MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence
Memo No. 208 September 1970

TEACHING PROCEDURES IN HUMANS AND ROBOTS

Carl Hewitt

This paper was originally presented at the Conference on Structural Learning, April 5, 1970, Philadelphia, Pa.

Work reported herein was supported by the Artificial Intelligence Laboratory, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002.

Teaching Procedures in Humans and Robots

1. Abstract

Analysis of the structure of procedures is central to the foundations of problem solving. In this paper we explore three principle means for teaching procedures: telling, canned loops, and procedural abstraction. The most straightforward way to teach a procedure is by telling how to accomplish it in a high level goal-oriented language. In the method of canned loops the control structure that is needed for the procedure is supposed and the procedure is deduced. In the method of procedural abstraction the procedure is abstracted from protocols of the procedure on examples.

2. The Importance of Procedures in the Structural Foundations of Problem Solving

Several fundamental questions must be faced by any foundation for problem solving. A foundation for problem solving must specify a goal-oriented formalism in which problems can be stated. Furthermore there must be a formalism for specifying the allowable methods of solution of problems. As part of the definition of the formalisms the following elements must be defined: the data structure, the control structure, and the primitive procedures. The problem of what are allowable data structures for facts about the world immediately arises. A foundation for problem solving must confront the problem of change: How can account be taken of the changing situation in the world? What are good ways to express problem solution methods and how can plans for the solution of problems be

is distinct from nondeterministic control. However, under certain conditions a process using one of the control structures can simulate a process using the other [3.]. PLANNER is a high level, goal-oriented language in which one can specify to a large degree what one wants done rather than how to do it.

We shall briefly explain some of the ideas behind PLANNER to support our contention of the extreme importance of a procedural basis for the foundations of problem solving. Since a formal definition of PLANNER is beyond the scope of this paper, our comments are necessarily somewhat vague and philosophical. PLANNER will play a crucial role in our theory of procedural teaching. One basic idea behind the language is a "duality" that we find between certain imperative and declarative sentences. For example consider the statement (implies A B). As it stands the statement is a perfectly good declarative statement. It also has certain imperative uses for PLANNER. For example it says that we should set up a procedure which will note whether A is ever asserted and if so to consider whether B should then be asserted. Furthermore it says that we should set up a procedure that will watch to see if it ever is our goal to try to deduce B and if so whether it is wise to make a subgoal to deduce A. Exactly the same observations can be made about the contrapositive of the statement (implies A B) which is (implies (not b) (not a)). Statements with such things as universal quantifiers, conjunctions, disjunctions, etc. also

have both declarative and imperative uses. PLANNER theorems are being used as imperatives when they are being executed and as declaratives when used as data.

Our work on PLANNER has been an investigation in PROCEDURAL EPISTEMOLOGY, the study of how knowledge can be embedded in procedures. The PRINCIPLE OF PROCEDURAL EMBEDDING says that intellectual structures can be analyzed through their procedural analogues. The following all have procedural analogues:

descriptions
recommendations
theorems
proofs
grammars
models of procedures
patterns

Descriptions have procedural analogues in the form of PLANNER procedures which recognize the objects described. Theorems in the predicate calculus correspond to PLANNER theorems for making deductions. Mathematical proofs correspond to plans in PLANNER for generating a valid chain of deductions. The PROGRAMMAR language of Terry Winograd provides a procedural analogue to obtain the kind of information that is supposed to be supplied by transformational grammars. Intricate patterns can be specified in procedural pattern matching languages. Models of programs are defined by procedures which state the relations that must hold between the variables of the program as control passes through various points.

From the above observations, we have constructed a language that permits both the imperative and declarative aspects of statements to be easily manipulated. PLANNER uses a pattern-directed information retrieval system. When a statement is asserted recommendations determine what conclusions will be drawn from the assertions. Procedures can make recommendations as to which theorems should be used in trying to draw conclusions from an assertion, and they can recommend the order in which the theorems should be applied. Goals can be created and automatically dismissed when they are satisfied. Objects can be found from schematic or partial descriptions. Provision is made for the fact that statements that were once true in a model may no longer be true at some later time and that consequences must be drawn from the fact that the state of the model has changed. Assertions and goals created within a procedure can be dynamically protected against interference from other procedures. Procedures written in the language are extendable in that they can make use of new knowledge whether it be primarily declarative or imperative in nature. Hypotheses can be established and later discharged.

The logical deductive system used by PLANNER is subordinate to the hierarchical control structure of the language. PLANNER theorems operate within a context consisting of return addresses, goals, assertions, bindings, and local changes of state that have been made to the global data base.

Through the use of this context we can guide the computation and avoid doing basically the same work over and over again. For example, once we determine that we are working within a group (in the mathematical sense) we can restrict our attention to theorems for working on groups since we have direct control over what theorems will be used. PLANNER has a sophisticated deductive system in order to give us greater power over the direction of the computation. In several respects the deductive system is more powerful than the quantificational calculus of order omega. We have tried to design a deductive system together with an elaborate control structure so that lengthy computations can be carried out without blowing up. Of course the control structure can still be used when we limit ourselves to using resolution [8.] as the sole rule of inference. In general a uniform proof procedure gives us very little control over how or when a theorem is to be used. The problem is one of the level of the interpreter that we want to use. A digital computer by itself will only interpret the hardware instructions of the machine. We can write a higher level interpeter such as LISP [9.] that will interpret assignments and recursive function calls. At a still higher level we can write an interpreter such as MATCHLESS which will interpret patterns. At the level of PLANNER we can interpret assertions, find statements, and goals. In general higher level interpreters have greater choice in the actions that they can take since instructions are phrased more

in terms of goals to be achived rather than in terms of explicit elementary actions. The problem that we face is to raise the level of the interpreter while at the same time keeping the actions taken by it under control. Because of its extreme hierarchical control and its ability to make use of new imperative as well as declarative knowledge, it is feasible to carry out very long chains of inference in PLANNER. Examples of some of the kinds of statements that can be made in the language are:

Find the second smallest integer that is sum of its factors.

Pick up all the red cubes that are on top of blue cubes and put them in the yellow box.

Assert that all the people in this room are older than Jack.

Find all the employees at MIT that are related to each other and give the relationship of each to the others.

We are concerned as to how a theorem prover can unify structural problem solving methods with domain dependent algorithms and data into a coherent problem solving process. By structural methods we mean those that are concerned with the formal structure of the argument rather than with the semantics of its domain dependent content. Examples of structural methods are the use of subgoals in PLANNER and the consequences of the consequent heuristic. By the consequences of the consequent heuristic, we mean that a problem solver should look at the consequences of the goal that is being attempted in order to get

an idea of some of the statements that could be useful in establishing or rejecting the goal. We need to discover more powerful structural methods. PLANNER is intended to provide a computational basis for expressing structural methods. One of the most important ideas in PLANNER is to bring some of the structural methods of problem solving out into the open where they can be analyzed and generalized. There are a few basic patterns of looping and recursion that are in constant use among programmers. Examples are the "for" statement of MATCHLESS, the "find" statement in PLANNER, and recursion on the car and the cdr in LISP. The "find" and "for" primitives are explained in the MATCHLESS and PLANNER documentation [5.]. The patterns represent common structural methods used in programs. They specify how commands can be repeated iteratively and recursively. One of the main problems in getting computers to write programs is to use these structural patterns with the particular commands that are available in a given problem solving domain. It is difficult to decide which if any of the basic patterns of recursion is appropriate in any given problem. The problem of syntesizing programs out of canned loops is formally identical to the problem of finding proofs by mathematical induction. Indeed many proofs can be fruitfully considered to define procedures which are proved to have certain properties. We have approached the problem of constructing procedures out of goal oriented language from two directions.

The first is to use canned loops (such as the find statement) where we assume a-priori the kind of control structure that is needed. The second approach is to try to abstract the procedure from protocols of its action in particular cases.

3. Teaching Procedures

3.1 How are the Foundations of Problem Solving Applicable to Teaching Procedures?

Crucial to our understanding of the phenomenon of teaching is the teaching of procedures. Understanding the teaching of procedures is crucial because of the central role played by the structrual analysis of procedures in the foundations of problem solving. How can procedures such as multiplication, algebraic simplification, and verbal analogy problem solving be taught efficiently? Once these procedures have been taught, how can most effective use of them be made to teach other procedures? In addition to being incorporated directly as a black box, a procedure which has already been taught can be used as a model for teaching other procedures with analogous structure. One of the most important methods of teaching procedures is telling. For example one can be told the algorithm for doing symbolic integration. Telling should done in a high level goal-oriented language. PLANNER goes a certain

distance toward raising the level of the language in which we can express a procedure to a computer. The language has primitives which implement fundamental problem solving abilities. Teaching procedures is intimately tied to what superficially appears to be the special case of teaching procedures which write procedures. Only by teaching good methods for constructing procedures out of goal-oriented language can we hope to teach ony but the most primitive procedures. The process of teaching a procedure should not be confused with the process of trying to get the one being taught to guess what some black box procedure really does (as is the case in in sequence extrapolation problems for example). The teacher is duty bound to tell anything that might help the one being taught to understand the properties and structure of the procedure. We assume that the teacher has a good model of how the student thinks. Also, just because we speak of "teaching", we do not thereby assume that anything like what classically has been called "learning" is taking place in the student. However, this does not exclude the possiblity that the easiest way to teach many procedures is through examples. We can give protocols of the action of the procedure for various inputs and enviroments. By "variabilization" (the introduction of identifiers for the constants of the examples) the protocols can be formed into a tree. Two branches of a protocol tree will be said to be compatible if the same actions are taken on both

branches. Two nodes on the protocol tree will be said to be compatible if the corresponding branches below the two nodes are compatible. Then a recursive procedure can be generated by identifying compatible nodes on the tree. We call the above procedure for constructing procedures from examples the PROCEDURAL ABSTRACTION of protocols. The program which is obtained by the process of procedural abstraction is not necessarily the one intended by the teacher. However, the teacher can always give more protocols to eliminate ambiguity or he can otherwise indicate how the program should be changed. Procedural abstraction can be used to teach oneself a procedure.

3.2 Examples of Procedural Abstraction

3.2.1 Computing a numerical function

For example suppose we are given the following protocols for a function f. An expression such as "new [5 * 4]" means that we are binding an identifier (whose name we do not necessarily know) with the value 5 * 4 = 20. We shall use polish prefix notation and enclose function calls between the characters "(" and ")". The protocols presented below are simplifications. In practice we can use a great deal more information within the protocol. For example when binding an identifier with a new value v by "new [v]", we might know that the identifier being bound is the same one bound in another place in the protocol.

```
(f Ø) := (new [Ø] TRUE: (= Ø Ø) SO 1)
Thus (f Ø) = 1

(f 1) := (new [1]
 FALSE: (= 1 Ø) SO
        1 * new [1-1] TRUE: (= Ø Ø) SO 1
Thus (f 1) = 1

(f 2) := (new [2]
 FALSE: (= 2 Ø) SO
        2 * new [2-1] FALSE: (= 1 Ø) SO
                1 * new [1-1] TRUE: (= Ø Ø) SO 1
Thus (f 2) = 2

(f 3) := (new [3]
 FALSE: (= 3 Ø) SO
        3 * new [3-1] FALSE: (= 2 Ø) SO
                2 * new [2-1] FALSE: (= 1 Ø) SO
                        1 * new [1-1] TRUE: (= Ø Ø) SO 1
Thus (f 3) = 6
```

By the process of "variabilization", we conclude that the above protocols are compatible with the following program which is in the form of a tree (which we shall call the protocol tree).

```
(f x) := (new [xØ= x]
if (= xØ Ø) then 1
        else xØ * new [x1=(xØ-1)] if (= x1 Ø) then 1
                else x1 * new [x2=(x1-1)] if (= x2 Ø) then 1
                        else x2 * new [x3=(x2-1)] if (= x3 Ø)
then 1
                                        else...
```

Now we identify compatible nodes on the protocol tree. For example

```
(new [x3=(x2-1)] if (= x3 Ø) then 1
        else...
```

is compatible with

```
(new [x2 = (x1 -1)] if (= x2 Ø) then 1
        else x2*new [x3= (x2-1)] if (= x3 Ø)
                else...
```

We can "identify" the two nodes as one node N by renaming identifiers.

```
N:= (new [x = (x2-1)] if (= x Ø)
        then 1
        else x*(N (x-1))
```

By identifying all of the compatible nodes of the protocol tree we obtain.

```
(f x) :=
if (= x Ø) then 1
        else x *(f (x-1))
```

The reader will note that f is in fact the factorial function. PLANNER procedures and theorems can be taught in precisely the same fashion. For example the computer can be taught to build a wall or recognize a tower from examples. We assume that the teacher has a good working model of the student who is being taught. The teacher attempts to convey a certain body of knowledge to the student. Of course the student will be told anything which might help him to understand the material faster.

3.2.2 Reversing a List at All Levels

We would like to give an example of procedural abstraction in a domain with structured data. The function "first" will take the first element of a list. The functions first and rest are named "car" and "cdr" respectively in LISP.

For example (first ((e f) a (j k))) is (e f). The function "rest" will take the rest. For example (rest ((e f) a (j k)) is (a (j k)). There is an ambiguity as to whether (a (rest (e f g)) d) should evaluate to (a (f g) d) or (a f g d). We shall resolve the ambiguity by introducing a new pair of delimiters for function calls "<" and ">" so that (a (rest (e f)) d) will evaluate to (a f g d). The function (atom x) will be true if x is an "atom" and thus cannot be broken down withe the functions first and rest.

Consider the following protocols for a procedure r:

```
(r a) := (new [a]
TRUE: (atom a)
SO a)
```

thus (r a) is a

```
(r (n)) := (new [(n)]
FALSE: (atom (n))
     SO
        (<new [(rest (n))]
        TRUE: (atom ())
             SO ()>
        (new [(first (n))]
        TRUE: (atom n)
             SO n)))
```

thus (r (n)) is (n)

```
(r (a b)) :=(new [(a b)]
FALSE: (atom (a b))
     SO
        (<new [(rest (a b))]
        FALSE: (atom (b))
             SO
```

```
                (<(new [(rest   (b))]
                TRUE: (atom   ())
                     SO ()>
                (new [(first   (b))]
                TRUE: (atom b)
                     SO  b))>
          (new [(first (a b))]
          TRUE: (atom  a)
               SO a)))
```

thus (r (a b)) is (b a)

```
(r ((a))) :=(new [((a))]
FALSE: (atom ((a)))
     SO
        (<(new [(rest ((a)))]
        TRUE: (atom ())
             SO ()>
        (new [(first ((a)))]
        FALSE: (atom (a))
             SO
                (<(new [(rest (a))]
                TRUE: (atom ())
                     SO ()>
                (new [(first (a))]
                TRUE: (atom a)
                     SO a)))>
```

thus (r ((a))) is ((a))

We obtain the following protocol tree:

```
(r x) := (new [x1= x]
if (atom x1)
     then x1
     else
        (<new [x2=(rest x1)]
        if (atom x2)
             then x2
             else
                (<new [x3=(rest x2)]
                if (atom x3)
                     then x3
                     else...>
                (new [x4=(first x2)]
                if (atom x4)
                     then x4
                     else...))>
```

```
(new [x5=(first x1)]
if (atom x5)
     then x5
     else
        (<new [x6=(rest x5)]
        if (atom x6)
             then x6
             else...>
        (new [x7=(first x5)]
        if (atom x7)
             then x7
             else...)))
```

By identifying compatible nodes we obtain:

```
(reverse-at-all-levels x) :=
        (if (atom x)
             then x
             else
                (<reverse-at-all-levels (rest x)>
                        (reverse-at-all-levels (first x))))
```

3.2.3 Finding the Description of a Stick

Suppose that we have the following data base:

```
(block a)
(block b)
(glued a b)
```

See figure 1 for an interpetation of the above data base. Suppose that we are told that the above situation represents a stick on the basis of the following protocol:

```
(goal (stick a b)) ;set up a goal to prove that there is a stick from a to b
        (new [NO-VALUE NO-VALUE NO-VALUE] ; we have three new identifiers that initially do not have values
        consequent: (stick a b) ;the goal causes activation of a PLANNER theorem whose consequent matches (stick a b)
        TRUE: (proved? (glued a b)) ; if it has already been established that a is glued to b
             SO (return true)
```

Thus the above situation really does represent a stick. Now suppose that the data base is:

```
(block a)
(block b)
(block c)
(glued a b)
(glued b c)
(between a b c)
```

Figure 2 gives the interpretation of the above data base.

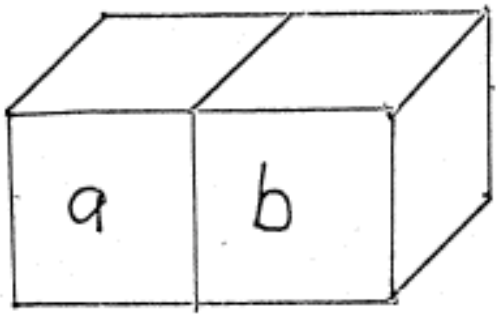
Suppose we obtain the following protocol on the above data base:

```
(goal (stick a c))
        [new NO-VALUE NO-VALUE NO-VALUE]
        consequent: (stick a c)
        FALSE: (proved? (glued a c))
             SO
                (proved? (block a))
                (goal (glued a b))
                (proved? (between a b c))
                (goal (stick b c))
                        [new NO-VALUE NO-VALUE NO-VALUE]
                        consequent: (stick b c)
                        is (proved (glued b c))
                             SO (return true)
```

By variabilization we obtain the following protocol tree:

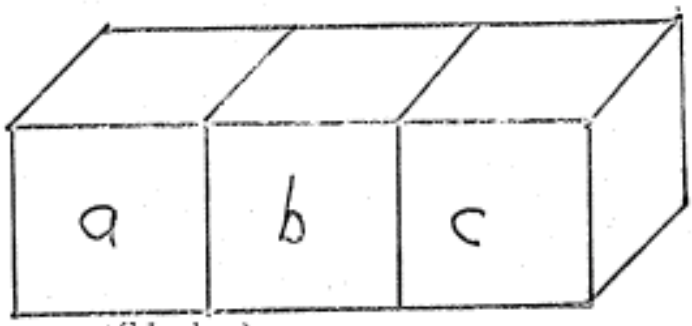
```
(goal (stick u v))
        [new x y z]
        consequent: (stick x z)
        if (goal (glued x z))
             then (return true)
             else
                (proved? (block x))
                (goal (glued x y))
                (proved? (between x y z))
                (goal (stick y z))
                        [new x1 y1 z1]
                        consequent: (stick x1 z1)
                        if (goal (glued x1 z1))
```

Figure 1.



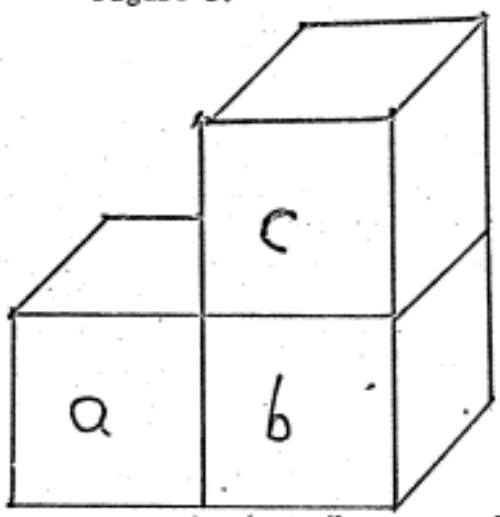
(block a)
(block b)
(glued a b)

Figure 2.



(block a)
(block b)
(block c)
(glued a b)
(glued b c)
(between a b c)

Figure 3.



(block a)
(block b)
(block c)
(glued a b)
(glued b c)

(not (between a b c))

```
                    then (return true)
                    else
                       (proved? (block x1))
                       (goal (glued x1 y1))
                       (proved? (between x1 y1 z1))
                       (goal (stick y1 z1))
```

By identifying compatible nodes we obtain the following consequent theorem which is the description of a stick.

```
(define stick-description (consequent
        (x y z) ;declare x y and z to be local identifiers
        (stick x z) ;this description is for statements of the
form (stick x z)
                (if (goal (glued  x z))
                     then (return true)
                     else
                        (proved? (block x))
                        (goal (glued x y))
                        (proved? (between x y z))
                        (goal (stick y z))))))
```

We see that (there is a stick from x to z in one of the following two cases:

1: x is glued to z

2: The block x is glued to some block y between x and z such that there is a stick between y and z.

The method of procedural abstraction is very much like a generalized form of compilation. The relationship between the compiled version and the interpreted version can be very subtle. The interpreted version can be the implicit behavior of an amorphous collection of general purpose goal-oriented language (say in PLANNER). The compiled procedure is the explicit solution of the problem in a precise algorithmic form. In classical compilers the relationship is much more

straightforward. Every time that the interpreter for the language changes the compiler must change. In fact the interpreter and compiler are two modes of what is essentially one program: an interpreter-compiler. In compile mode it would actually produce the code for the source code; in interpret mode it would take the actions corresponding to the compiled code that would be produced in compile mode.

4. Teaching Procedures by Deducing the Bodies of Canned Loops

If the type of control structure is known a priori, then the rest of the function can often be deduced. Often the control structure needed is a very commonly used loop such as the "for" loop in MATCHLESS, recursion on the tree structure of lists, or one of the loops in PLANNER such as "try", "find", or "act". We shall call loops such as the above "canned" loops since we will often pull them out and use them whole when we are in need of a control structure for a routine. The approach of using canned loops is the one used by Kleene [7.] for constructive realization functions for intuitionistic logic. Also, a very similar approach is used in [10.] and [1.]. Suppose that we know the following theorem about the predicate (REVERSEP x y) which means that y is the reverse of x. For example (reversep aa aa) and (reversep (1 2 (3 4)) ((3 4) 2 1)) are true. We shall use !< and >! as meta angle brackets for <

and > respectivly. The delimiters !(and)! are the meta braces for (and). The function "identity" is the the identity function. For example (identity b) evaluates to b and (s <identity (t u)> v) evaluates to (s t u v). The reason that we use meta-brackets is that we shall use a pattern directed formalism to talk about programs as objects. The function last will return the last element of a list and the function butlast will return all but the last. Thus (last ((a:b) c (e f))) evaluates to (e f) and (butlast ((a b) c (e f))) evaluates to ((a b) c). Using procedural abstraction the following definition can be produced from a few well chosen examples.

```
(reversep x y) :=
        (if (or (atom x) (atom y))
            then (= x y)
            else
               (and
                        (= (first x) (last y))
                        (reversep (rest x) (butlast y))
```

By mathematical induction we can show that (reversep a b) is true if the following PLANNER expression succeeds.

```
        (if (goal !(atom a)!)
            then
               ;If a is an atom then b should be equal to a.
               (goal (= a b))
            else
               (goal (not !(atom a)!))
               (goal !(reversep !(rest a)! c)!)
               ;Otherwise let c be the reverse of the rest of
a.
               (goal (= (!<identity c>! !(first a)!) b))
               ;the identity function is used to convert c into
the initial segement of b))
```

The above PLANNER expression gives methods by which a goal of the form (reversep a b) can be established. We would like to find a function reverse from lists to lists such that (reversep x (reverse x)) is always true. The PLANNER expression above suggests that we try to use linear induction on lists as the control structure. The schema for linear induction applied to attempt to construct a function reverse which satisfies the condition (reversep x (reverse x)) is:

```
!(reverse x)! :=
        !(if !(atom x)!
             then
                (temprog (Y) ;Y is a new local
                         ;Here we compute the code of what to do
if x is atomic.
                         (assert !(atom x)!)
                         (goal !(reversep x Y)!)
                         ; Find a Y which is the reverse of the
atom "x" and return it as value.
                         (return Y))
             else
                (temprog (Y)
                         ;Here we compute the code of what to do
if x is not atomic.
                         (assert (not !(atom x)!))
                         (assert !(reversep
                                 !(rest x)!
                                 !(reverse !(rest x)!)!)!)
                         ; Make the inductive hypothesiis that
the reverse of the rest of x satisfies the condition.
                         (goal !(reversep x Y)!)
                         ; Find a Y which the reverse of x and
return it as value.
                         (return Y)))
```

The above expression evaluates to the following definition:

```
(reverse x) :=
        (if (atom x)
             then x
             else (<identity (reverse (rest x))> (first x)))
```

4. Comparison of the Methods

There is not much to be said about teaching procedures by telling. It is not always clear whether the procedure should be taught from the top down or the primitives should be taught first. However, the basics of the method are simple and direct. Unfortunately the teacher will not always know the code for the procedure which is to be taught. He might be engaged in wishful thinking hoping to find a procedure with certain properties. The method of canned loops is often applicable to such cases. Trying to use the method of canned loops has the problem that the control structure must be supposed. Often it is very difficult to guess the kind of control structure which will prove appropriate. Also the method of canned loops works on the problem in the abstract as opposed to specific examples where the identifiers are bound to actual values. The advantage of the abstract approach is that if it succeeds then the procedure will be known by its construction to have certain properties. On the other hand it is often easier to see what to do on concrete cases. The approach of procedural abstraction is to combine together several concrete cases into one supposed general procedure. Properties of the general procedure must then be established by separate argument. If the protocols of the examples are produced by a goal-oriented language such as PLANNER, then there will be points along the

protocols where certain predicates are known to be true. The predicates express the fact that some goal was established as true at that point. Often it is possible to show by mathematical induction that the corresponding properties in the abstracted procedure are always true when the procedure passes through the points. In this way a problem solver can have a partial model of his problem solving procedures. The models can be expressed naturally in PLANNER. Also the method of procedural abstraction has the advantage that the control structure does not have to be supposed in advance. Often a problem solver will have the basic problem solving ability to solve any one of a certain class of problems. But he will not know that he has the capability. Writing a procedure which can be shown to solve the class enables the problem solver to bootstrap on his previous work. Procedural abstraction itself is further evidence for the Principle of Procedural Embedding. To implement the principle as a research program requires a high level goal-oriented formalism. PLANNER and some embellishments that we have made to the language are first steps toward realizing the Principle of Procedural Embedding.

5. Acknowledgements

Thanks to the referee for many helpful suggestions. In the course of doing this work, I had useful discussions with Marvin Minsky and Seymour Papert.

6. Bibliography

1. Green C. C. (with Bob Yates), Application of Theorem Proving to Problem Solving. Proceedings of IJCAI. June 1969.

2. Hewitt, C. and Paterson, M., Comparative Schematology: Proceedings of Conference on Concurrent Systems and Parallel Computation at Woods Hole Mass. June 1970. Sponsored by Project MAC, M. I. T. June 1970.

3. Hewitt, C., More Comparative Schematology: Project MAC Artificial Intelegence Memo. June 1970

4. Hewitt, C., PLANNER: A Lanugage for Proving Theorems and Manipulating Models in a Robot, Proceedings of IJCAI.

5. Hewitt, C., PLANNER: a Language for Proving Theorems, Artificial Intelligence Memo 137, Massachusetts Institute of Technology (project MAC), July 1967. Revised June 1970.

6. Hewitt, C., Functional Abstraction in LISP and PLANNER, Artificial Intelligence Memo 151, Massachusetts Institute of Technology (project MAC),

7. Kleene, S. C., Introduction to Metamathematics: North Holland.

8. Robinson, J. A., A Machine-oriented Logic based on the Resolution Principle. Journal Association of Computing Machinery. Vol. 12, pg. 23-41.

9. McCarthy, J.; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; and Levin, Michael I. 1962. Lisp 1.5 Programmer's Manual, M. I. T. Press.

10. Waldinger and Lee, PROW: A Step Toward Automatic Program Writing. Proc. IJCAI.