DIRECTORY OF RESEARCH PERSONNEL
IN AI AND LANGUAGE PROCESSING

Donald E. Walker
SRI Menlo Park, Ca 94025

Don Walker is compiling for the National Institute of Education a directory of people engaged in research on artificial intelligence who are interested specifically in natural language processing. Anyone who has demonstrated competence in this area and who has not received a form from Don should submit the following information to him:

Name, Institution, Mailing Address, Telephone
Statement of major research interests in the area of artificial intelligence and language processing (200 words or less)
List of most important publications and reports in this area
Keywords to characterize personal interests:

System Building	Narrative Analysis
Question Answering	Conversational Analysis
Speech Understanding	Problem Solving
Comprehension	Inference
Instruction	Logic
World Modeling	Deduction
Belief Modeling	Induction
Planning	Syntax
Decision Making	Semantics
Protocol Analysis	Pragmatics
Discourse Analysis	Language Acquisition

(respondents are invited to add items to this list)

SYMBOL-MAPPING: A Technical Problem in PLANNER-like Systems

Drew McDermott
MIT AI Lab Cambridge, Mass 02139

This is a brief report on a frustrating problem (or constellation of problems) that has been a source of controversy at our lab recently. It is a summary of the ideas of many people, but, since even the formulation of the problem is controversial, your reporter is solely responsible for describing it in the following terms.

PLANNER-like systems (<Hewitt, 1972> <Rulifson, 1972> <Sussman, 1972>) accomplish deduction by operations on patterns, such as matching and instantiation. Since these operations are under program control, the patterns must be "indexed" so that they may be referred to at random, via other matching patterns. (Most theorem provers just grind lists of clauses against each other.) It is usually assumed that indexing is a relatively cheap operation, requiring a little manipulation of hash tables or discrimination nets, or some other simulation of an associative memory.

In large data bases, however, a large fraction of the data items that are used are implicit: they must be accessed by some sort of simple deduction rather than by a straightforward hash-table look-up. To store all of these potentially deducible facts explicitly requires a great deal of memory space and indexing time, most of which would be wasted, since only a small subset of them will ever be used.

The best examples arise in connection with inheritance of properties down an "IS-A" hierarchy. When you hear that Clyde is an elephant (an example due, like much in this report, to Scott Fahlman), many facts about Clyde become "accessible," including the shape of his nose, his color, his preference in legumes, etc. (Didn't all those things come to mind "effortlessly"?) They presumably follow from some procedural analogue of

(1) (is-a ?x elephant) ⊃
  ((color ?x gray) ∧ (shape (nose ?x) long)
  ∧ (likes ?x peanuts) ∧ (is-a ?x mammal) ∧ ...)

Furthermore, since the large conjunction on the right of (1) includes (is-a ?x mammal), many more properties are implicitly included.

AI languages give us two obvious ways to represent (1): as a consequent theorem (if-needed method, etc.); or as an antecedent theorem (if-added method, WHEN statement, etc.). Neither of these works. If the goal (color Clyde gray) is proposed, consequent methods tell us to propose the goal (is-a Clyde elephant). However, the goals (is-a Clyde battleship), (is Clyde (mother Whistler)), (is-a Clyde Confederate-uniform), etc., are all inevitably proposed as well. (If the goal had been (color Clyde ?c), or a mammalian property like (number-of-limbs Clyde 4), the situation would have been even worse, but people do not have any extra trouble.)

The antecedent method does not work either. It would require assertion of all the properties of elephants, mammals, animals, and physical objects, specialized to Clyde. (Again, one can point to the case of humans, who do not pause for a long time just upon hearing that Clyde is an elephant.)

This looks like a good place to use the "context" or "possible world" mechanism of PLANNER, QA4, GOL <Pople, 1972>, Conniver, et al. In Conniver (with which I am most familiar), a context is a list of context layers, each of which includes some data. A datum is present if it is indexed <u>and</u> in some layer of the current context. (This is oversimplified.) A glamorous thing to do with contexts is to treat a layer as a package of knowledge that can be strung like a bead with other layers to fit a specialized situation. How to play chess, for example, might be such a package. The information would be indexed just once (when it was learned), but only findable when the current layer was plugged in. Plugging in takes a couple of RPLACD's.

An idea like this (the "packet" concept) was used by Fahlman in his <1973> thesis proposal, part of which appears in <Minsky, 1974>.

Unfortunately, it will not work for the case at hand without modification. What we want to plug in is the package {(color ?x gray), (shape (nose ?x) long), ...}, but with <u>?x bound to Clyde</u>. We just cannot get this without further indexing, if 'Clyde' is to point to each of these facts. (It appears to do no good just to omit ?x, since there may be more than one animal or nose around.) This is why this problem is about "symbol mapping"; we want ?x (or "canonical elephant" in many versions) to be mapped into Clyde (and whatever other elephants there are).

The controversy surrounding this problem was sparked by Fahlman's recent work (soon to appear as an A.I. memo), in which he abandons his packet approach because of this and related problems. Fahlman proposes the use of special parallel hardware to do searches through IS-A hierarchies. (His entire scheme includes much more. It is impossible to describe it here.) He and other people have despaired of a solution on a conventional computer.

On ideological and other grounds, others of us have continued to think about solutions on a serial machine, but not always to exactly the same set of problems as Fahlman. Bob Moore has shown that the IS-A problem may be solved by using typed variables and terms, where the types are the classes in the IS-A hierarchy, and then indexing on the types of terms instead of the terms themselves. Thus (color ?x/(elephant) gray) would appear directly in the data base. It would work for all elephants because the indexer and matcher could efficiently pair this with, e.g., (color Clyde/(elephant) gray). Its main drawback is that it works only for an IS-A hierarchy, and only one which is not too "tangled." A tangled hierarchy is one in which Clyde could be many things-- a male, a herbivore, a big animal, a lover of Martha the elephant next door, etc. It is an open question whether the IS-A network should have to do this much work. On psychological grounds, however, Moore and others feel that it should not.

It seems to me that this problem is an instance of the standard "binding" issue of computer science: when shall variables be assigned values. The choice of consequent or antecedent deduction is a choice between deep and shallow binding <Moses, 1970>. When environments or packets are to be switched in and out frequently, shallow binding becomes time-consuming (and, in the case of deduction, space-consuming). On the other hand, lookup can be extremely slow in the case of deep binding. A hybrid scheme has been devised by Chris Reeve of the MIT Dynamic Modeling Group's MUDDLE language project, which is as fast as deep binding at switching environments, and as fast as shallow binding on lookups, except on the first reference of a variable whose binding has changed. I have devised an analogous scheme for the deduction case. I see it as an alternative to the antecedent or consequent way of modeling implication. It has nothing to do with IS-A as such, which may be an advantage or disadvantage.

Schemes like this are unacceptable to Fahlman and others because their implementation requires a bit of bookkeeping on the first request for Clyde's color, which might slow the process too much. In addition, some care is required in managing changes to the concept of mammal or elephant after some elephant has come to mind. In general, the binding approach will not work unless "coming to mind" is well-defined. If all objects are dumped into one global data base, space and time will be consumed just as in the consequent case.

Other people (notably David Marr, in a work in progress) have argued that a heuristic approach will suffice. Just point 'Clyde' at a small list of important elephant properties (which might well duplicate some mammal properties), and make it more expensive to get at any others. Other roles Clyde might have must be made secondary. Many properties will be of the form (color) = gray, with no "?x." The ambiguity so generated can be detected and tolerated. Those who advocate such an approach can point to the general poor performance of people in retrieving obscure properties of concepts. They claim Fahlman's system will solve too much.

We would appreciate hearing of other people's solutions or despair with respect to this problem.

References

Fahlman, Scott (1973) *A Hypothesis-Frame System for Recognition Problems*, Cambridge: MIT AI Lab Working Paper 57.

Hewitt, Carl (1972) *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*, Cambridge: MIT AI Lab TR-258.

Minsky, Marvin (1974) *A Framework for Representing Knowledge*, Cambridge: MIT AI Lab Memo 306.

Moses, Joel (1970) *The Function of FUNCTION in LISP*, Cambridge: MIT AI Lab Memo 199.

Pople, H.E., Jr. (1972) "A Goal Oriented Language for the Computer," in *Representation and Meaning-- Experiments with Information Processing Systems*, H. Simon and L. Siklossy (eds.), Prentice-Hall.

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. (1972) *QA4: A Procedural Calculus for Intuitive Reasoning*, Menlo Park: SRI Technical Note 73.

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

A THEOREM ABOUT AUTOMATIC PROGRAMMING[1]

Peter Kugel
Computer Science Program -- Fulton 406
Boston College Chestnut Hill, Mass. 02167

Introduction

An automatic programming system is a system (usually a programmed computer) that generates a program from partial information about that program (Figure 1). Looked at in this general way, automatic programming systems are hardly new. Assemblers are automatic programming systems. So are compilers, debugging systems, and even loaders. In this paper, I propose to focus on systems that "do more" than such systems in the sense that they require less complete information about the programs they are to generate.

```
Partial Information                           Full
   About Program                             Program
        I                                       I
        V                                       V
    --------        ---------------         --------
    I  \\\I         I  Automatic  I         I\\\\\\I
    I \\\\I   -->   I Programming I   -->   I\\\\\\I
    I   \\I         I   System    I         I\\\\\\I
    --------        ---------------         --------
        I                                       I
        V                                       V
```

Figure 1: An Automatic Programming System

Let us call an automatic programming procedure an *exemplary programming* procedure if it takes (as input) examples of what the program that it will generate is supposed to do and produces (as output) a program that does that job.

```
  Inputs-Outputs                              Full
of Desired Program                           Program
        I                                       I
        V                                       V
    --------        ---------------         --------
    I      I        I  Exemplary  I         I\\\\\\I
    I      I  -->   I Programming I   -->   I\\\\\\I
    I      I        I   System    I         I\\\\\\I
    --------        ---------------         --------
        I                                       I
        V                                       V
```

Figure 2: An Exemplary Programming System

For example, an exemplary programming procedure might take in samples of sales figures and the kinds of reports such figures were supposed to produce and then create a program that would produce similar reports from different figures. Or it might take in sample chess games and develop a program that would play legal, or even good chess. Our main result, stated informally, is the following:

An exemplary programming procedure that uses a totally computable procedure (a debugged computer program) to produce programs can produce only finitely many programs.

This is a very strong limitation on exemplary programming (by totally computable procedures) since the set of all programs for almost any reasonable problem domain is infinite.

[1] Presented at 1975 Computer Science Conference, Washington, D.C.