From Problems to Programs via Plans: The Content and Structure of Knowledge for Introductory LISP Programming*

Elliot M. Soloway
Beverly Woolf

COINS Technical Report 80-19

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

* An earlier version of this paper appeared in the Eleventh ACM Technical Symposium on Computer Science Education, Kansas City, Feb. 1980.

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant Nos. DAHC19-77-G-0012 and MDA903-80-C-0508.

Abstract

When a programming language is taught, typically the syntax and the semantics of the language are emphasized, and little emphasis is placed on the *relationship* of *problems* to each other, and to the *constructs* of the language. We, however, report on an organization of information for teaching LISP, which puts *primary* emphasis on the structure of and relationships between: a *problem*, a *program*, and, an intermediate abstraction, a *plan*. This organization is based on an analysis of the underlying structure of ostensibly different problems, and their program solutions. We present qualitative observations on the use of this organization gleaned from actual classroom teaching. Finally, we speculate on the utility of this approach with respect to other programming languages.

1. Introduction

Those doing research into how people actually understand various topics, have observed that traditional sources of information, e.g., textbooks, do not usually convey knowledge which is key to a correct understanding of that subject matter. For example, Rissland [9] discusses a new structure for mathematical knowledge which emphasizes new categories of information, e.g., types of examples (start-up, reference, etc.), types of concepts (basic definitional, culminating, etc.). Brown, Collins, and Harris [2] use the term "deep structure" -- in contrast to "surface structure" -- to emphasize the importance of the underlying goals, plans, purposes of entities in a subject domain. From this perspective, they go on to provide an analysis of 3 subject domains (stories, electronic circuits, arithmetic).

In this paper, we describe work done in this vein with respect to the knowledge underlying a sound understanding of introductory LISP programming. Typically, when a programming language is tuaght, the syntax and semantics of constructs in the language are emphasized; besides a perfunctory one-line description, little time is given to issues such as when to use a construct, how constructs are related, what functional role a construct plays in a program, or what the deep, plan structure of programs are. Similarly, little recognition is paid to issues such as how problems are related to each other, and how problems relate to constructs in a language. In what follows, we present a new structure for information about introductory LISP programming, which does infact emphasize the neglected issues mentioned above. 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, i.e., abstractions, each of which captures the essential features of a class of problems, and corresponding solution programs.

As we shall attempt to show, the knowledge underlying introductory LISP programming is quite structured. That is, while there are a lot of details that one must know in order to program, these details can be highly organized. Emphasis then is placed on the organization of the knowledge rather than the amount of knowledge.

We begin by developing a scheme for classifying the problems usually offered as exercises 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  (LST ARG)
   (COND
    ( (NULL LST)  NIL )
    ( (EQ (CAR LST) ARG)  T )
    ( T  (MEMBER (CDR LST) ARG))))
```

If we step back from the particulars of the problems and the programs, we notice first that both problems 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 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, this similarity of function can be reflected in the programs.

Below we list a <u>template</u> which captures the commonalities and differences in the above two programs. (2)

```
( 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 <u>slots</u> 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 understand the relationships abstracted in the template, one needs to add <u>explanatory comments</u> to the clauses of the template. Thus, we define a <u>plan</u> to be a <u>template</u> plus <u>explanatory comments</u>. 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 "smaller" problems; recursion is at the heart of this process. Comments reflecting this point could be:

TEMPLATE CLAUSE	EXPLANATORY COMMENT
1p	Is the list empty? Are we done processing?
2p	Do it to the first; test the "head" of the list.
3e	Do it to the rest; continue 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. Note that an AND-PREDICATE can stop processing if it finds 1 element which does not meet the specified criteria. Alternatively, if an OR-PREDICATE, such as MEMBER, has visited each element in the list and has not succeeded in finding the desired property, it will return false (NIL in LISP). Explanatory concepts in the plan which describe these goals could be:

TEMPLATE CLAUSE	EXPLANATORY COMMENT
1e	Return a truth value after all elements have been visited: 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 no element which satisfies the test in 2p has been found, thus return False.
2e	Return a truth value when a particular element has been found: 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; quit 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. Before proceeding however, three comments are in order. First, in the classroom, 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 are trying to foster. Second, we do not view a template as a type of equation into which numbers or functions can be plugged blindly. The misuse of equations by students in this manner in fields such as physics and engineering has been documented previously (cf. [4]). 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 third issue is the <u>level</u> of abstraction of the plans described here. One could imagine plans which do <u>not</u> 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 <u>without</u> 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. [8]), 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 [4]. With this caveat in mind, we encourage the search for even more general abstractions. (We return to this point in section 9.)

2.1 <u>Attacking a Problem: Using Plans to Ask Questions</u>

Another way to look at the knowledge surrounding a plan is in terms of <u>questions</u> 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. {3} 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 the result of some action (remove, insert, etc.) on the elements which satisfied the test criteria, as well as all the elements which did not. 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 traversed further. However, for an AND-BUILDER, such as REMOVE-ALL-MEMBERS, the second alternative is required in order to continue down the list.

Now, clause 2c 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   (LST OLD NEW)
   (COND
    ( (NULL LST)  () )
    ( (EQ (CAR LST) OLD)  (CONS NEW ((CDR LST)) )
    ( T  (CONS (CAR LST) (SUBSTITUTE ((CDR LST) OLD NEW))))
```

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 (4):

```
( 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 3c the BUILDER template uses 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 means 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 interpretation 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 copying. In particular, consider 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; all that changes is the particular test for the desired elements, and the action to be performed on such elements.

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 Lexicographically 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: {5}

```
( selector  (LST ARG1 ARG2 ... )
   (COND
    ( (NULL LST)   () )
    ( (test (CAR LST) ARG1 ...)
          < (CDR 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 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 preceding sections can be neatly depicted in a tree structure (see Figure 1); there are three classes of problems, each class having a similar structure. We do not claim that the scheme depicted above is unique, canonical, or complete. As we note in the concluding remarks, we feel that other problems and other languages might suggest or require other classification characteristics, i.e., other dimensions along which a problem taxonomy can be based. 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 accommodate the inconsistencies is a powerful learning technique.

The basis for the above taxonomy can be traced to the work of McCarthy [7], in developing LISP, to Friedman [5], who has written a self-teaching text for LISP, and to Hardy [6], who uses the template idea in a program synthesis system. Allen [1], Burge [2], and Winston [10], in their analyses of recursive programming in general, and LISP in particular, develop program structures larger than unit expressions which they too proceed to evolve.

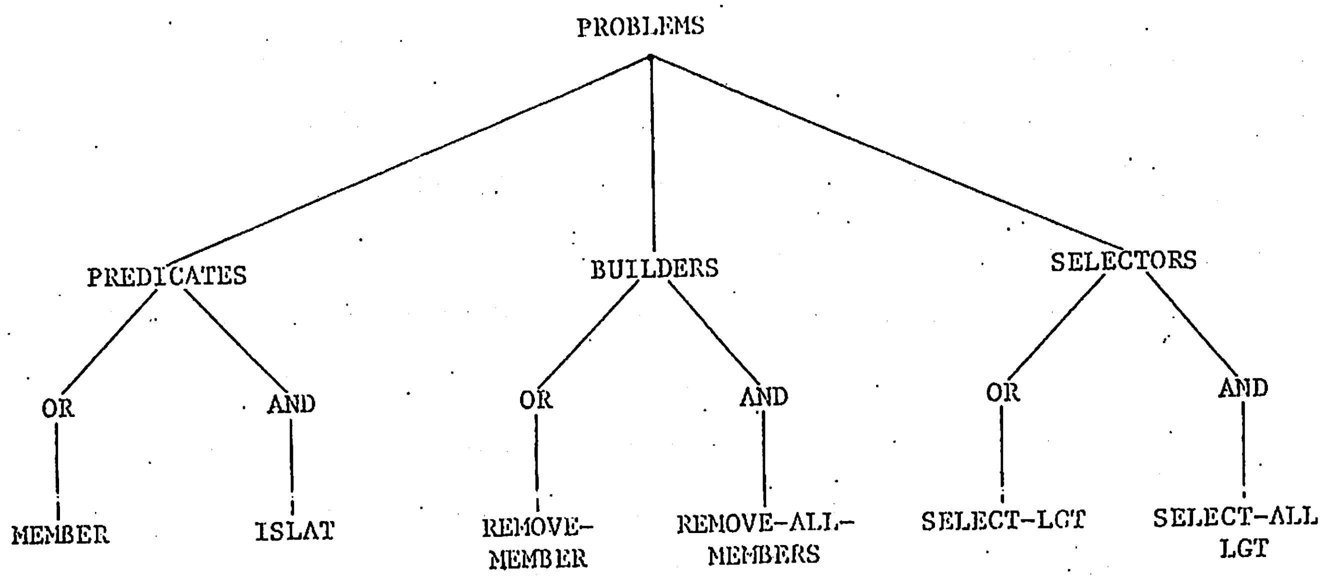

Figure 1.

A taxonomy of problems.

6. Generalizing the Plans to Handle an Exception

The problem,

> Write a program which takes the union of two lists.

is an AND-BUILDER; it is actually just a "generalized" version of REMOVE-ALL-MEMBERS, with MEMBER replacing EQ. However, the code

```
( UNION  (SET1 SET2)
   (COND
    ( (NULL SET1)  SET2 )
    ( (MEMBER (CAR SET1) SET2) (UNION ((CDR SET1) SET2))
    ( T  (CONS (CAR SET1) (UNION (CDR SET1) SET2)))))
```

does not completely follow the template for AND-BUILDERS. 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 BUILDER plan (and most probably, the other plans too). 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 admit of exceptions is precisely the kind of problem solving skill that one wants to encourage.

7. Generalizing the Plans to Handle Nested Lists

In the preceding discussion we have assumed that the input lists would be simple (flat) lists of atoms, e.g., (A B C). However, if we want to input nested 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 {6} is encountered, and the program must call itself again on this portion of the list, as well as on the rest of the list. {7} Thus, if we wanted to write REMOVE-ALL-MEMBERS to handle this type of list, the code for this new program, REMOVE-ALL-MEMBERS-* {8}, might be:

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

Two observations can now be made. First, students notice that the above AND-BUILDER works exactly like the old REMOVE-ALL-MEMBERS, i.e., REMOVE-ALL-MEMBERS-* will delete all occurrences of the desired atom from each of the sublists. Based on this observation, students quickly examine the OR-BUILDER case to see if it too can be extended {9}.

Second, in trying to see how to make AND-PREDICATES and OR-PREDICATES work on nested 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. (10) That is, just like CONS, AND and OR "glue together" the result of operating on the head of the list, with the result of operating on the tail of the list. 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)) ) )
```

In both of the above cases, key program examples are used to make analogies and adjustments to the <u>plans</u>. Since the plans serve as a basis for generating <u>new</u> programs, making this little change in the plans will actually have far-reaching effects!

8. <u>Two Issues: Analogies Between Lists and Numbers, and Types of Recursion</u>

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 list COPY program (reprinted below).

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

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. (11)

Now, consider the following code for PLUS,

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

After some thought, students come to realize that this PLUS is adding "on the way down" while the previous PLUS was 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 exercise makes explicit the different types of recursion actually being used.

9. <u>Concluding Remarks</u>

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 [12]. 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 farther and farther 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 accommodate 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.

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 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.,

```
TOTAL = staring-value, {0 1 ... }
FOR I = 1 TO n
    TOTAL = TOTAL {+,*} new term
NEXT I
```

This paper has emphasized the knowledge which goes beyond the syntax and semantics of constructs in LISP. Currently, we are using this same approach to analyze the knowledge students' need in order to understand Pascal programming. Before crisp generalizations can be made, more of this specific type of research must be carried out. Our classroom experience with LISP, while admittedly impressionistic, gives us confidence that this enterprise has a great deal of promise.

Notes

{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 the students' misunderstanding is a fair price to pay for easing them 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.

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

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

where 'p' stands for predicate, and 'e' stands for expression.

{3} 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.

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

{5} 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 ((dr LST)...)) >
```

{6} We assume here that EQ is undefined if its arguments are not atoms.

{7} This observation was pointed out by John Lowrance in a course he taught on LISP.

{8} The naming convention of * for procedures which require double recursion is taken from Friedman [5].

{9} In "some sense" the OR-BUILDER-* REMOVE-MEMBER-* removes the first occurrence of the desired atom, i.e., it removes the first first occurrence of the desired atom from each <u>nested</u> <u>list</u>, <u>except</u> when occurrence of the desired atom from each nested list is preceeded by the desired atom. This rule that nested list is preceeded by the desired atom. This rule applies to <u>all</u> sublists at the same depth of nesting.

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

{11} Also, note that ZEROP and NULL serve analogous functions. [5]

{12} The names RAC, RDC, and SNOC are also taken from Friedman [5].

Bibliography

[1] Allen, J., 1978, Anatomy of LISP, (New York: McGraw-Hill Book Co.).

[2] Brown, J.S., Collins, A., and Harris, G., 1978, Artificial Intelligence and Learning Strategies, H.F. O'Neil Jr. (Ed), in Learning Strategies, H.F. O'Neil Jr., (Ed), (New York, N.Y.: Academic Press).

[3] Burge, W., 1976, Recursive Programming Techniques, (Reading, Mass.: Addison-Wesley).

[4] Clement, J., Lochhead, J., and Soloway, E., 1980, Positive Effects of Computer Programming on Students' Understanding of Varibles and Functions, Proc. of the National ACM Conference, Nashville.

[5] Friedman, D., 1974, The Little LISPer, (Menlo Park, Calif.: Science Research Associates).

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

[7] McCarthy, J., et al., 1965, LISP 1.5 Programmer's Manual, (Cambridge, Mass.: MIT Press).

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

Rissland (Michener), E., 1978, The Structure of Mathematical Knowledge, Technical Report No. 472, M.I.T. Artificial Intelligence Lab., Cambridge.

[10] Winston, P., 1977, Artificial Intelligence, (Reading, Mass.: Addison-Wesley).