MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A.I. LABORATORY

Artificial Intelligence
Memo No. 270

October 1972

TEACHING OF PROCEDURES--PROGRESS REPORT

Gerald Jay Sussman

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

Progress Report — Teaching of Procedures

by Gerald Jay Sussman

The idea of building a programmer is very seductive in that it holds the promise of massive bootstrapping and thus ties in with many ideas about learning and teaching. I will avoid going into those issues here. It is necessary, however, to explain what I am not working on. I am not interested in developing new and better languages for expressing algorithms. When FORTRAN was invented, it was touted as an automatic programmer, and indeed it was, as it relieved the user of complete specification of the details of implementation. Newer programming languages are just elaborations (usually better) of that basic idea. I am, however, interested in the problem of implementation of a partially specified algorithm rather than a complete algorithm and a partially specified implementation. This problem is truly in the domain of Artificial Intelligence because the system which "solves" this problem needs a great deal of knowledge about the problem domain for which the algorithm is being constructed in order to "reasonably" complete the specification. Indeed, a programmer is not told exactly the algorithm to be implemented, he is told the problem which his program is expected to solve.

A programmer hardly ever starts with no program at all. Usually, he has a program which is almost but not quite applicable to the need. In this case, the programmer determines how the given program is to be modified to display the desired behavior; he makes a patch. This

technique is closely related to debugging, in which a program designed to solve a known problem misbehaves in some case. The bug must be understood and a patch concocted. In general, under good conditions, the creation of a program to solve some problem can be viewed as debugging an existing program.

Thus, the purpose of this project is to produce a "programmer." I am relying deeply on introspective concepts of how I do programming.

I. Automatic Program Construction

I have been constructing a program called "HACKER" which writes and debugs programs in the BLOCKS world. It currently shows signs of life, writing some elementary programs by debugging and patching. The program HACKER as well as the programs it operates on are written in CONNIVER.

The physics of the BLOCKS world is as follows: there are blocks on a table and a hand. The hand can lift only one block at a time. Hence the hand cannot lift a block if there are others on it. A block can only be placed on another if there is enough space on the other. Here we introduce one primitive which moves blocks in this world. (PUTON A B) puts block A on block B if A's top is clear and B has space for A on it, otherwise it produces an error. HACKER does not consider the structure of PUTON just as I don't think about CONS. HACKER does, however, know about PUTON and gets the error comments from it when it is incorrectly called. HACKER starts out with one well-commented but very limited program to work with:

```
(IF-NEEDED I-F-ON (IMPERATIVE-FOR (ON !(X (ATOM !,X)) !(Y (ATOM !,Y))))
          (NEEDS (AND (CLEARTOP !,X) (SPACE-FOR !,X !,Y))
                 (PUTON X Y))
          (ADIEU 'OK))
```

This program is called by pattern-directed invocation; if we need an imperative for getting atomic object X (not a tower) on an atomic object Y, it is called. Its body consists of a call to PUTON commented by its prerequisites. The (ADIEU 'OK) informs the caller of success.

HACKER also knows:

```
(IF-NEEDED M-O-CLEARTOP
        (MEANING-OF (CLEARTOP !'X)
                    (NOT (EXISTS !,(Y (GENSYM))
                                 (ON !,Y !,X))))
        (NOTE))
```

i.e. $\text{cleartop}(x) \triangleq \neg \exists y \ni. \ \text{on}(y,x)$

```
(IF-NEEDED S-F-NOT-ON
        (SUFFICES-FOR (NOT (ON !'X !'Y))
                (EXISTS !,(Z (GENSYM))
                        (WHERE (ON !,X !,Z) (NOT (= !,Z !,Y)))))
        (NOTE))
```

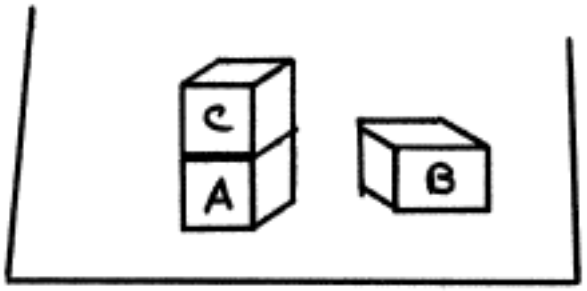
i.e. $\exists z \neq y \ni. \ \text{on}(x,z) \supset \neg \text{on}(x,y)$

Further concepts will be introduced as needed. In section II (Heuristic Programming) we will deal with more difficult concepts concerned with

space.

We talk to HACKER by asking it to achieve a desired final state, e.g. (ACHIEVE (ON A B)) or, say (ACHIEVE (AND (ON A B) (ON B C))). ACHIEVE first tests if its goal is already achieved and if so it returns OK. If not, it searches for a way to accomplish the goal.

Consider (ACHIEVE (ON A B)). If A has a clear top and there is space for A on B, then when I-F-ON is invoked, it calls (PUTON 'A 'B) which does the job. Suppose, however, the situation was:



(ON A TABLE)
(ON B TABLE)
(ON C A)

I-F-ON is invoked, it calls PUTON. PUTON gets angry and returns the error comment:

UNSATISFIED PREREQUISITE (NOT (ON C A))

HACKER then backtraces, checking the truth of the comment (NEEDS (AND (CLEARTOP ,X) (SPACE-FOR ,X ,Y))...). It finds it untrue. Since this is a NEEDS comment, it writes code to achieve the prerequisites and patches it in. Now I-F-ON looks like this:

```
(IF-NEEDED I-F-ON (IMPERATIVE-FOR (ON !(X (ATOM ,X)) !(Y (ATOM ,Y)))
    (SETUP (CODE-FOR (AND (CLEARTOP !,X) (SPACE-FOR !,X !,Y))
                     (PROG "AUX" ((PROTECTEDS NIL))
                           (PROG (ACHIEVE (CLEARTOP !,X))
                                 (PROTECT (CLEARTOP !,X)))
                           (PROG (ACHIEVE (SPACE-FOR !,X !,Y))
                                 (PROTECT (SPACE-FOR !,X !,Y)))
                           (UNPROTECT PROTECTEDS)))
           (PUTON X Y)))
```

SETUP is a comment explaining the reasons for the code written; to setup for (PUTON X Y). CODE-FOR is a comment explaining that its second argument was written to achieve the first argument. The code written binds a variable PROTECTEDS and initializes it to NIL. It then achieves and protects each subgoal, unprotecting them before leaving. Protection is a mechanism for catching bugs of interaction between subgoals (to be explained later).

This code was generated by a process I call pattern-directed macro-expansion from a macro in HACKER'S bag of coding tricks. (The concepts of a "bag of tricks" is due to Richard Greenblatt). This trick is fairly complex and looks like:

```
(IF-NEEDED C-F-AND (CODE-FOR (AND . !'L) !!CODE)
    "AUX" ((P NIL))
    (FOR-EACH-ELEMENT G L
         (CSETQ P (CONS (LIST 'PROG (LIST 'ACHIEVE G)
                                    (LIST 'PROTECT G))
                        P)))
    (CSETQ CODE (APPEND '(PROG "AUX" ((PROTECTEDS NIL)))
                        (REVERSE P)
                        '((UNPROTECT PROTECTEDS))))
    (NOTE))
```

The complexity stems from the ability to code for AND of any number of elements. HACKER then backs up the stack and starts running from (SETUP...). He gets to (ACHIEVE (CLEARTOP !,X)) before running into trouble. (CLEARTOP A) is not true so he looks for an imperative for cleartop. Not finding any he realizes it's time to write some code. Finding M-O-CLEARTOP he sees that CLEARTOP is a defined concept and thus he writes, by pattern-directed macro-expansion:

```
(IF-NEEDED (IMPERATIVE-FOR (CLEARTOP !X))
           (MEANING-OF (CLEARTOP !,X)
                       (ACHIEVE (NOT (EXISTS Z1 (ON Z1 !,X)))))
           (ADIEU 'OK))
```

HACKER then runs the new imperative but he needs to achieve the not exists expression. It is not yet true, has no imperative, has no explicit meaning, but we have a trick for this case:

```
(IF-NEEDED (CODE-FOR (NOT (EXISTS !V !'G))
                     (FOR-EACH !,V !!G1
                               (ACHIEVE (NOT !!G2))))
           (CSETQ G1 (SUBST (LIST '/! V) V G)
                  G2 (SUBST (LIST '/!/, V) V G))
           (NOTE))
```

Thus we expand the ACHIEVE, patching to get:

```
(IF-NEEDED (IMPERATIVE-FOR (CLEARTOP !X))
           (MEANING-OF (CLEARTOP !,X)
                       (CODE-FOR (NOT (EXISTS Z1 (ON Z1 !,X)))
                                 (FOR-EACH Z1 (ON !Z1 !,X)
                                           (ACHIEVE (NOT (ON !,Z1 !,X))))))
           (ADIEU 'OK))
```

FOR-EACH is a canned loop commonly found in CONNIVER programs. This code is run until it hits the expression (ACHIEVE (NOT (ON !,Z1 !,X)). Here we expand the macro S-F-NOT-ON getting:

```
(IF-NEEDED (IMPERATIVE -FOR (CLEARTOP !X))
    (MEANING-OF (CLEARTOP !,X)
            (CODE-FOR (NOT (EXISTS Z1 (ON Z1 !,X)))
                    (FOR-EACH Z1 (ON !Z1 !,X)
                        (SUFFICES-FOR (NOT (ON !,Z1 !,X))
                            (ACHIEVE (EXISTS Z2
                                        (WHERE (ON !,Z1 !,Z2)
                                            (NOT (= !,Z2 !,X))))))
            (ADIEU 'OK))
```

We run this new code until we have to expand the expression (ACHIEVE (EXISTS ...)). Here we need the trick:

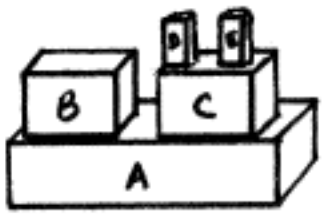
```
(IF-NEEDED (CODE-FOR (EXISTS !V (WHERE !'G !'Q))
            (PROG "AUX" (!,V)
                  (CHOOSE !,V !,Q (POSSIBLE !,G))
                  (ACHIEVE !,G)))
            (NOTE))
```

Thus we get:

```
(IF-NEEDED (IMPERATIVE-FOR (CLEARTOP !X))
    (MEANING-OF (CLEARTOP !,X)
        (CODE-FOR (NOT (EXISTS Z1 (on Z1 !,X)))
            (FOR-EACH Z1 (ON !Z1 !,X)
                (SUFFICES-FOR (NOT (ON !,Z1 !,X))
                    (CODE-FOR (EXISTS Z2 (WHERE (ON !,Z1 !,Z2)
                                                (NOT (= !,Z2 !,X))))
                              (PROG "AUX" (Z2)
                                    (CHOOSE Z2 (NOT (= !,Z2 !,X))
                                               (POSSIBLE (ON !,Z1 !,Z2)))
                                    (ACHIEVE (ON !,Z1 !,Z2)))))))))
    (ADIEU 'OK))
```

This code runs, CHOOSE (by magic of its own) makes Z2=TABLE. (ACHIEVE (ON C TABLE)) (remember Z1=C) finds I-F-ON and calls it recursively, solving the problem.

Note that by just running this specific example we get I-F-ON patched for situations where X is not CLEARTOP and a completely general CLEARTOP routine which not only solves this specific case but also any recursively or iteratively complex example. E.g. the performance program can now, without modification, achieve (CLEARTOP A) in:



Note also that the knowledge used in writing this program is divided into two independent classes:

1) Knowledge of the proglem domain:
   I-F-ON, M-O-CLEARTOP, S-F-NOT-ON
2) Domain independent programming knowledge:
   i.e. all CODE-FOR methods.

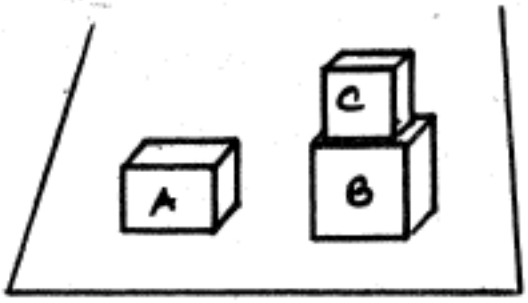
It is my hope and honest belief, that almost all "programming" knowledge can be coded into a small number (perhaps 50) coding tricks. New problem domain dependent knowledge can be easily added; for instance, I can define the concept of a 3-tower as:

```
(IF-NEEDED (MEANING-OF (3-TOWER !'X !'Y !'Z)
                       (AND (ON !,X !,Y) (ON !,Y !,Z))))
```

II. Heuristic Programming

So far we have been dealing with well-defined and understood concepts like CLEARTOP (which even has an explicit definition), NOT-ONness (for which we have a sufficient condition) and ONness. The blocks world also contains a much fuzzier concept, SPACE-FOR, which we can not so clearly work out. For achieving CLEARTOP, there is only one program. If it cannot do the job there is no hope. For SPACE-FOR, however, we have various strategies which can be tried, no one of which is guaranteed to work, nor is the failure of one an indication of ultimate failure. We now investigate the methods required to handle such concepts and the constructs which are entailed.

Suppose that the scene is:



(ON A TABLE)
(ON B TABLE)
(ON C B)

and we want to put A on B. (CLEARTOP A) is true but we get stuck at (SPACE-FOR A B). Since it has no direct route to achieving the goal, HACKER looks around for a way to assign blame for this failure. Here we introduce a new set of facts.

```
(IF-NEEDED M-H-SPACE-FOR-1
   (MAY-HURT (SPACE-FOR !'X !'Y) (CLUTTERED !,Y))
   (NOTE))

(IF-NEEDED M-O-CLUTTERED
   (MEANING-OF (CLUTTERED !'X)
          (EXISTS !,(Y (GENSYM)) (WHERE (ON !,Y !,X)
                                        (NOT (PROTECTED (ON !,Y !,X))))))
   (NOTE))

(IF-NEEDED M-H-SPACE-FOR-2
   (MAY-HURT (SPACE-FOR !'X !'Y) (HAPHAZARD !,Y))
   (NOTE))

(IF-NEEDED M-O-HAPHAZARD
   (MEANING-OF (HAPHAZARD !'X)
      (EXISTS !,(Y (GENSYM)) (WHERE (BADLY-PLACED !,Y !,X)
                                    (ON !,Y !,X))))
   (NOTE))

(IF-NEEDED C-F-NOT-WHERE
   (CODE-FOR (NOT (WHERE !'X !'Y)) (ACHIEVE (NOT !,X)))
   (NOTE))
```

These facts tell HACKER:

1) Lack of space can be attributed to two possible causes; a cluttered surface or a haphazardly arranged surface.
2) A surface is cluttered if it has on it objects which are not on it for a reason (if they were, the ONness relation would have been protected by the program which needs it preserved).
3) A surface is HAPHAZARD if it is not packed, that is, if there are "badly placed" objects on it.

HACKER comes up with M-H-SPACE-FOR-1 in searching for a blameful fact. He tests (CLUTTERED B) and finds it true. Thus he asigns it blame for the failure. Since this is an uncertain situation (as triggered by MAY-HURT) it gets compiled into a very special piece of code using a function STRATEGY-FOR. Also, since SPACE-FOR is a named concept (like CLEARTOP) we make an imperative for it:

```
(IF-NEEDED (IMPERATIVE-FOR (SPACE-FOR !X !Y))
   (STRATEGY-FOR (SPACE-FOR !,X !,Y)
                 (ACHIEVE (NOT (CLUTTERED !,Y))))
   (ADIEU 'OK))
```

This is then expanded (by straight pattern-directed macro-expansion as before) into:

```
(IF-NEEDED (IMPERATIVE-FOR (SPACE-FOR !X !Y))
   (STRATEGY-FOR (SPACE-FOR !,X !,Y)
      (MEANING-OF (NOT (CLUTTERED !,Y))
         (CODE-FOR (NOT (EXISTS Y3 (WHERE (ON Y3 !,X)
                                          (NOT (PROTECTED (ON Y3 !,X))))))
            (FOR-EACH Y3 (WHERE (ON !Y3 !,X)
                                (NOT (PROTECTED (ON !Y3 !,X))))
               (SUFFICES-FOR (NOT (WHERE (ON !,Y3 !,X)
                                         (NOT (PROTECTED (ON !,Y3 !,X)))))
                             (ACHIEVE (NOT (ON !,Y3 !,X))))))))
   (ADIEU 'OK))
```

Since (ACHIEVE (NOT (ON !,Y3 !,X))) is a problem already solved, this does it. STRATEGY-FOR is special in that it sets up an interrupt condition on its first argument being true. Thus if the situtation was: and the problem was (ACHIEVE (ON D C)) the FOR-EACH loop to get rid of A and B would not run to termination. As soon as A was removed (and put down on the table) control would return to STRATEGY-FOR because the (SPACE-FOR D C) would become true.

I have not described yet how to use the HAPHAZARD concept; I leave that to the next section on debugging for an example. Note, however, that only a few new programming concepts, CODE-FOR-NOT-WHERE, MAY-HURT and STRATEGY-FOR have been added in this section, but lots of new problem-domain concepts were introduced.

III. Automatic Debugging

Consider what will happen if we tell HACKER to (ACHIEVE (3-TOWER A B C)). He will write:

```
(IF-NEEDED (IMPERATIVE-FOR (3-TOWER !X !Y !Z))
   (MEANING-OF (3-TOWER !,X !,Y !,Z)
      (CODE-FOR (AND (ON !,X !,Y) (ON !,Y !,Z))
         (PROG "AUX" ((PROTECTEDS NIL))
            (PROG (ACHIEVE (ON !,X !,Y))
                  (PROTECT (ON !,X !,Y)))
            (PROG (ACHIEVE (ON !,Y !,Z))
                  (PROTECT (ON !,Y !,Z)))
            (UNPROTECT PROTECTEDS))))
   (ADIEU 'OK))
```

This program has a bug! The first clause of the (CODE-FOR (AND ...) ...) gets A on B. The second tries to get B on C. First it tries to (CLEARTOP B) so B can be grasped. This forces an attempt to (PUTON A TABLE) but the first clause protected (ON A B) so A can't be moved. Thus we get the error comment from PUTON: PROTECTION VIOLATION (ON A B). At this point, HACKER is entered via a BUG routine called from PUTON with the error comment as its argument. From this vantage point HACKER looks up the stack to see if there are any alternatives to be considered. He reasons: I would have violated the protection by putting A anywhere; I had to put

A somewhere to get it off B. Even if I had another way to get it off B I couldn't use it because the protection is on (ON A B). This restriction contradicts the meaning of (CLEARTOP B) and thus contradicts that goal. But (CLEARTOP B) is necessary to setup for (PUTON B C) so it doesn't help to try to achieve (SPACE-FOR B C) first. (a possible alternative -- a PLANNER program would have tried it, and failed). Thus all of the moves are forced from (ACHIEVE (ON B C)). This subgoal is thus inconsistent with the restriction; but it is necessary for the goal (AND (ON A B) (ON B C)), but above this the restriction is gone. In fact, the restriction was placed in order to protect the previously achieved subgoal (ON A B). Thus we have no way to achieve those subgoals in the given order so I'll patch it so the offending step is before the offended one:

```
(IF-NEEDED (IMPERATIVE-FOR (3-TOWER !X !Y !Z))
   (MEANING-OF (3-TOWER !,X !,Y !,Z)
      (CODE-FOR (AND (ON !,X !,Y) (ON !,Y !,Z))
         (PROG "AUX" ((PROTECTEDS NIL))
            (PROG (ACHIEVE (ON !,Y !,Z))
                  (PROTECT (ON !,Y !,Z)))
            (PROG (ACHIEVE (ON !,X !,Y))
                  (PROTECT (ON !,X !,Y)))
            (UNPROTECT PROTECTEDS))))
   (ADIEU 'OK))
```

This program is correct. But not all bugs are so simple; let us consider another:

The situation is that all blocks (A, B, and C) are on the table and the problem is: (ACHIEVE (U A B C)) where:

```
(IF-NEEDED (MEANING-OF (U !'X !'Y !'Z)
                       (AND (ON !,X !,Z) (ON !,Y !,Z)))
           (NOTE))
```

In this case the code generated is:

```
(IF-NEEDED (IMPERATIVE-FOR (U !X !Y !Z))
   (MEANING-OF (U !,X !,Y !,Z)
      (CODE-FOR (AND (ON !,X !,Z) (ON !,Y !,Z))
         (PROG "AUX" ((PROTECTEDS NIL))
            (PROG (ACHIEVE (ON !,X !,Z))
                  (PROTECT (ON !,X !,Z)))
            (PROG (ACHIEVE (ON !,Y !,Z))
                  (PROTECT (ON !,Y !,Z)))
            (UNPROTECT PROTECTEDS))))
   (ADIEU 'OK))
```

This code is nearly OK, but when we execute (ACHIEVE-(ON A C)) A goes on the middle of C (a heuristic to maximize stability). This blocks the (SPACE-FOR B C) becuase C isn't large enough to hold B with A on its middle. What does HACKER do? The performance program is in (STRATEGY-FOR

(SPACE-FOR B C) ...) when it discovers that it cannot win by the clutter strategy as (ON A C) is protected. Since there are no other strategies currently coded, STRATEGY-FOR asks HACKER for help. HACKER finds another strategy, M-H-SPACE-FOR-2; the HAPHAZARD strategy, and finds, indeed, that (HAPHAZARD C) is true. This is then coded up as before as an alternative strategy: (from now on I will leave out details of expansion, filling in with ... and only showing relevant segments.)

```
(STRATEGY-FOR (SPACE-FOR !,X !,Y)
   (MEANING-OF (NOT (CLUTTERED !,Y)) ... )
   (MEANING-OF (NOT (HAPHAZARD !,Y)) ... ))
```

This code indeed solves the problem but we must introduce some more primitives.

```
(IF-NEEDED (IMPERATIVE-FOR (PACKED !X !Y))
   (NEEDS (CLEARTOP !,X) (PACK X Y))
   (ADIEU 'OK))
```

```
(IF-NEEDED (SUFFICES-FOR (NOT (BADLY-PLACED !'X !'Y)) (PACKED !,X !,Y))
   (NOTE))
```

Note: STRATEGY-FOR takes any number of strategies and exhausts them in the given order.

Note: PACK pushes its first argument as far as it can from the center of its second argument, without falling off.

This patch, though it works in this case, and is often useful, is not a good patch. In fact, HACKER realizes the trouble quickly. We see that in running the patch, A is pushed to a corner of C. But A was the last object moved, (HACKER keeps a record of this) and if an object is moved twice with no intervening other manipulations, it should have been moved to the correct place the first time. This is not just a matter of efficiency — if the problem had been: (AND (ON A C) (ON D A) (ON B C) (ON E B)) we would have had trouble pushing the towers (D A) or (E B). Though there is a permutation of the AND which will work, the problem is compounded with compound objects (not yet discussed) as in (AND (ON (TOWER D A) C) (ON (TOWER E B) C)). In any case, HACKER has an eye peeled (an interrupt set) for double movements in execution of a single command. He allows the execution to proceed but leaves a note to himself to investigate the problem when the command returns successfully. Each time an object is moved, the "physics" leaves a note on the HISTORY list with the activation of the mover. Thus when the program successfully returns HACKER sees that the first mover of A was:
(PUTON A C) called by (ACHIEVE (ON A C))
and that the second was: (PACK A C) which left A in a good position. Thus he edits the first goal to read: (ACHIEVE (ON A C PACK)) which will put it down packed the first time (I lied to you about I-F-ON and PUTON: they really have an optional position defining argument whose default is "center").

The DEBUGGING knowledge appearing in this section is not as explicitly stated as in the preceeding sections because it is not yet coded. A major criterion for the code is that it break up into BLOCKS and programming knowledge.