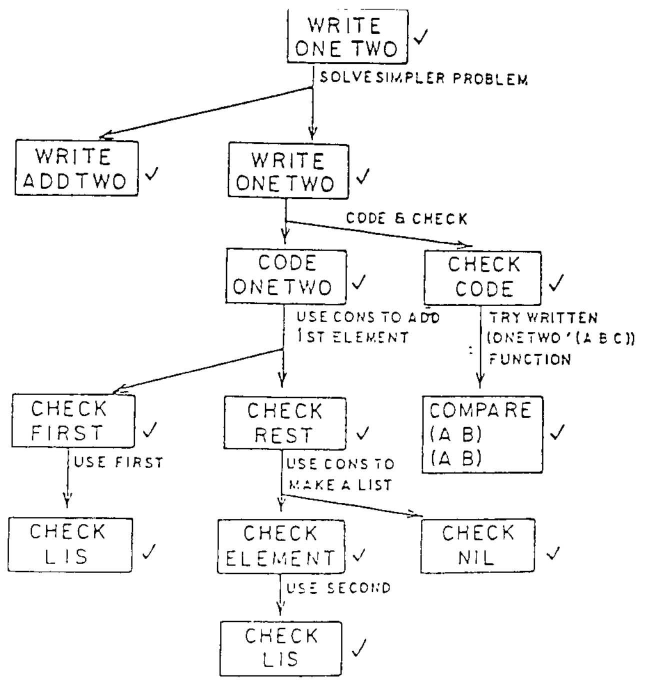

Figure 6

One of the operators that GRAPES compiled summarizes the problem solution illustrated in Figure 5 that started with the goal of creating a list of a single element and resulted in the action of CONSing that element with NIL. The compilation procedure recognizes that the various aspects of the concrete example and its code are intermediate results and are not essential to the final answer. It traces through these steps to determine if there are any connections from the top goal to the final CONSing action. The summary operator built is:

```
        IF the goal is to code a list consisting of
              one argument
    THEN CONS that argument with NIL
            and set as subgoals to code that argument
```

Similarly, an operator is compiled to correspond to the outer CONS in the ADDTWO function. It has the form:

```
        IF the goal is to code into a list consisting
              of argument1 and argument2
    THEN CONS argument1 into a list consisting of
              argument2
            and set as subgoals to code argument1
            and to code a list consisting of argument2
```

Conclusions from Simulation. This example illustrates a number of conclusions that we have come to in simulating the problem-solving of our subjects. The first conclusion is that their behavior is hierarchically organized according to goal trees like the ones illustrated in Figures 1-6. The second conclusion is that subjects prefer to solve problems by analogy to concrete cases. In this episode the subject created an example at the top level which she then tried to map over into the code for the function.

The third conclusion is the importance of knowledge compilation in extracting from an example problem operators that will streamline the solution of later problems. There are numerous technical issues in deciding how much to collapse into a single operation and we have hardly solved them all. However, for a discussion of what progress we have made consult Anderson(1982) and Anderson, Farrell, and Sauers(1982). The hierarchical organization imposed by goal structures such as the one in Figure 5 is important to the compilation process. It serves to indicate what aspects of the original problem-solving episode should go together in the compiled operator.

The fourth conclusion concerns the impact of working memory failure on performance. We saw that CONS was incorrectly applied because the subject momentarily misrepresented its definition. I will return later in this paper to the issue of how working memory failure impacts on incorrect application of a function.

Representation

How one represents a problem has a strong impact on one's problem solving. We have been trying to document the impact of data structure representations on subject's problem solving. Typically, LISP is now taught with respect to parenthesized notation. Subjects conceive of the effects of basic LISP operators like CAR, CDR, CONS, and LIST in terms of changes made on marks on a page. We think this representation has strong implications for the relative difficulty of the various functions and the types of errors.

```
(CAR '((A) (B))) = (A)
VERY EASY

(CDR '((A) (B))) = ((B))
MORE DIFFICULT
TYPICAL ERROR:   =  (B)

(LIST '(A) '(B)) = ((A) (B))
FAIRLY EASY

(CONS '(A) '(B)) = ((A) B)
MOST DIFFICULT
TYPICAL ERROR:   = ((A) (B))
```

Figure 7

Figure 7 illustrates the four functions. CAR takes the first element of a list and subjects have little difficulty with it beyond overcoming the non-mnemonic character of the function name. CDR returns the remander of a list and subjects have more difficulty with it. A frequent error is illustrated in Figure 7. When the tail of the list contains a single list, subjects will return that list with one set of parentheses missing. The function LIST subjects find both mnemonic and easy to understand. They have the greatest difficulty with CONS. CONS inserts its first argument into the list that is its second argument. A frequent error is leaving an extra set of parentheses around the second argument.

Exactly why these errors are made is a little uncertain. Robin Jeffries has developed a GRAPES model that will produce these errors. It and many similar explanations turn on the fact that CDR and CONS are explained to subjects as moving parentheses. For instance, this instruction is quite explicit in Siklossy(1976). Subjects think that CDR operates by deleting the first parenthesis and the first element and inserting the parenthesis in front of the rest of the list. If the last insert operation is omitted it is possible to have an error. Subjects think of CONS as deleting the left parenthesis of the second argument, inserting the first argument in front of that, and then inserting left parentheses. If they omitted the deletion operation they could produce the error in Figure 7. Subjects also think of CAR and LIST as moving parentheses, but omission of any step in CAR and LIST results in a nonsensical result as does omission of any of the other steps in CDR and CONS. Thus, the problem with CDR and CONS is that there are critical steps which, if omitted, will lead to non-detectable errors. Consistent with this view is that the same subject will randomly make and not make the errors. This is what we would expect if it were produced by occasionally forgetting a step.

```
(CAR ((A.NIL).((B.NIL).NIL))) = (A.NIL)
VERY EASY

(CDR ((A.NIL).((B.NIL).NIL))) = ((B.NIL).NIL)
VERY EASY

(LIST (A.NIL) (B.NIL))= ((A.NIL).((B.NIL).NIL))
DIFFICULT
TYPICAL ERROR:          = ((A.NIL).(B.NIL))

(CONS (A.NIL) (B.NIL)) = ((A.NIL).(B.NIL))
FAIRLY EASY
```

Figure 8

One might think that the problems with CDR and CONS are unavoidable. However, this is certainly not the case. Originally, LISP was taught as operating on a box structure, which is much closer to the actual computer-implementation of LISP. Figure 8 illustrates how the functions apply to the dotted-pair equivalent of the box notation (Weissman, 1967). We performed an experiment in which we explained the operation of these LISP functions with respect to the dotted-pair equivalent of the box notation and contrasted this with the list notation. As we have informally observed, CDR is harder than CAR with the list notation and this is largely due to forgetting parentheses in examples like Figure 7. This difference went away with the dotted-pair notation. Again we confirmed our informal observation that CONS was more difficult than LIST and this was in part due to failing to put parenthesis around the second argument. We found that the difficulty reversed with the dotted-pair notation and the errors with LIST derived from failing to embed its second argument in a separate dotted-pair structure. This clearly indicates that the difference is not a matter of the greater mnemonic value of the word LIST rather than CONS, as some have suggested.

Now, I do not want to be read as advocating that LISP instruction go back to the box notation. There are good justifications for using the parenthesized list notation as basic and only later introducing the box representation. People find lists easier to reason about and the appropriate data structure usually turns out to be a list. However, there are applied consequences of understanding how representation impacts on difficulty of functions. The basic problem with CDR and CONS in the representation is that they destroy and reconstruct the structure of their arguments. If the destroy and reconstruct are not carried out properly, non-detectable wrong answers can result. There are ways to characterize the operations of CDR and CONS to avoid this difficulty. Rather than deleting the left parenthesis, the first element, and reinserting the left parenthesis, CDR could be characterized as erasing the first element. Rather than characterizing CONS as deleting the left parenthesis of the second argument, adding the first argument, and adding the left parenthesis, we could characterize CONS as squeezing its first argument into the front of the second.

Problems of Memory Load

One of the surprises to me in studying LISP novices is how much of their programming time is spent recovering from errors of memory. We intend to quantify this for each of our protocol subjects, but informally I would guess it averages around 50%. This includes time spent trying to reconstruct what forgotten subgoals are. Worse is when the subject misremembers a subgoal and solves a different one. Also subjects will often retrieve the wrong function to perform a task. Subjects do not have the experience of so much wasted time because in retrospect they tend to forget their failures of memory and only remember their correct steps. So you might get remarks like, "I don't know how I could have spent two hours on solving this. I must have been asleep."

Experts do not have the same frequency of memory errors on problems. In part this is because their short term memories are better, perhaps due to something like chunking. But they are also more reliable at recalling things that occurred half an hour earlier in the problem solving protocol (Jeffries, Turner, Atwood & Polson, 1981). This cannot be simply due to chunking. Their long term memories are more reliable for the domain and become reliable extensions of short-term memory. This better working memory for the domain of expertise seems a fairly general phenomenon. Chase and Ericsson (1981) have studied some subjects who reliably increased their digit span up to 80 digits from the usual 7 $\pm$ 2. They also were able to show that this depended on reliable storage and retrieval from LTM. The interesting observation was that subjects with 80 digit spans were no different in other ways. In fact, their letter span was 7 $\pm$2. I think the same thing happens in programming -- we develop better working memories for partial information computed in the process of programming and this better working memory is specific to programming.

One of the classic memory retrieval errors in novice LISP programming is using LIST instead of CONS. Anderson, Farrell and Sauers have shown how this can be explained in terms of what the subject's representation of what CONS and LIST do. The interesting observation is that these errors appear to be more frequent in the context of writing a complex function. Thus, it seems when the working memory load increases, errors in function recall increase.

We did an experiment to see if embedding an extra level of processing would result in increased errors. Figure 9 illustrates the material used in the experiment. There were three types of tasks. One involved recognizing the missing function that would transform the input into the output. The second involved evaluating the function as applied to a series of arguments. The third involved recognizing the argument which, if provided to the function, would produce the output.

```
(a) FUNCTION RECOGNITION
60% (?  '(A) '((B) (C) (D)))=
                 ((A) (B) (C) (D))
48% (?  '(A) (REVERSE '((B) (C) (D)))) =
                 ((A) (D) (C) (B))

(b) Evaluation
52% (CONS '(A) '(B C D)) = ?

50% (CONS '(A) (REVERSE '(B C D))) = ?

(c) ARGUMENT PROVISION
67% (CONS 'X ?) = (X (Y Z))

55% (CONS 'X (REVERSE ?)) = (X (Y Z))
```

Figure 9

The two types of functions used were LIST and CONS, although only CONS examples are illustrated in Figure 9. Thus, for function recognition the subjects choice was implicitly between LIST and CONS, although occasionally an APPEND was given. In evaluation and argument provision, the typical errors were incorrect number of parentheses with error much more frequent for CONS. Subjects got 79% of the LIST problems correct, but only 41% of the CONS problems. This is in line with earlier remarks about the relative difficulty of the two functions.

The principal manipulation of interest was whether the function had a REVERSE composed in. REVERSE simply changes the order of elements in a list; it does not change the appropriateness of a CONS or a LIST in the function generation problems. In evaluation and argument provision, it does not change the correct parentheses -- and we only scored these problems for parentheses, not whether the subject had correctly done the reversing. However, as can be seen from the percentages correct in Figure 9, errors did increase when a REVERSE was involved. Thus, adding an extra piece of information to working memory increased the error rate. This is an example of working memory spill over -- extra memory load in one aspect of the problem impacts on performance in a logically independent aspect. The effect on function recognition is particularly relevant here. We see that whether we credit a student with knowing the difference between LIST and CONS may depend on the working memory demands of the test context.

Many situations contain working memory demands that pose needless difficulty for novices. Novices become overburdened in extracting needless detail and fail to extract the critical information for learning. One of the most frustrating instances of this in LISP is parenthesis counting. Novices will be on the verge of an insight, go off to check whether the parentheses are balanced for some expression, and lose any chance of getting the insight after they are finished with the balancing. The implication is that tutorial techniques that reduce working memory load should facilitate learning.

References

Anderson, J. R. Tuning of search of the problem space for geometry proofs. *Proceedings of IJCAI-81*.

Anderson, J. R. Acquisition of cognitive skill. *Psychological Review*, 1982, *89*, 369-406.

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

Anderson, J. R., Farrell, R., and Sauers, R. Learning to plan in LISP. ONR Technical Report ONR-82-2, 1982.

Anderson, J. R. and Kline, P. J. A learning system and its psychological implications. *Proceedings of IJCAI-79*.

Chase, W. G. and Ericsson, K. A. Skilled memory. In J. R. Anderson (Ed.) *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum, 1981.

Jeffries, R., Turner, A. A., Polson, P. G. and Atwood, M. E. The processes involved in designing software. In J. R. Anderson (Ed) *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum, 1981.

Larkin, J. Enriching formal knowledge: A model for learning to solve textbook physics problems. In J. R. Anderson (Ed) *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum, 1981.

Siklossy, L. *Let's Talk LISP*. Englewood Cliffs,N.J.: Prentice-Hall, 1976.

Soloway, E. M. From problems to programs via plans: The context and structure of knowledge for introductory LISP programming. Coins Technical Report 80-19. University of Massachusetts Amherst, 1980.

Weissman, C. *LISP 1.5 Primer*. Belmont, Ca.: Dickenson, 1967.

This research was supported by the Personnel and Training Research Programs, Psychological Services Division, Office of Naval Research, under Contract No.: N00014-81-C-0335, Contract Authority Number, NR 157-465 to John Anderson.