MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence Memo. 137 July 1967.

PLANNER
A Language for Proving Theorems

Carl Hewitt

SCHEMATISE

The following is a description of SCHEMATISE, a proposal for a program that proves very elementary theorems through the use of planning. The method is most easily explained through an example due to Black.

Given

```
 t1:     (p4 c1 c2)
 t3:     (p4 c3 c4)
 t2:     (p4 c2 c3)
 t4:     (forall (x y) (implies (p4 x y) (p5 x y)))
 t5:     (forall (x y z) (implies (and (p5 y z) (p4 x  y)  )
(p5 x z)))
```

Prove

```
t6:     (p5 c1 c4)
```

The above problem has an interpretation that makes it a much easier one for humans.

Given

```
 t1:     (in pencil desk)
 t2:     (in desk home)
```

t3: (in home county)

t4: (forall (x y) (implies (in x y) (at x y)))

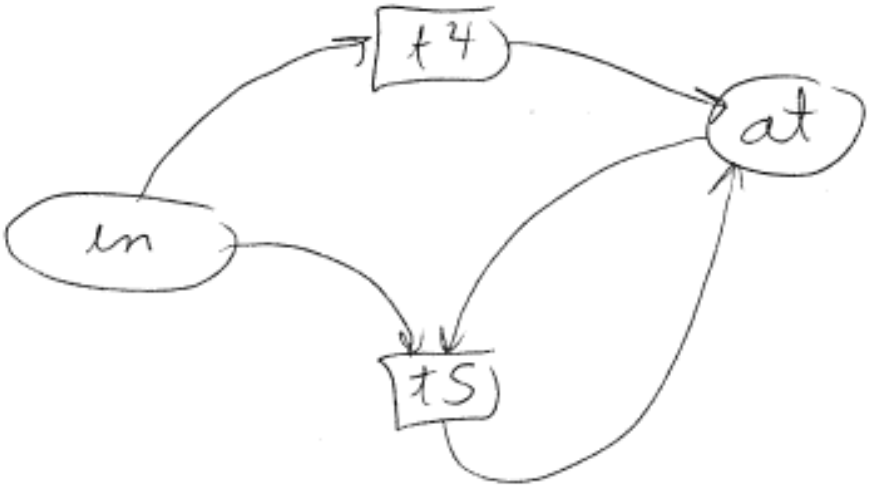
t5: (forall (x y z) (implies (and (in x y) (at y z)) (at x z)))

Prove

t6: (at pencil county)

As far as the theorem prover is concerned it is no easier to solve the problem given one representation rather than the other without additional information. A human would much rather work with the second representation since he has a well defined model for it. In order to make this paper more readable we shall work with the second representation. However, the reader should be careful to remember that in the sequel the computer program has a very different perspective from his own.

First the program connects all the theorems together into a net which we shall the theorem net. It is understood that the theorem net also reflects the structure of the rules of inference (operators) of the underlying logical system. In the system which we shall study the sole rule of inference will be modus ponens together with the simultaneous instantiation of any free variables with constants. The theorem net for our example is diagrammed in figure s0.

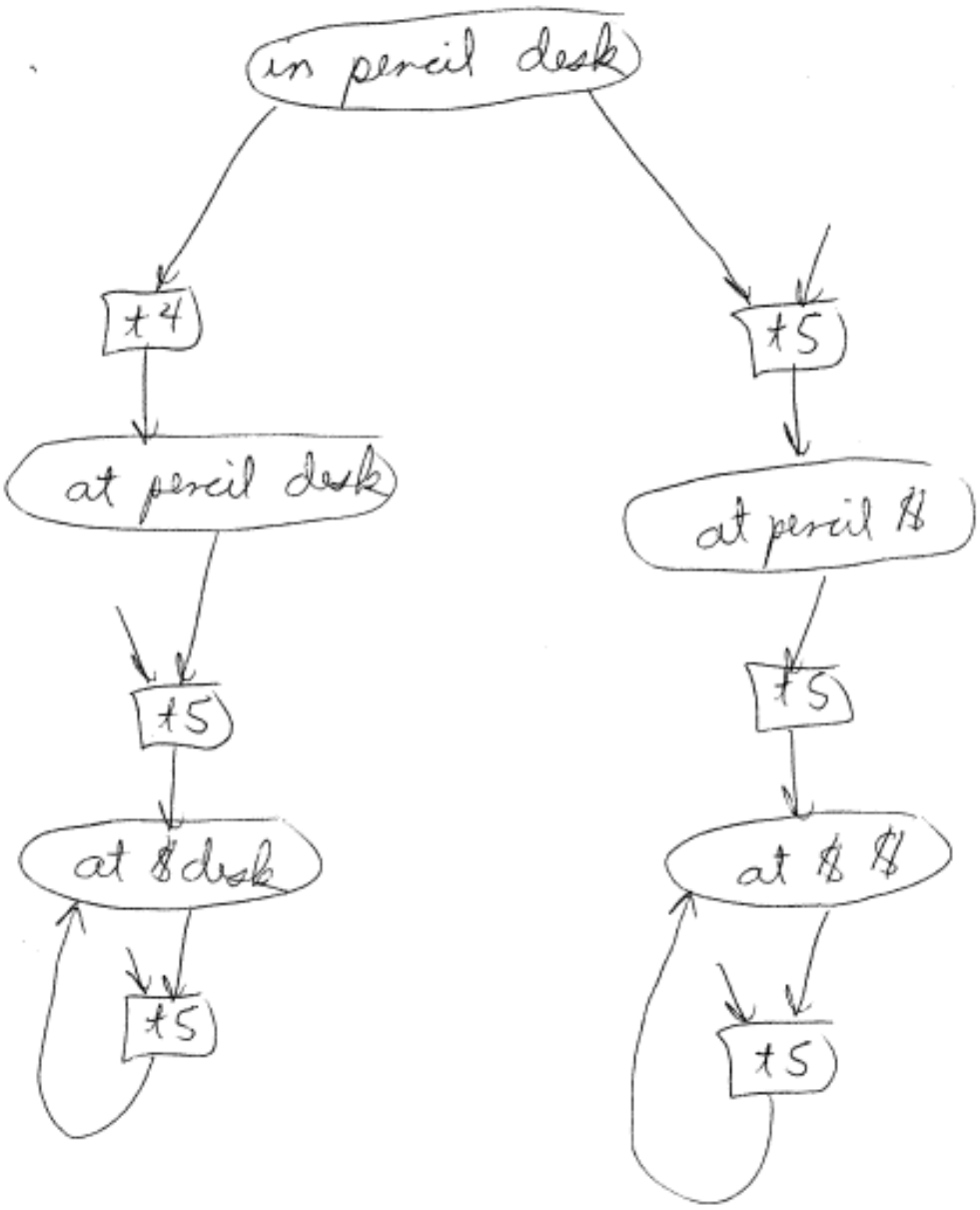


Theorem Net
figure 50

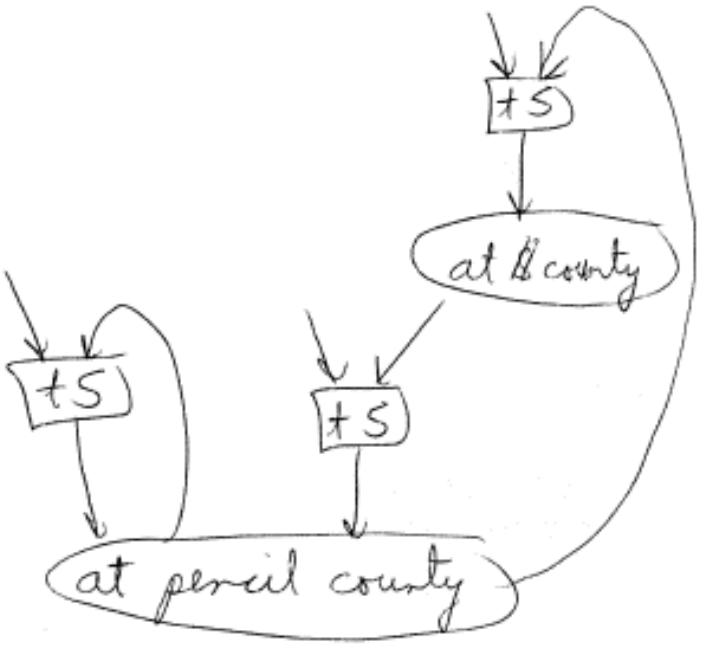
Each box (such as the one containing theorem t2) stores the information for binding variables in that particular theorem. Without some insight into the problem structure there is not much that the theorem prover can do except to initiate a straight backwards search for the proof of the theorem. But suppose that the theorem prover is more fortunate. It might expect that (in pencil desk) is relevant to the proof of (at pencil county) since the constant pencil appears no where else in the problem. Or perhaps that the theorem prover is not allowed to use theorem t1 and asked to prove theorem t6' instead of theorem t6 where

t6': (implies (in pencil desk) (at pencil county))

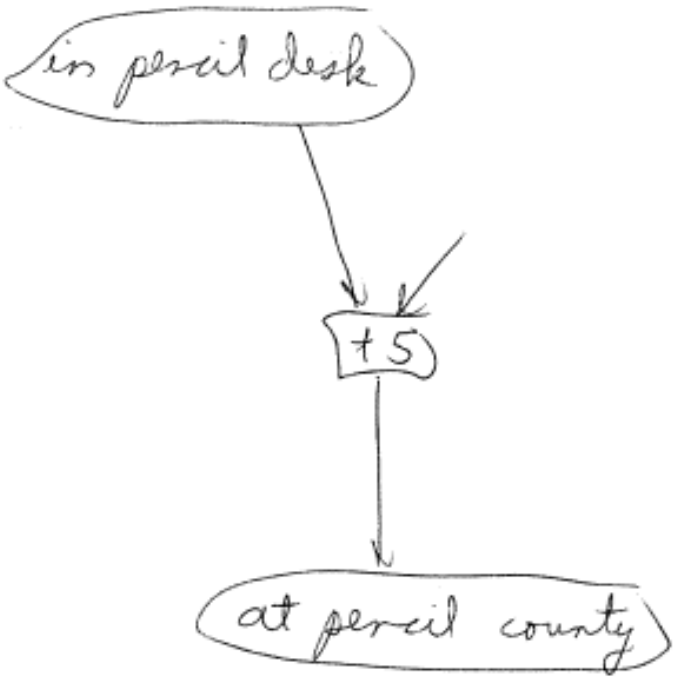
Now the theorem prover has enough information to try to form a plan. Following Minsky whe shall call the statements (such as (in pencil desk)) that the theorem prover thinks are relevant to the proof <u>islands</u>. We define the <u>difference</u> between two statements A and B to be the subgraph of the theorem net that can possibly carry A into B. In our case the difference between (in pencil desk) and (at pencil county) is the whole theorem net. We might proceed to reduce the difference between our antecedent and consequent by chaining forward from our island to try to match the consequent. The $ stand for constants that are presently unknown to us.



Chaining Forward From 0 (in pencil desk)
figure 51

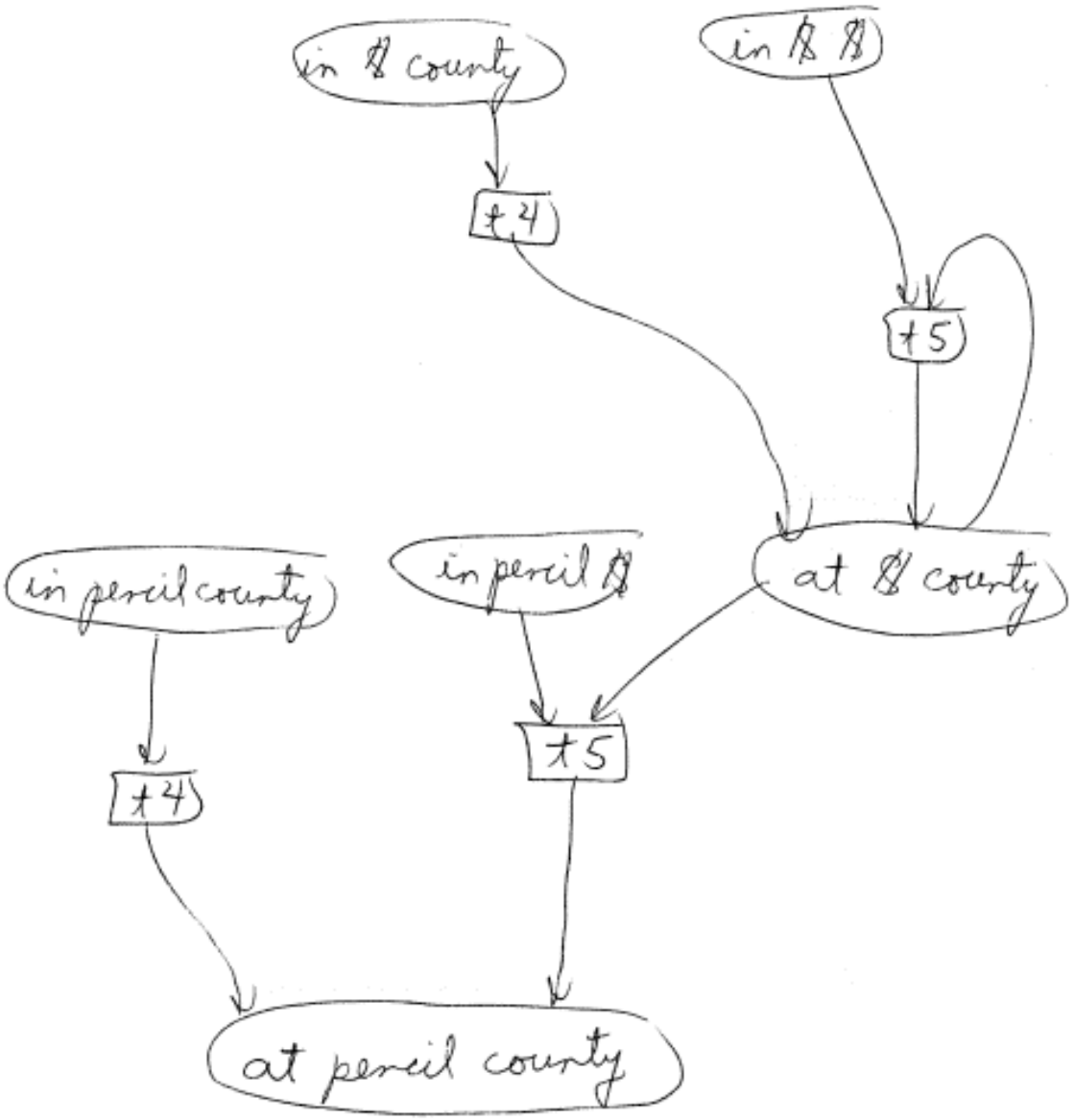


plugging (at pencil county) Into (at ℓℓ ℓℓ) And Then Chaining Backwards figure 52



Plugging (at pencil coonty) into (at pencil ?)
And Then Chaining Backwards

figure 53

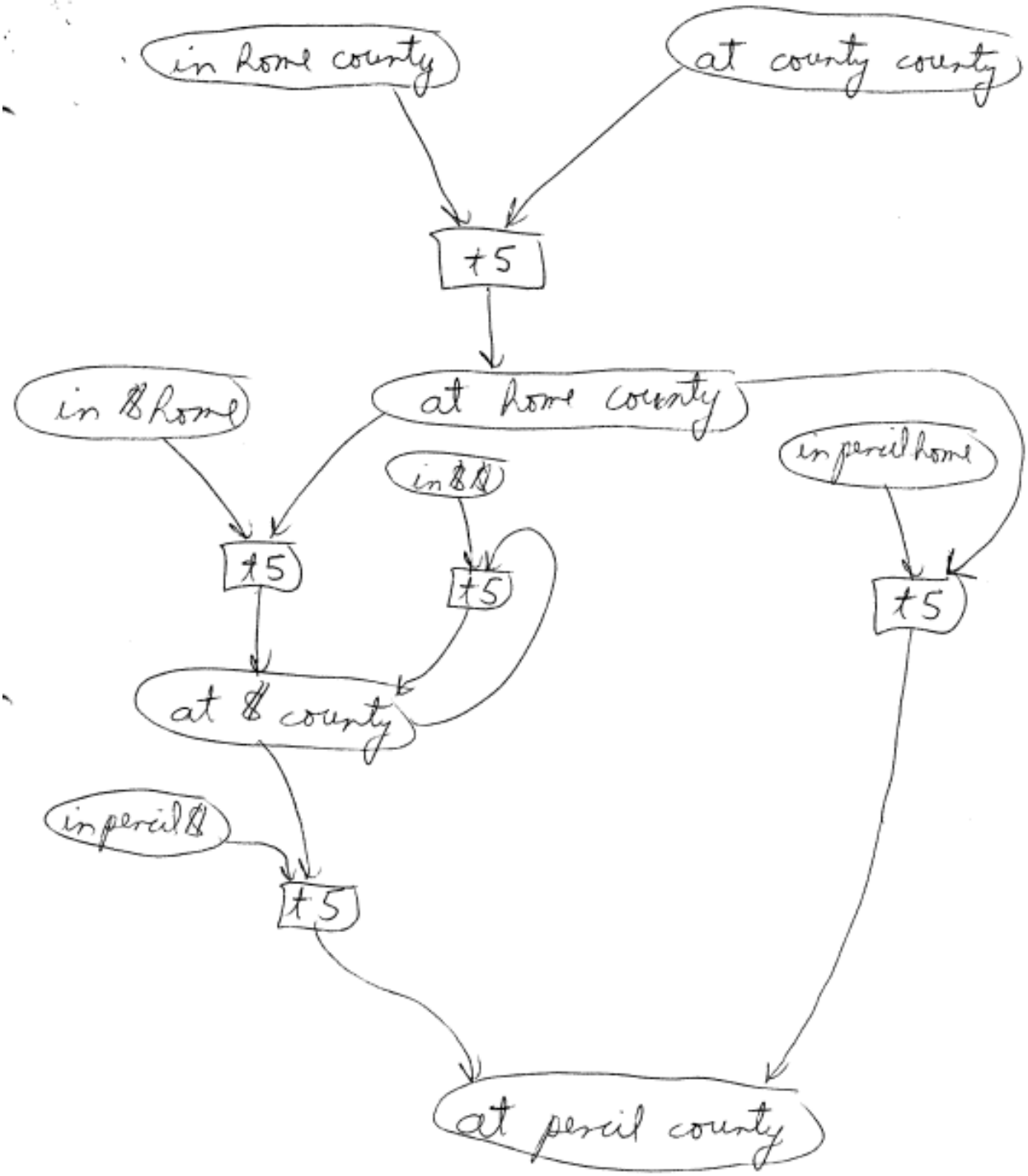


Schematized Goal Tree

figure 54

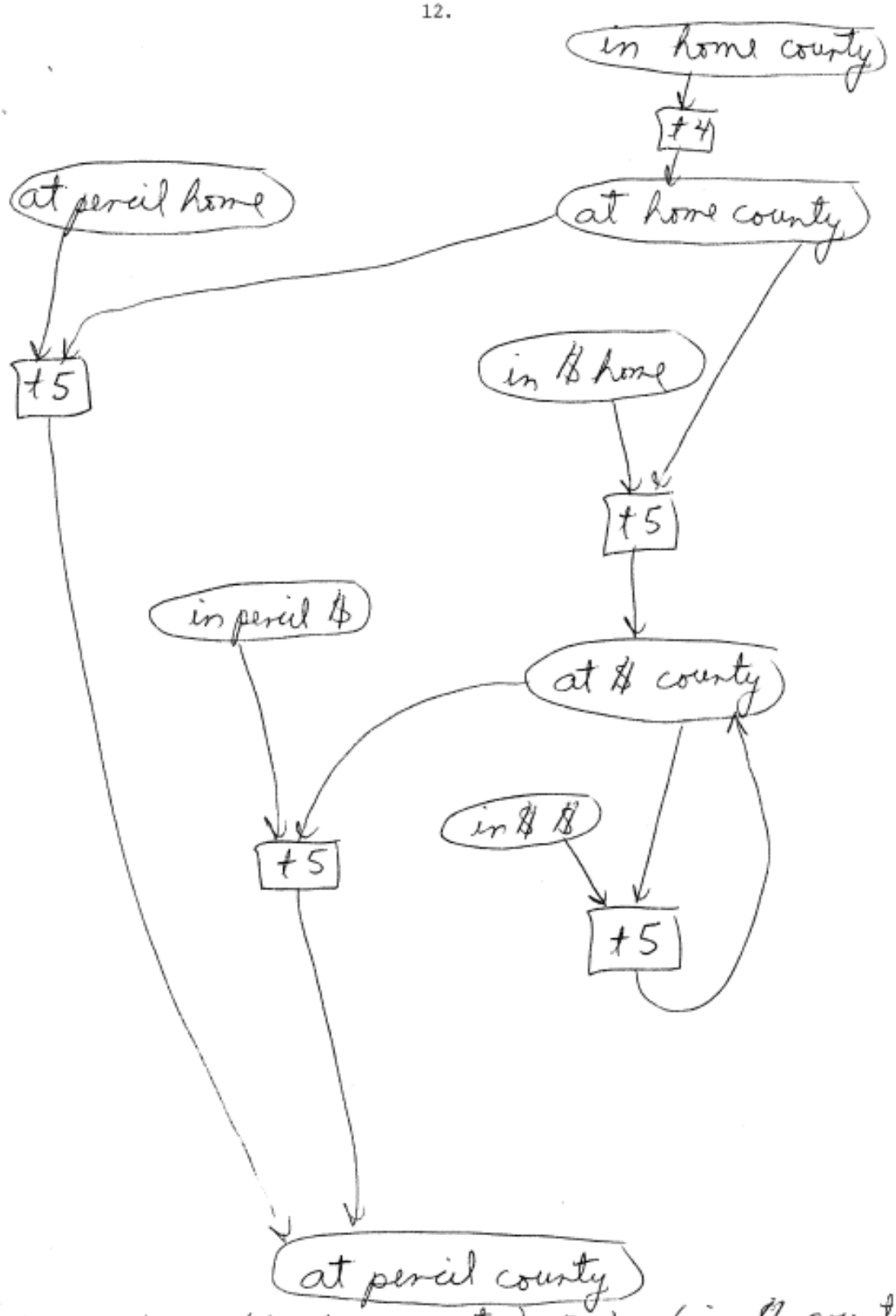
Pushing (in pencil desk) through t4 leads us to a dead end. But pushing through t5 leads us to two expressions that match (at pencil county). See figure s1. We want to reverse the process and chain back from the consequent to our island. A good heuristic is to pick the expression that best matches the consequent as the first one to chain back from. If we violate the heuristic and chain back from (at $ $) we obtain figure s2. As the reader can see chaining back from (at $ $) does not take us all the way back to our island (in pencil desk). Therefore we must abandon the above plan. Chaining back from (at pencil $), we obtain figure s3. The theorem prover realizes that in order to complete the proof of (at pencil county) it must prove the lemma (at desk county).

We would like to consider the example from a slightly different viewpoint. Suppose that we transpose chaining forward from our island and the chaining backward from the goal. Chaining backward from (at pencil county) we obtain a schematised goal tree (figure s4). Every proof of (at pencil county) has a graphical homomorphic image in the schematized goal tree. The theorem prover can obtain all proofs by grinding away inside the schematized goal tree. Suppose that our island is (in home county). We note that (in home county) matches both (in $ $) and (in $ county). Unfortunately the plan (figure s5) generated by plugging (in home county) into (in $ $) cannot be fulfilled since (at county county) is unproveable. Even if the theorem prover



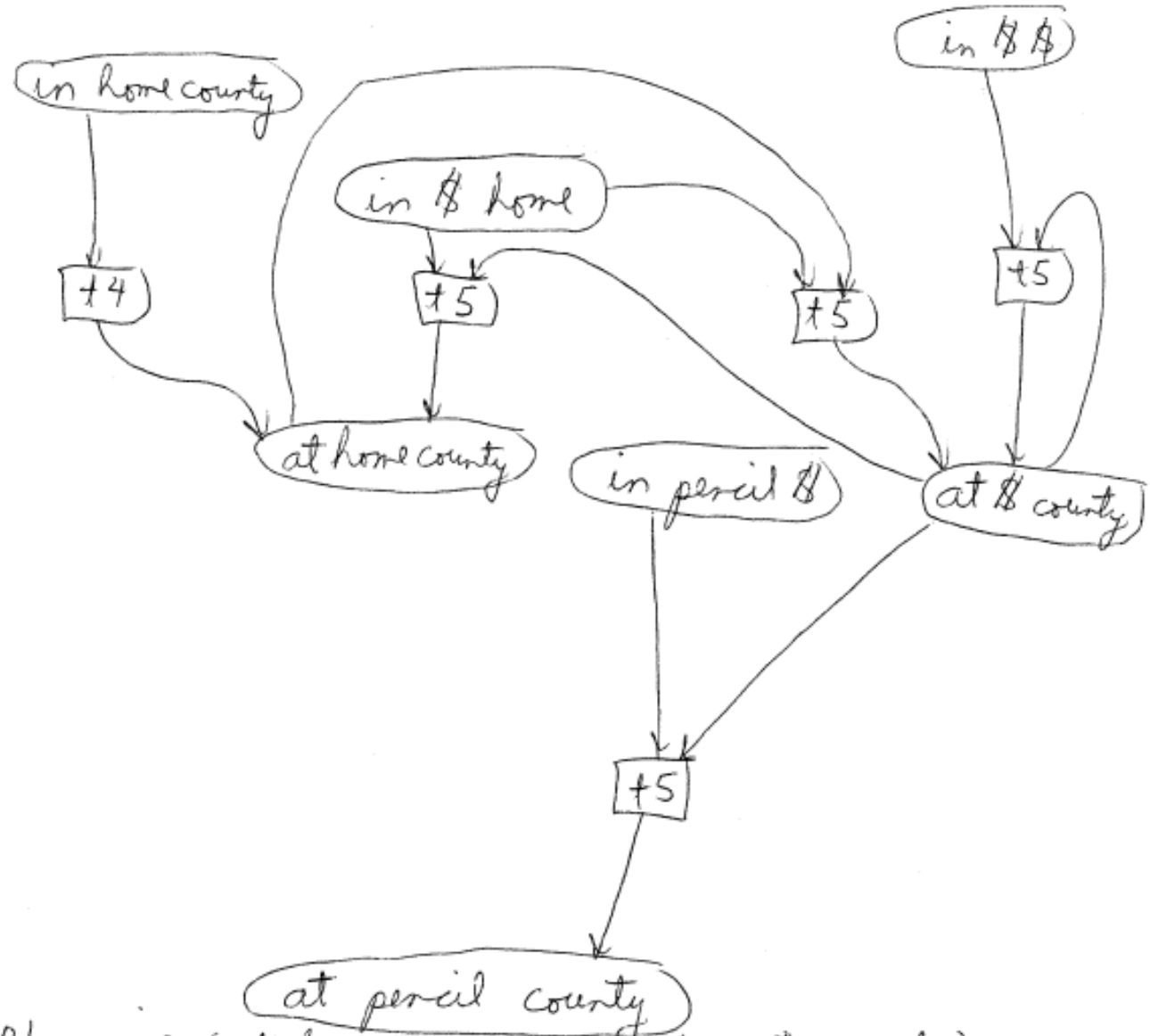
Plugging (in Rome county) Into (in ß ß)

figure 55

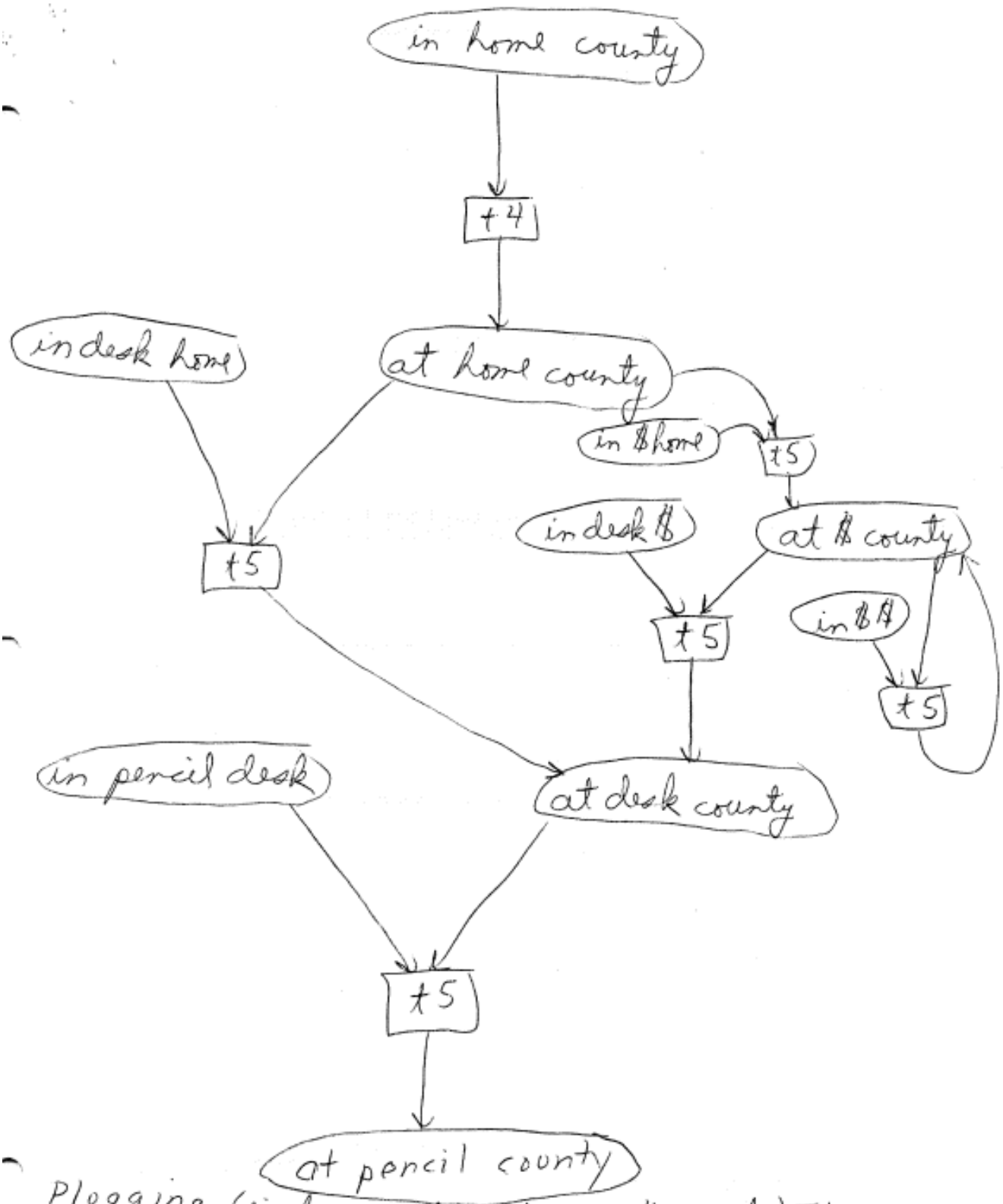


Plogging (in home county) into (in $ county)

figure 56



Plugging (at home county) into (at $ county)
figure 57



Plogging (in home county) into (in $ county) Then
Plogging (in pencil desk) into (in pencil $)
figure 58

had a model it would be misled if the model attached the interpretation true to (at county county). Plugging (in home county) into (in $ county) leads to a plan (figure s6) that can be fulfilled. Note that two applications of therorem t5 are needed to prove the theorem. Of course if we are given two islands which are both relevant to the same proof we can often form a much better plan with both than with either one alone.

Each time the theorem prover plugs an island into a schematized goal tree, it should check to see if the resulting plan is circular. Thus (in pencil desk) cannot be plugged into (in $ $) in figure s7 since it forces (at $ county) to become (at pencil county) which makes the plan circular. Therefore (in pencil desk) can only be plugged into (in pencil $). The resulting graph is figure s8.

In each case we have in fact constructed a whole schema of plans. Within any one schema we can have many degrees of freedom.

1. There can be more than one route from the island(s) to the consequent.

2. The constants represented by the $ must be found.

3. Some loops in the plan can be unwound.

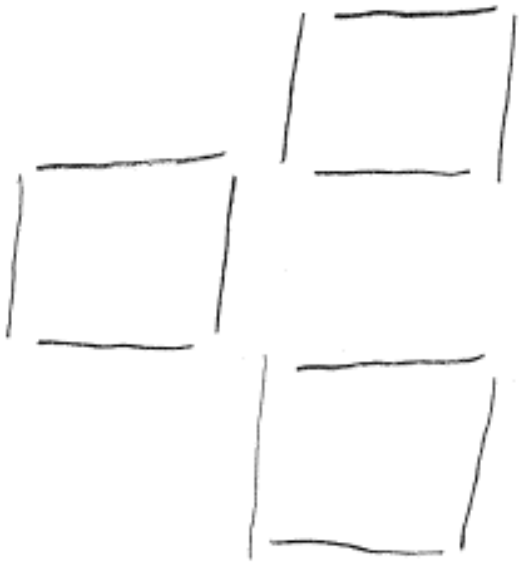
Flexiblility in planning is a mixed blessing. A good deal of rigidity is necessary in order to proceed

straightforwardly from the plan to a rigorous proof. On the other hand our plans must be somewhat adaptable so that we are not stymied by the first difficulty. The planning mechanism of G.P.S. illustrates the usefullness of controlled flexibility in planning.

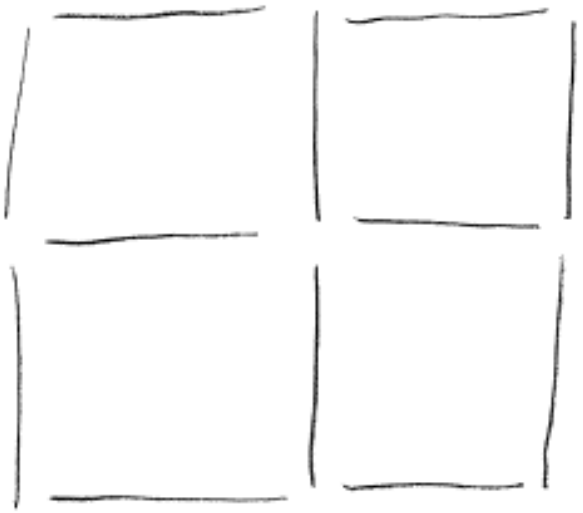
We would now like to turn to the question of where to obtain islands for planning. One source is pure syntactic analysis. The theorem prover should carefully examine the consequences of the expression to be proved. The consequences of the consequent heuristic is try to find islands among the consequences of the consequent of the theorem to be proved. For example in the problem to prove (at pencil county) suppose we had the additional theorem

t7: (implies (not (in pencil desk)) (not (at pencil county)))

Using the consequencies of the consequent heuristic the theorem prover should find the island (in pencil desk). How can one transform the stick figure below into one with four squares by moving at most three sticks?



In this case our consequence is that we have four squares. But if we are to make four squares from only twelve sticks then at least four sticks must each have two squares in common. Thus our island is the figure below



Another syntactic trick is to try to trace the constants back to their source in the data base of the problem. Recall that in our example that the constant pencil could only have come from the expression (in pencil desk). Hypotheses usually make excellent candidates as islands. Of course, there are exceptions to the rule. For example, if the theorem prover were asked to prove (implies (in home Texas) (at pencil county)) it would try to use (in home Texas) as an island only to find that (in home Texas) is actually irrelevant.

Models, special knowledge, and analogies are often sources of islands. Suppose the theorem prover knew how to inscribe a circle in a given triangle. It could use an

analogous method to inscribe a sphere in a given tetrahedron. Models often contain links that are useful in the construction of islands. The links make some relevant expressions in the data base of the problem more accessible to the theorem prover.

Suppose the theorem prover had the following information about cubes a, b c, and d.

v1: (directlyabove d a)

v2: (restingon d c)

v3: (restingon c b)

v4: (forall (x y) (equivalent (supporting x y) (restingon y x)))

v5: (forall (x y) (equivalent (dierectlyabove x y) (directlybelow y x)))

v6: (forall (x y z) (implies (and (directlybelow y z) (directlybelow x y)) (directly below x z)))

v7: (forall (x y) (implies (supporting x y) (directlybelow x y)))

A model for the situation might look like figure s9.

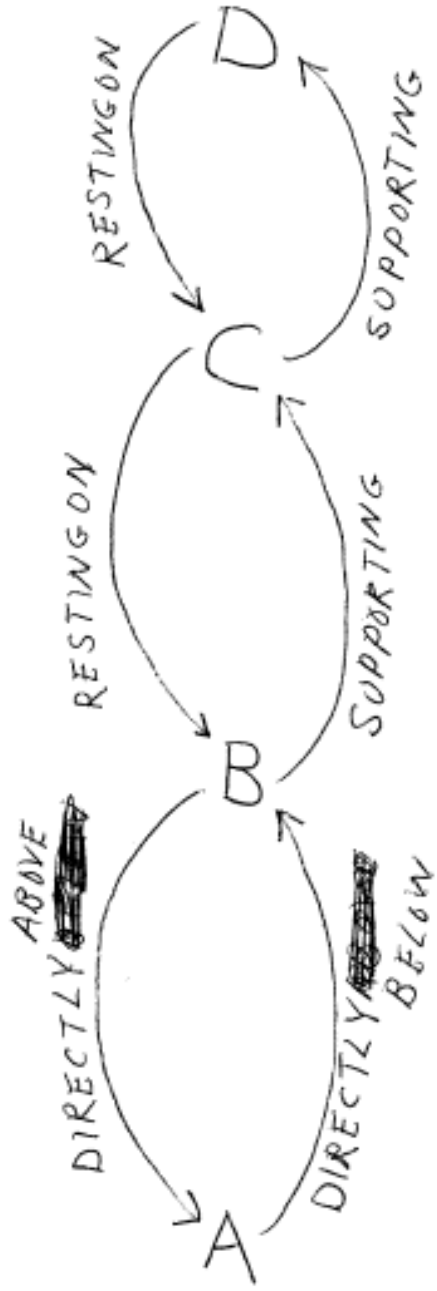

D
RESTING ON
SUPPORTING
C
RESTING ON
SUPPORTING
B
ABOVE
DIRECTLY
DIRECTLY
BELOW
A

Figure 59.

Clearly the constant b is more accessible from the constant a in the model than from the theorems. Thus models are important in planning not only for pruning those expressions that are false in the interpretation of the model but also for the connections they provide which suggest new plans. A model keeps the state of the system in a continually updated canonical form. All other information about the system is derived from the canonical form.

Theorem provers should do more thinking about their problems before blindly beginning a huge tree search. An intelligent problem solver would try to find key nodes (such as (at $ county) in figure s4) in the schematized goal tree. Of course most of our analysis for statements as islands goes dually for theorems. Also the theorem prover should do a series-parallel loop analysis to get a better idea of the size of the problem that it faces and to learn more about its structure. I have been trying to develop a theory of the decomposition of planning nets analogous to the existing series-parallel decomposition theory for sequential machines. There are clues in the difference between the island(s) and the consequent that can save the theorem prover a great deal of work. For example MATCHLESS can quite easily be made to find all the loop structures of the form of (figure s11) in the schematized goal tree. After a proving a statement that satifies A, one would often like to go back through theorem t1 to see if anyting else can picked up almost for free.

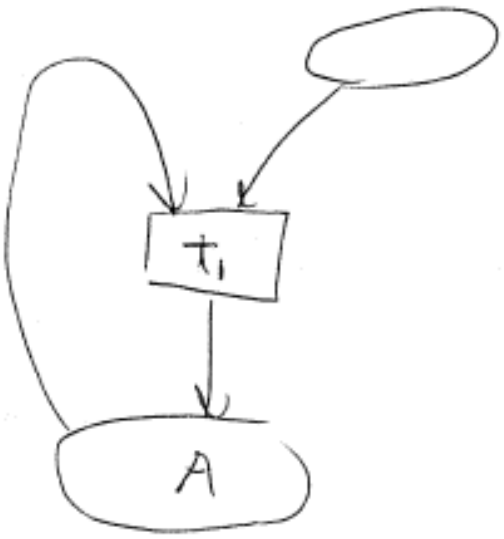


figure 511

The distinguishing characteristic of SCHEMATIZE is that the relationship between the islands and the statement to be proved is shown. To have some islands with no indication as to how they should be used simply limits the size of the tree to be searched; the theorem prover must still do heuristic tree searching. In a stronger system such as SCHEMATIZE the executive of the theorem prover does not search the goal tree but rather follows along a plan using the indicated relations between islands. Of course SCHEMATISE works only for problems with a simple logical structure. It can be generalized slighlty to more powerful deductive procedures using natural deduction but the analysis become exremely complicated. Sometimes one can usefully analyze part of a more complicated problem using SCHEMATISE.

MATCHLESS

MATCHLESS is a pattern matching program witten in LISP. It is most succinctly described as a cross between SNOBOL and CONVERT. The most important respect in which MATCHLESS differs form CONVERT is that MATCHLESS doesn't have a dictionary. Bindings for MATCHLESS variables are kept on the LISP push down list. Consequently MATCHLESS can very conveniently be used within LISP progs. I think that it is an important principle in language design that the user should be allowed to write in the level that he thinks is most suitable for his task. The various levels of code should all be compatible with one another so that it is possible to use them all in a single body of code. For example it would be useful to be able to write LAP code in the middle of LISP progs. Furthermore low level code should not have to run slower simply in order to preserve the ability for users to write at a higher level.

A variable v in MATCHLESS can match three types of objects:

(1) atoms (the first character of the name of v must be a $).

(2) S-expressions (the first character of the name of v must be a =)

(3) fragments of lists (the first character of the name of v must be an *)

Any type of variable may be in any one of four modes: VAR (for variable), CONST (for constant), GENERIC, or COMP (for computed). The idea for the VAR and CONST modes comes form SNOBOL; the GENERIC mode from CONVERT and AXLE. The COMP mode allows one to introduce new modes and types into MATCHLESS. All of MATCHLESS could be written in the COMP mode. The following atoms are given special interpretations by MATCHLESS.

(1) $ will match any atom.

(2) = will match any S-expression.

(3) * will match any fragment of a list including the null fragment.

Furthermore MATCHLESS has the standard Boolean operators on patterns. For example if pat is a pattern then (=NOT= pat) will match any S-expression that doesn't match pat. Similarly =and= may be used to require that an S-expression match a number of patterns and =or= may be used to require that an S-expression match any one of a number of patterns. The idea for Boolean operations on patterns comes from CONVERT and AMBIT.

There is a MATCHLESS pattern associated with each MATCHLESS variable. The idea of associated patterns for pattern variables comes from CONVERT and AXLE. Executing

(var v w) or (generic v w) will put v in the VAR or GENERIC mode respectively with the associate pattern w. A variable in the var or generic mode has no value; it matches according to its associated pattern. When a variable in the VAR mode matches it takes the value of the expression that it matches and its mode changes to CONST. A GENERIC variable differs form a VAR variable in that it is not modified when it matches. A variable in the CONST mode matches according to its value. Executing (msetq v r) will put v in the CONST mode with value r. The default mode is VAR with an innocuous associated pattern.

Suppose $B, =A, and *C are all in the VAR mode. If MATCHLESS attempts to match the pattern (* $B A =A $B *C) against (K H5 G A (K) H H A (L) M H I A (M) I B C) then the variables mode will be changed to CONST and they will have the following values.

$B: I

=A: (M)

*C: -B C-

The dashes in the value of *C are meant to indicate that the value of *C is a fragment of a list.

PLANNER

SCHEMATISE can be conceived to be searching through a planning space. Indeed any LISP program can be conceived as a tree searching program since the computation itself is a tree. Thus it would be useful to have a powerful tree searching language in which we could write SCHEMATISE and other tree searching theorem proving procedures. PLANNER is a theorem prover which hopefully represents still another step toward such a general tree searching language. PLANNER gets its name from the fact that it was originally created as a language in which a robot could formulate plans about its possible actions. The theorems of PLANNER are executable data structures. An example of such a theorem is TRANSITIVE (below) which expresses a necessary condition for a transitive relation.

```
(TRANSITIVE  (THPROG ($PREDICATE $X $Y $Z)
    (CONSEQUENT ($X $PREDICATE $Z))
    (PROVED (TRANSITIVE $PRED))
    (PROVEABLE ($X $PREDICATE $Y))
    (PROVEABLE ($Y $PREDICATE $Z))
    (FINISHED)
) )
```

All theorems for PLANNER are imperatives. Imperative theorems have certain advantages and disadvantages compared to declarative theorems. The chief advantage of imperative theorems is that they can be extremely powerful. Arbitrary LISP computations are permitted in imperative theorems. Thus each theorem can contain the heuristics for its own use. For example a theorem might recommend certain theorems for some subproblem that it creates. I would be very grateful to any reader of this paper who sends me examples of types of heuristics that cannot naturally be incorporated into theorems for PLANNER.

The central function of PLANNER is thprog which is like prog except that it treats PLANNER functions in a special way. When a failure occurs thprog backs up to the last executed PLANNER function and tries again on a different branch of the tree that it is searching. Some of the functions of PLANNER are

(proveable a switch listoftheorems): If the pattern a is proveable without erasing anything then execute the next statement; otherwise fail. If switch is FIRST (ONLY) then listoftheorems are the first (only) theorems that will be used to try to chain back from a.

(goal a switch listoftheorems): Goal is like proveable except that erasures are permitted.

(consequent a): a is declared to be the consequent of the theorem. Whenever a goal is created which matches the

pattern a, the theorem will be used to try to chain backwards from the goal.

(antecedent a): a is declared to be the antecedent of the theorem. Whenever a statement is asserted which matches the pattern a, the theorem will be used to try chain forwards from the statement.

(assert a b switch listoftheorems): Record a as proved with reason b. If switch is FIRST (ONLY) then listoftheorems are the first (only) theorems that will be used to try to chain forward from a.

(finished): Indicates the end of the theorem.

(threturn a): Returns a as the value of the thprog in which it appears. Threturn has not yet been implemented.

(eraseable a): Record that a is eraseable.

(uneraseable a): Record that a is uneraseable.

(erase a switch listoftheorems): If a is eraseable then erase it and the statements that depend on it; otherwise fail. If switch is FIRST (ONLY) then use listoftheorems as the first (only) theorems to try to chain forward from the fact that the pattern a is being erased.

(fail): Causes a failure.

(thfail): Causes the theorem to fail.

(hypothesize a): Assert a with the reason that it is a hypothesis.

(discharge): Discharge the last made hypothesis.

(thset a b): Thset is like set except that the old value of a is remembered so that it can be restored in case of failure.

(thrplaca a b) and (thrplacd a b): These functions are like rplaca and rplacd respectively except that the old value of a is remembered so that it can be restored in case of failure. The functions thset, thrplaca, and thrplacd are very useful for manipulating models.

(thgo a): Thgo is like go except that in case of failure control is returned to the place from where the transfer was made.

The erase feature of PLANNER enables it to quite easily manipulate models. The following two theorems enable PLANNER to build a tower three cubes high.

```
(TOWER (THPROG ($BLOCK1 $BLOCK2 $BLOCK3 )
    (CONSEQUENT (CAN TOWERAT HERE))
    (GOAL ($BLOCK1 AT HERE LEVEL 0))
    (UNERASEABLE ($BLOCK1 AT HERE LEVEL 0))
    (GOAL ($BLOCK2 AT LPLACE LEVEL 1))
    (UNERASEABLE ($BLOCK2 AT HERE LEVEL 1))
    (GOAL ($BLOCK3 AT HERE LEVEL 2))
```

```
     (ERASEABLE ($BLOCK2 AT HERE LEVEL 0))
     (ERASEABLE ($BLOCK1 AT HERE LEVEL 1 )))
     (FINISHED) )      )

 (MOVE  (THPROG
       ($BLOCK  $OTHERBLOCK  $NEWLEVEL  $NEWPLACE  $OLDLEVEL
$OLDPLACE)
     (CONSEQUENT ($BLOCK AT $NEWPLACE LEVEL $NEWLEVEL))
     (PROVED (BLOCK $BLOCK))
     (ERASE ($BLOCK AT $OLDPLACE LEVEL $OLDLEVEL))
     (UNPROVED ($OTHERBLOCK AT $OLDPLACE LEVEL (=EVAL= (PLUS
$OLDLEVEL 1)))
     (STATEFINSHED) )      )

 (BLOCK BLOCK1)
 (BLOCK BLOCK2)
 (BLOCK BLOCK3)
 (BLOCK1 AT P1 LEVEL 0)
 (BLOCK2 AT P1 LEVEL 1)
 (BLOCK3 AT P2 LEVEL 0)
```

The theorem MOVE has a condition that it will not move any cube that has another cube setting on top of it. Professer Papert pointed out that one could write the theorem MOVE differently. Instead of prohibiting the removal of the cube, it could bring the cubes above the removed one crashing down! Thus we could have written

```
 (MOVE (THPROG
         ($BLOCK $OTHERBLOCK $NEWLEVEL  $NEWPLACE  $OLDLEVEL
$OLDPLACE)
     (CONSEQUENT ($BLOCK AT $NEWPLACE LEVEL $NEWLEVEL))
     (PROVED (BLOCK $BLOCK))
     (ERASE ($BLOCK AT $OLDPLACE LEVEL $OLDLEVEL))
     (THCOND  (  (PROVED  ($OTHERBLOCK  AT  $OLDPLACE  LEVEL
(=EVAL= (PLUS $OLDLEVEL 1)))
                 (ASSERT (FALLING $OTHERBLOCK)))
     )
     (STATEFINISHED) )   )

 (FALLING (THPROG ($BLOCK, $OTHERBLOCK, $LEVEL, $PLACE)
     (ANTECEDENT (FALLING $BLOCK))
     (ERASE (FALLING $BLOCK))
     (ERASE ($BLOCK AT $PLACE LEVEL $LEVEL))
     (ASSERT ($BLOCK AT $PLACE LEVEL (=EVAL=  (MINUS  $LEVEL
1))))
     (THCOND ( (PROVED ($OTHERBLOCK AT $PLACE LEVEL  (=EVAL=
(PLUS $LEVEL 1)))
               (ASSERT (FALLING $OTHERBLOCK)))
     )
     (FINISHED) ) )
```

To form a plan as to how to build a tower does not complete the job. The tower must still be constructed! One way in which that might be accomplished is as follows. After the planning phase is completed the system should

write out PLANNER programs that actually carry out the job. For example the theorem TOWER above might create something like the following theorem.

```
(MAKETOWER (THPROG ()
    (GRASPBLOCK (AT P1 LEVEL 1))
    (MOVETO (HERE LEVEL 0))
    (RELEASEBLOCK)
    (CHECK (CUBE AT HERE LEVEL 0))
    (GRASPBLOCK (AT P1 LEVEL 0))
    (MOVETO (HERE LEVEL 1))
    (RELEASEBLOCK)
    (CHECK (CUBE AT HERE LEVEL 1))
    (GRASPBLOCK (AT P2 LEVEL 0))
    (MOVETO (HERE LEVEL 2))
    (RELEASEBLOCK)
    (CHECK (CUBE AT HERE LEVEL 2))
    (FINISHED) )     )
```

Of course if we wanted to build a tower ten blocks tall instead of only three, then both MAKETOWER and tower would have loops in them.

One important problem in actually having the robot carry out an operation is that of unexpected input from the sense organs. For example a cube might slip out of its hand. At that point the robot must do some fast calculation to determine what has happened and what it should do next. It would be interesting to know whether the controlled back

up feature of PLANNER would help or hinder solutions to this problem.

PLANNER can do simple proofs in a decidable subtheory of the quantificational calculus. The following theorems enable it to prove the transitivity of set theoretic inclusion.

```
(NEC     (THPROG ($A $B)
    (THIMPLIES     (THPROG ($X)
                        (THIMPLIES     (ELEMENT $X $A)
                                       (ELEMENT $X $B)
                        )
                   )
                   (SUBSET $A $B)
    ) )     )

(SUFF     (THPROG ($A $B)
    (THIMPLIES     (SUBSET $A $B)
                   (THPROG ($X)
                        (THIMPLIES     (ELEMENT $X $A)
                                       (ELEMENT $X $B)
                        )
                   )
                   (SUBSET $A $B)
    ) )     )
```

Note that THPROG serves as the universal quantifier. In our notation transitivity of set theoretic inclusion is expressed by

```
(TRANSET (THPROG ($A $B $C)
    (THIMPLIES (THAND (SUBSET $A $B)
                      (SUBSET $B $C)
               )
               (SUBSET $A $C)
    )
)    )
```

Within the quantificational calculus there are essentially two ways to prove a theorem of the form (R x) where x is in the VAR mode. The first method is to assume (thprog (x) (thnot (R x))) and then attempt to derive a contradiction. The other method is to derive as many consequences as possible from (R x)--say (R1 x), (R2 x), ... , (Rn x)-and then attempt to cons up an object that will satisfy the consequences. For example we might write the following theorems to construct the midpoint of a line segment:

```
(CONSTRUCT (THPROG ($P1 $P2 $P3)
    (CONSEQUENT (EQUAL (DISTANCE $P1 $P3) (DISTANCE $P3
$P2)))
    (PROVED (POINT $P1 ))
    (PROVED (POINT $P2))
    (THCOND ( (CONSTP $P3) (THFAIL)) )
    (MSET (QUOTE $P3) (QUOTE (MIDPOINT $P1 $P2)))
    (ASSERT (POINT $P3))
    (FINISHED)
```

```
)     )
(POINT A)
(POINT B)
```

CONSTP is a predicate which tests to see if its argument is in the CONST mode. If asked to prove (EQUAL (DISTANCE A $Y) (DISTANCE $Y B)) where $Y is in the VAR mode, PLANNER would give $Y the value (MIDPOINT A B). Theorems such as CONSTRUCT can cause PLANNER to go into a loop if they are not used with care. PLANNER can do simple proofs by contradiction if it is told the statement to be contradicted. For example if we wanted to prove (not a) by contradiction on c we could say (hypotheise a) (proveable c) (proveable (not c)) (discharge). In this way we could prove (not a) from the theorem (implies a (not a)). Of course in order to do general proofs in the quantificational calculus we would have to write considerably more complicated theorems. My present goal in this area is to make PLANNER prove that the limit of the sum of two sequences is the sum of the limits of the sequences.

I would like to thank Professor Minsky for suggesting that I investigate the problem of getting PLANNER to swap the contents of two machine addresses on an IBM 7094. Suppose that a is in address1, b is in address2, and randomness is in address3. The following theorems will swap the contents of address1 and address2.

```
(SWAP (THPROG
    ($ADD1 $ADD2 $A $B $ADDA $ADDB)
    (CONSEQUENT (SWAP $ADD1 $ADD2))
    (PROVED (CONTAINS $ADD1 $A))
    (PROVED (CONTAINS $ADD2 $B))
    (PROVED (CONTAINS $OTHERADD =))
    (GOAL (MOVE $ADD1 $OTHERADD))
    (PROVED (CONTAINS $ADDB $B))
    (GOAL (MOVE $ADDB $ADD1))
    (PROVED (CONTAINS $ADDA $A))
    (GOAL (MOVE $ADDA $ADD2))
    (FINISHED)
) ) )

(CONTAINS ADDRESS1 a)
(CONTAINS ADDRESS2 b)
(CONTAINS ADDRESS3 0)

(MOVE (THPROG
    ($ADD1 $ADD2 $CONTENTS)
    (CONSEQUENT (MOVE $ADD1 $ADD2))
    (PROVED (CONTAINS $ADD1 $CONTENTS))
    (ERASE (CONTAINS $ADD2 =))
    (ASSERT (CONTAINS $ADD2 $CONTENTS))
    (FINISHED) ) )
```

If we were to examine the protocol produced by PLANNER as it solves the problem, we would find that it goes up a couple of blind alleys before it finally finds the correct

solution. We would like to try to write theorems for PLANNER that would enable it to analyze simple protocols and try to make changes in the theorems that produced the protocols in order to make the solutions more straightforward. Thus the fourth statement of the theorem SWAP might be changed from (PROVED (CONTAINS $OTHERADD =)) TO (PROVED (CONTAINS (=AND= $OTHERADD (=NOT= (=OR= $ADD1 $ADD2))) =)). Similarly we could try to get PLANNER to generalize the theorem TOWER (above) by replacing HERE throughout by $PLACE. Then PLANNER would be able to build a tower anywhere instead of only at the place HERE. Inserting and deleting statements from theorems are other simple manipulations which are feasible for PLANNER. PLANNER will have to do a gigantic inefficient search in order to learn to do some simple class of tasks. Subsequently, it should be able to proceed straightforwardly for the class of problems. If it should hit a snag, PLANNER should attempt to modify the procedure that has worked in the past in order to get around the difficulty. Protocol analysis provides important clues to show where and how theorems should be modified. Attempting to make changes in theorems without the help of protocols from those theorems appears to be an untractable problem. It has been suggested that the protocol analyzing theorems be used to try to improve themselves. Unfortunately it will be quite a long time before bootstrapping in this way will be fruitful. At the present time it is necessary to write very complicated

theorems to analyze even the most trivial protocols. Furthermore any increase in the power of the protocol analysis theorems would seem to call for even more complex theorems.

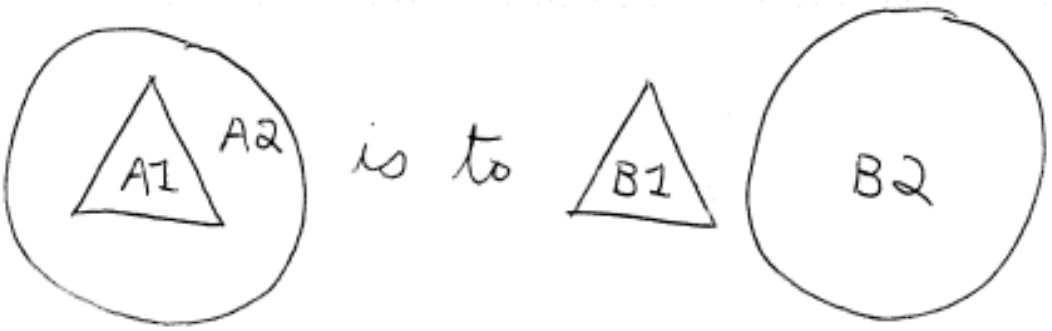
The following two theorems illustrate how PLANNER can be made to solve geometric analogy problems such as those solved by Evans's program.

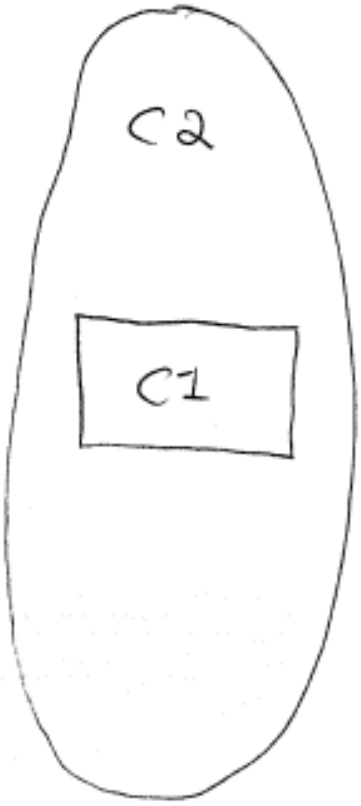
```
(ANALOGOUSBA (THPROG ($B $A $TYPEB)
     (CONSEQUENT (ANALOGOUSBA $B $A))
     (PROVED (TYPE $B $TYPEB))
     (PROVED (TYPE $A $TYPEB))
     (FINISHED) )     )

(ANALOGOUSAC (THPROG ($A $C $PREDICATE *ARGSA1 *ARGSA2
*ARGSC1 *ARGSC2)
     (CONSEQUENT (ANALOGOUSAC $A $C))
     (THCOND   (          (PROVED (TESTANALOGOUSAC $A $C))
(FINISHED)) )
     (PROVED (OBJECT $A))
     (PROVED (OBJECT $C))
     (THCOND ( (PROVED (TESTANALOGOUSAC $A (=NOT= $C ) ))
(FAIL)) )
     (ASSERT (TESTANALOGOUSAC $A $C))
     (PROVED (RELATION $PREDICATE))
     (PROVED ($PREDICATE *ARGSA1 $A *ARGSA2)))
     (PROVED ($PREDICATE *ARGSC1 $C *ARGSC2))
```

```
     (PROVEABLE (CORANALOGOUSAC (*ARGSA1) (*ARGSC1)))
     (PROVEABLE (CORANALOGOUSAC (*ARGSA2) (*ARGSC2)))
     (FINISHED) ) )

(CORANALOGOUSAC  (THPROG ($A *A $C *C)
     (CONSEQUENT (CORANALOGOUSAC ($A *A) ($C *C)))
     (THCOND ( (PROVED (TESTANALOGOUSAC $A $C)) (THGO REST))
)
     (PROVEABLE (ANALOGOUSAC $A $C))
REST (PROVEABLE (CORANALOGOUSAC (*A) (*C)))
     (FINISHED)
) )

(TYPE TRIANGLE)
(TYPE RECTANGLE)
(TYPE CIRCLE)
(RELATION INSIDE)
(RELATION LEFTOF)
(COBJECT C1)
(TYPE C1 RECTANGLE)
(COBJECT C2)
(TYPE C2 ELLIPSE)
(AOBJECT A1)
(TYPE A1 TRIANGLE)
(AOBJECT A2)
(TYPE A2 CIRCLE)
(BOBJECT B1)
```

```
(TYPE B1 TRIANGLE)
(ROBJECT B2)
(TYPE B2 CIRCLE)
(CORANALOGOUSAC () ())
(INSIDE A1 A2)
(INSIDE C1 C2)
(LEFTOF B1 B2)
```

If you ask PLANNER to prove (ANALOGOUSAC A1 $X) where $X is in the VAR mode, then it will give $X the value C1. Using the above theorems as models the reader should be able to write the other theorems necessary to solve the analogy below





as

is to ?

Analogy and generalization play a very important role in theorem proving. For example the proofs of the uniqueness of the identity element, the zero element, and inverses in semi-groups are closely related. The definitions are

(equivalent (identity e) (forall (a) (implies (equal (times a e) (times e a) a))))

(equivalent (zero z) (forall (a) (implies (equal (times a z) (times z a) a))))

(implies (identity e) (equivalent (inverse b1 b) (equal (times b1 b) (times b b1) e))) If we suppose that e', z', and b1' are respectively identity, zero, and inverse elements then the solutions are

(equal e (times e' e) e')

(equal z (times z' z) z')

(equal a1 (times a1' a a1) a1')

Thus the general form of the solution is (equal w string w') where string algebraicly simplifies to w and w'. It would be a straightforward to write theorems for PLANNER that would enable the program to recognize the very particular above kind of analogy. But this is not the way in which we would ultimately like to approach the problems of analogy and generalization in theorem proving. What we need is a way to search analogy space and generalization space in a manner similar to the way in which SCHEMATIZE searches island space for plans. A reasonable approach toward accomplishing this would be to construct a heirarchy

of theorems around the the various predicates for types of analogy such as those introduced above.

The ability of PLANNER to write theorems which it can later execute potentially gives it very great powers of generalization and abstraction. We do not yet know how to effectively utilyze this power. At present the principal use of this ability has been to aid in the implementation of a decidable subtheory of the quantificational calculus. Perhaps it will prove fruitful for planning programs to create PLANNER theorems such as MAKETOWER above in order for the robot to carry out the plan. Also the ability to create programs enables the computer to put several small procedures together in order to accomplish a larger task. Usually some fudging is necessary between the smaller procedures in order to make them work together.

PLANNER was first developed as a simple planning mechanism and model manipulator for a robot. Its structure permits the use of macro steps in constructing plans of action for the robot. There might be some confusion as the purpose of PLANNER as a general theorem prover. PLANNER is not intended to show that a computer theorem prover does not need knowledge and expertise in the domain in which it works. To the contrary PLANNER should be used as a meta-theorem prover with as much knowledge as possible built into its theorems. In the last decade many programmers have constructed heuristic tree searching problem solvers. Many of the the problem solvers have detailed knowledge of their

intended domains built into their structue. I would like to investigate how much of the knowledge of the programs can be naturally incorporated into theorems for PLANNER.

A great deal remains to be done to remove some of the limitations associated with general theorem provers. Much of our trouble stems from the fact that we do not yet even have the beginnings of a mathematically rigorous theory of tree searching. Nor do we have much theoretical understanding of the process of proving theorems. Independent of more general theoretical considerations, general theorem provers have important limitations relative to more special purpose problem solvers. In any given problem area, PLANNER will certainly prove to be less efficient than a special purpose problem solver in the problem area for which the latter was designed. However, part of this inefficiency can be avoided by writing very special theorems for PLANNER. Also, PLANNER will become much more efficient when I have completed a compiler for it. PLANNER is sufficiently powerful that I can write the compiler in PLANNER and then bootstrap it. The argument of loss of efficiency in general theorem provers will lose much of its force once we have the hardware to search the branches of the goal tree in parallel.

The problem of representation for theorem proving systems has attracted increasing attention from researchers in recent years. The problem has the following aspects:

1. Input representation of problem
2. Internal representaion of problem
3. Ability of the theorem prover to change representation

Inadequate ability to represent concepts necessary for the solution of problems can severely affect the performance of a problem solver. For example the fixed inflexible input format of GPS-2-6 greatly limits the generality of the problems that it can solve. In PLANNER there is no distinction between input representation and internal representation. Furthermore theorems for PLANNER make no distinction between a problem to change the representaion of a problem and any other kind of problem. For example it is straightforward to write theorems for PLANNER which will enable it to recognize that the games tic-tac-toe, number scrabble, and jam are isomorphic. The chief difficulty with the theorems to recognize the isomorphism is that they are applicable only to this very specific problem. What we need to do is to write a system of theorems that can recognize isomorphism within a very wide class of games.

Almost all the problems that have been solved by general theorem provers thus far have been rather trivial. The chief reason for this has been that heretofore enough core storage to attempt more ambitious problems has not been available. Thus the following question has arisen: Can the techniques which have been developed to handle toy problems be extended and generalized? I think that we will see the

question answered in the affirmative in the next few years as more core memory becomes available to users. I am presently working on two problems to show how theorems for PLANNER can solve harder problems. The first problem is to write SCHEMATISE in PLANNER. The second is to write theorems which solve the problem of how to put Soma Cubes together to make fairly arbitrary block figures. Both problems require a large number of theorems arranged in interlocking hierarchies. Hopefully, this work will eventually become part of a thesis.

BIBLIOGRAPHY

Black, F., 1964. A Deductive Question Answering System, doctoral dissertation, Harvard University, Cambridge, Mass.

Christensen, C., 1964. AMBIT: A Programming Language for Algebriaic Symbol Manipulation, AFCRL-64-909.

Cohen, K., and Wegstein J., 1965. AXLE: An Axiomatic Language for String Transformations, Communications of the Association for Computing Machinery, November, 1965.

Feigenbaum, E. A., and Feldman, J. (eds.), 1963. Computers and Thought, New York , N. Y.: McGraw-Hill.

Fenichel, R. R., and Moses, J., 1966. A New Version of CTSS LISP, Artificial Intelligence Memo 93, Massachusetts Institute of Technology (Project MAC), Cambridge, Mass.

Guzman, A. and McIntosh, H. V., Convert, Communications of the Associaton for Computing Machinery, August, 1966.

Kalenich, W. A. (ed.), 1965. Information Processing 1965, International Federation for Information Processing, New York City, Washington D.D.: Spartan.

McCarthy, J. 1959. Programs with common sense, Proceedings of the Symposium on Mechanisation of Thought Processes, National Physical Laboratory, Teddington, England, London: H. M. Stationery Office, pp. 75-84.

McCarthy, J., et al., 1962. LISP 1.5 Programmer's Manual,Cambridge, Mass.

Minsky, M. L., 1961. Steps toward artificial intelligence, in Computers and Thought, pp. 406-450.

Newell, A., and Ernst, G., 1965. The search for generality, in Information Processing 1965, pp. 17-24.

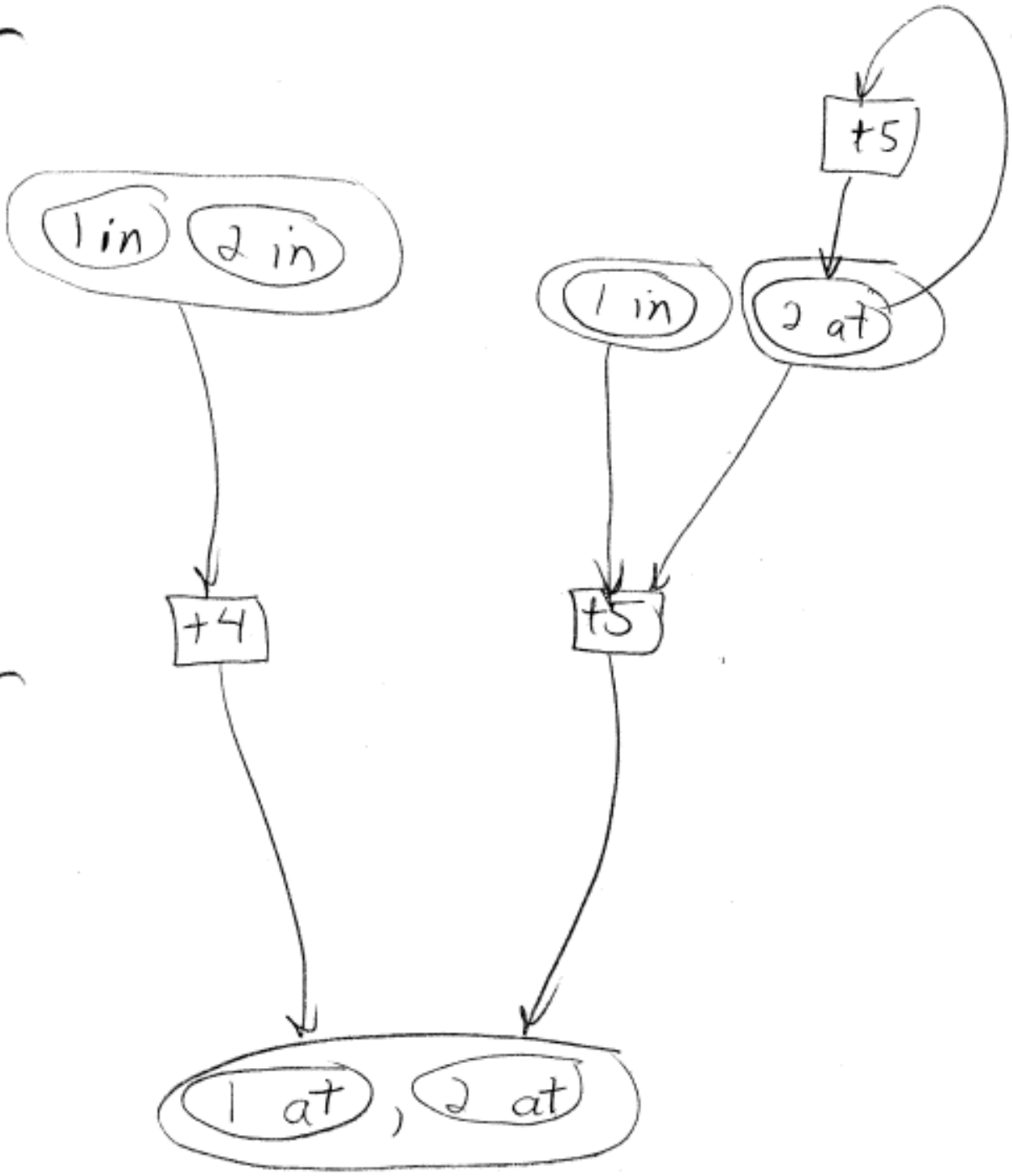
Newell, A., Shaw, J. C., and Simon, H. A., 1959. Report on a general problem-solving program, Proceedings of the International Conference on Information Processing, Paris: UNESCO House, pp. 256-264.

Slagle, J., 1965. Experiments with a deductive question-answering program, Communications of the Associaton for Computing Machinery, December, 8:792-798.

The Philosophy of the System

SCHEMATISE was designed and MATCHLESS was programmed in the fall term of 1966-1967. SCHEMATISE is intended to provide global methods for the computer to use to prove theorems in logic systems of a very simple type. One idea is to use invariants of the theorem net in order to more quickly find proofs. For example one invariant of a theorem net is its place holder goal tree. Let (n pred) be the nth place holder of the predicate pred. The place holder goal tree for the predicate at in the theorem net in the first chapter is shown in the figure below. The advantage of using invariants of the theorem net for this purpose is that although they can be used in many problems they only have to computed once for each theorem net.

Another global method in SCHEMATISE is the use of schematized goal trees. The philosophy behind the idea of schematized goal trees is very simple. We interpret subgraphs of the theorem net as subtheories. Quotient graphs are syntactic planning theories. The space used by G.P.S. in which connectives are left out of propositional formulas is an example of a syntactic planning theory. Homomorphisms from one space into another are interpreted as analogies between theories. Thus there is always an analogy between a theory and one of its syntactic planning theories.



Place Holder Goal Tree of Predicate "at"

A trivial kind of homomorphism is exemplified by the symmetry of variables recognized by Gelenter's geometry theorem prover. These homomorphisms are trivial because they map predicates identically onto themselves. Also the use of these homomorphisms is covered by a constructive metatheorem. A metatheorem is interpreted as a theorem about certain classes of theorem nets. Constructive metatheorems represent the best of all possible worlds for the theorem prover. Metatheorems save the theorem prover the work of doing basically the same kind of proof over each time it comes to a problem for which it has a relevant metatheorem. Working in a theory over the theorem nets (that is metatheoretically), the theorem prover can obtain insights and results that would otherwise be impossible. A more interesting example of a homomorphism is given by the analogous proofs by diagonalization of the incompleteness theorem and of the existence of nonrecursive predicates. Suppose there is a homomorphism h1 from theorem net T1 into theorem net T2 and a homomorphism h2 from theorem net T2 into theorem net T3. One can sometimes obtain a proof of a proposition Q in T3 by first obtaining a schematized proof P1 in T1, extending h1(P1) to a proof P2 in T2, and then finally extending h2(p2) to be a proof of Q. Thus we have an elementary theory of multi-pass theorem provers. It is also useful to look at the homomorphic images fo the given theorem net since they induce quotients back on the given space.

PLANNER is designed to be an extendable flexible system for manipulating an internal model of a dynamic world. In its world arbitrary objects can be created and destroyed at will. The world of PLANNER contrasts strongly with the static world of the quantificational calculus. In order to obtain greater flexiblility and generality, PLANNER does not maintain a rigid barriar between the imperatives (theorems) that take some action in its world and complicated declaratives (theorems) that state some fact about its world. PLANNER often uses the same S-expression as an imperative and as a declarative depending on the context. There is an analogous situation in mathmematics where the quantificational calculus (a theory of declaratives) is strong enough to represent the recursive functions (the imperatives). Furthermore one can regard a wff in the quantificational calculus with free variables as a predicate. Assume that for each procedure P we have a declarative D (called the <u>intention</u> of P) that says how P is intended to be used. Now D does not determine P. If there is one way to realize a given intention then there are infinitely many ways to realize it. Also it may happen than an intention cannot be realized at all. For example the following intention is realized by no procedure P: P solves the halting problem. Using intentions the theorem prover can proceed to debug a procedure P as follows: given an argument a such that P(a) violates its intention check the subprocedures of P to see if any of them violated their

intention while P(a) was being computed. If so then proceed to find the bug in the sub-procedure of P. If not then there is a bug in P. Check the intentions of the statements of P to find the statement of P whose intention was violated. For example in the problem to exchange the contents of two locations the intention that $OTHERADD be different from both $ADD1 and $ADD2 was violated. As illustrated by the example of building a tower, the standard way in which PLANNER trys to find a procedure which satisfies a given intention is to first figure out an algorithm which satisfies the intention and then write a procedure which uses the algorithm. Given the knowledge of how to cause an imperative to act, PLANNER can sometimes find out what the imperative means. For example in Rosser's axiom system for the propositional calculus the meaning of the general procedure by which you can obtain a proof of (implies a b) from the proof of b starting from a is the Deduction Theorem for Rosser's axiomatization. Conversely we can try to analyze how a theorem can be applied. Classical theorem provers keep their declarative knowledge rigidily separated from their imperative knowledge.

From the point of view of program graphs, there are essentially two kinds of heuristics in PLANNER theorems. The first kind (called selectors) choose which branch of the problem tree to search next. The second kind (called rejectors) determine when to stop working on a branch of the problem tree. At a high level selectors should use

planning, analogies, and links in models to help make plausible choices. Rejectors can try to prove the negation of a proposed goal or try to find a counterexample to it. Due to overriding considerations of efficiency, a practical theorem prover must be as close to a decision procedure as possible. Rejectors and selectors supplement each other. For example in elementary plane geometry, counterexamples from diagrams make such a good rejection heuristic that very good selectors are not needed. On the other hand since Samuels's checker player can choose the correct move at each ply over half the time, it is less dependent on its static evaluator as a rejector. One can view rejectors as special cases of selectors in which the null choice is made.

PLANNER has many of the features that are desireable for a language in which to write a domain independent planner for proving theorems. A domain independent planner is a program that operates by accepting as input knowledge of a domain D (including both declaratives and imperatives) and a theorem T in the domain D and outputting a plan for the proof of T. The justification for a domain independent planner is the thesis that there is a large body of techniques and strategies common to mathematical domains such as logic, algebra, set theory, and analysis. The ultimate goal is for a domain independent planner to be able to read a book written in a formal language on some mathematical domain D and then be able to constuct plans for the proof of theorems in D. Admittedly it is not an easy

task to construct a domain independent planner for even a trivial class of domains. Up to the present time the pattern of research in artificial intelligence has been to construct programs that straightforwardly search goal trees at the level of the given axioms and rules of inference of some specialized domain. Although the previous work has made valuable contributions, it does not automatically produce a domain independent planner. There are many difficult problems in the construction of a domain independent planner that have not arisen in the work on specialized domains. Thus work on domain independent planner has its own independent right to existence as a problem in artificial intelligence.

In a limited sense PLANNER is "aware" of what it is doing when it is trying to prove some result since each theorem has complete access to the subgoals and procedures that are being used to try to obtain the result. For it to be "aware" of what it is doing in a deeper sense, PLANNER must be able to easily translate from intentions to procedures which realize those intentions and from procedures to the meaning of those procedures. In particular it must be able to do so for the intentions and procedures that constitute the theorem prover.

Added to A.I. 137 (July 1967)
57 in March 1968

PLANNER is a language for proving theorems and manipulating models. It can prove theorems in a typed second order quantificational calculus with erasing. The declarative-imperative duality of its theory is basic to understanding how the language works. The language itself is domain independent. We may think of the language as divided into two parts: bookeeping and default conditions.

The default conditions constitute the domain independent knowledge of the theorem prover. Suppose the goal of the theorem prover is to prove (implies a b). In a default condition PLANNER will usually assume a and try to prove b. If that doesn't work, then it will usually assume (not b) and try to prove (not a). The default conditions allow PLANNER to do a reasonable amount of search. Thus the default conditions would not assume (not (implies a b)) and attempt to derive a contradiction. Of course this does not mean that we connot do resolution in PLANNER. It simply will not be done by default. Another example of a default condition is that (not (not c)) simplifies to c. The last example is interesting because it points out an interesting parallel between theorem proving and algebraic manipulation. The two fields face similar problems on the issues of simplification, equivalence of expressions, intermediate expression bulge, and man-machine interaction. Of course in

any particular case, the theorems need not allow PLANNER to lapse into its default conditions.

All domain dependent knowldege is contained in the theorems (imperative and declarative). The theorems can do arbitrary amounts of computation. They have the full recursive power of PLANNER available to help them with their problems. For example an assertion can recommend theorems to be used to attempt to draw conclusions from what is asserted. Messages can be sent to lower level theorems to try to get them to produce better answers to some question that a higher level theorem is asking. Consider the following functions,

```
(defprop among
 (thlambda (l) (thprog ()
 start (thcond ((null l) (fail)))
     (setq l (cdr l))
     (failp (failto start))
     (threturn (car l)) )) expr)

(defprop foo (thlambda (b a) (thcond
                    ((greaterp a b) a)
                    (t (fail))))
 expr)
```

Thus the value of (foo 5 (among (quote (2 4 6)))) is 6. If the predicate failp detects a failure then it executes its argument. The function failto causes failure to the tag

which is its argument. The function among successively takes on the elements of its argument 1; i.e. 2 then 4 then 6. Clearly the computation would be faster if foo were to assert that "among" should produce a value greater than 5 and "among" were to take this advice. It will sometimes happen that the heuristics for a problem are very good and that the proof proceeds smoothly until almost the very end. At that point the program gets stuck and lapses into default conditions to try to push through the proof. On the other hand the program might grope for a while trying to get started and than latch onto a theorem that knows how to polish off the problem in a lengthly but fool proof computation. PLANNER is designed for use where one has a great number of procedures (theorems) that might be of use in solving some problem along with a general plan for the solution of the problem. The language helps to select procedures to refine the plan and to sequence through these procedures in a flexible way in case everything doesn't go exactly according to the plan. The present default conditions can and will be extended but I don't think that by themselves they will ever be able to prove deep mathematical theorems. Nor do I believe that computers can solve difficult problems where their domain dependent knowledge is limited to a finite-state difference table of connections between goals and methods.

A straight forward compilation of programs written in PLANNER produces very inefficient code. For the language to

practical, the compiler must be able to cut corners while at the same time producing correct code. Thus the compiler should be able to prove that in each case it has indeed compiled a program correctly. McCarthy and his students have used the alternative approach of trying to prove the compiler correct once and for all. The PLANNER compiler should be extendable in that it should accept new heuristics at any time and that programs to be compiled should be albe to make recommendations as to how they should be compiled. Thus we are led to consider the equivalence problem for programming languages. In the following the s-expressions enclosed in <> are first order intentions. Intentions are defined to be predicates that should be true when control passes through them. Variables that begin with a single "<" are first order intention variables. Intentions are allowed to reference but not to modify program variables. For example the following programs are all equivalent.

```
(defprop fact1
 (lambda (n) (cond ((lessp n 1) 1)
                   (t (<t (lambda (<b>) (equal <b> (factorial
n))))>
                         (times n (fact1 (sub1 n))))))
 expr)
 (lap fact1a subr)
 ((<block> ((<n> (addr 1)))
      (push p 1)
```

```
      (movei 2 (quote 1))
      (call 2 (function *less))
      (jumpe 1 a)
      (movei 1 (quote 1))
      (jrst 0 b)
      a
      (move 1 0 p)
      (call 1 (function sub1))
      (call 1 (function fact1a))
      (<(equal (addr 1) (factorial (sub1 <n>)))>)
      (move 2 0 p)
      (call 2 (function *times))
      b
      (sub p (% 0 0 1 1))
)
(popj p)
()
```

In fact1a (addr 1) is the address of accumulator 1 and <block> declares the intention variable <n> to be equal to (addr 1).

```
(defprop fact2
 (lambda (n) (<block> ((<b> n)) (prog (a)
     (< (advice (prooftype of (equal a (factorial <b>)) is
numerical-induction on n from 0 to infinity))>)
     (setq a 1)
```

```
again (cond ((lessp n 1)
            ((<(equal a (factorial <b>))>) (return a))))
    (setq a (times n a))
    (setq n (sub1 n))
    (go again) ))) expr)

(lap fact2a subr)
((<block> ((<n> (addr 1 (<p> (addr p)))
          (push p (% 0 0 (quote 1)))
     (push p 1)
     a
     (movei 2 (quote 1))
     (move 1 0 p)
     (call 2 (function *less))
     (jumpe 1 b)
     (move 1 -1 p)
     ((<(equal (addr 1) (factorial <n>))>))
     (jrst 0 c)
     b
     (move 2 -1 p)
     (move 1 0 p)
     (call 2 (function *times))
     (movem 1 -1 p)
     (move 1 0 p)
     (call 1 (function sub1))
     (movem 1 0 p)
     (jrst 0 a)
```

```
      c
      (sub p (% 0 0 2 2))
     ((<(eq <p> (addr p))>)
     ((<(no-side-effects)>)
     ((<(no-free-variables)>))
 )
 (popj p)
 ()
```

The function <block> declares that the intention variable <b> should have the value of n and returns its second argument as value.

```
 (definition (iff (equal (fact3 n) a)
     (or (and (lessp n 1) (equal a 1))
          (and (not (lessp n 1)) (equal a (times n (fact3
(sub1 n)))))))))
 (defprop fact4
   (lambda (n) (prog (a b)
          (setq a 1)
          (cond ((lessp n 1) (return a)))
          (setq b 1)
  again   (cond     ((equal b n)
                    (return a))
           )
          (setq b (add1 b))
          (setq a (times b a))
```

```
        (go again)
)) expr)
```

Fact1 and fact2 are schematically equivalent to each other, i.e. they are equivalent solely on the basis of the meaning of the LISP logical primitives. Proving that fact4 is equivalent to the others is slightly nontrivial since it involves the associativity and commutativity of multiplication. The quantity included in the <> is the intention for that piece of code. The first s-expression in the <> is the intention before the function is entered; the second, if present, is the intention for the value. Note that in this connection that it is convenient to treat the quantificational calculus as a programming language.

The intentions of a program are useful for a variety of purposes. Included in the protocols of a function, they provide valuable information on how to combine the protocols in functional abstraction. They are very useful in attempting to prove that a function satisfies its overall intention or to prove that two functions are equivalent. If PLANNER constructs a function which supposedly builds towers, it needs to be able to prove that the towers that the function can construct are all stable. Without out any ultimate loss in efficiency, a function can be run in a debugging mode in which the processor checks the intentions of the function as it executes the function to make sure that they are satisfied. At the present time computers are much worse than mathematicians at proving that two programs

are equivalent. Even if computers were to greatly improve, there are grave difficulties in using them as a practical solution to the debugging problem. It is difficult to find reasonable intentions that adequately characterize the output in terms of the input for many functions (for example system programs and very highly recursive programs). Nevertheless, work on this approach to debugging problem is independently valuable. It increases the power of the computer to prove facts about functions. Only experience can teach us how severe the practical difficulties are toward this approach to the debugging problem.