Problems, Plans, and Programs

Elliot M. Soloway
Beverly Woolf

COINS Technical Report 79-18

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. DAHC19-77-G-0012.

Abstract

Being able to make abstractions is a critical skill in today's fast changing world. We report herein on material developed which emphasizes this skill in the context of teaching introductory LISP problem solving and programming. This enterprise is based on a taxonomy of problems we developed which groups ostensibly different problems into classes based on underlying similarities. Moreover, similarities among programs in a class lead to the development of plans, which are program templates with slots, and annotated with explanatory comments. Exceptions and further generalizations to the plans are considered. Finally, we speculate on the utility of this approach with respect to non-applicative programming languages (e.g., FORTRAN, APL, PASCAL).

1. Introduction

Since the specific content of courses is changing so quickly, the challenge to education is teaching students how to learn [Simon 78]. In this regard, an important skill is the ability to develop abstractions, i.e., to construct classification schemes which highlight similarities and differences. In this paper we shall outline the content of a course which attempts to teach this skill in the context of teaching introductory LISP programming and problem solving. The key to this enterprise has been the development of:

1. a taxonomy of problems, i.e., a classification scheme which groups problems into classes based on specific criteria, and
2. a set of plans, each of which captures the essential features of a class of problems, and corresponding solution programs.

In this paper, we begin by developing a scheme for classifying the problems often found in introductory LISP courses. We then examine the LISP programs which solve problems in the various classes and abstract higher level structures called 'plans.' Here we view a plan as a program template plus comments describing the goals and reasons for the various expressions in the template. Next, we build on the set of plans to include new problems. Finally, we speculate briefly on the utility of our taxonomy with respect to programming in languages such as FORTRAN, APL, or PASCAL.

2. PREDICATE Type Problems and Programs

In order to teach a "simple" type of recursion {1}, one often uses the following two problems:

> Write a program which returns TRUE if and only if all the elements in the input list are atoms.

The LISP code for this problem could be:

```
( IS-LIST-OF-ATOMS  (LST)
   (COND
     ( (NULL LST)  T)
     ( (NOT (ATOM (CAR LST)))  NIL )
     ( T  (IS-LIST-OF-ATOMS (CDR LST)) ) ) )
```

Next,

> Write a program which returns TRUE if and only if the first argument (which is an atom) is a member of the second argument (which is a list).

The code for this problem could be:

```
( MEMBER  (ARG LST)
   (COND
     ( (NULL LST)  NIL )
     ( (EQ (CAR LST) ARG)  T )
     ( T  (MEMBER ARG (CDR LST))) ) )
```

If we step back from the particulars of the problems and the programs, we notice first that both <u>problems</u> belong to the class of PREDICATES, i.e., problems which require True or False for an answer. Moreover, the first problem is an AND-PREDICATE type, since it requires

{1} By "simple" we mean that the answer to the problem is available "at the bottom" of the recursion; students often skip over the fact that the answer must still be "passed back up." We feel that this misunderstanding is a fair price to pay for easing students into recursion; when the next type of problems is introduced, the notion of recursion becomes more refined. In Section 8, we explicitly contrast these two "types" of recursion.

that ALL the elements of the input list have the desired property, while the second problem is an OR-PREDICATE type, since it requires only that one of the elements have the desired property. Next, if care is taken in writing the programs, similarity in function can be reflected in the programs. Below we list a template which captures the commonalities and differences in the above two programs. {1}

```
( predicate  (LST ARG1 ARG2 ... )
   (COND
    ( (NULL LST)  <T, NIL> )
    ( (test (CAR LST) ARG1 ... )  <NIL, T> )
    ( T  (predicate (CDR LST) ARG1 ARG2 ...)) ) )
```

Note that the lowercase names represent variables or slots that are filled in for particular problems, e.g., in MEMBER 'test' would be replaced by 'EQ'. The angle brackets and the comma, e.g., <T, NIL> in clause 1e, represent choices, one of which must be chosen.

A naked template is not enough; in order to be able to truly understand the relationships abstracted in the template, one needs to add explanatory comments to the clauses of the template. Thus, we define a plan to be a template plus explanatory comments. The intuition is that a plan represents actions to be performed and reasons and goals for those actions. For example, a key goal in LISP programming is the reduction of a problem into identical "smaller" problems; recursion is at the heart of this process. Comments reflecting this point could be:

{1} The notation we will use to refer to clauses in the CONDitional expression is:

```
(COND (1p, 1e)
      (2p, 2e)
      (3p, 3e) )
```

Template Clause	Explanatory Comment
2p	Do it to the first; test the first element.
3e	Do it to the rest; continue processing.
1p	Is the list empty? Are we done processing?

The goals of the different types of PREDICATES must also be reflected in this template and in the comments. For example, since IS-LIST-OF-ATOMS is an AND-PREDICATE it must visit each member of the list before it can respond True, hence only when the empty list is detected in 1p is IS-LIST-OF-ATOMS's work really done. On the other hand, if MEMBER satisfies clause 1p, then it has not found the element with the desired property, and since it is an OR-PREDICATE it must return FALSE, or in LISP, NIL. Comments explaining this aspect of the template could be:

Template Clause	Explanatory Comment
1e	Return a truth value: AND-PREDICATE - if this point is reached then all the elements must have passed the test in 2p, thus return True. OR-PREDICATE - if this point is reached then a single element which satisfies the test in 2p has not been found, thus return False.
2e	Return a truth value: AND-PREDICATE - Aha, found an element which does not meet the condition in 2p; thus AND could never be true, so stop processing and return False. OR-PREDICATE - Aha, found an element that satisfies the desired condition in 2p; quite processing early and return True.

Throughout the rest of this paper we shall be engaged in the enterprise exemplified above, namely, looking for commonalities among problems, among programs, and abstracting plans based on these observations. We have used the material in this paper in teaching introductory LISP three times, and have found the abstractions to be powerful aids in teaching students problem solving and LISP programming. In this process, we have also stressed the more important meta-lesson that finding abstractions is a good idea. Thus, we have no objections when students argue about our particular classification scheme, or encoding of the programs or templates. In fact, we encourage them to do so, since a discussion about alternative abstractions is precisely the activity we try to foster.

Before proceeding, we need to make two further comments about the use of plans and templates. First, we do not view a template as a type of equation into which numbers or functions can be blindly plugged. We are all aware of the misuse of equations in fields such as physics and engineering [cf. Clement, Lochhead, and Soloway, 1979]. We stress that a template without explanatory comments explaining the why is not all that useful; in fact, blind substitution can result in major errors.

A second issue is the level of abstraction of the plans described here. One could imagine plans which do not have as strong a procedural component as ours do, i.e., one could describe MEMBER without a commitment to either recursion or iteration. The development of more abstract plans is not inconsistent with our major teaching goals, namely, teaching students about constructing abstractions. The major danger we see, however, is that without an explicit procedural component, plans will tend to look more like algebraic equations. Since

a discussion of the merits of this issue are beyond the scope of this paper (cf. Papert [1971]), we only point out that recent empirical studies have indicated that programming can enhance problem solving ability, when compared with using algebraic equations as a solution language [Clement, et al., 1979]. With this caveat in mind, we encourage the enterprise of finding even more general abstractions. (We return to this point in section 9.)

2.1 Attacking a Problem: Using Plans to Ask Questions

Another way to look at the knowledge surrounding a plan is in terms of questions which need to be answered in order to develop a correct program solution. For example, if the student can determine that the problem is in the PREDICATE class, then he/she can ask:

1. Is the problem an OR-PREDICATE or an AND-PREDICATE?
2. Which of the arguments will be examined, i.e., what is the CDRing variable?
3. What property or test must be applied to the head (typical) element?

Answers to these questions determine which components of the PREDICATE template are selected for the desired final program and determine how the slots are to be filled. For example, an answer to question 3 will determine how the slot in clause 2p will be instantiated. {1} Of course, in order to answer the questions, the student needs to have a

{1} In order to test the completeness of these questions, we have written a computer program which first queries a user with the above questions (plus two other bookkeeping questions) and generates a program based on the questions. Success with this program suggests that we have isolated the key components of certain types of problems.

grasp of the reasons why these questions are in fact critical. In effect, the questions serve as a systematic strategy for attacking problems.

3. BUILDER Type Problems and Programs

In order to develop a more complete and sophisticated understanding of recursion, problems and programs of the following sort might be presented.

> Write a program which deletes an atom equal to ARG from the input list.

The code for this program, REMOVE-MEMBER, might be:

```
( REMOVE-MEMBER  (LST ARG)
   (COND
    ( (NULL LST)   () )
    ( (EQ (CAR LST) ARG)  (CDR LST) )
    ( T  (CONS (CAR LST) (REMOVE-MEMBER (CDR LST) ARG))) ) )
```

We call problems and programs of this sort BUILDERS; the defining characteristic of the BUILDER problem class is that a list is returned which contains all the elements which did not satisfy the test criteria, plus the result of some action (remove, insert, etc.) on the elements which did. Another example of a BUILDER type problem would be:

> Write a program which deletes all the atoms equal to ARG from the input list.

And the code for this problem might be:

```
( REMOVE-ALL-MEMBERS  (LST ARG)
   (COND
    ( (NULL LST) () )
    ( (EQ (CAR LST) ARG)  (REMOVE-ALL-MEMBERS (CDR LST) ARG) )
    ( T  (CONS (CAR LST) (REMOVE-ALL-MEMBERS (CDR LST) ARG))) ) )
```

A number of abstractions can now be made. First, we see that BUILDERS have OR-like and AND-like functions just as PREDICATES do; REMOVE-MEMBER is an OR-BUILDER since it is satisfied when the first occurrence of the desired element is found, while REMOVE-ALL-MEMBERS is an AND-BUILDER, since it must visit all the elements of the list. Second, if we compare the code in the above two programs, we see that they are the same except for clause 2e. Also, we can abstract from the particular test in clause 2p to generate the following first pass at a template for BUILDERS.

```
( builder  (LST ARG1 ARG2 ... )
   (COND
    ( (NULL LST)   () )
    ( (test (CAR LST) ARG1 ... )  < (CDR LST),
                                    (builder (CDR LST) ARG1 ... ) > )
    ( T  (CONS (CAR LST) (builder (CDR LST) ARG1 ARG2 ... ))) ) )
```

For an OR-BUILDER, such as REMOVE-MEMBER, the first alternative in clause 2e is selected, since the list need not be further traversed. However, for an AND-BUILDER, such as REMOVE-ALL-MEMBERS, the second alternative is required in order to continue down the list.

Now, clause 2e in the above BUILDER template is not quite general enough to handle other BUILDER problems. For example,

> Write a program SUBSTITUTE which replaces the first occurrence of the atom OLD with the atom NEW in the list LST.

The LISP code for this problem is:

```
( SUBSTITUTE   (OLD NEW LST)
   (COND
    ( (NULL LST)   () )
    ( (EQ (CAR LST) OLD)  (CONS NEW (CDR LST)) )
    ( T  (CONS (CAR LST) (SUBSTITUTE OLD NEW (CDR LST)))) ) )
```

The key issue is that now some action must be performed on the desired

element. Taking into consideration the AND-BUILDERS a revised BUILDER template would be {1}:

```
( builder  (LST ARG1 ARG2 ...)
   (COND
    ( (NULL LST)  () )
    ( (test (CAR LST) ARG1)  < (action LST),
                               (CONS (action (CAR lst)
                                     (builder (CDR LST) ARG1 ..))>)
    ( T  (CONS (CAR LST) (builder (CDR LST) ARG1 ARG2 ...))) ) )
```

Thus, in the REMOVE-MEMBER case, the 'action' slot would be filled by 'CDR', while in the SUBSTITUTE case it would be filled in by '(CONS NEW (CDR LST))'.

Comparisons between PREDICATES and BUILDERS can also be made. For example, we see that in clause 3e the BUILDER template includes the list building function CONStruct, whereas the PREDICATES do not. Also, while NIL and () represent the same object in LISP, they can have different interpretations in different contexts. Thus, though NIL is returned in clause 1e in both templates, the NIL in each case <u>means</u> something different. The plan comments must explain that the NIL in the PREDICATE case stands for False, while the NIL in the BUILDER case stands for the empty list, onto which elements of the list will be CONSed. Finally, exactly the same set of questions which were used in the PREDICATE case can be used in the BUILDER case. Again, comments must point out that the <u>interpretation</u> of OR and AND in either case is different, which results in different code.

If we move one more step back, we can see that underlying BUILDER problems and programs is the notion of <u>copying</u>. In particular, consider

{1} Counterexamples can readily be found to this template, e.g., UNION. We will address this issue shortly.

the program COPY which returns a copy of the input list by actually tearing it apart and putting it back together:

```
( COPY  (LST)
   (COND
    ( (NULL LST)  () )

    (T (CONS (CAR LST) (COPY (CDR LST)))) ) )
```

The blank line in the COND expression is there on purpose. It indicates that BUILDERS have a basic shell, and that all that changes is the particular test for the desired elements, and the action required on them.

4. SELECTOR Type Problems and Programs

The third class of problems and programs which we shall consider here can be called SELECTORS; e.g., assuming we have a built-in predicate, LGT, which is Lexicographically Greater Than, the following problem would be a SELECTOR:

> Write a program which returns the first atom in the input list which is Lexicograhically Greater Than the given atom.

The code for this function might be:

```
( SELECT-LGT  (LST ARG)
   (COND
    ( (NULL LST)   () )
    ( (LGT (CAR LST) ARG)   (CAR LST) )
    ( T  (SELECT-LGT (CDR LST) ARG)) ) )
```

By this time, one can predict that in this class also there will be OR-SELECTORS and AND-SELECTORS. The above problem is an OR-SELECTOR; replacing "first" with "all" will make it an AND-SELECTOR.

The template for this type of problem is: {1}

```
( selector  (LST ARG1 ARG2 ... )
   (COND
    ( (NULL LST)   () )
    ( (test (CAR LST) ARG1 ...)  <(CAR LST),
                                  (CONS (CAR LST)
                                        (selector (CDR LST) ARG1 ...) >)
    ( T  (selector (CDR LST) ARG1 ARG2 ...)) ) )
```

As evidenced by a comparison of the BUILDER plan and the SELECTOR plan, the key difference between these two types of problems is that SELECTORS do NOT return elements which do not meet the test requirements, while

{1} Consistent with clause 2e in the BUILDER template, note that clause 2e in the SELECTOR template will probably need to be generalized as follows:
<(action (CAR LST)), (CONS (action (CAR LST)) (selector (CDR LST)...))>

BUILDERS do, i.e., compare clauses 3e and 2e in each template. Clearly, comparisons to the PREDICATE plan can also be made.

5. A Taxonomy of Problems

The taxonomy described in the preceeding sections can be neatly depicted in a tree structure (see Figure 1); there are three classes of problems, each class having a similar structure. The basis for this grouping can be traced to the work of McCarthy [1965], in developing LISP, to Friedman [1965], who has written an excellent text for LISP, and to Hardy [1975], who uses the template idea in a program synthesis system.

We do not claim uniqueness for this scheme. As we note in the concluding remarks, we feel that other problems and other languages might require other classification characteristics. Nonetheless, the key point is that looking for abstractions is a powerful idea, and the tree in Figure 1 is presented to the students as one concrete example of this enterprise to follow. Moreover, finding exceptions (see next section) and developing new structures to accomodate the inconsistencies is a powerful learning technique (cf. [Piaget, 1969]).

6. Generalizing the Plans to Handle an Exception

The problem

Write a program which takes the union of two lists.

is an AND-SELECTOR; each element of the first list is visited to see if they are members of the second list, and if so, such elements are discarded. However, the code

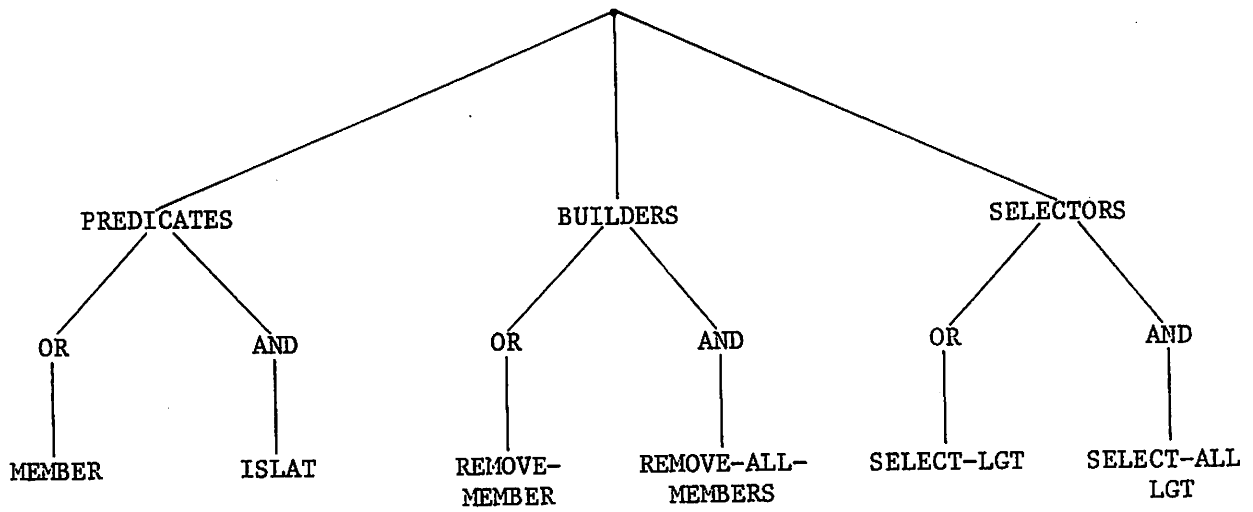

Figure 1.

A taxonomy of problems.

```
( UNION  (LST1 LST2)
   (COND
    ( (NULL LST1)  LST2 )
    ( (NOT (MEMBER (CAR LST1) LST2))
              (CONS (CAR LST1) (UNION (CDR LST1) LST2) )
    ( T  (UNION (CDR LST1) LST2)) ) )
```

does not completely follow the template for AND-SELECTORS. That is, in clause 1e UNION returns the second list, while the template suggests that NIL, the empty list, be returned.

Clearly, what needs to be done is to generalize the SELECTOR plan. However, what is most interesting about this apparent inconsistency is not the specific change to the plan, but rather, the process of recognizing a conflict between an example and an abstraction. Becoming aware that abstractions often admit of exceptions is precisely the kind of problem solving skill that one wants to encourage.

7. Generalizing the Plans to Handle Structured Lists

In the preceding discussion we have assumed that the input lists would be simple lists of atoms, e.g., (A B C). However, if we want to input structured lists, e.g., (A B (C D) E B), some additional machinery must be developed. Simply put, a new line of code needs to be added to "catch" the possible list element before EQ {1} is encountered, and the program must call itself again on this portion of the list, as well as on the rest of the list. {2} Thus, if we wanted to write REMOVE-MEMBER to handle this type of list, the code for this new program, REMOVE-MEMBER-* {3}, might be:

{1} We assume here that EQ is undefined if its arguments are not atoms.
{2} This observation was pointed out by John Lowrance in a course on LISP.
{3} The naming convention of * for procedures which require double recursion is taken from Friedman[1974].

```
( REMOVE-MEMBER-*  (LST ARG)
   (COND
    ( (NULL LST)  NIL )
    ( (LISTP (CAR LST))  (CONS (REMOVE-MEMBER-*
                                  (CAR LST) ARG)
                               (REMOVE-MEMBER-*
                                  (CDR LST) ARG)) )
    ( (EQ (CAR LST) ARG)  (REMOVE-MEMBER-* (CDR LST) ARG) )
    ( T  (CONS (CAR LST) (REMOVE-MEMBER-* (CDR LST) ARG))) ) )
```

Two observations can now be made. First, students notice that the above OR-BUILDER does not work exactly like the old REMOVE-MEMBER, i.e., REMOVE-MEMBER-* will delete the first occurrence of the desired atom from <u>each</u> of the sublists, while REMOVE-MEMBER will delete only the first occurrence of the desired atom from the simple, non-embedded list. While this generalization is consistent with what an OR-BUILDER does, it is nonetheless extending is scope. Based on this observation, students quickly examine the AND-BUILDER case to see if it too will be extended.

Second, in trying to see how to make AND-PREDICATES and OR-PREDICATES work on structured lists, students encounter a snag: BUILDERS have a CONS in clause 2e in order to bind together the results, but PREDICATES can't use a CONS. To deal with this problem they <u>invent</u> the analogue to the list builder CONS, namely, the logical builders AND and OR. {1} Thus, the code for the OR-PREDICATE, MEMBER-*, would be

```
( MEMBER-*  (LST ARG)
   (COND
    ( (NULL LST)  NIL )
    ( (LISTP (CAR LST))  (OR (MEMBER-* (CAR LST) ARG)
                             (MEMBER-* (CDR LST) ARG)) )
    ( (EQ (CAR LST) ARG)  T )
    ( T  (MEMBER-* (CDR LST) ARG)) ) )
```

{1} Do not confuse the LISP functions AND and OR with the abstract notion of AND and OR with respect to PREDICATES, BUILDERS, etc.

In both of the above cases, key program examples are used to make analogies and adjustments to the plans. Since the plans serve as a basis for generating programs, this "little effect" can actually have far-reaching effects! Here is another case in which the utility of thinking in terms of abstractions is demonstrated.

8. Two Issues: Analogies Between Lists and Numbers, and Types of Recursion

We have developed a great deal of machinery in order to cope with problems about lists. It would be quite useful if, in looking at a new data type, e.g., numbers, some of that machinery might carry over to the new domain. While we have not as yet worked out this aspect in detail, some prelimenary observations might be thought-provoking.

Consider the following program, PLUS, which adds two numbers together:

```
( PLUS  (N1 N2)
   (COND
    ( (ZEROP N1)  N2 )
    ( T  (ADD1 (PLUS (SUB1 N1) N2))) ) )
```

Now recall the COPY program. An analogy can be made between the two programs by noting that if PLUS were given two numbers, n and zero, to add, PLUS would effectively return a copy of the original argument, n. Thus, one can point out that in the number domain, ADD1 (and PLUS) can serve as a "NUMBER-BUILDER" just as CONS served as a LIST-BUILDER and AND/OR served as LOGICAL-BUILDERS. {1}

{1} Also, note that ZEROP and NULL serve analogous functions. [Friedman 1974]

Now, consider the following code for PLUS

```
( PLUS  (N1 N2)
   (COND
    ( (ZEROP N1)  N2 )
    ( T  (PLUS (SUB1 N1) (ADD1 N2))) ) )
```

After some consternation, students come to realize that this PLUS is adding "on the way down" while the previous PLUS is adding "on the way back up." Two observations quickly follow. First, students realize that this latter PLUS "looks like" a PREDICATE program, since the answer is actually available "at the bottom" in both cases, i.e., there is nothing to do but pass the answer back up. Next, students typically ask "well, can't we make COPY build on the way down too, i.e., can't we move CONS inside?" The students go on to modify COPY to include a second argument which is used to hold the list being built "on the way down," e.g.,

```
(COPY  (LST1 LST2)
   (COND
    ( (NULL LST1)  LST2 )
    ( T  (COPY (CDR LST1) (CONS (CAR LST1) LST2))) ) )
```

The students also quickly see how to modify the BUILDER and SELECTOR plans to incorporate this type of list construction. This exericse makes explicit the different types of recursion actually being used.

9. Concluding Remarks

Once the enterprise of finding abstractions becomes ingrained, generalizations start to pop up all over. For example, the utility of MAPping functions is readily recognized; MAPping functions are, in effect, built-in templates. After the BUILDER 'MAPCAR' is introduced, then PREDICATE MAPping functions such as MAPOR and MAPAND follow quite

naturally. Or, when the problem of REVERSing a list is solved, the students come to see that working from the left is not sacred. Thus operators such as RAC, RDC, and SNOC, which are counterparts to CAR, CDR, and CONS, but which work from the right, come into being {1}. Moreover, one can see how these new functions can systematically replace CAR, CDR, and CONS in the templates and plans.

As more and more examples of problems and programs are examined, the level of the plans grows further and further away from actual LISP code. Consider this problem:

> Write a program wich returns True if and only if every other element in the input list is an atom.

In order to accomodate this AND-PREDICATE, we need to generalize clause 3e to permit "bigger chunks" of the list to be consumed in the recursion step, i.e.,

```
( predicate (CDR (CDR LST)) )
```

Problems which require that clauses 2p and 2e also be generalized can be readily generated. The result is that the plans --- the templates and the comments --- become more abstract. For example,

```
( predicate   (argument list)
   ( COND
     ( (NULL list)  < T, NIL > )
     ( (test element of CDRing variable)  < NIL, T > )
     ( T  (predicate (reduce CDRing variable))) ) )
```

More thought is required to use such plans; students can not count on making simple substitutions in order to produce correct solutions. Nonetheless, these abstractions provide a context in which to think about a particular problem or program.

{1} The names RAC, RDC, and SNOC are also taken from Friedman [1974].

At this point, a valid question to ask is: what do the particular generalizations discussed in this paper have to do with programming in BASIC, FORTRAN, APL, or PASCAL? Clearly, one could use function subprograms to mimic the functional decomposition used in LISP, but the particular taxonomy of PREDICATES, BUILDERS, and SELECTORS may not be valid. The major types of problems addressed by LISP are those which deal with non-numeric entities structured into lists. FORTRAN, APL, etc. have arrays and numbers as key data types, e.g., a BUILDER might not have a clean analogue in the context of arrays. Nonetheless, the search for problem-program abstractions may prove worthwhile. For example, in teaching introductory BASIC, we emphasized a class of problems based on the theorem in mathematics which states that all functions can be approximated by a series expansion. We dubbed the program structure which captures this set of problems the "running-total template", e.g.,

```
FOR I = 1 TO n
     TOTAL = TOTAL {+,*} new term
NEXT I
```

This generalization was quite well received by the students.

Our classroom experience with the LISP material has been exciting and rewarding. We felt that, generally speaking, the students developed a keen understanding of problem solving and programming in LISP and an appreciation for the enterprise of finding abstractions, as demonstrated by their participation in class discussions and by their homework. It would be very interesting to follow them to see if this approach has any lasting effect. Currently, we are encoding the knowledge described in this paper about problems and programs into the knowledge base of a

computer-assisted instruction system, MENO-II [Soloway, et al., 1979]. Possibly in this context, more controlled experiments can be performed to ascertain the effect of our approach.

Acknowledgments

We have had the good fortune to discuss this work with a number of different individuals; we greatly appreciate their efforts. Thank you Jeff Bonar, Dan Corkill, Allan Collins, Dan Friedman, Steve Levitan, John Lowrance, Edwina Rissland, and Piet Vermeer. We would also like to thank Janet Turnbull whose quite competence facilitated our getting this paper "in shape and out the door on time."

Bibliography

Clement, J., Lochhead, J., and Soloway. E. (1979) "Translating Between Symbol Systems: Isolating A Common Difficulty in Solving Algebra Word Problems," COINS Technical Report 79-19, Computer and Information Science Department, University of Massachusetts, Amherst.

Friedman, D. (1974) The Little LISPer, Science Research Associates.

Hardy, S. (1975) "Synthesis of LISP Functions from Examples," Fourth International Joint Conference on Artificial Intelligence, Tbilisi, U. S. S. R.

McCarthy, J., et al. (1965) LISP 1.5 Programmer's Manual, MIT Press, Cambridge, MA.

Papert, S. (1971) "Teaching Children to be Mathematicians versus Teaching about Mathematics," MIT A. I. Lab Memo 249, Cambridge, MA.

Piaget, J. and Inhelder, B. (1969) Psychology of the Child, Basic Books, NY.

Simon, H. (1978) "Problem Solving and Education," abstract of a presentation at the Conference on Problem-Solving and Education: Issues in Teaching and Research, Carnegie-Mellon University, Pittsburgh.

Soloway, E., Rissland, E., Bonar, J., Vermeer, P. and Woolf, B. (1979) "A Description of the ICAI System MENO-II," Department of Computer and Information Science, University of Massachusetts, Amherst, in preparation.