WORKING PAPER 71

# ADVICE ON THE FAST-PACED WORLD OF ELECTRONICS

by

DREW MCDERMOTT

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

May 1974

Abstract

This paper is a reprint of a sketch of an electronic-circuit-designing program, submitted as a Ph.D. proposal. It describes the electronic design problem with respect to the classic trade-off between expertise and generality. The essence of the proposal is to approach the electronics domain indirectly, by writing an "advice-taking" program (in McCarthy's sense) which can be told about electronics, including heuristic knowledge about the use of specific electronics expertise. The core of this advice taker is a deductive program capable of deducing what its strategies should be.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advance Research Projects Agency of the Department of Defense and monitored by the office of Naval Research under Contract Number N00014-70-A-0362-0005.

Table of Contents

## I. Introduction

It is by now a trite observation, but it is worth mentioning, that Artificial Intelligence research tends to have two contradictory goals: the production of intelligent expertise, and the capture of the general nature of intelligence. The exercise of subduing a particular intellectual domain by formalization has now been done for several domains: assembly-line balancing, checkers, symbolic integration, and mass-spectrogram interpretation, to mention a few. Critics of our assumptions can point out with much justification that doing such an exercise fails to capture the notion of intelligence-in-general.

Some of us retaliate by claiming that there is no such notion, if we examine matters closely. This claim has merit when used to refute the criticism that doing a formalization exercise removes the domain involved from intelligence altogether, which is absurd. (Hubert Dreyfus <1972> is a good source of such inane criticism.)

But most workers have recognized that there is an problem here. People have spent much effort on the problem of generality since AI began. Projects that were or are concerned with achieving it include GPS <Newell and Simon, 1963>, QA3 <Green, 1969> and other theorem provers, MULTIPLE <Slagle and Bursky, 1968>, and STRIPS <Fikes and Nilsson, 1971>. I think all of them

can be characterized by their attitude that "problem solving in general" is a domain for which there ought to be an expertise. The programs that result often use obviously useful techniques, such as goal achievement and deduction, in an extremely elegant fashion.

But none of them are experts in anything. All of them can try to work on any problem that can be expressed to them, and they are designed so that a wide class of problems can be expressed. But on interesting problems they fail. Evidently "problem solving in general" is not a coherent domain apart from a lot of special knowledge in each field. This is only fair; human beings are not great in a new field without a lot of practice.

But let us say GPS were modified to take hints. Each time a decision as to which goal to work on came up, it would pose this as a goal. Then hints, in the form of suggestions for choosing goals, would allow it to make more directed choices of how to proceed.

This in itself is just not going to work, although the principle is good enough for me to adopt it among others in Sect. II. It is too magical on the face of it. Closer consideration reveals that the goal-chooser is going to have to tie together a lot of very disparate information on any interesting problem. This information is going to be scattered around a large data base, so the system has traded a bush of simple goals for a goal

of great complexity.

Sussman <1973a> has pointed out the problem here. The expert has a procedure for solving problems in its domain. The hints do not get in the way precisely because the problems of what to do next, at some level, has been taken care of--just follow the procedure. The procedure knows what deductions to make, what hints to use, what goals to propose or attack.

The fact that procedures are so good at this has led some to embrace the Procedural Utopia view of AI, a more subtle restatement of the claim that there is no "intelligence-in-general." The utopian view is that hints and knowledge are just procedures; that all we should be allowed to tell a program is more program; that any natural-language or other declarative input must be converted to imperatives. (Cf. <Winograd, 1971>.) Then, since universality is a simple property for a programming language to possess, generality must some day follow. To be sure, it helps to have sophisticated languages and programs, with features like pattern-directed procedure calls <Hewitt, 1972>, multiple co-routines <Sussman and McDermott, 1972>, "multiple-body interrupts" <Brown, 1973>, etc. There may also be some simple (even declarative) information about procedures, such as recommendations <Hewitt, 1972> or frames <Minsky, in progress>. But the procedure must be king.

It has been my experience that this approach will not succeed; that is, not at creating an intelligence. There are

two reasons for this: (1) a program can only apply the knowledge it has in ways that were foreseen; and (2) a new program that is added to the system is not at all guaranteed to work correctly with its old programs. These two problems are related.

Allan Newell <1962> has the best example of (1). He cites the difficulty of getting a chess program to answer a slightly different question from the one it was written to answer. The only controls you have on the behavior of such a program are the inputs you wrote it to look at. If they aren't the right ones, the program can't help you. The problem is clear: some knowledge went into writing the program, but the program is not that knowledge; it is at best a knower. The phrase "procedural embedding of knowledge" <Hewitt, 1972> is misleading. (Cf. <Hayes, 1974>.)

As an example of (2), I can draw on my own TOPLE program for finding plausible interpretations of declarative sentences. <McDermott, 1974> When interpreting ambiguous sentences about the spatial relation of two objects, it tried to visualize the desired relation. If told "object1 is in object2," it tries to believe object1 is smaller than object2. It if knows nothing else, it reasons from its knowledge of what kinds of objects 1 and 2 are; it doesn't like to hear that a table is in a ball.

Having decided that object1 is smaller than object2, it next inquires whether object2 is enclosed (e.g., a covered box as opposed to a pen). If it is, object1 must be less tall than

object2 as well. In isolation, tallness is like overall size (a floor lamp is not likely to be in a music box), so TOPLE looks for facts about individuals, then reasons from categories. But, since often dimensions are considered together, it would be nice if the system knew, "All other things being equal, the bigger object is likely to be the taller." This might save it some tedious computation about what kinds of objects it is thinking about.

But there is no way to tell the dimension-comparison routine this fact, short of rewriting it. That routine doesn't look for advice; it just computes. And there is no way to add an entirely new routine so that it can communicate with the old one. The old one is not taking calls.

One example like this may not be convincing. You can always imagine a better, "more modular" way to have written the old routine in the first place. But experience with computer programs leads me to believe that all modularity has a finite lifetime. You leave some slots and interfaces where later changes can occur, but they get used up. Eventually a change straddles two interfaces, or demands the creation of a new slot. Besides, as Sussman <1973a> has argued, there is a conflict between modularity and efficiency. Procedures work precisely because they ignore a lot of information that might be relevant.

In this proposal, I will trace the outline of a system that is both expert and potentially general, that is willing to ignore

probable irrelevancies, but able to make them relevant when they are pointed out by a human. That is, I have in mind a system like McCarthy's Advice Taker <McCarthy, 1968>, a program that is not necessarily brilliant, but is able to use any hint that is given to it. I will use this name for want of something better. Since even simple schemes often sound good on paper, I intend to describe a concrete domain -- proposal of electronic-circuit designs. The advice taker's expertise is intended to be sound enough so that the circuit proposer may interface with other parts of the electronic circuit-design project being organized by Gerry Sussman <Sussman, 1973b; Brown, 1973>. In what follows, I first outline the structure of a deductive advice taker, then describe its application to electronic-circuit design, and finally attempt to meet the many objections that I anticipate.

## II. Ideas for the Structure of an Advice Taker

To start as down-to-earth as possible, let us imagine that a decent problem solver must always think of itself as executing at least one procedure, at some level. This procedure is associated with one or more goals the system believes that it will help accomplish. In keeping with standard terminology, I call the execution of such a procedure a process.

The current process may be the result of the interpretation of a program, a "plan," or some other such structure, by an appropriate interpretive procedure, or it may be an abstraction, such as a commitment to review every step with respect to a global strategy. An important example is the control on a deductive process represented by a declarative belief. For example, one way (IMPLIES A B) might direct a deduction is to instruct a suitable interpreter to propose (PROVE A) as a subgoal of (PROVE B).

There may be more than one active process; that is, a process that "thinks" it is now running; whose interpreter has a next step in mind. If one process knows the language in which another process's interpreter thinks, the first can alter what the second thinks, or "put ideas in its head," i.e., advise it. It should also be able to give it control.

To facilitate such communication, it seems right to make each interpreter (except for the machine, or Conniver interpreter)

speak the same declarative language. Each interpreter keeps a data base in this language that describes the state of interpretation of its procedure. (I am indebted for many ideas such as this to Scott Fahlman.) Fetching from such a data base may be as general as a deduction, or it may be as narrow and efficient as desired.

Thus the current interpreter of a robot executing a plan might keep a data base describing the current piece of the plan being worked on, and the current state of the world the plan was affecting. "Careful-mode" execution of the plan could be enforced by adding to the data base a statement, "Between plan steps, the next step is: check for agreement between the expected world model (as given by comments on the plan) and the actual result of each step." (This won't work unless the interpreter reads such messages. Having a common declarative language is not sufficient to make messages understandable. Further conventions will be necessary.)

The example may be extended. In testing a plan, the robot will want to run it without affecting the real world. This requires a new interpreter which simulates the effect of each real-world action (or just the old one with still other statements that redefine the meaning of action).

Finally, a plan is merely an object of some sort, which may be altered or inspected as well as executed. We might as well have it be denoted by an expression in our declarative language.

So far, except for a certain type of uniformity, this system is no different from generalized control-structure <Bobrow and Wegbreit, 1973>, ACTOR's <Hewitt, 1973>, Fahlman's frame system <1974>, or any of a number of systems whose good ideas I am trying to use.

The really important notion is that deduction is to be a smoothly-integrated part of the system. A deduction is just the operation of a (deductive) interpreter, using a data base as a program. The deductive goal is to prove something from this data base. (For technical reasons, the real goal may be to deduce a contradiction from the negation of the given goal.) There is a conceptually separate data base which records the current state of the deduction, and holds advice to the deductive process, just as for other processes. (This will be described in detail shortly.)

Deductions are to occur, at least in principle, whenever a process (including an interpreter) wishes to know something. For example, a robot might ask, did execution of that step result in what I expected? A circuit designer might ask, does the proposed coupling circuit between two stages load the second stage? A chess-move explorer, upon noticing a problem with a move, might ask, is the reason this move failed likely to be a problem for other moves as well? (...If so, amend the plausible-move generator with a procedure that reminds every subsequent move explorer to check this problem first.)

The last example is an instance of an important kind of knowledge: how to exploit "bugs." <Sussman, 1973a>

The emphasis on deduction is likely to worry the many people who feel deduction is a trap for naive AI researchers. (If the following list of advantages is not convincing, section IV, countering objections, may help. A helpful remark to make is that I am using "deduction" in a very broad sense, analogous to that in the phrase "probable deduction." <Hume, 1955> I allow reasoning processes such as induction, and "buggy reasoning" (e.g., from general statements that are not true in every case), in addition to necessary deduction. I could give this process a new name (like "duction"), but I think a revival of the original word in Sherlock Holmes' sense <Doyle, 1967> is in order.)

Here is my list of advantages of deduction:

(1) (i) It is the obvious way to utilize information expressed in declarative form.
(ii) It is a good way to discover relations among previously unrelated data.
(iii) Deduction is a good framework for study of the modularity problem.
(iv) A deductive system may look at much the same information in simple or complicated ways.
(v) Communication conventions between deductive processes are easy to establish.

In detail:

(i) I take it for granted that some information is declarative.

(ii) The pattern matching used in deduction is able to express looser relations between variables in formulas being

combined than PLANNER-type matching, because matching semantics is more general than assignment semantics. If discovery of new relations seems to be too grandiose a goal, I will settle for the ability to state new lemmas to the advice taker in a general language.

(iii) A deductive framework is a good one for the study of the modularity problem, because we can always take at least the step of adding a new piece of information. Then, in many cases, knowledge of how to use that datum may be accumulated in the form of comments on it.

Let me explain this further. Remember that a data base may be regarded as a "program" to a deducer. It is not perfectly ordered, because more than one rule of inference may be applicable at each stage. If it is too expensive just to try them all, the deducer may use comments about them to decide which rule to try (this is another deduction); or it may set up a longer-range plan (e.g., "this rule, then that one on its output") in the form of an abstract process which the deducer is aware that it is responsible for; or it may try one rule or path, and use its knowledge of typical bugs associated with that technique to make a better guess if it fails. (Cf. <Sussman, 1973a>.) At least while studying the modularity problem, this framework helps to organize the different kinds of information that are around.

In the long run, it is most probably true that more powerful

methods of organization (analogous to compilation) will be necessary. See sect. IV.C.

(iv) A deductive framework enables a system to look at much the same information in simple or complicated ways. Sometimes a deduction is a brute-force "filtering" of simple assertions. Other times, it is a sophisticated problem-solving process, each step of which must be considered carefully. This is not always a domain-dependent variation; the same domain may require both kinds of deductions. For example, in solving a design problem, a goal of the form "does there exist a circuit that does so-and-so" may arise in two different ways: when a complete such circuit, completely thought out, is desired; and when planning is being done for a higher-level circuit, and quick verification that a proposed module is feasible is required. In the latter case, elaborate testing and criticism of suggested plans is out of place. I believe that, in a deduction, these phases may be skipped by alteration of appropriate inference rules. It would obviously be harder to alter the behavior of an expert procedure.

(v) It is easier to establish communication conventions between parts of a deductive process, or between a deductive process and some other type, because the objects manipulated in a deduction are so natural: deductive goals, rules of inference, "brother goals" (e.g., in (PROVE (AND A B)), A and B are brothers).

Conventions are important for the reason mentioned before:

you can't send a computer a message unless it is listening. Two arbitrary processes cannot communicate unless each knows the data and control structures of the other. Any Conniver program has all the potential power of the system I have been describing, but it is usually illusory.

For example, a chess program might formulate its search for a move as a deduction

(AND (PLAUSIBLE-MOVE ?M) (ACHIEVE ?M (CURRENT-GOALS)))

(This might seem a little strange for a deductive formulation as opposed to a procedural one. This point will be addressed below, Sect. IV.A.) Let us say a move M is proposed in deducing (PLAUSIBLE-MOVE ?M). The brother goal then might discover a problem with the move, a threat by the opponent that refutes it. A plausible course of action might be to find a move M' to meet the threat, and instruct the deductive interpreter (via appropriate assertions) to return to the previous goal, now formulated as (PLAUSIBLE-MOVE M'). Because a deductive process is so transparently organized, it is easy to test its state and alter it.

This last point is not completely obvious without some comments on the concrete structure of the deductive interpreter. Any process at all is clearly organized at the level of subroutine calls, iteration, etc. A clear deductive interpreter must be organized in terms of higher-level entities: goals and

inference rules.

Within this constraint, there are several ways to organize a deducer. The method explored by Nevins <1972, 1974a> in his theorem-proving research seems excellent to me. It uses both forward and backward deduction when appropriate, and is capable of managing very subtle interactions in achieving brother goals. It represents knowledge procedurally when that is the obvious representation.

There are several changes and additions I would make to such a theorem prover. First, I would make explicit some of the information implicit in Nevins' program. For example, a goal of the form (AND A1 A2...An) may be attacked in several orders: some of the A's should be tried first in parallel; others should be postponed and used to filter the initial early results. All of these possibilities should be stated and pondered explicitly.

Second, alternatives such as these must be considered whenever a decision comes up as to what inferences to make. So that this may be reasonably efficient, the deducer should follow certain principles like, "Generally, keep working on the current goal." When an action is tentatively selected, a quick search should show whether there is any possibility of some other action beating it out. Only if relevant advice appears should it be thought about. (This is a crucial example of being able to treat information simply and efficiently when necessary (point (1iv) above). The most common deduction of what rule to apply next is

of the form, "This looks good. Any objections? No? Then....")

It should be noted that the actions to be picked from are inference rules of all sorts, not just a member of a finite set. Besides rules like

(2)(i) "if A is proved and (IMPLIES A B) is proved, you may assert B,"

we might also have

(ii) "if you can't prove there are any blocks in the box, you may assert that there are no blocks in the box."

(This rule is handled procedurally by something like the THNOT device. <Hewitt, 1972> Notice that this latter method requires the decision as to what inference rule to use to be specified by the caller of the deducer. Eg., the two statements (THGOAL (NOT (EXISTS X (IN ?X BOX1)))) and (THNOT (THGOAL (EXISTS X (IN ?X BOX1)))) have quite different procedural semantics, where no such underlying difference exists. The only relevant information here is how much knowledge you think you have about BOX1. A box you can see deserves a THNOT; others may not.)

Information about choosing between rules may be called "quasi-procedural" in that it specifies a preferential ordering of actions. For example, an electronics designer might know the following two coupling tricks:

(3)(i) An appropriate buffer amplifier may be used to couple any two circuits

(ii) A transformer sharing a winding with an inductor is a good way to couple something to a tuned circuit.

Then it should also have the further knowledge that (3ii) is more

specific than (3i), hence, is preferable if they both apply. Such orderings have been studied by Richard Brown <1973>, and been shown useful in a procedural framework. The present approach has the advantage of explicitness; the reasons for an ordering, for example, may be stated in a natural way. Further, there is no more order specified than is justified.

It is very important to be able to represent facts of the form, "Generally so-and-so." This requires use of modal and non-monotonic inference rules. (The THNOT rule is an example of the latter; see Sect IV.A.) An instance of a rule for using such a fact is from <McCarthy and Hayes, 1969>: "From (NORMALLY P) and inability to prove (NOT P), infer P." (Of course, the concept "I am able to prove so-and-so" must be represented procedurally.) Another such rule is, "From (NORMALLY (IMPLIES A B)) and A, infer (NORMALLY B)." Connectives like "presumably" have been studied by logicians in recent years, in mostly uninteresting ways. <Prior, 1967; Kripke, 1963> I will freely use such concepts in what follows.

## II. Electronic Circuit-Design Proposal

The program I expect to write as part of this project is intended to be a module in an electronic circuit designer, which is a joint venture between Gerry Sussman, Allen Brown and me, and possibly others. The structure of this entire device is beginning to be sketched now. The basic idea is that circuits are designed by a process of proposal and testing. My module, the proposer, is to use its knowledge of electronics to propose solutions to electronics problems, and criticize and alter them until they are ready for testing. The tester (a human being in early incarnations of the system) reports how the circuit performed, and whether it works properly. If it doesn't, it is handed to the "bug localizer" to be built by Allen Brown <1973>, which reports, after any necessary tests, on which module is malfunctioning (and possibly why). Then control is given back to the proposer, with enough new criticism added for the proposer to know what the problem is and try something else.

In this section, I will describe the structure of the circuit-design proposer. In keeping with my desire for explicitness, all of this structure will be in the form of knowledge in a uniform language rather than hidden in an expert program. However, a given piece of knowledge may appear in a variety of formats. For example, an instruction to "do this, then do that," may be expressed as a procedure ("this; that")

which is chosen and interpreted, as the left side of an implication ("this and that imply desired result"), or as a statement that "this is preferable to that as a method to achieve the result." In this part of my proposal, I suspend judgment as to the best representation for each piece of information. However, my bias toward the deductive representation should be obvious by now. Particularly for the high-level organization of the design proposer, the deductive notation seems most perspicuous.

## A. Recalling Old Results

When given a design problem, the proposer first attempts to see if it knows the answer already. That is, it tries to use knowledge about particular circuits that solve "about the same" problem as the one given. Recognizing such similarities is, of course, a very hard problem in general, but I envisage here nothing more sophisticated than template matching. Generalizations will be as simple as having "10MHz" in the problem statement match "high-frequency" in the stored circuit. One could imagine a complex indexing scheme that enabled the proposer to pull out appropriate circuits, but I prefer to keep the indexing simple, and treat this as a deductive problem. Other advantages of this decision will appear shortly.

For example, a desire for a "high voltage-gain AC amplifier" should cause the common-emitter circuit to be proposed:

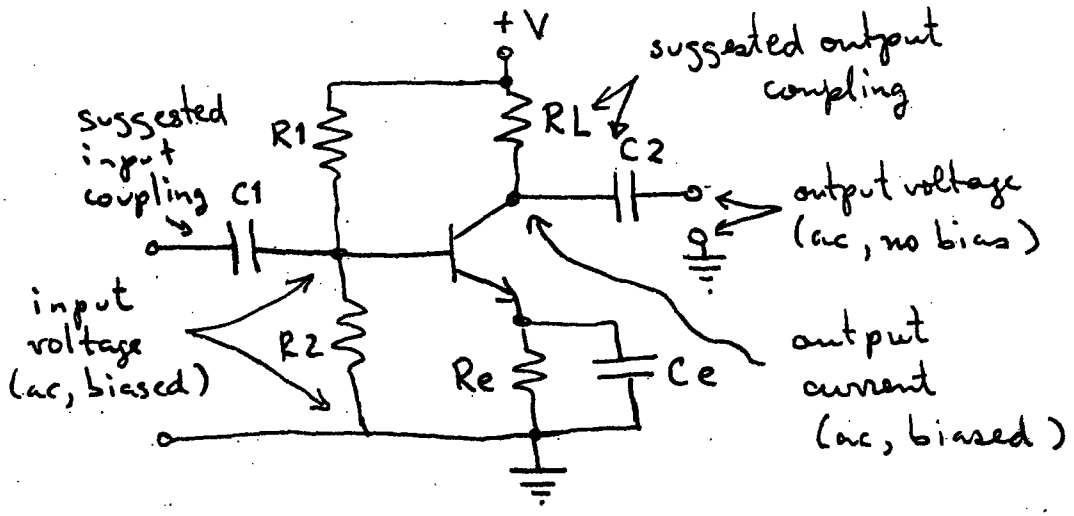
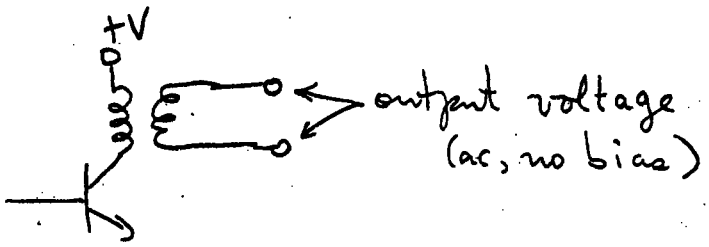



Figure 1.

This diagram, which is incomplete, expresses a lot of different kinds of knowledge. The visual part states the basic topology of a common-emitter connection. This topology is "what comes to mind" when a common-emitter amplifier is under discussion. Parts of it, however, are more essential than others. The transistor is the necessary transconductance that is part of any amplifier. Almost everything else is merely suggested, or "typical," as indicated by the comments surrounding the diagram. R1, R2, and Re are part of a typical biasing network. C1 and the RL-C2 network are suggested capacitive coupling to other stages. They are only suggestions; deeper comments explain what role they play, so that they may be replaced by more useful circuits that serve the same purpose plus others that may be required. For

example, RL and C2 convert a dc-offset current into a decoupled ac voltage with no offset. For some purposes, where an impedance match with the next stage is harder to obtain, they may be replaced by:




How knowledge like this may be used will be described later.

When a circuit is fetched from memory, the most immediate task is to relate the circuit parameters to the given desired parameters. In the circuit of Fig. 1, there must be comments to the effect that:

(4) if there is no Re, or Ce is present, the (incremental) gain of the stage is beta*RL/Rs (where beta is transistor gain and Rs is source resistance). If Re alone is present, the gain is RL/Re.

When the circuit is pondered, to some degree these facts constrain beta, RL and Rs automatically, in the process of pattern-matching. There are, of course, other constraints on RL and Rs, and the proposer must be aware that it has little control over beta. In addition, the fact as given must cause the proposer to think about which of Re and Ce is present. This depends on several considerations, including the type of transistor, the bias stability needed, and the purpose of the amplifier. (Ce would be wasted on a dc amplifier.) All of this

is by way of illustrating that pulling a circuit from memory is a very active process, which must automatically cause suggestions to be considered, constraints to be posed, and subproblems to be put forth.

I visualize this as a deductive process, with advice on how to proceed being on hand at the appropriate moments. The reason for using deduction is that it provides the required active framework for what is essentially index-searching. When there is a goal of the form (IS ?X (AMPLIFIER (GAIN 100))), a simple indexer suffices to fetch

```
(IMPLIES (IS ?X (COMMON-EMITTER (GAIN ?G)))
         (IS ?X (AMPLIFIER (GAIN ?G))))
```

from the data base. The pattern-matcher binds X and G appropriately, and proposes a goal (IS ?X (COMMON-EMITTER (GAIN 100))). Immediately this goal makes relevant the clauses of Fact (4). At this point, straightforward deduction would generate two goals: "?X has Re only"; and "?X has Re or Ce." These lead through other facts to consideration of the role of the amplifier and of RL and Re in it. Details are not possible to give now, but it is clear that exactly these considerations are relevant. (Although some, like biasing, should be postponed.)

B. Inventing New Results

The proposer will not always know a type of circuit adaptable to the problem at hand. In this case, it must invent one. Nothing mystical is meant by this; we cannot expect a program as intelligent as the average technician to invent something brand-new, like the superheterodyne radio. Instead, the proposer must leave the domain of standard circuits and enter the domain of standard tricks. Many of these are methods of breaking problems down along useful dimensions, then solving the subproblems generated, and the subproblem of hooking the solutions together.

Here is a preliminary list of tricks:

(5) (i) Frequency division
(ii) Time division
(iii) Cascading
(iv) Substitution

In more detail:

(i) If a circuit is to have different behavior at different frequencies, a group of circuits, each of which does the correct thing in its frequency range, connected in parallel, will solve the problem:

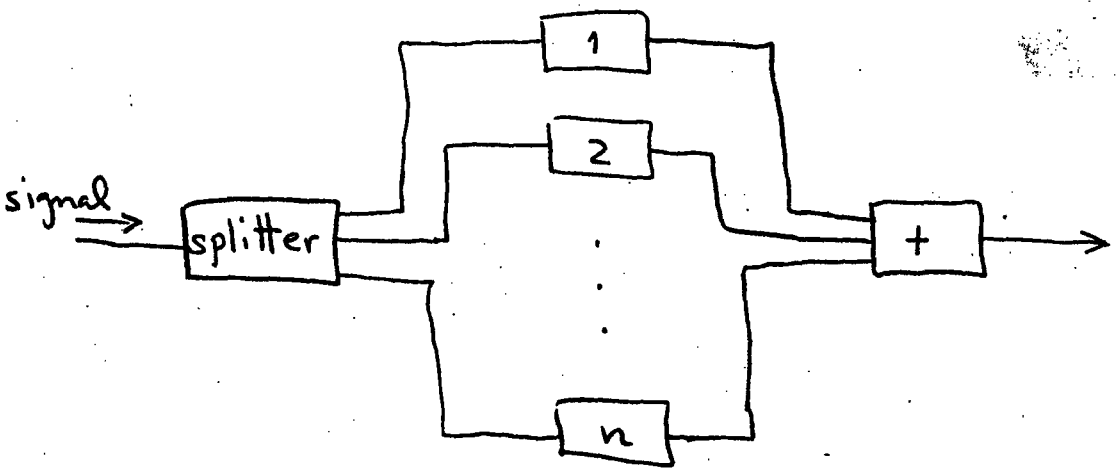



Figure 2.

(Each module is assumed to turn out a zero signal for frequencies it doesn't handle.)

(ii) If the circuit's response is to be piecewise linear (like a detector), do much the same trick, only decompose the signal over the time domain, and use diodes to apportion the pieces:

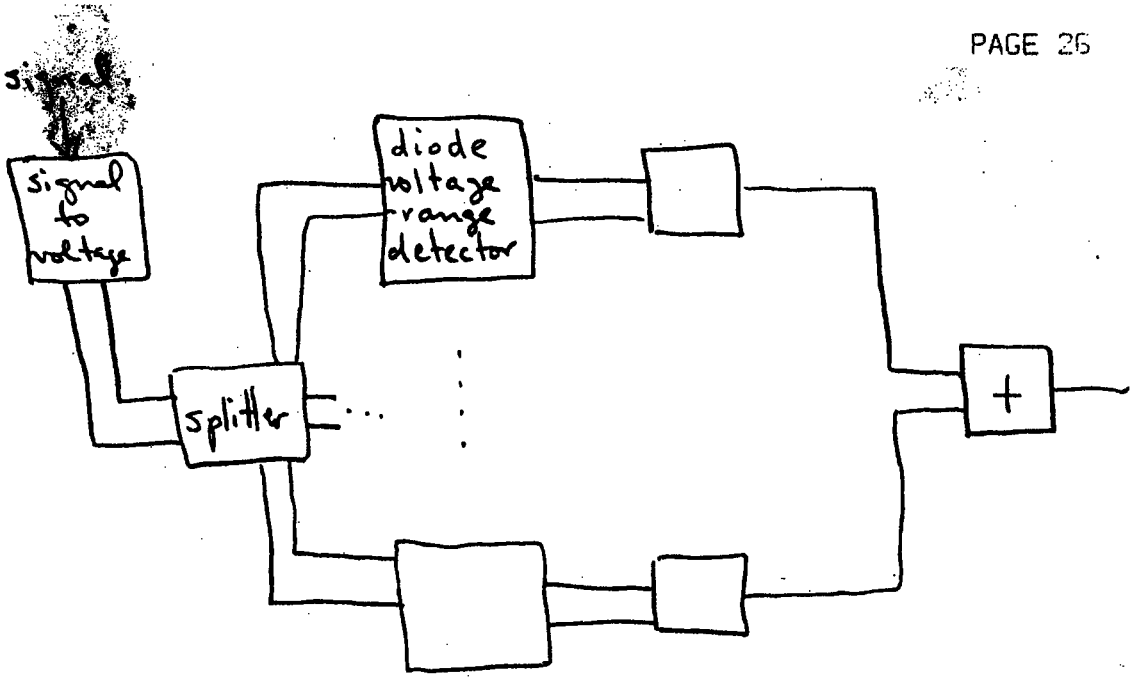



Figure 3.

(Again, each parallel module must give a 0 signal for ranges it is not concerned with.)

(iii) If a circuit behavior can be expressed as a product of behaviors, a cascade of circuits, each of which realizes a factor, will do the job:


$C1 \times C2 \times \cdots \times CN = C$

Figure 4.

A cascade is a more complicated idea than might first appear. Usually, two circuits may not be hooked together without some kind of coupling network. The following fact is relevant:

(6) if there is a coupling network cc that does not load the output of c1 or the input of c2, which transforms c1's output-signal representation into c2's input representation, then c1--cc--c2 is a cascade of c1 and c2.

Notice that this fact may be useful in applying tricks (i) and (ii) as well. (In this fact, "loads" means "alters the behavior of." This requires study to pin down precisely; it is exactly the sort of knowledge that is <u>not</u> taught in textbooks on the subject.)

(iv) The next trick, substitution, is tougher to state, and may just be an addendum on number (iii). It involves the replacement of a part of a known circuit with a new part that does what the old one did, and performs the new function as well. Practically stated, this is a cascade guided by the old circuit as though it were a plan for the new one. That is, if the problem is broken down as for Trick (iii), and this breakdown, except for a box or two, matches the breakdown for a previously generated cascade, then use the old circuit, with the new, differing blocks cascaded in in place of the old. This trick is a little glib, as stated. The intent is to be able to make use of old coupling circuits and special knowledge (e.g., special multi-stage biasing tricks) noticed while assimilating the old cascade. I admit that this is non-trivial. An example of it will be given below. (Research may show that this trick is actually a variant strategy in applying all the other tricks. That is, it is an attempt to use an old plan of any sort.)

Let me trace an example of the invention of a new circuit, the tuned amplifier. Specifically, the problem is to design an AC amplifier (low gain), which amplifies signals around 455 kHz, rejecting all others.

Trick 5(iii) applies here. I will gloss over how it factors this description (this ability requires a lot of electronics knowledge), and assume it can do it. In visual terms, then, the response

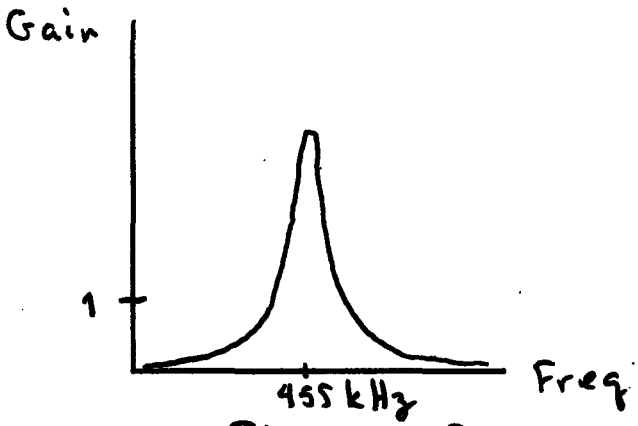


Figure 5.

has been factored into

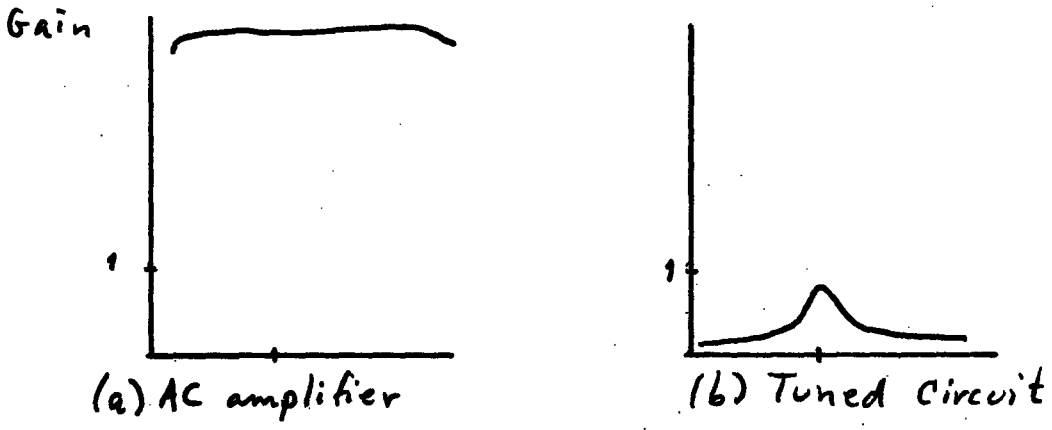
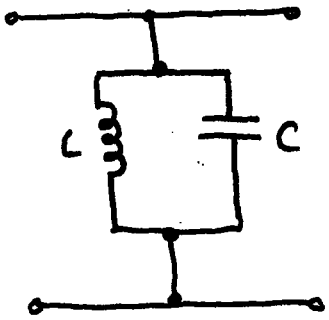



Figure 6.

where the AC amplifier is as given in Fig. 1, and the tuned circuit is




Resonant Frequency $= \frac{1}{2\pi\sqrt{LC}}$

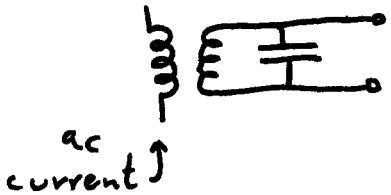
Figure 7.

The cascade trick has generated some subproblems: design an AC amplifier, find an LC "tank" with resonant frequency 455kHz, and cascade them. Presumably, the "problem factorer" doesn't blindly generate breakdowns of problems, but uses it knowledge of circuit responses to propose good blocks. As it fetches them, using the same active index-search process earlier, some submodule design problems are solved "automatically." In

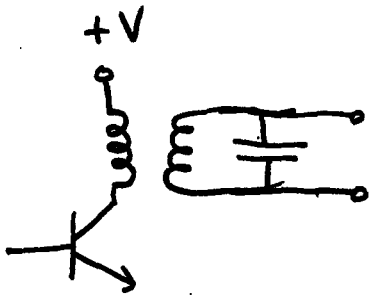
particular, before cascading, the range of beta and the values of RL, Re, L and C will be picked.

Now, to cascade the amplifier and LC circuit, we need a coupling network. The "suggested coupling" (capacitive), unfortunately "loads" the tuned circuit, in the sense of altering its resonant frequency, which is fixable, and lowering its quality, Q, which is not.

This failure, I assume, occurs inside a goal generated by Fact (6). The proposer tried the coupling that "came to mind" and it failed. This should cause an "informed backtrack" (as executed by, e.g., Fahlman's <1973> "gripe-catchers.") That is, attached to the tuned-circuit description should be the notion that




is a good coupling circuit for an LC tank. (This comment doesn't have to be present, but if it isn't, the proposer will have to thrash a bit.) Since a current is a possible output for an amplifier, this suggestion results in the coupling-circuit generator suggesting:



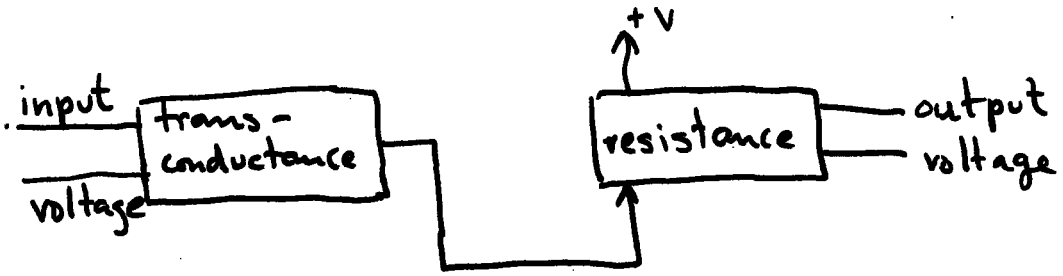

as the coupler.

Now we are essentially done. However, whenever a circuit (at least an "important" one) has been generated, it is usually worthwhile to think a little more about it. For one thing, it must be filed away with comments, the cascade plan that led to it, etc. I shall have something to say about this later, under "Learning."

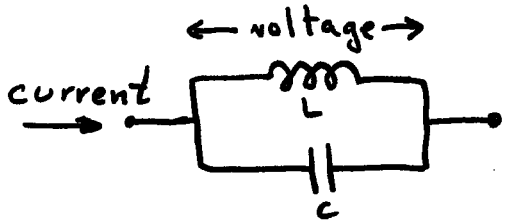
But before doing this, the proposer should make sure it has done the best it can. In this case, it has had to introduce an expensive inductor on the amplifier side. Although this may be justified, let us assume the existence of a demon that catches use of expensive parts (like a very large capacitor) and concludes that a substitution (Trick (5iv)) might save money. (Because a substitution replaces parts as well as adding them, in fact, it may be that Trick (iv) should always be tried first, but that doesn't affect the overall plan.)

Is there a cascade plan for a device that approximately matches "high, flat gain" + "tuned impedance"? If there are adequate comments on the IF amplifier, it will do! The amplifier

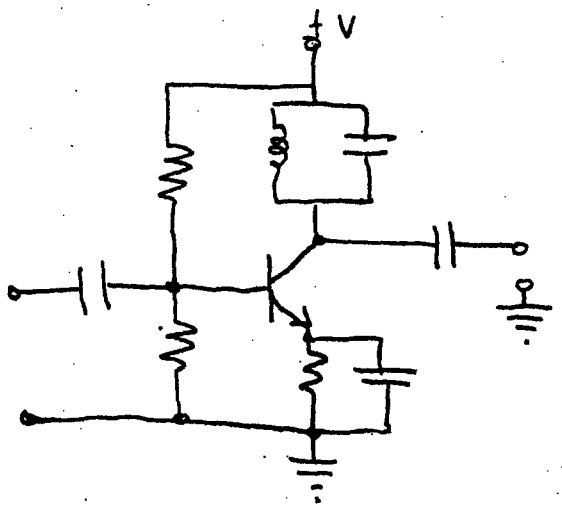
should have a comment that it is a cascade of the form




where a transconductance is a wide-band voltage-controlled current valve. (in this case, a transistor). Now we need a circuit that takes a current and converts it to a voltage the way Fig. 6(b) suggests. This is just the tuned circuit




(from which the circuit of Fig. 7 may have been derived using Trick (5i)). The resulting substitution yields




(with comments) as a new version of the tuned amplifier, with only one inductor.

This concludes the sketch of the proposer, except for a few comments. First, it is obvious that the Circuit Tricks (5) are not the only knowledge the proposer has. There must also be gripe-catching, problem-factoring, and other electronics knowledge. These have been assumed, and occasionally alluded to, in what I have said.

Second, there is some question as to the dividing line between circuits and invention tricks, which are distinguished only by having more comment (or "proposal") and less actual circuit. In some cases, the line between recall and invention will be quite fuzzy. The recaller might discover a very definite plan for a problem, whose boxes are not filled in with complete circuit detail. For now, I classify all such things as tricks.

The reason tricks are special is that they involve

"recursive" calls to the proposer to generate sub-circuits. In the example, these were subproblems that could be solved immediately, by recall of generic devices. Thus there was no substantive recursion of the entire proposer.

It is to be hoped that, at least on a first pass, the proposer may retain this simplicity. There are lot of problems with the intelligent organization of a recursive tree of problem-solving processes. However, for other problem domains, like digital design and program writing, subproblems are generated which have never been solved before. This is not done blindly, but in such a way as to propose subproblems that "look easy." The solutions to these subproblems are often postponed until the higher level is worked out in detail, or even debugged. This looks like another degree of complexity, whose introduction should be postponed as long as possible.

## IV. Objections and Replies

Predicate-calculus theorem provers have gotten a bad name in AI research, and most agree they deserve it. It seems that this is because too many logicians have been involved in it, who have spent too much time worrying about abstruse issues like completeness and consistency, and too little worrying about computation and understanding. These people have obscured the really valuable work in the field, by people like Gelernter <1963>, Bledsoe, Boyer, and Henneman <1971>, and Nevins <1972, 1974a> (to name only a few). I have attempted to follow their lead in allowing the use of information of all sorts when appropriate, not just axioms and theorems, and ignoring abstract completeness when practical completeness, the ability of a system to deduce something in the next year or so, is still an issue. Even consistency is, as Minsky <in progress> has observed, a red herring at best; even a formally consistent theorem prover is going to make practical mistakes (when its axioms fail to conform to reality), and must learn how to correct them. An inconsistent theorem prover can use the same techniques to work around its inconsistency.

Despite my ideological purity, my espousal of formal reasoning is going to encounter objections. In this section, I attempt to meet them in advance.

## A. Logisticity

Marvin Minsky <in progress> has noticed several drawbacks of "logistic" systems. Briefly, they seem to be:

(7) (i) The dependence of the success of a logistic system upon the isolation of a relevant micro-world small enough for it to operate on.
(ii) The absence of THNOT in addition to NOT. (The inability to draw conclusions from the inability to draw other conclusions.)
(iii) Monotonicity: the inability of new information to block a deduction.
(iv) Inability to specify qualifications on how some belief is to be used. (Example: "near-to," a somewhat transitive concept, whose transitivity cannot be applied repeatedly or, worse yet, inductively.)

Another frequent criticism is:

(v) The requirements of logical rigor lead to axiom systems that are too remote from real problems. For example, rigorous geometry axioms require proof of collinearities which people usually take for granted in order to get on to the interesting questions. (The presumed reason is that relaxing this vigilance leads to fallacies, i.e., inconsistencies.)

My system is not logistic in Minsky's sense, but it is likely to appear as a fellow-traveler to a lot of people, so it is worth checking how it fares against these points.

(i) This point is undoubtedly correct. Any deductive plan must notice when it is really just bewildered, and should stop and think first about questions like, what is the real structure of the problem before me? The domain in which this question is asked must be "micro" enough to make the answer clear. Otherwise, it should switch to the following two-step plan:

a. Ask, is the problem really the one I should be working on?

b. If so, ask a human for more information (for a good plan, an overlooked lemma, etc.)

This point appears to apply equally well to a procedural system. In fact, the use of a program does not apply at all to a completely unstructured domain. In this case, we must start with declarative information and figure out what the program should be. (Cf. <Sussman, 1973a>.)

(ii) & (iii) The monotonicity problem is real, but does not occur in the system I have sketched. (Sect. II.) Besides, a general analysis shows it to be part of a more interesting problem, the problem of "passivity." Theorem provers tend to be incapable of "actions," in the sense of "conscious actions consciously taken." This is unfortunate, since in many cases there are external actions that may be taken for deductive or information-gathering ends. For example, there may be many logical principles, memories, etc., that might tell you whether there is a person in front of you; at a bank window, e.g., it is likely there is a teller there. But it is much better just to look and see rather than attempt a proof, in almost any case. Logistic systems cannot usually state the former strategy, although the latter is trivial:

```
(IMPLIES
    (AND (ATROBOT ?W) (IS ?W BANK-WINDOW))
    (EXISTS X (AND (IS ?X HUMAN) (FRONTROBOT ?X))))
```

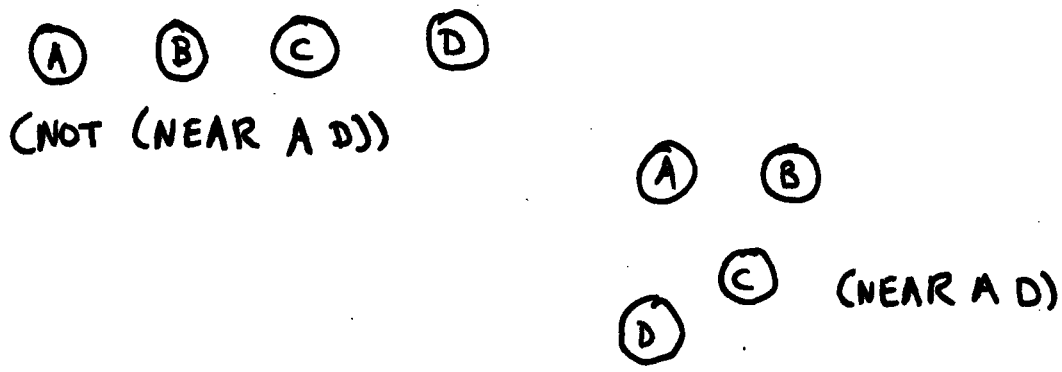
It is easy, in Planner, to write a program to execute the "look and see" strategy, but that avoids the usual issue ducked by the

procedural approach: how do you state the strategy, e.g., as an alternative being considered?

This is not a serious problem. A proposition of the form, "under such-and-such condition, it is good to include 'look for a person in front of you' in the current plan," is close to what is needed.

But this solution to the passivity problem solves the monotonicity problem as well! All we have to do is let "try to deduce so-and-so" be an action analogous to "look in front of you." It is a primitive action, which, like other actions, should be done only when it is cheap and useful compared to the alternatives. I have stated it in Sect. II as a part of an inference rule, inference rules being the actions available to a deduction routine.

(iv) Like Minsky, I have no theory right now about nearness. But I think I have some top-level suggestions to make. First, you need a rule that says, whenever you apply near-transitivity, be sure to verify that you are not using it excessively. This rule might be turned on conditionally. For example, if B is very long, (NEAR A B) and (NEAR C B) do not imply (NEAR A C). If (NEAR A C) was deduced from (NEAR A B) and (NEAR B C), verify that A and C are not on opposite sides of B before concluding (NEAR A D) from (NEAR C D):




Second, suppose there is some kind of inductive rule like

```
(IMPLIES (TRANSITIVE ?P)
         (FOR-ALL Y (IMPLIES (EXISTS S (SEQUENCE ?S ?P ?X ?Y))
                             (?P ?X ?Y))))
```

where (SEQUENCE ?S ?P ?X ?Y) means ?S is a sequence S1, S2,...Sn such that (?P ?Si ?Si+1) and S1=?X, Sn=?Y. Then, we must amend this rule to include the precondition (INDUCTIVE ?P). In this terminology, if NEAR is transitive, it must not be inductive.

(v) This traditional criticism of logistic systems does not carry over to more general deductive systems. For example, where Gelernter's <1963> geometry-theorem prover was logistic, and required a diagram as a subgoal filter, Nevins' <1974b> recent program thinks in higher-level, diagrammatic terms from the start, and achieves much more directed behavior.

A program like this is, of course, as susceptible to fallacies, in this case from a misleading or subtly impossible diagram, as a human would be. Thus such a program will occasionally generate inconsistencies like, "All triangles are isosceles." This would bother a lot of theorem-proving researchers, but there is no reason for it to. The proper conclusion from such an inconsistency is, "I made a mistake,"

not, e.g., "I must be insane." The right course of action is to debug the data base to remove the problem. (Ways of doing this in simple cases are discussed in McDermott <1974>.)

There is a stylistic criticism to be answered here as well. Deduction as classically investigated seems to be in opposition to plausible inference. People who believe in this distinction are apt to think I am cheating when I use formulations (cf. Sect. II) of chess-move generation like "look for plausible moves and see if they work." They might say this is not really deductive; a deduction would have to be of the form, "Prove beyond a shadow of a doubt that some x is the best move available." Others might say it is idle to cast it as a deduction; I am just hiding a procedure.

This last criticism is the best; it may indeed be correct that this level of chess-move generation should be procedural. The issues, however, are not whether chess is a proper topic, or the necessity of the answer desired, but are more like: are the concepts really separable as shown (into plausibility and verification)? is chess well enough understood so that an expert should be written once and for all, at least at this level? is the chess schema an instance of a more general problem of this structure (e.g., the problem of "satisficing" search, that is, finding a "good" element of a set in a reasonable length of time)?

B. Why Deductive Interpretation?

It might seem that we can dispense with a deductive interpreter, and express all the knowledge we need in programs directly. That is, instead of having

(8)(i) machine interprets deducer interprets data base

we might have

(ii) machine interprets Planner theorems

or something like them. (By "machine," I mean "Lisp interpreter," or the equivalent.) Then, instead of a data base, we would have a set of "theorems" compiled from a data base. Indeed, this would be possible, but forcing this scheme in all domains seems completely unnecessary, for the following reasons:

a. To begin with, in the cases where (8ii) is more natural, my proposal allows its use. Some knowledge is naturally expressed as procedures, and should be so expressed.

b. Experience has shown that the use of procedures is successful only if they are well organized and well commented. <Sussman, 1973a> <Fahlman, 1973> If this approach is carried to its logical conclusion, the resulting programs seem equivalent to commented declaratives.

c. The declaratives have the advantage of making as much information as possible explicit rather than buried in the syntax of a program. A declarative goal like (AND A B) may be tackled in more than one order, depending on circumstances. This is

harder to arrange in a program.

d. Uniform use of procedures requires translation of all incoming information into procedures. Most declaratives correspond to more than one possible program. (IMPLIES A B), for example, may be used in an "antecedent" or "consequent" manner, either to deduce B from A, or pose (GOAL A) from (GOAL B). In the many cases where it is unclear how to pick the proper program, or even how to generate it among other choices, it seems harder in an imperative system to represent the uncertainty involved.

e. A layer of interpreter always gives more control over any process. This is because an interpreter maintains data structures regarding the process it is performing, which are close to the surface representation of the structure that it is interpreting, and which may be inspected or altered. Compilation suppresses this information for efficiency's sake. This is a bad idea for a process elaborated from an undebugged structure (a program or data base).

f. Besides, the real efficiencies depend on how well the system understands the structure of its problems. This understanding results from knowledge of mini-worlds, including knowledge of how to use other knowledge. This requirement must be met one way or another, and is independent of whether the knowledge is represented procedurally or otherwise. (Bob Moore pointed this out to me.)

C. Building on Good Ideas

This section is aimed at a criticism that I think lies behind most I have received: that by embracing deduction I am rolling back the clock, and abandoning all the good ideas of the last few years in A.I. From people's comments on early drafts of this paper, I have compiled the following list of such "endangered" good ideas:

(9) (i) Subproblemization
(ii) Ignoring information unlikely to be relevant
(iii) Goal orientation
(iv) "Defaults" and when they are overriden (one of many concepts associated with "frames" <Minsky, in progress>)
(v) Consequent vs. antecedent computations
(vi) "rational form criteria" <Goldstein, 1974> (perhaps the same as "local optimization criteria")
(vii) Gripe catching <Fahlman, 1973>, criticism <Sussman, 1973a>
(viii) Demons <Charniak, 1972>
(ix) Plausible Inference
(x) Information about using information.

In fact, I am aware of these ideas, and consider them very important. Not to use them would be to admit stagnation in A.I., which is out of the question. Some of them, like subproblemization, goal orientation, and consequent vs. antecedent computations, obviously fit in my scheme. Others, especially expression and use of information about information, especially information about what to ignore, I have dealt with at length in Sect. II and elsewhere, precisely because recognition

of them would be the major advance of my deducer over previous ones. The other issues I am aware of, but could not really claim to have dealt with. If, however, the system I design is not capable of exploiting them, I will consider it a failure. What I expect is that the organization I have described will in fact make it easier to exploit them.

I am comforted in this research by my impression that disparate sections of the A.I. community are coming together over a lot of these issues. "Proceduralists" are having to recognize the uses of declarative comments <Sussman, 1973a; Goldstein, 1974>, and "declarativists" are recognizing that their data bases are essentially non-deterministic programs. (Cf. Hayes <1973> and Kowalski <1974>, two old theorem-provers whose current work was brought to my attention after the first draft of this proposal was written. They have things to say about implications as programs which are similar to my theory, and are equally vague as to the exact nature and use of a deductive control language.)

## D. Learning from Experience

Two contradictory objections must be met in this section, for some people will say that my account of advice taking has left learning from experience out altogether, and others will think that I have implicitly appealed to it too often.

I think this kind of learning has been downgraded too much by many workers in AI. The reasonable observation that there are strong limits on what humans can discover has been used to conclude that independent learning is not a very important component of everyday adult intelligence. The claim is made that this kind of learning has been confused with learning "from a teacher," which is much easier to tackle, and much less mysterious. <Winston, 1970> I obviously agree that understanding is more profitable to study than discovery, or I would not be studying advice taking, but I think that there is an important component of discovery in understanding.

To illustrate what I mean, let me pursue an example of how a "Mark 0" Advice Taker might learn from teaching. Assume it has been taught about dc amplifiers and biasing, and about low-frequency incremental models of transistors. These enable it to design good ac amplifiers. It must know about capacitors and passbands in order to choose coupling capacitors that do not block signals. So, when I give it a problem requiring the design of a high-frequency amplifier, it will just go ahead and design

it as before, and it won't work. The Advice Taker won't know this until I tell it (or Brown's bug-finder tells it) why the amplifier failed: because it neglected transistor capacitances. Once I tell it about these problems, including ways of calculating them and minimizing their effect, it should be more careful.

But here's a problem; now it is altogether too cautious. Its inclination will be to make sure with every amplifier it builds that the capacitors are not causing trouble. This is an old AI problem; a program must usually be aware of how to use a piece of knowledge as well as what it is, or it bogs down examining irrelevancies. In this case, it is clear that one's model of the transistor should depend on the frequencies involved. This is a kind of "antecedent," or forward-deductive knowledge.

Since this is a humble Mark 0, I will just go ahead and tell it when to include transistor capacitances and when not. But I think most people are smarter than that. They know that the simple model used to work, and must still work, for the simpler amplifiers. They figure out when to stay with the simple model from their knowledge that essentially one detail, the frequency, has changed. There is nothing brilliant about this reasoning, but we still don't know how to automate it. Winston <1970> has studied reasoning like this under more isolated circumstances, but a design situation appears to be more loosely structured than

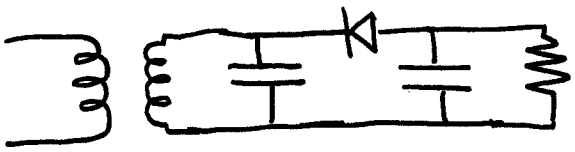
a Winston similarity network.

(Note: it may well be that for the particular example I have given, some people might think instead, "Let me start with the full model of the transistor. Well, right away I can neglect all the capacitors, which look open at signal frequency." This just pushes the problem back to how to learn about capacitors.)

As ever, the learning problem has many facets. Here are four concrete problems I expect to encounter:

a. In the long run, there must be more and more information about assimilating new information. Initially, this will just be an indexer of declaratives. But just having a piece of information does not tell you how to use it. This must be deduced from the way it is stated, knowledge about the domain involved, and experience in trying it.

The most pragmatic such problem is that of understanding new electronics ideas and circuits. When a human technician is shwon a sample circuit of a given class, say a detector:



he can usually understand what is the essential idea (here, the diode), the signal path, merely the suggested coupling or application (the left half of the circuit), the idealized load (the resistor), etc.

As explained above, for now I would like merely to study how to tell this information to a machine. How to derive it from the meaning of circuits and conventions in writing them is a problem Allen Brown, Gerry Sussman and I plan to attack later.

b. When a problem has been solved, the answer should be stored in a form useful for the solution of future problems. This is the closest thing to "independent learning" that I wish to study. The problem here is that the way in which an answer is stored influences whether it will be found later. For example, a receiver filed away under "10MHz" will never be found again; "high frequency" is much more useful. Clearly, how to store particular circuits, cascade plans, etc. depends on electronics knowledge. (This actually seems like a relatively easy problem, at least for minimal results.)

c. The most interesting question is whether the Advice-

Taking Proposer can learn from its brother modules' experience in testing proposed circuits. Initially, a human begin will check its output and tell it the things it has obviously overlooked. In the long run, the proposer, the bug-localizer, and possibly a global "Learning Module" must conspire to set the proposer straight. (Whether there is learning in each module, or global learning, or both, is a problem we are keeping in mind.)

d. A seemingly interesting problem that has been indefinitely shelved is the problem of compilation of knowledge into procedures. This was studied enough by Sussman <1973a> to show that it is crucial and complex, but we are aiming for a deeper knowledge of electronics than his HACKER had of blocks. Compilation is not conceptually necessary (cf. Sect. IV.B), and seems to follow a preliminary exploration of a domain. Besides, proper study of it seems to involve the question of program design, which is more complicated (apparently) than circuit design (cf. Sect. III, last comment).

To avoid any grandiosity, let me admit that I have little understanding of these problems now, without having built a Mark 0. Humans seem to have an ability to summarize the difference between one problem and another, as as to provide plenty of clues later on as to which models and facts to use on a subsequent problem. My hope is that a well-constructed Mark 0 will be a good laboratory for the study of ways of attaining this ability.

## Bibliography

Bledsoe, W.W., Boyer, R.S. , and Henneman, W. H. (1971) Computer Proofs of Limit Theorems, Proc. Inter. Joint Conf. Art. Intel. 2, p. 586.

Bobrow, D. and Wegbreit, B. (1973) "A Model and Stack Implementation of Multiple Environments," CACM 16:10, p. 591.

Brown, A. (1973) Qualitative Knowledge, Causal Reasoning, and the Localization of Failures, Cambridge: M.I.T. A.I. Lab WP-61.

Brown, R. (1973) Use of Multiple-body Interrupts in Discourse Generation, Cambridge: unpublished M.I.T. S.B. thesis.

Charniak, E. (1972) Toward a Model of Children's Story Comprehension, M.I.T. A.I. Laboratory Technical Report 266.

Doyle, A.C. (1967) The Annotated Sherlock Holmes, New york: Clarkson N. Potter, Inc.

Dreyfus, H.L. (1972) What Computers Can't Do: A Critique of Artificial Reason, New York: Harper & Row.

Fahlman, S. (1973) A Planning System for Robot Construction Tasks, M.I.T. unpublished S.M. thesis.

Feigenbaum, E.A. and Feldman, J. (1963) Computers and Thought, New York: McGraw-Hill Book Company.

Fikes, R.E. and Nilsson, N.J. (1971) "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," AIJ 2:3/4, p.251.

Gelernter, H. (1963) "Realization of a Geometry-Theorem Proving Machine," in Feigenbaum and Feldman, p. 134.

Goldstein, I. (1974) Understanding Simple Picture Programs, Cambridge:M.I.T. A.I. Lab TR-294.

Green, C. (1969) "Theorem-Proving by Resolution as a Basis for Question-Answering Systems," in Meltzer and Michie, p. 183.

Hayes, P.J. (1974) "Computation and Deduction," Essex University, to appear in Proc. MFCS Conf., Czech Acad. of Sciences.

Hewitt, C.E. (1972), Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot, M.I.T. A.I. Laboratory,

Technical Report 258.

Hewitt, C.E., Bishop, P., and Steiger, R. (1973) "A Universal Modular ACTOR Formalism for Artificial Intelligence, IJCAI 3, p. 235.

Hume, D. (1955) An Inquiry Concerning Human Understanding, Indianapolis: The Bobbs-Merrill Company, Inc.

Kowalski, R. (1973) Predicate Logic as Programming Language, Edinburgh: University of Edinburgh, School of Artificial Intelligence, Dept. of Computational Logic Memo. 70..undent 3 Kripke, S. (1963) "semantical Considerations on Modal Logic," Acta Philosophica Fennica 16, p. 83.

McCarthy, J. (1968) "Programs with Common Sense," in Minsky (1968), p. 403.

McCarthy, J. and Hayes, P.J. (1969) Some philosophical problems from the standpoint of artificial intelligence, in Meltzer and Michie, p. 463.

McDermott, D.V. (1974) Assimilation of New Information by a Natural Language-Understanding System, Cambridge: M.I.T. A.I. Lab, forthcoming technical report.

Meltzer, B. and Michie, D. (1969) Machine Intelligence 4, New York: American Elsevier Publishing Company, Inc.

Minsky, M. (1968) (ed.) Semantic Information Processing, Cambridge: MIT Press.

Minsky, M. (in progress) A Framework for Representing Knowledge.

Nevins, A. (1972) A Human Oriented Logic for Automatic Theorem Proving, M.I.T. A.I. Laboratory Memo. No. 268.

Nevins, A. (1974a) A Relaxation Approach to Splitting in an Automatic Theorem Prover, Cambridge: M.I.T. A.I. Lab Memo. 302.

Nevins, A. (1974b) Plane Geometry Theorem proving Using Forward Chaining, Cambridge: M.I.T. A.I. Lab Memo 303.

Newell, A. (1962) Some problems of basic organization in problem-solving programs, in Self-Organizing Systems--1962 (M. Yovitts, G.T. Jacobi, and G.D. Goldstein, eds.), New York: Spartan.

Newell, A. and Simon, H.A. (1963) "GPS, A Program that Simulates Human Thought," in Feigenbaum and Feldman, p. 279.

Prior, A. (1967) Past, Present, and Future, Oxford: Clarendon Press.

Slagle, J.H. and Bursky, P. (1968) "Experiments with a Multipurpose Theorem-Proving Heuristic Program," JACM 14:4, p. 687.

Sussman G.J. (1973b) A Scenario of Planning and Debugging in Electronic Circuit Design, Cambridge: M.I.T. A.I. Lab WP-54.

Sussman, G.J. (1973a) A Computational Model of Skill Acquisition, Cambridge: M.I.t. A.I. Lab TR-297.

Sussman, G.J. and McDermott, D.V. (1972) From PLANNER to CONNIVER--A Genetic Approach, Proc. FJCC 41, p. 1171.

Winograd, T. (1971) Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, M.I.T. Project MAC Technical Report 84.

Winston, P.H. (1970) Learning Structural Descriptions from Examples, M.I.T. Project MAC Technical Report 76.