Seek-Whence: A Model of Pattern Perception

by

Marsha J. Ekstrom Meredith

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 214

Seek-Whence: A Model of Pattern Perception

by

Marsha J. Ekstrom Meredith

September, 1986

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the degree of Doctor of Philosophy.

Douglas R. Hofstadter
Douglas R. Hofstadter, Ph.D., Chair

Daniel P. Friedman
Daniel P. Friedman, Ph.D.

Mitchell Wand
Mitchell Wand, Ph.D.

William C. Perkins
William C. Perkins, D.B.A.

August 20, 1986

DEDICATION

To Sam, for all the hours and all the encouragement.

ACKNOWLEDGMENTS

The work presented here has benefitted from years of support and critique by a number of people. First and foremost is Professor Douglas R. Hofstadter, without whose inspiration, encouragement, patience, and friendship over the years none of this would have been possible. Professor Daniel Friedman has been a constant source of encouragement, and always gave generously of his time. Professor Mitchell Wand's careful reading and clear thinking have helped me enormously, and Professor William Perkins has consistently gone out of his way to be helpful.

I would also like to thank two Systems Managers -- David Plaisier at Indiana and Barbara Zimany at Blackburn -- and their staffs for their help in getting my program on its feet. Thanks also to Paul Reynolds and Audrey Wehking for their work on human pattern perception, and to Dr. Ivan Liss and his CS110 class at Blackburn for serving as guinea pigs in our experiment.

Finally, I would like to thank several people who have helped me on a personal level over the years. Thanks: to Jill Porter for her mighty efforts to keep all my records in order; to Katy, Ken, and Regina Ratcliff for their hospitality and friendship; to Ivan Liss for the pep talks; to my parents, Fred and Margaret Ekstrom, for encouragement and understanding; to my sister Maggie for all those summers of help and support; to Kirsten and Kelsey for being more understanding than could be expected of children; and to Sam, for everything.

Of course, in spite of the efforts of all these people, there may be errors of formulation or execution in this work. For these, I take full responsibility.

PREFACE

In an era when programs have been written to perform medical diagnoses, find oil, analyze soybean diseases, and even rediscover 19th-century chemistry, I have written a program -- and one of some size -- that seemingly does almost nothing.

The program, called Seek-Whence, is designed to discover, model, and reformulate patterns presented as sequences of nonnegative integers. The patterns are not mathematically complicated ones -- they are based on little more than the successorship and sameness relations between pairs of integers -- yet they can become arbitrarily complex, challenging even for humans. Our work on Seek-Whence represents only the barest beginnings in exploring this domain space; the program can handle only a few types of problems of moderate complexity. Nonetheless, we believe that our goals and approach are sufficiently important to warrant further work and much concentrated study.

But sequence extrapolation is a solved problem, handled by Pivar and Finkelstein [Pivar 64] twenty years ago -- is it not? Not in its full generality. The Pivar-Finkelstein system concentrated on extrapolating sequences with underlying mathematical formulas. Hence, these sequences could often be solved by applying a battery of mathematical techniques until an explanatory formula (or collection of formulas) was found. Their domain and approach are quite distant conceptually from ours.

Those who have worked on the formulation and implementation of Seek-Whence are interested in modeling the human ability to discover patterns and to find multiple and/or changing patterns in an evolving situation. Integer sequences happen to be an excellent domain for our purposes for several reasons.

First, we can strip away enough complicating detail to get at core issues. For example, by eliminating knowledge of mathematical operations (such as addition, multiplication, squaring, etc.), we can divest the nonnegative integers of all but their most fundamental properties. They can then serve as atomic units -- structures without internal pattern -- in our pattern domain.

In addition, by presenting sequence terms one at a time, we can explore the ways in which perceptions about a pattern change as it evolves. Humans are able to move from one plausible pattern characterizaton to another without entertaining a host of unrelated and implausible characterizations along the way. We want to model this ability.

Finally, we can test the adequacy of the system's pattern perception by asking for:

1) a characterization of the pattern;

2) an extrapolation of the sequence according to that characterization.

In summary, although Pivar and Finkelstein explored mathematical sequence extrapolation, their work -- and that of their successors -- has left the important and difficult problem of pattern perception in the domain of integer sequences unexplored. The following claim will emphasize the importance we attach to this problem:

Finding patterns in sequences, developing a model to describe the perceived pattern, and reformulating the model on the basis of new evidence is nothing less than scientific induction in microcosm.

This dissertation is organized into five chapters. In the first chapter, we discuss the foundations of our work, including both underlying questions and extant systems that influenced our ideas and approach. The subsequent two chapters document the current implementation of the Seek-Whence program. In chapter four, we compare the Seek-Whence approach and program to several related systems. Finally, in chapter five we present implementation details, review some shortcomings of the system, and set some directions for future research.

ABSTRACT

Seek-Whence is an inductive learning program that serves as a model of a new approach to the programming of "intelligent" systems. This approach is characterized by:

structural representation of concepts;
the ability to reformulate concepts into new, related concepts;
a probabilistic, biologically-inspired approach to processing;
levels of abstraction in both representation and processing.

The program's goals are to discover patterns, describe them as structural pattern concepts, and reformulate those concepts, when appropriate. The system should model human performance as closely as possible, especially in the sense of generating plausible descriptions and ignoring implausible ones. Description development should be strongly data-driven. Small, special-purpose tasks working at different levels of abstraction with no overseeing agent to impose an ordering eventually guide the system toward a correct and concise pattern description.

The chosen domain is that of non-mathematically-sophisticated patterns expressed as sequences of nonnegative integers. A user presents a patterned number sequence to the system, one term at a time. Seek-Whence then either ventures a guess at the pattern, quits, or asks for another term. Should the system guess a pattern structure different from the one the user has in mind, the system will attempt to reformulate its faulty perception.

Processing occurs in two stages. An initial formulation must first evolve; this is the work of stage one, culminating in the creation of a hypothesis for the sequence pattern. During stage two, the hypothesis is either verified or refuted by new evidence. Consistent verification will tend to confirm the hypothesis, and the system will present the user with its hypothesis. An incorrect guess or refutation of the hypothesis by new evidence will cause the system to reformulate or abandon the hypothesis.

Reformulation of the hypothesis causes related changes throughout the several levels of Seek-Whence structures. These changes can in turn cause the noticing of new perceptions about the sequence, creating an important interplay among the processing levels.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

FOUNDATIONS

A. INTRODUCTION

Humans are excellent pattern perceivers. From the tiny baby learning to recognize its mother's face to the scientist whose perspiration is rewarded by a sudden inspiration, we spend much of our lives noticing patterns. Although we find nothing amazing about being able to recognize a friend at a distance of three blocks — a common ability — we do prize the pattern-discovery ability of those who are especially good at it in some domain.

For example, when Larry Bird has one of those special games of which he is capable, we watch in amazement, trying to capture the experience with such phrases as "seing the whole court" or "playing out of his mind". We can feel that he "understands" the court, that he knows where everyone is, where they will be, and what they will be doing. He has a sense of <u>system</u>, of how things fit together, that escapes almost everyone else. "Playing out of his mind" is literally true, in the sense that he need only follow the mental structure he has created to be successful.

The basketball situation outlined above strikes us as very similar to that of the scientist having a "breakthrough", when things simply "come together" or "fall into place" — that is, when important connections are made. We feel that both of these situations, along with a multitude of the more common, everyday kind, are at the core of human creativity. To be precise, the core of creativity is the ability to find unexpected relationships and to discover previously-unnoticed patterns.

BONGARD PROBLEMS

Bongard problems let us experience the "natural" human ability to create and reformulate pattern characterizations. The problems, first posed by Mikhail Bongard [Bongard 70], present the solver with twelve drawings, six on either side of a dividing line. The object is to characterize the difference between the figures on the left and those on the right -- in essence, to explain why the dividing line "makes sense". Several Bongard problems are reproduced in the Appendix.

In solving Bongard problems, we move from one half-formed and tenuously-held idea to another, can feel notions bubbling up from somewhere in our minds, and arrive at unexpected but immediately accepted characterizations. For example, after a group of people worked for a moment or two on problem #21 -- shown in the Appendix -- one person suddenly called out "puppies are allowed!", and the group immediately agreed. Such ideas crystallize suddenly, and feel right. This certainty is not a result of dealing with overly simplistic or common notions. In fact, the favored characterization is often a phrase rather than a simple term, and different people will come up with different but acceptable characterizations that share an underlying notion, the one "conceptual skeleton" (to use Douglas Hofstadter's term) that fits.

My first encounters with Bongard problems were in two classes given by Douglas Hofstadter at Indiana University -- one, a seminar on his book Gödel, Escher, Bach: an Eternal Golden Braid [Hofstadter 79], and the other a class in artificial intelligence. During the seminar, we were made aware of the potential afforded by these problems as a vehicle for exploring human intelligence, and, in a wider sense, were made aware of the unexplored territory opened by such domains as opposed to those typically studied in A.I. -- the knowledge-intensive, the "difficult", the relation-entangled. The discussion of Bongard problems given in GEB, and the problems themselves -- in Bongard's own book, Pattern Recognition [Bongard 70] -- are valuable reading for anyone interested in the mechanisms and structural depth of human intelligence.

In our artificial-intelligence class, we began to explore the Bongard problems themselves a bit more deeply. We tried to watch ourselves solve the problems, tried to verbalize what was going on as our minds seemed to "leave us behind" on some of the problems and come up with solutions. On other problems, we consciously tried different characterizations, our attempts often being colored by our experience with previous problems.

Hofstadter has found or created many terms to describe what goes on in our minds as we attempt to solve these problems. Such terms as "reformulate", "focus and filter", "deform", "structural similarity", "sameness detector", "levels of description", "slipping", "meta-description", "template",and "flexibility" achieve special meaning in this context. Perhaps most important of all:

> "One can think of the Bongard-problem world as a tiny place where 'science' is done -- that is, where the purpose is to discern patterns in the world."
> [Hofstadter 79, p. 659]

BIRTH OF SEEK-WHENCE

The intriguing perspective on intelligence presented in the Hofstadter courses made a strong case for the importance of exploring this new universe of the non-verbalizable, the mental undercurrent, the "subcognitive". All that was required was a suitable domain, one that captured the essence of the problem without being tied to too many extraneous and complicating variables. A fully general Bongard-problem-solver was clearly beyond reach because of the limits of visual processing systems and the overhead they would entail. We needed quicker access to the central issues of perception and reformulation. It was then that a previous project in sequence extrapolation leapt to the fore.

As have many students in artificial intelligence classes, I wrote a program to extrapolate integer sequences. Typically enough, the program could recognize smallish primes and Fibonacci numbers, and could untangle interleaved sequences of fixed- or patterned- length period, such as:

1 1 3 3 3 1 1 3 3 3 . . ., or

1 0 2 2 0 3 3 3 0 . . . .

It could finite-difference its way to solutions of many pathological problems humans would never solve (except by finite-differences, and only under duress) -- for example:

1 2 5 15 42 98 . . .

(a sequence whose second differences are every third prime).

Although pleased that the program could solve so many intricate sequences, I was disturbed in particular by its total lack of "intelligence". The program was "mechanistic", blindly recursive, and not at all sensitive to pattern, as would be a human. The same solution machinery was applied to all sequences, regardless of their form or content.

The juxtaposition of the two projects -- a Bongard-like pattern-discovery and reformulation program with an overly mechanistic, pattern-insensitive sequence-extrapolator -- made for an obvious conclusion, and so the Seek-Whence project was born. Sequence terms have simple descriptions. By ignoring "mathematical" sequences we could concentrate on "the processes of recognizing patterns" [Hofstadter 1982c, p. 10] -- the essence of both Bongard problems and science -- without becoming mired down in "large amounts of specialized knowledge about mathematics and arithmetic" (p.10). The project's name reflects both our domain interest -- we can "seek whence" terms arise in a patterned "seq-uence" -- and the multiple perspectives one must often have of a single object -- in this case, the project's name -- in order to understand it fully.

SOME TYPICAL PROBLEMS

In Seek-Whence, terms of a sequence are presented one by one to the solver by the presenter. The solver's goal is to guess the pattern the presenter has in mind. Clearly, for any given initial segment there are multitudes of possible patterns; however, the solver usually finds the correct solution to a reasonable pattern after seeing relatively few patterned groups of terms.

In order to give a sense of what we mean by "correct" solutions and "reasonable" patterns, we list below a dozen sequences. These sequences were actually presented in the manner described above to each of twenty-five students at Blackburn College, in an experiment to determine the types of complications most troublesome to human pattern perceivers [Meredith 83]. Their experience can be approximated by sampling the sequences one term at a time, making hypotheses as one goes along. The "parsed" sequences follow.

THE BLACKBURN DOZEN

1) 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5...

2) 1 2 3 4 5 6 7...

3) 2 1 2 2 2 2 2 3 2 2 4 2 2 5 2...

4) 1 2 2 3 3 3 4 4 4 4 ...

5) 1 8 5 8 1 8 5 8 1 8 5 8...

6) 2 1 2 2 2 3 2 4 2 5...

7) 2 3 1 2 3 2 2 2 3 3 3 3 2 3 4 4 4 4...

8) 1 2 2 3 3 4 4 5 5 6...

9) 1 2 3 3 4 4 5 5 5 6 6 6...

10) 9 1 9 2 9 3 9 4...

11) 1 8 1 2 1 8 1 2 3 2 1 8 1 2 3...

12) 1 8 5 5 8 1 1 8 5 5 8 1...

THE PARSED DOZEN

1) 1 * 1 2 * 1 2 3 * 1 2 3 4 * 1 2 3 4 5 *...

2) 1 * 2 * 3 * 4 * 5 * 6 * 7...

3) 2 1 2 * 2 2 2 * 2 3 2 * 2 4 2 * 2 5 2...

4) 1 * 2 2 * 3 3 3 * 4 4 4 4 *...

5) 1 8 5 8 * 1 8 5 8 * 1 8 5 8 *...

6) 2 1 * 2 2 * 2 3 * 2 4 * 2 5...

7) 2 3 (1) * 2 3 (2 2) * 2 3 (3 3 3) * 2 3 (4 4 4 4) *...

8) 1 2 * 2 3 * 3 4 * 4 5 * 5 6...

9) 1 * 2 * 3 3 * 4 4 * 5 5 5 * 6 6 6...

10) 9 1 * 9 2 * 9 3 * 9 4...

11) 1 8 1 * (2 1) 8 (1 2) * (3 2 1) 8 (1 2 3) *...

12) 1 8 5 5 8 1 * 1 8 5 5 8 1 *...

We call a run of terms between asterisks (*) in the parsed versions a "template". In order to demonstrate an understanding of the pattern, the solver must complete the current template and fill out the next one -- which is what people usually do anyway when presented these problems.

B. THE SEEK-WHENCE APPROACH

The Seek-Whence system, like any human problem-solver, is presented sequence terms one at a time by the user (presenter). As each term is presented, the system tries to come up with a hypothesis, or characterization of the sequence pattern. If subsequent terms confirm the hypothesis, the system will venture a guess -- not simply by supplying the next template (although it does this), but by showing the user a synopsis of its model. On the other hand, should subsequent terms refute its model, the system attempts to reformulate the model to conform to the new data as well as to the old. If successful at this reformulation effort, the system has a new working hypothesis, open for confirmation or refutation

GETTING AT THE ESSENCE

To be sure, there are some differences between Seek-Whence and a full-blown Bongard-type program. Most obvious is that we chose to deal with one sequence, not a set of twelve drawings. This requires us to predict successive terms, rather than to come up with a verbal characterization. However, the fact that we require construction of a predictive model mitigates this difference somewhat, in that we are attempting to characterize the sequence in some explicit way.

Another difference is that we chose to present the sequence to the system one term at a time, rather than as a whole, as is the case with Bongard problems. This models the scientific method by forcing Seek-Whence to react to new evidence, to reformulate its model of the sequence in the light of new terms. We believe that our choices have made the sequence problem an appropriate domain for the study of the phenomena in which we are interested.

EXTRAPOLATOR LESSONS

One lesson learned in writing the sequence extrapolation program for our artificial-intelligence class was that one must be careful not to build in too many clever devices. The success of that program was directly proportional to the number of tricks and special sequences the programmer could devise. In reaction to that, we have not permitted Seek-Whence to work on derived sequences (e.g., first differences, first ratios, even-numbered terms, etc.) of any kind. Such manipulations as separating interleaved subsequences, pulling out group lengths, and the like are "high-level" actions that can only be employed after the initial noticing of patterns has taken place. To introduce such operations too soon would be to run the risk of overly directing the program's actions, and so of doing its work for it.

When the programmer does get to the point of supplying a "bag of tricks" such as noticing interleaving, or whatever, the program should be able to select tricks from that bag by itself, based on its perceptions at the time -- as people do -- and not based on some "canned", pre-determined hierarchy of techniques. As is pointed out in a later chapter, the Seek-Whence system is just now becoming ready to employ top-down approaches such as these.

C. REPRESENTATION ISSUES

The central concern of the Seek-Whence project is to explore the ability to discover patterns, an ability that requires the development and reformulation of pattern (concept) descriptions. The representation of concepts is critical to the success of the system, because the concept descriptions must express salient information -- where salience is not predefined -- and so must be amenable to fluid and continual modification. In the following sections, we will outline our approach to concept representation and processing in Seek-Whence, beginning with a discussion of our distinction between "complex" and "complicated" systems.

COMPLEX VS. COMPLICATED

Consider this interchange between a college Dean and a faculty member, which occurred in the middle of a discussion about replacing a just-resigned member of the faculty:

Prof: "I guess we should advertise as soon as possible now that it's official. Aahh -- looks like I lost a button off this shirt."

Dean: "It's always sad when a faculty member loses his buttons."

Prof: "Yes, but not as sad as a Dean who loses his faculties."

This conversation, similar to many that occur each day, is nothing spectacular, special, or difficult to understand. Those same actors could also have engaged in a complicated discussion of international law or faculty politics -- a discussion too complicated for many non-specialists or outsiders to understand. The exchange of (sad) puns is, however, a prime example of what we consider to be a complex (as opposed to complicated) interchange. Few elements are being related or discussed, no web of tangled linkages is involved, and no technical terms are used. Rather, the cleverness comes from finding and using unexpected relationships among the elements.

Here is another complex but everyday discussion, this time between a three-year-old and her mother at 9 A.M.:

Child: I want to go visit Toby.

Mom: OK, but you'll have to wait until after lunch.

Child: May I have a peanut butter sandwich now?

Again, we have a situation where nothing difficult is being discussed, but there are obvious important rumblings going on beneath the surface. One can almost see -- cartoonlike -- a little bump appear in the ground and travel from one place to the next, simply disturbing the surface as it passes along below. Something subtle has gone on in the child's mind, but it is unexpected, and it takes us a little while to "catch on".

The terms we will be using -- "complex" and "complicated" -- may not be the best to capture the two underlying notions, the implementation of which may map quite well onto Michalski's "structural- vs. attribute- based" descriptions [Dietterich 83, p.42]. Nonetheless, they will serve as pegs on which we can perhaps hang meanings.

By "complicated", we mean a big, "busy", tangled system of linkages, with much data involved -- a "tropical jungle" of concepts. Complicated systems include murder-mystery plots, automobile engine diagrams, and typical expert-system domains. In computer applications, the concept representations involved tend to be frame-based, with fixed slots to go with the predefined linkages. The focus is on following the proper links to get from one concept to another.

In contrast, a "complex" domain is deep rather than broad -- more like an iceberg field than a jungle. There may be some clear linkages, but some apparently separate bergs are actually connected below the surface of the water. The concept space is relatively uncluttered and the linkages often subtle. Complex domains include puns, some poems, and patterns. In computer applications, the focus would be in finding interesting relationships among the few concepts, which would tend to have structural descriptions. (Winston [Winston 75] and Ronald Brachman, with his KL-ONE system [Brachman 77; 85] have made some progress in the area of structural description of concepts.) We find a helpful metaphor for our distinction between "complicated" and "complex" in the comparison between unraveling a murder-mystery and understanding a short but allusive poem.

There are certainly some domains -- speech recognition and the writing and understanding of stories come immediately to mind -- that are both complex and complicated. In fact, there are probably elements of both in almost every problem. What is of note, though, is that the complex dimension seems to have been virtually ignored so far in most AI research.

SEEK-WHENCE CONCEPT REPRESENTATION

In Seek-Whence, we attempt to begin opening the "can of worms" outlined in the previous section. Seek-Whence compound concepts are represented as networks of primitive concepts. The primitives are fixed, as are most base-level relations. That is, we describe a compound concept in terms of primitive concepts and links, so that a concept's structure holds much information about it. This "complex", structural representation of concepts will permit the use of structural similarities as "virtual links" in the system. That is, we can relate two concepts by noting similarities in their structures and/or structural building blocks, rather than simply looking at their lists of attributes. Moreover, a concept's representation is not unique -- it can be "rephrased" or, as we say, reformulated. In fact, as new sequence terms are presented to it, the system is constrained to change its pattern description in the light of the new evidence. In addition, however, the representation can be changed even though the current model is accurate, simply to see if a different representation "looks better". These miniature paradigm-shifts are termed "slipping", and are crucial if the system is to model fluid movement from one concept to another.

WINSTON AND STRUCTURAL DESCRIPTIONS

The idea of using structural descriptions in a computer system is certainly not new. Patrick Winston, in his important structure-learning program [Winston 75], was keenly interested in employing such descriptions in order to capture notions such as "table", "tent", and "arch". Moreover, in "learning" these notions from a succession of examples and near-misses, his program first created a concept description and then modified it to conform to new evidence. In addition, once the program had learned several concepts, one of its goals was: "To compare some scene with a list of models and report the most acceptable match" [Winston 75, p. 200].

The use of positive and negative evidence, the construction of structural models, and the use of these models to categorize new block figures all have a Bongard-like flavor that we find very interesting and appealing. However, we have had to face some additional representational issues, which we will discuss after first describing our approach to structural representation.

D. SEEK-WHENCE DIAGRAMS

As a first major step in understanding what we were about, our group (Hofstadter, Clossman, and Meredith) devised a set of primitives and a structural representation technique that we called "Seek-Whence diagrams". These expressive visual diagrams, which to some extent have been implemented in the current system, give a sense of how we envision reformulation to take place and how various distinct concepts can be seen to be related through "closeness" of their structural representations.

THE PRIMITIVES

There are eight primitive notions in Seek-Whence diagrams, each of which is represented by a node that takes at least one input value. The function of each primitive is to return a value when queried -- or hit, as we say. A primitive returns no value when an input lies outside of the appropriate domain or when the processing would produce a result out of the range of nested groups of nonnegative integers. A returned value may in turn be used as input to another primitive or may be returned as a final result. The primitives are:

Constant (k) -- returns the value k, a nonnegative integer --

<u>Example:</u> (Constant 4) ---> 4;

Countup (k) -- returns k, then k+1, then k+2,. . . on successive hits --

<u>Example</u>: (Countup 4) ---> 4, 5, 6 . . . , (on successive hits);

C-group (val,n) -- a "copy-group": returns n copies of val, grouped in a pair of parentheses --

<u>Example</u>: (C-group 5 3) ---> (5 5 5);

S-group (k,n) -- a "successorship group": returns the grouped terms (k, k+1, k+2, . . ., k+n-1) --

<u>Example</u>: (S-group 6 4) ---> (6 7 8 9);

P-group (k,n) -- a "predecessorship group": returns the grouped terms (k, k-1, k-2, . . ., k-n+1) --

<u>Example</u>: (P-group 7 3) ---> (7 6 5);

Y-group (first, mid, last) -- a "symmetry group": returns the grouped elements (first, mid, last), where "last" is a mirror image of "first". If "mid" is simply the word "nil", Y-group returns (first, last) --

<u>Examples</u>: (Y-group (5 2) 3 mirror) ---> (5 2 3 2 5)
(Y-group (6 3) nil mirror) ---> (6 3 3 6);

Tuple (arglist) -- returns a group of its arguments' values, evaluated in the order given in "arglist" --

<u>Example</u>: (Tuple (5 3 9)) ---> (5 3 9);

Cycle (arglist) -- returns the value of successive members in "arglist" on successive hits, in a cyclic fashion --

<u>Example</u>: (Cycle (5 3 9)) ---> 5, 3, 9, 5, . . . on successive hits.

Figure 1 shows our diagrammatic representation of all but the two simplest primitives. In Figures 1 and 2, each line represents one hit or query of the given structure.

The primitives can be compounded, with the output of one structure serving as input to another. A hit on the topmost structure causes the propagation of hits throughout the network. The bottommost structures return their values to their calling structures, which then use the returned values to calculate their own values, and so on upwards. A simple example of this is shown in the first diagram of Figure 2. The top-level Y-group requires a value from the Tuple, and so hits it, receiving "(1 4)" from that structure. It then uses this value to compute its own -- "(1 4 4 1)". More examples of compounding are shown in later figures.

Seek-Whence networks can also employ shared structures, as shown in the second and third diagrams in Figure 2. In the first of these, a "Countup" structure is shared by two inputs to the Tuple. When the Countup is hit by the first input, its value -- 3 -- is fed to both inputs, giving the Tuple a value of "(3 1 3)" . Similarly, the next hit of Tuple returns a "(4 1 4)", and so on.

An analogous shared structure is shown in the last diagram of Figure 2. This time, however, the sharers are two inputs to a Cycle, and so we get a different sort of result. The first hit of Cycle causes its first input to be hit, so Countup is hit in turn and feeds both sharing structures -- the first and third inputs to Cycle. The Cycle then returns a 3. On the second hit of Cycle, the middle input is hit, and returns a 1. Then, on the third hit of Cycle, the third input is hit, causing it to hit the Countup again. Countup then returns a 4 to the first and third inputs of Cycle, overwriting both "3"'s at once with "4"'s, and consequently the third input returns a value of 4 to Cycle, which reports it.

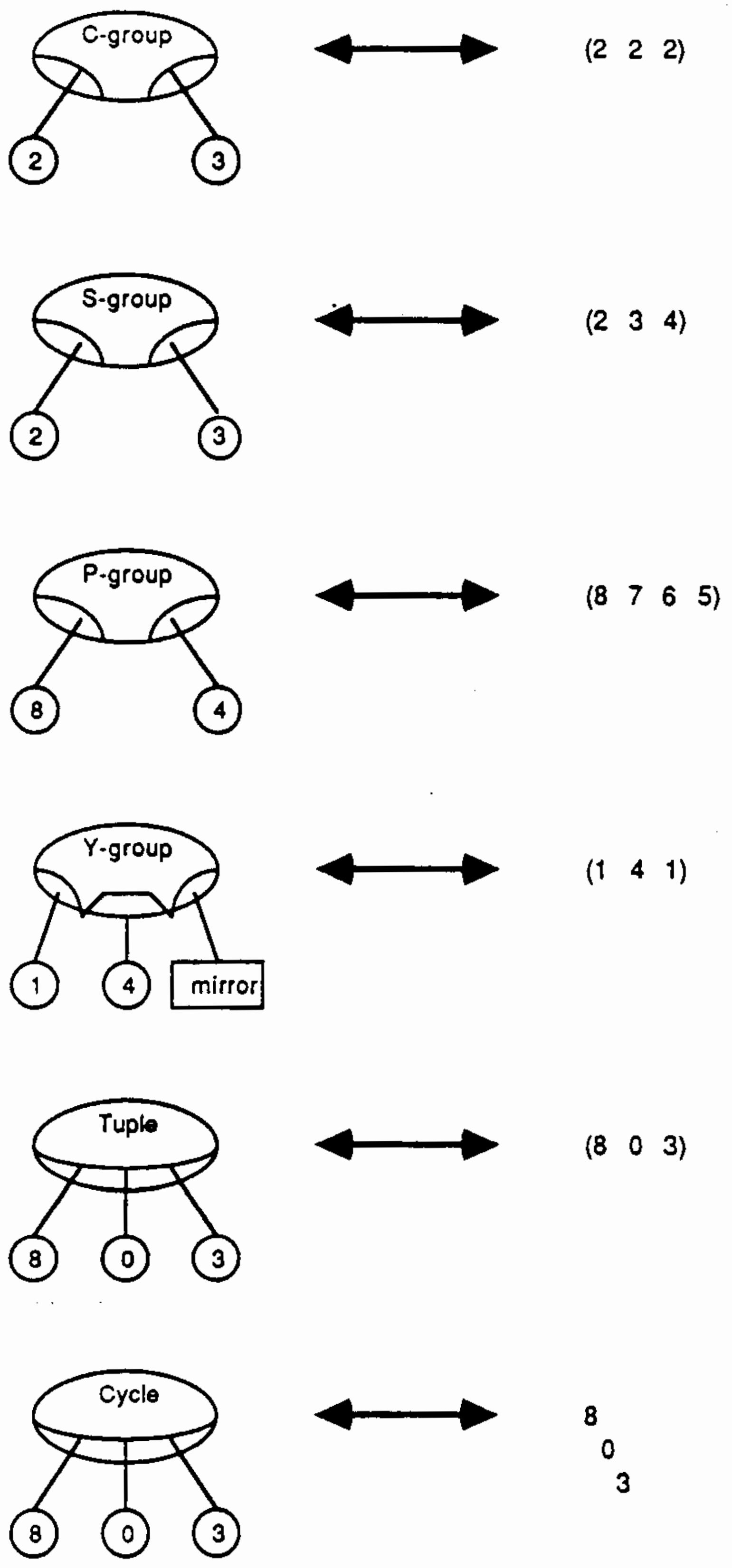


Figure 1 -- The major primitives

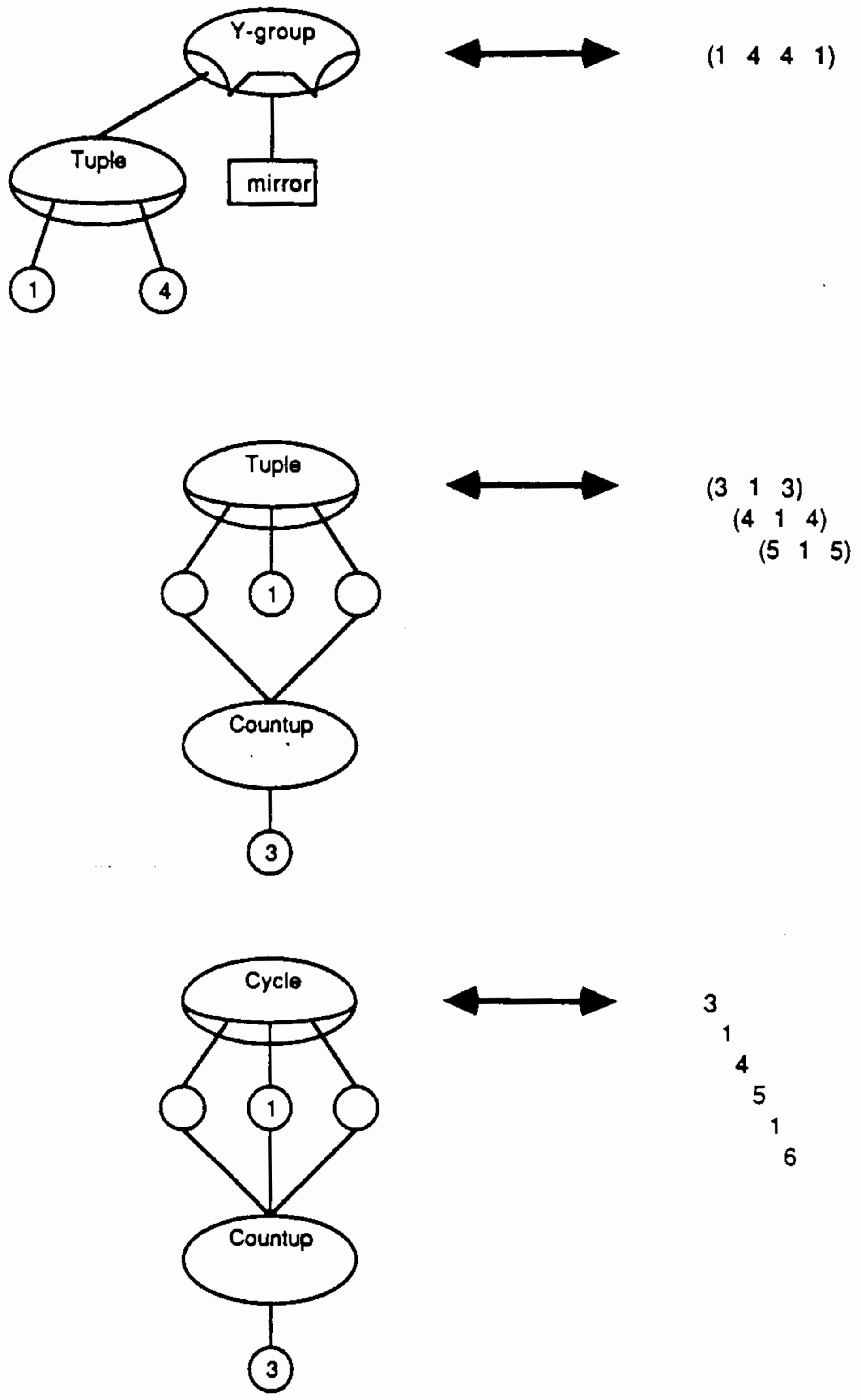


Figure 2 -- Seek-Whence diagrams with some shared structures

Note that if the Cycle above had two different Countups under its inputs instead of a single shared structure, the results returned would have been different. Then it would have returned 3, 1, 3, 4, 1, 4, . . . on successive hits.

E. MODELING SEQUENCE PATTERNS

People presented with the first few terms of a sequence have a strong tendency to formulate a hypothesis about the underlying pattern. One of our goals in creating Seek-Whence diagrams was to be able to model such hypotheses in an understandable, expressive, and flexible (both modifiable and extensible) pictorial form.

Given below are several possible hypotheses based on the initial segment "1 1 2":

```
(1)   1 * 1 2 * 1 2 3 * 1 2 3 4 * . . .
(2)   1 1 * 2 2 * 3 3 * 4 4 * . . .
(3)   1 1 * 2 2 * 1 1 * 2 2 * . . .
(4)   1 1 * 2 * 1 1 * 2 * . . .
(5)   1 * 1 2 * 1 * 1 2 * . . .
(6)   1 1 * 2 1 * 3 1 * 4 1 * . . .
(7)   1 1 * 2 2 2 * 3 3 3 3 * 4 4 4 4 4 * . . .
(8)   1 1 2 * 1 2 2 * 1 3 2 * 1 4 2 * 1 5 2 * . .
(9)   1 (1 2) * 2 (1 2) * 3 (1 2) * 4 (1 2) . . .
(10)  1 1 2 * 2 1 3 * 3 1 4 * 4 1 5 * . . .
(11)  1 1 2 * 3 1 4 * 5 1 6 * 7 1 8 * . . .
(12)  1 * 1 2 1 * 1 2 3 2 1 * 1 2 3 4 3 2 1 * . . .
(13)  (1 1) * 2 * 3 ** (1 1) * 2 * 3 ** (1 1) * 2 * 3 ** . . .
(14)  (1 1) * 2 * 3 ** 1 * (2 2) * 3 ** 1 * 2 * (3 3) ** (1 1) . . .
```

These are all reasonable extensions of the initial segment, although some are more likely than others to come to mind immediately. The last of these, the "marching doubler", is Gray Clossman's invention. It poses some interesting representational problems, as shown in Figure 6. The other parses are given as diagrams in Figures 3 - 5.

In diagram (1) of Figure 3, we have an S-group ("successorship" group) structure. Its first input -- the start value -- is a constant, 1. This means that each hit of the top-level structure will be a successorship group counting up from 1. The second input -- which tells us the group length -- is here the result of hitting a Countup structure. Thus, the S-group lengths will vary, increasing by one on each successive hit. The first length will be 1. Therefore, the first hit on the diagram will return an S-group starting at 1 and of length 1 -- i.e., "1". The second hit's result again begins at 1, but will be of length 2 -- i.e., "1 2". Successive hits give us successively longer successorship runs (with success).

In diagram (2) of Figure 3, we see a top-level C-group ("copy" group), whose first input -- the value to be copied -- changes, but whose second input -- the length or number of copies -- remains constant at 2. Because the first input is fed by a Countup structure, the value to be copied will be successive integers starting at the Countup's start-value -- 1, in this case.

In diagram (6), there is a top-level Cycle. When hit, it will return the value of a hit to one of its inputs. Thus, the first hit of the Cycle results in "1" being returned -- the result of a first hit to the Countup. The next hit of Cycle causes it to return "1", but this time thanks to a hit of its second input. A third hit of Cycle brings us back to the Countup, so a "2" is returned. Successive hits will then generate the indicated pattern.

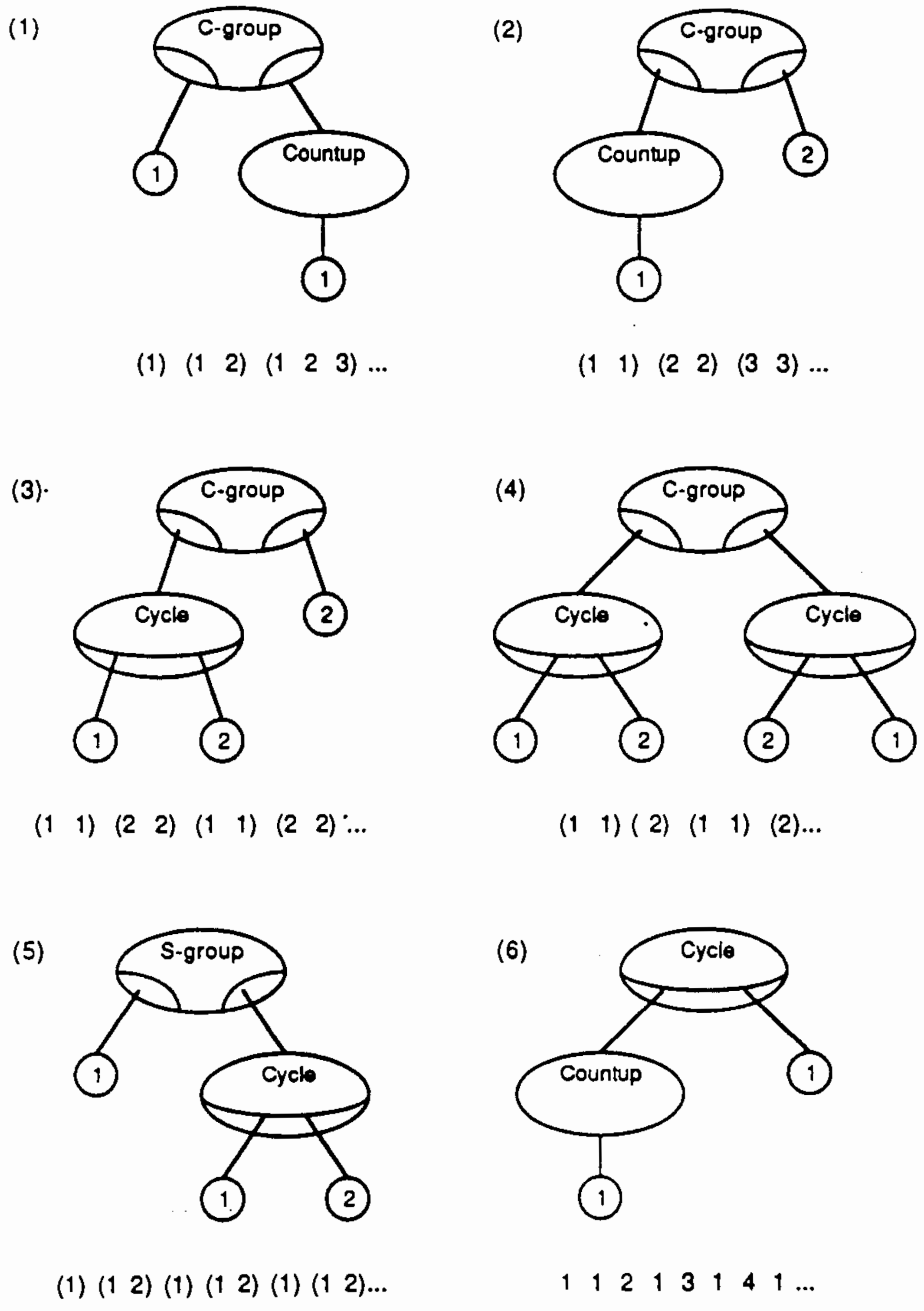
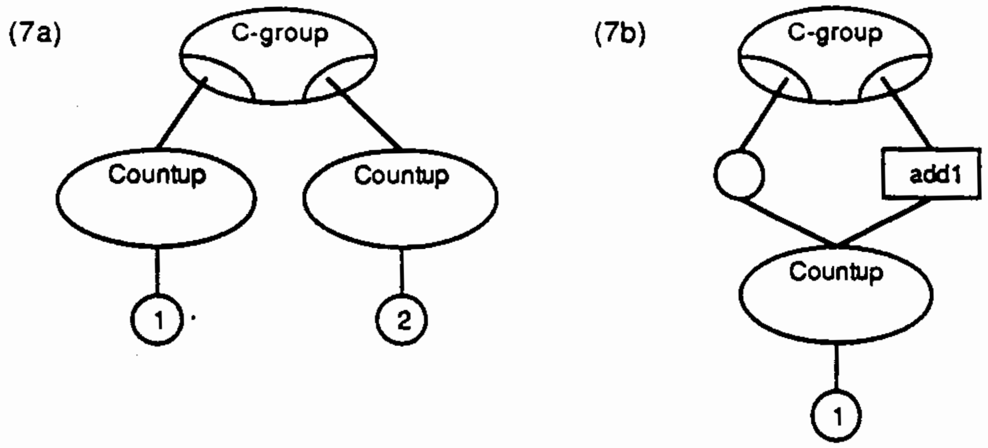


Figure 3 -- Some parses of "1 1 2"



(1 1) (2 2 2) (3 3 3 3) . . .

Figure 4 -- Two different representations of a single parse

In Figure 4, we see an example of two different representations of the same pattern concept. In this particular case, the representations are not apparently very different, since both use "C-group" as the basic organizing notion. The only real difference is that in (b) the successorship relationship between the content and length of each group is made explicit by means of the rectangular "add1" box, whereas in (a) it is not. This small difference can result in very different generalizations of the pattern, however. For example, if asked to generalize from "1" to "2", a program holding representation (a) would give us the sequence :

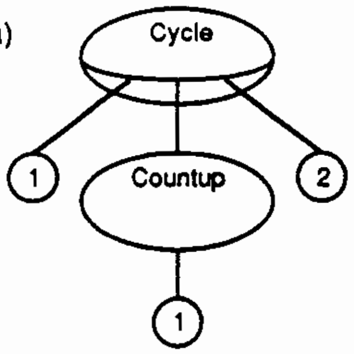
2 2 3 3 3 4 4 4 4 5 5 5 5 5 . . .

whereas a program holding representation (b) would generalize to:

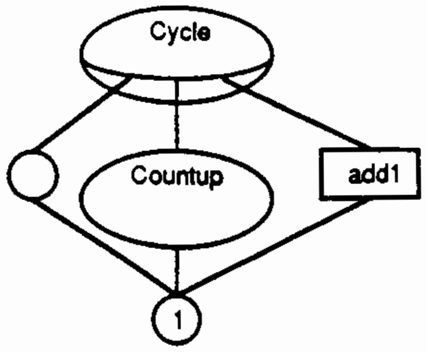
2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 . . . .

No one can say which is the "correct" generalization -- it depends upon the presenter's pattern concept. What we can say is that both are "reasonable".

(8a)

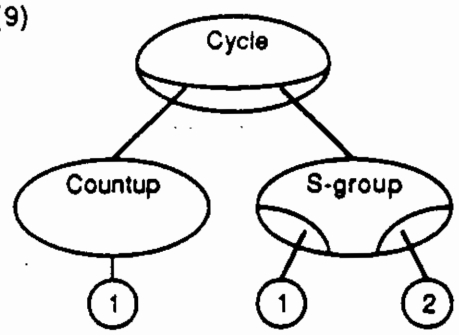


(8b)

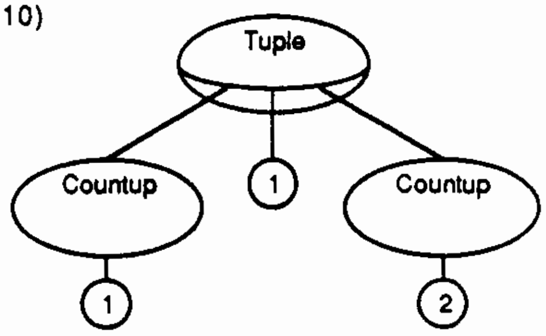


1 1 2 1 2 2 1 3 2 1 4 2 . . .

(9)



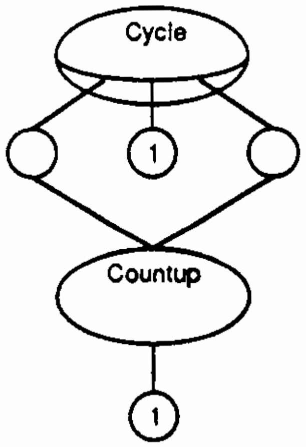
(10)



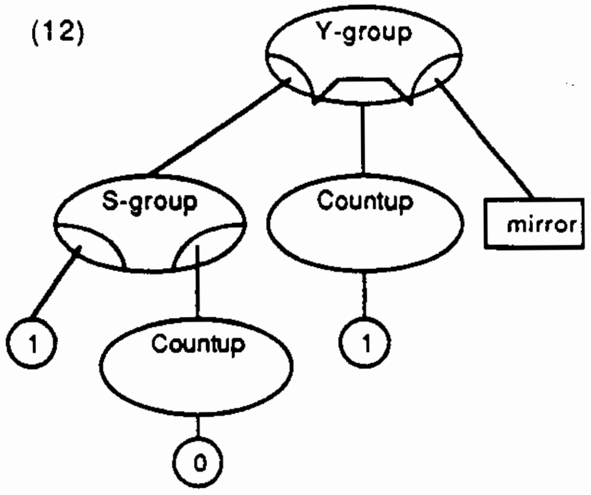
1 (1 2) 2 (1 2) 3 (1 2) . . .

(1 1 2) (2 1 3) (3 1 4) . . .

(11)

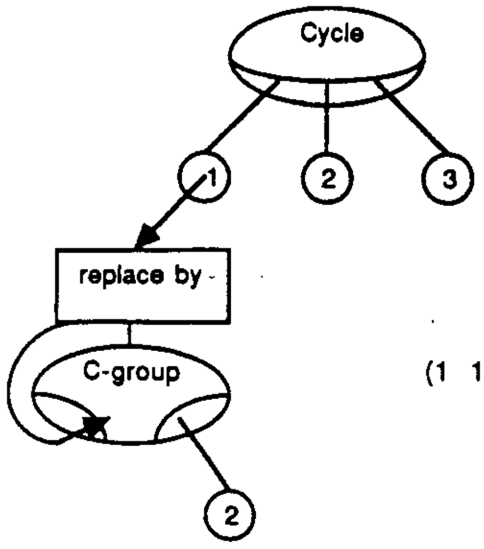


(12)

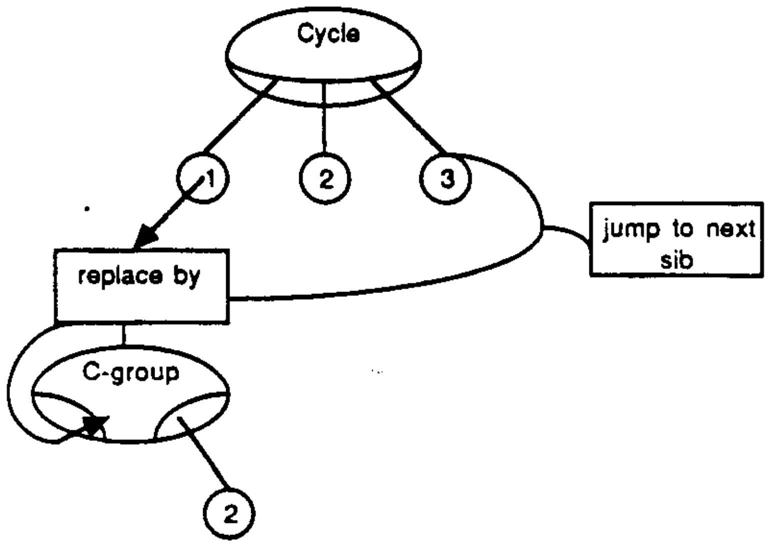


1 1 2 3 1 4 5 1 6 . . . (1) (1 2 1) (1 2 3 2 1) (1 2 3 4 3 2 1) ...

Figure 5 -- More parses of "1 1 2"



(1 1) 2 3 (1 1) 2 3 . . .



(1 1) 2 3 1 (2 2) 3 1 2 (3 3) (1 1) 2 3 . . .

Figure 6 — The doubler and Clossman's "marching doubler"

In Figure 6, we again encounter rectangular "instruction" boxes, indicating modifications to be done on the fly. In the first diagram, the "1" will be replaced by a C-group of length 2 whose value is taken as the "1". In the second diagram, the same sort of replacement is done, but when the "jump" box is encountered (after each hit of the "3"), the entire replace-box structure moves over to the next sibling below the Cycle. Here, it moves cyclically from "1" to "2" to "3" to "1", and so on.

COMPARISON WITH WINSTON

As we noted earlier, Winston's work on structural descriptions colored our thinking on Seek-Whence. But whereas his program had to find discrete objects and then describe the physical relationships among them, our program is given the discrete objects and must describe patterns formed by neighboring groups of them. Winston's program did use grouping as a way of simplifying descriptions. However, block groups were defined in a strict, algorithmic way on the basis of shared properties. Once formed, a group became a permanent unit in the scene description. Our grouping mechanism is more fundamental to our system, in that groups are continually being created and destroyed as the system attempts to formulate a pattern description. Grouping goes on simultaneously with description. Our difficulties, then, lie in finding structures simultaneously with comparing those structures in the "correct" way. For example, in the sequence:

2 1 2 2 2 2 2 3 2 2 4 2 . . .

we can easily tell exactly what the terms are and who is next to whom. We can even note that there is a group of five "2" 's , starting with the third term -- a Seek-Whence "(C-group 2 5)". None of this is relevant, however. What we must notice in order to analyze the pattern is that the aforementioned C-group must not be viewed as such. It has to be torn apart, and its pieces recombined with other sequence fragments in order to make a parse of the sequence reflecting its underlying rule.

Winston's diagrams almost look like the object they describe. We can see the three elements of an arch, and the fact that the supports serve symmetric functions in the whole. In contrast, in Seek-Whence, "a pattern has not been fully understood if the diagram representing it itself contains a pattern. For that means either that some aspect of the pattern was missed or that the notation lacks the power to characterize that aspect and therefore had to copy it verbatim" [Hofstadter 82a, Appendix 1, p.8].

The implementations of the two systems bring out additional distinctions between them. The structures created by Winston's program were essentially static, designed to be viewed and modified. In contrast, the Seek-Whence structures have an active facet — they "act" as well as "are". They need to compute and return values, a process that often requires some sort of memory in each node — of what was hit last, of what value was last computed, and so on.

In summary, we owe much to Winston and his notion of modifiable structural descriptions. However, our domain and interests involve us in a world where the objects to be related must be discovered and described simultaneously, and where the physical relationships between inputs are only fragments of the information needed to describe an underlying pattern. His domain is more like Bongard's in the use of positive and negative evidence to determine membership in a set — be it "arch" or "left-hand-side". Ours is more like Bongard's in the requirement of coming up with a characterization of perceived pattern rather than a description of physical reality.

F. SYSTEM ORGANIZATION

Representation issues as discussed above are closely intertwined with processing and organization in Seek-Whence. The system employs simulated parallel processing with non-cooperating processes working independently and under no overseeing agent. "Triggered" processes — those awakened by recent events — are chosen at random to perform their duties, the choice being affected (but not determined) by the weights or "urgencies " of the candidate processes.

HEARSAY II

The HEARSAY II speech-understanding system [Reddy 76] contributed much to our conception of Seek-Whence. First, it used level-based concept representation, wherein the utterance under consideration was represented differently at different levels, in a language appropriate to the level. Lower levels provided evidence for a higher-level hypothesis, and whenever a support was weakened, the higher-level notion was also weakened. Similarly, whenever a high-level construct was called questionable by some higher-level criterion, the lower-level supports for it were also weakened. This interplay among levels of representation is, we believe, one of the most important contributions of HEARSAY II.

Certainly, the "knowledge source" approach to processing was another contribution. Self-activated, independently-acting processes operated in parallel, communicating only by the process trace they left behind. The trace of a process consisted of the structures it created or modified on the "blackboard" — a global, three-dimensional data structure -- and the triggered (or "awakened") processes left in its wake. This approach, taken to its logical conclusion as in Minsky's "society of mind" notion [Minsky 86], seems to us to be the wave of the future.

COMPARISON WITH HEARSAY II

Our system organization is similar in some ways to that of HEARSAY II, but our processes are smaller and less powerful than its "knowledge sources" and our global data structure is much simpler than its blackboard. Seek-Whence does not physically maintain a collection of alternative hypotheses as did HEARSAY II. Rather, it maintains one "reigning" hypothesis and the ability to reformulate that hypothesis into an alternative one as the "evidence" -- the pressure to change -- mounts. The success of this approach in general will depend upon the system's ability to reformulate easily and reasonably -- a tall order.

F. THE HOFSTADTER CONNECTION

Certainly, Douglas Hofstadter has deeply influenced my work on Seek-Whence, from conception through representation and organization to implementation. Notions he has developed and those that we have developed in innumerable discussions together and with Gray Clossman have become inextricably intertwined, and their realizations have begun to emerge (we hope) in Seek-Whence. These include such notions as active symbols that are composed of groupings of lower-level units, which are in turn groupings of even lower-level units..., reformulation and the importance of "natural" human abilities, conceptual skeletons, slipping, fluid concepts, focusing and filtering, the "terraced scan" approach to processing, the elusive quality of salience, roles and the importance and difficulty of recognizing similarity, the simultaneous creation and use of categories made "on the fly" as needed, the importance of non-cooperating processes and randomness in lieu of an overseeing all-powerful agent designed to make "important decisions", recognition of the complexity and subtlety of perception and its central place in human intelligence, the importance of "toy worlds" and the frictionless universe in getting to the heart of a problem.

A clear and direct exposition of some of the central notions underlying Hofstadter's work is given in the paper "Artificial Intelligence: Subcognition as Computation" [Hofstadter 82b]. This is important reading for anyone deeply interested in exploring intelligence rather than chasing its shadows. The paper, and some subsequent thoughts, are reprinted in the book, Metamagical Themes [Hofstadter 85a] (as Chapter 26).

SEEK-WHENCE AND ITS FAMILY

The Seek-Whence project presented here is only one of a family of Hofstadter-inspired works designed to address the issues of perception, reformulation, and similarity. Other members of the family are Jumbo -- an anagram-solver; Copycat -- a pattern analogy program; and Letter Spirit -- a style-extrapolation system operating in the domain of visual letterforms "a" --> "z". The Fluid Analogies Research Group (FARG) at the University of Michigan is currently working on or has completed work on each of these projects [Hofstadter 85b].

JUMBO

The eldest member of the Hofstadter-inspired family is Jumbo [Hofstadter 83]. This system explored the domain of word "jumbles" (anagrams). As the game is usually played, the anagram solver is given a word whose letters have been scrambled -- such as "toonin". The solver's object is to unscramble the letters to reveal the unique word that can be formed from them. Jumbo strays from this norm in that it does not actually have to come up with real words -- it has no dictionary of the English language. Rather, its object is to create word-like entities -- things that could be English words -- from the given letters. This modification goes to the heart of the matter -- how people go about solving jumbles -- while bypassing the side issue of dictionary lookup. The system must "judge its progress on purely internal criteria of coherency at several levels of structure at once." [Hofstadter 84, p.11]

Jumbo has knowledge of how consonants and vowels "like" to be grouped into clusters, how clusters can be made into reasonable syllables, and how syllables can be combined into words. The system, knowing only these affinities and using a probabilistic, simulated-parallel control structure similar to Seek-Whence's, consistently comes up with good word-like objects from its input letters. Macro-level order emerges from micro-level chaos, chaos of processing as well as of input.

In Jumbo, Hofstadter also began exploring the ideas of terraced scan, temperature and self-watching. A "terraced scan" is a technique for progressively deepening the exploration of several different pathways in parallel. The most fruitful or interesting pathways tend to be explored more deeply, while less plausible pathways are seldom visited.

Briefly summarized, the "temperature" of a system both describes and emerges from the activity level in the system. When the temperature is high, even unlikely pathways may be explored. Conversely, in low temperatures only very plausible pathways are explored. In Jumbo, the system's temperature is controlled by the "happiness" of the structures it has created. Initially, when single letters ("unhappy" because they "want" to be combined with other letters) are introduced, the temperature is high, encouraging the letters to mingle and combine. Later, when a suitable word-like entity has been created and all letters are included in it, temperature falls off to the freezing point, inhibiting any further activity.

Self-watching is an important notion in any system lacking overseers that check for loopiness of behavior. In a system such as Jumbo or Seek-Whence, structures are continually being created and destroyed. It is certainly possible that such a system will recreate a structure time and again. This sort of loopiness is not a problem unless it takes over the system -- that is, unless it takes place at a high enough level that it inhibits other processing. Jumbo had no effective controls for such behavior, relying on externally-imposed temperature changes to destroy recurring structures. Seek-Whence goes a step further by remembering encapsulations of previously-generated hypotheses in order to prevent their re-use. More sophisticated self-watching is being incorporated into the Copycat system, the third member of the FARG family.

COPYCAT

Copycat [Hofstadter 84; 85, ch. 24] is the principal current focus of attention at FARG. Like Seek-Whence, it involves noticing patterns, but this time in a slightly different "idealized domain" and with explicit attention to one of Hofstadter's major interests -- analogies. The Copycat system is given three strings of letters, each string being one element in a four-part analogy problem. The system is to complete the analogy by discovering the fourth string. For example, if given the input:

ABC ==> ABD ;   PQR ==> ?

the system should respond with another alphabetic string as its answer. ("PQS" would be good, "PQD" would be defensible, and "ABS" would be strange.)

Like Bongard problems and Seek-Whence problems, Copycat analogies require a good deal of thought and ingenuity to solve in all generality. Attention must be given both to the "face-value", the actual letters involved --

their "extensional" or "syntactic" identities -- and to the roles those letters play in the strings in which they are seen -- their "intensional"or "semantic" identities. The depth of difficulty in defining roles and evaluating their meaning is explored in [Hofstadter 80; 85, ch. 24].

Not surprisingly, the notion of salience pops up in Copycat as it did in Seek-Whence. In our example above, is it important that "A" is the first letter of the alphabet, or is that fact just "noise", interfering with our ability to find a good solution? Do the lengths of the strings matter or not? How do we identify the important facets of the first half of the analogy and then translate those accurately to the second half? These questions are not easy to answer in general. FARG might have tried to create a letter-analogy "expert", but instead opted for the usual Hofstadter system organization -- simulated parallelism among small tasks. The tasks are non-cooperating, with no overseeing agent to direct system activity. Rather than construct alternative high-level hypotheses, a terraced scan [Hofstadter 84, pp. 13 - 14] is used to explore many low-level pathways simultaneously. The most successful and appealing paths will tend to be pursued most actively. As in Seek-Whence, a current hypothesis will be reformulated when the weight of evidence turns against it. Thus, the process of discovery that the Copycat system must go through is very similar to that required of Seek-Whence. The explicit use of analogy makes the connection to Bongard problems clear.

In developing Copycat, the members of FARG have begun implementing a "Slipnet" similar to but more sophisticated than the one used in Seek-Whence. The Slipnet structure, absent from Jumbo, is a repository of information about the Platonic concepts known to the system. Its nodes and links "form a storehouse of conceptual proximities (slippability links) and semanticities (centrality values)" [Hofstadter 84, p. 20]. The Slipnet is crucial in supporting fluid yet controlled passage of activation from a concept to its neighbors. This "spreading activation" causes some of the concepts to be more "interested" in the ongoing problem-solving activity than others. Those that are most interested will tend to come forth as potential organizing notions, "popping to mind" as it were.

A well-developed and fluid Slipnet is necessary for the complete exploration of relationships among the atomic entities of the system -- be they letters or numbers -- and among any perceived groupings of those entities. It is also difficult to implement. There must be enough activity so that new ideas keep coming as needed. On the other hand, a "hyperactivated" Slipnet, wherein nearly all the concepts are active most of the time, is too confusing to be helpful. Gray Clossman has become very interested in taming the Slipnet as well as in creating uniform structures for all levels of abstraction in the system [Clossman 85].

Both Seek-Whence and Copycat are charged with finding a useful description of their input -- a description that "works" in solving the problem posed. In Copycat, the first two letter strings must be contrasted to show a clean distinction. The first and third must provide fodder for translation, including the translation of the difference between the first two! David Rogers at FARG has begun to attack these problems in a unique way -- by creating potentially schizophrenic structures [Rogers 86]. For example, in the string "ABD", the "D" will feel a little uncertain about its identity, because the "B" and the string "AB" would like to be followed by a "C", and will continually ask the "D" if it is, in fact, a "C". Thus, the unusual elements of a string may be pointed out by the system's structures themselves.

Seek-Whence does not have as many distinct elements as Copycat to work with, since it operates on a single sequence of integers. This is a boon in allowing the system to focus its attention in one place, but a bane in imposing fewer constraints -- the help it could get in parsing the sequence by looking at two structures that are known to be similar. Nonetheless, the two systems obviously share a conceptual skeleton and are deeply concerned with perceptual mechanisms as the foundation for even the highest levels of cognition.

LETTER SPIRIT

The "youngest" member of our project family -- Letter Spirit -- involves what may be the purest domain for the exploration of perception. The task of this system is to extrapolate the style of a given letterform to other letters of the alphabet. Some workers at FARG have begun to attack this problem, but the challenge is great. Perhaps the best indicator of the difficulty of this undertaking is to note that the final Bongard problem -- problem 100 -- consisted of six "a"'s on one side of the dividing line and six "b"'s on the other.

H. CONCLUSION

In the Preface to this dissertation, we claimed that pattern perception is scientific induction in microcosm. To be sure, we recognize that scientists rely on a great deal of factual knowledge and that the scientific method requires careful experimentation and evaluation of evidence. In this respect, we are exploring only a small region of a vast territory. However, the creative essence of science is the inductive part, the ability to find connections where none were previously known. We believe we can explore this essential region through programs such as Jumbo, Seek-Whence, Copycat, and Letter Spirit. In a topology where complexity is the metric, our small domains for the study of discovery and perception may be of the same size as highly complicated scientific domains. We are, at least, certain that our problem -- the perception of patterns -- is, as almost everyone notes when first entering Dr. Who's Tardis, "bigger on the inside than on the outside".

CHAPTER TWO

SEEK-WHENCE: STAGE ONE -- HYPOTHESIS CREATION

A. INTRODUCTION

In the previous chapter, we presented our central problem -- finding patterns in sequences of nonnegative integers. We also developed "Seek-Whence diagrams", a structural representation system for describing such patterns. In this chapter and the next, we go on to describe the Seek-Whence system and to document those features that have been implemented in the current version of the program.

The program realizes most of the features of Seek-Whence diagrams in its structural representations of patterns -- called <u>hypotheses</u>. The most important omission is of the rectangular instruction boxes seen in Figures 5 and 6 of the last chapter. For many (but certainly not all) sequences, the program can create a hypothesis as it is presented the terms of a sequence, thereby building its own model of an unfolding pattern. Moreover, the system can often reformulate its hypothesis to form a new one when subsequent sequence terms prove the current hypothesis incorrect.

B. OVERVIEW OF THE SEEK-WHENCE SYSTEM

1. DOMAIN AND GOALS

As was mentioned in the Preface, the domain of Seek-Whence lends itself to the study of pattern perception. By eliminating knowledge of mathematical operations, we can avoid such problems as whether "4" should be interpreted as 2*2, 5-1 , or 100/25. This permits us to concentrate on "4" as an atomic element in a pattern. The value of the element may or may not have other significance, but it cannot be seen as having any internal pattern. For example, "4" is an element in the segment "2 3 4 5", and it also represents the length of that segment; it can also be viewed as the successor of "3" or as the predecessor of "5". Beyond that, it has very little structure.

Why did we choose such a "simple" domain? We wanted to study pattern perception, not finite differences or number theory. We can come up with some very difficult patterns in our little universe, yet the components are simple. This is just what we were after -- a domain wherein problem-solving difficulties clearly arise from the way in which the elements are combined and not from the elements themselves.

The patterns studied, therefore, are non-mathematically-sophisticated rules that generate sequences of nonnegative integers. In response to a prompt, a user presents to the system numbers which presumably follow some pattern the user has in mind. The system receives these terms one by one and, after each one, either ventures a guess at the underlying pattern, quits, or asks for more information (another term). Should the system guess incorrectly -- that is, guess an underlying rule different from the one the user has in mind -- the user will so indicate and the system will continue, probably by asking for more terms, and then using those as a basis for reformulating its faulty perception. The patterns presented can be very subtle or very simple, but in every case the system's guessed rules should be "reasonable", acceptable as possible solutions to a human observer; they should elegantly and economically explain the portion of the sequence already seen, as well as predict an infinite continuation.

Some "typical" pattern problems might start out as follows:

2 2 2 ...

1 1 2 2 3 3 ...

1 2 2 3 3 3 ...

1 2 1 2 3 1 2 3 4 ...

1 0 1 0 0 1 0 0 0 1...

1 2 8 3 4 8 5 6 8...

1 2 8 3 4 5 8 6 7 8 9 8...

3 7 3 7 3 7...

8 0 8 8 0 8 8 0 8...

1 1 2 1 2 2 1 3 2 ...

Some non-domain problems are:

2 3 5 7 11 ... ---------- "primes"is too mathematically sophisticated a notion;

-3 -2 -1 ... ---------- negative integers are "unknown";

1 7 9 15 18 ... ---------- "get bigger" is too amorphous; there is no canonical "next term".

In essence, we can assume the system is like a small child who is able to count and notice samenesses but who cannot do arithmetic, count by twos, recite primes, etc.. It is critical to emphasize that we are after pattern rather than mathematics here.

2. THE TWO STAGES OF PROCESSING

There are two stages of processing in Seek-Whence. An initial formulation must first evolve; this is the work of stage one, culminating in the creation of a hypothesis for the underlying rule. This "preliminary" stage is really quite complicated and very important. The structures created during stage one play a critical role in later processing, since all high-level actions inevitably affect them and are affected by them. As did the HEARSAY II system [Reddy 76], Seek-Whence operates simultaneously at several levels, from the most concrete -- the integers input at the terminal -- through the descriptive -- the hypothesis and its supporting concept descriptions -- to the most abstract -- the "ideal" primitive concepts. Low-level structures support the creation of higher-level ones, and indirectly even determine the course of high-level processing. When changed by high-level actions, as they inevitably are, the lower-level structures may have an unexpected effect on the higher-level ones. These reverberations, modeled on the "ripplings" among levels in HEARSAY II, are at the heart of SEEK-WHENCE's processing, and are necessary to cause the interplay of bottom-up and top-down activity required for Seek-Whence to work properly.

During stage two, the hypothesis is either supported or refuted by new evidence. Consistent verification, in the form of terms which support the hypothesis, will lead the system to a confirmation of the hypothesis and the venturing of a public guess. An incorrect guess (one that is rejected by the user) or refutation of the hypothesis by new evidence will cause the system to reformulate or, in rare instances, abandon the hypothesis. Hypothesis abandonment or "scrapping", which is analogous to a human's "let's start all over again", takes system processing back to the lower levels. This is not a total restart with a clean slate as though the sequence terms had never been seen, but rather a return to the term level, with all perceived groupings eradicated but with accumulated knowledge of term samenesses and other primitive relations maintained.

The major distinction between the two stages of processing is the existence of the hypothesis in stage two. Without it, the system has no model of the sequence and so cannot predict the next term to be encountered. Once a hypothesis is in place, all new evidence is "filtered" through it (checked for agreement with it). Confirming data -- new terms that fit the hypothesis -- are handled rapidly, essentially just being "swallowed" by the system. In contrast, entry of an unexpected term makes the system "sit back and look things over".

REFORMULATION AND THE SLIPNET

Changing a hypothesis is done by reformulation -- modification of the form of the hypothesis. Reformulation is accomplished by "slipping" from one Seek-Whence concept to another. The direction of change will be suggested by system processes, based upon the evidence gathered from that portion of the sequence already seen and guided by the "slipping knowledge" possessed by the system. A structure called the Slipnet, which maintains relationships among the primitive Seek-Whence concepts -- the "ideals" -- as well as pointers to salient structures at various levels of representation, contains much of the information needed in the reformulation process and thus serves as an important reference source for the system.

Reformulation of the hypothesis causes related changes throughout the several levels of Seek-Whence structures, changes made so that all levels of the system operate with the same pattern structure "in mind". These changes can in turn cause the bubbling-up or noticing of new perceptions about the sequence, creating an important interplay among the levels. Moreover, reformulation permits changes in the representation of concepts in order to facilitate the discovery of "structural" similarities (similarities of form) between them. Seek-Whence cannot as yet make such discoveries.

In Seek-Whence, concept descriptions are not necessarily atomic entities; they can be compound structures created by combining the primitive, atomic concept descriptions in simple or complicated ways. Thus, concepts can be related because their descriptions share the same or related structural building blocks, an important feature reflecting a similar human ability.

For example, we can sense that there are similarities among:

1 7 1 1 8 1 1 9 1 ...

2 3 2 2 4 2 2 5 2 ...

1 0 0 1 2 0 0 2 3 0 0 3 ...

even though the "face value" content, the actual numbers used, differs. Seek-Whence's facility for making such structural similarities manifest in its concept descriptions could prove very useful in the future for discovering analogies between sequence pattern concepts.

3. PROCESSING AND TASKS

A word about processing technique is in order. In Seek-Whence, all operations are carried out in task series, which run in simulated parallel. The tasks comprising various series are chosen at random for processing, so no assumptions can be made about which of two competing tasks will run first. In fact, the tasks in a given series may vary, because any task may alter the environment. A particular task may create, access, or modify some data structure, may request information from the user, or may set out other tasks -- place them on the taskrack, where they will stay until chosen and run.

A biological metaphor -- thanks to Douglas Hofstadter -- is the model for this type of processing. Within a cell, various enzymes are present. One of these may act upon a molecule, causing it to change in some way. This action will make the molecule more attractive to some enzymes and less attractive to others, thus affecting the course of later "processing" in the cell. The gloms in Seek-Whence's cytoplasm serve as molecules and the tasks as enzymes in our version of this biological model.

At certain times some cleaning is done (removing old tasks of certain types) but it is perfectly possible for an old task finally to be chosen and, when it runs, for it to find the Seek-Whence world quite different than when it was created. Such a task will probably do nothing, because the structures on which it was designed to operate no longer exist or are inaccessible to it. All tasks have "urgencies" (integer weights), and more urgent tasks will have a greater chance of being chosen than less urgent ones do.

Seek-Whence, then, depends upon order to emerge from chaos. Small special-purpose tasks, working at different levels of abstraction with no overseeing agent eventually guide the system toward convergence upon a working hypothesis.

4. STRUCTURES AND THE "PLASMS"

Like most other computer systems, Seek-Whence relies heavily on an assortment of data structures. Already mentioned was the hypothesis, an active formulation of the system's current view of the evolving pattern. Its lower-level counterpart, the <u>template</u>, is a transitory, weaker, less expressive, and less flexible structure used as a first rough statement of an emerging formulation. Below the template level are the central working structures of the system, namely, the <u>glints</u>, the <u>gloms</u>, and the <u>gnoths</u> (pronounced "knots", since they are used to "tie things together").

Briefly, glints are Seek-Whence representations of input terms, members of the class "Glints". The Glints form a distinguished subclass of the "Gloms" class, with glint structures being atomic and undissolvable. Gloms are structures representing collections of adjacent glints, hierarchically grouped for a variety of reasons. "Glomming" is the process by which two or more existing gloms combine to form a new glom. All gloms reside in the <u>cytoplasm</u>.

We view the Seek-Whence world as consisting of three levels, with two potential intermediating structures -- the hypothesis and the template. Our levels correspond to the Socratic vision of a "real world", a Platonic "ideal world", and a "perceived world" between them. At the lowest level is the cytoplasm, which represents our "real world" -- representations of the input integers and relations among them. The "purest" notions, corresponding to the Seek-Whence diagram "primitives", are housed in (of course) the <u>platoplasm</u>. Finally, the intermediate level -- our <u>socratoplasm</u> -- houses the system's representation of its parse of the sequence.

In order to represent its parse of a target sequence, the system needed structures with some permanence, so that a parse would remain intact, yet with the ability to interact with each other, so that the parse could be changed at the request of higher-level processes.

Gloms could not perform this parse-representation function for several reasons. First, they were designed to combine readily with each other, the combination occurring only because of "bottom-up" pressures. Secondly, when glomming occurs, the participants do not survive the operation. Rather, they are destroyed and a new glom is created from their subgloms. Therefore, the system cannot attach information to gloms and rely on its being available at any future time. Finally, gloms cannot change their basic structure in any way from the moment of their creation -- they cannot absorb other gloms or give away or recombine any of their subgloms without themselves being destroyed. Thus, gloms are too ephemeral and unpredictable to represent a sequence parse.

Our solution to the parse-representation problem was to create a whole new level in the Seek-Whence world -- the socratoplasm, or "perceived world", and to populate it with more stable structures than gloms, called "gnoths", amenable to "top-down" change. Gnoths, like gloms, represent groupings of gloms, but they are more permanent and a single gnoth can represent different glom clusters at different times. A gnoth can pass gloms to its neighbors, can withstand reformulation of its subgloms, and can even become a representative of concepts different from those associated with it at the time of its creation. The entire nature of a gnoth may change several times during its lifetime. A gnoth ceases to exist only when its subglom collection is empty. (For an interesting discussion of intensionality and the "meaning" of a representation structure, see [Hofstadter 80].)

Gnoths, then, are what the system uses to represent and restructure its current view of the sequence. They live in the socratoplasm, the middle level of Seek-Whence structures, and serve as bridges among the gloms, the hypothesis, and the "ideal notions" of the platoplasm [see Figure 1].

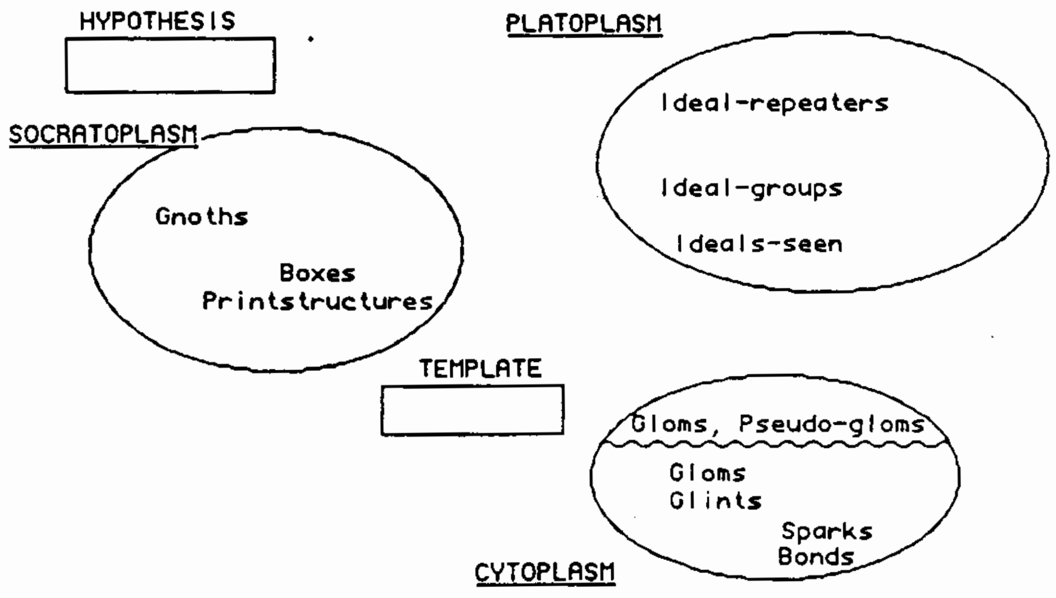


Figure 1 — The Seek-Whence world

THE PLATOPLASM AND IDEALS

As noted above, the platoplasm is the home of the "pure Platonic notions" or "ideals" of the Seek-Whence system. These include idealized versions ("types", if you will) of the input integers ("tokens" represented in the cytoplasm), the grouping structures known a priori, and some relations among them. Ideals are connected to the "real world" or cytoplasm through manifestation links and to the "perceived world" or socratoplasm through actualization links. For example, if the system groups three 2's, the glom representing this grouping in the cytoplasm becomes a "manifestation" of the ideal sameness notion (called "C-group"). If the glom is also crucial to the system's hypothesis for the sequence and so has a gnoth devoted to it, that gnoth becomes an "actualization" of the ideal [see Figure 2].

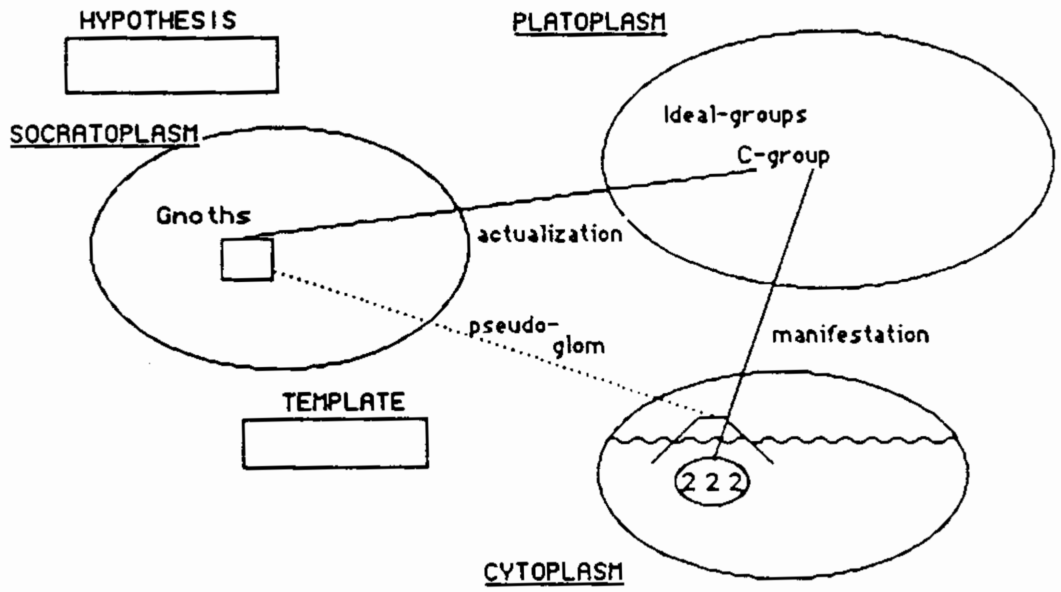


Figure 2 -- Some links between plasms

THE SOCRATOPLASM AND PERCEPTIONS

The socratoplasm is a working area and sometime battleground between the cytoplasm and the platoplasm. Its most important structures are the gnoths (from "gnothi seauton", the motto -- "know thyself" -- of the Socratic school of philosophy), roughly the socrato-level equivalents of the cytoplasm's gloms. However, whereas the hallmark of gloms is their ephemeral nature, their proclivity to combine and split, the main function of gnoths is to capture the current "view" -- parse or parenthesization -- of the sequence. The first gnoths are created contemporaneously with the first hypothesis and reflect its pattern description. From then on, gnoths must always be in agreement with the hypothesis (as described later in our sections on gnoth-hypothesis equivalence). Thus, gnoths are not as free as gloms to simply combine at will. They feel the "top-down" pressure of the hypothesis as well as the "bottom-up" pressure coming from lower-level activities. Conflicts between these pressures must be resolved through the gnoths. Each gnoth has a collection of subgloms from which it derives its structure, its view of the sequence. Any change to the gnoth's structure is realized by changing the subglom collection.

Much bubbling and pushing-up of groupings goes on in the cytoplasm. The requirement of conforming to the current hypothesis, with the related subsequent downward pushing and glom destruction, is added in the socratoplasm. Reformulation is implemented, also to be reflected in cyto-level activity. Mindlessness ceases here.

THE CYTOPLASM AND GLIMMERINGS

The cytoplasm is the bottom level of the Seek-Whence system. All changes to higher-level structures filter down here, are reflected here, and cause reactions which may "bubble up" new structures. All processing here is automatic and myopic -- no global views of the sequence are maintained. Any new structures, such as a newly-formed glom, or the glint made when a new term is entered, are immediately made centers of interest called active foci, and heightened activity takes place around them.

The main goal of this level is to hit upon a pattern of gloms which can be taken as a template for the sequence. Once a template is in place, a "cap" is, in effect, placed over the the top-level gloms to prevent their disappearance. This cap is in the form of a pseudo-glom, a glom that has subgloms but that is inert, unable to interact with other gloms. When such a cap is in place, new terms' glints still bond and even glom with other gloms but no changes that contradict the template can be made. Should a new term not fit the template, a review is begun which may lead to template modification or destruction.

If, meanwhile, the template has caused the formation of a hypothesis, new terms are filtered through the hypothesis rather than the template, and the template is virtually abandoned in deference to the more malleable, more expressive structure. The filtering process, by which new terms are checked for consistency with the hypothesis, can question the hypothesis' validity. This, in turn, can cause gnoth changes which may precipitate glom-, template-, and even hypothesis-modification or rejection.

5. SUMMARY

In summary, Seek-Whence is a program which attempts to discover and represent rules underlying nonmathematical sequences of nonnegative integers. It is not always successful. When it is successful, the complex but subliminal first stage of processing develops a hypothesis, an encapsulation of the perceived underlying pattern. The hypothesis is represented in such a way as to make the reformulation often found necessary in the second stage of processing not only possible but also simple to carry out in many cases.

Processing occurs in simulated parallel on several levels of abstraction at once. At the lowest level of the Seek-Whence world, the cytoplasm, glimmerings of grouping-ideas (gloms) derived from comparisons among the system's representation of the input terms (glints) are "pushed up" to be recognized as perceptions (gnoths) in the socratoplasm, the middle level. Recognition of useful cyto-groupings is aided by reference to the ideal notions of the platoplasm, the most abstract level. Suggestions continually bubble up, either to be pushed up further or to be rejected, sent back down. The interplay of bottom-up and top-down processing is central to the system's functioning.

C. SEEK-WHENCE IN DETAIL

1. THE PLATOPLASM — ABSTRACT NOTIONS

Currently, the platoplasm houses the ideal types — the primitive, built-in notions available to the system for use in constructing its view of a sequence. These can be seen as its vocabulary for the well-structured "phrases" it constructs. New notions, the newly-constructed phrases, may eventually come to be housed in the platoplasm as first-class citizens. We are developing a network of relations among the ideal types to aid the system's reformulation efforts.

ATOMIC IDEAL TYPES

The ideal-atoms are Seek-Whence analogues to the integers entered at the keyboard. An ideal-atom has predecessor and successor values, its own value, and manifestations in the cytoplasm. For example, the "ideal5" has predecessor "ideal4" and successor "ideal6", while "ideal0" has no predecessor but has "ideal1" as successor. Glints are certainly manifestations (cytoplasm-level instances) of these ideals, because they are the system's representations of the terms of the sequence. For example, in the sequence fragment "5 0 5", the first and last terms are manifestations of ideal5, while the middle term is a manifestation of ideal0. Certain other integer-valued quantities, such as group length, may also be important to the development of a good representation of a given sequence pattern, and so should also be viewed as manifestations. For example, the length of the group "(4 4 4 4 4)" might prove to be an important manifestation of ideal5. In the current version of the system, however, only glints are referenced as manifestations of ideal atoms -- a simplifying (and weakening) design decision. In the future, we hope to address the problem of what other quantities should be viewed as manifestations and under what circumstances they become important.

NON-ATOMIC IDEAL TYPES

There are eight non-atomic ideal types, each of which is associated with a <u>format</u> having one or more active parameters:

(typename start-value length actual-value).

In our descriptions, optional parameters which have been included will be given in brackets ("[ ]"). In each of the examples below, we show a <u>form</u>, an instantiation of the given format. When such a form is queried -- or "hit", as we say -- it returns a value. The results of successive hits are shown on separate lines. Note that these types and their formats correspond quite closely (but not exactly) to the Seek-Whence diagram primitives introduced in Chapter One.

Constant --- a structure that always returns one value, its argument, when queried.

format: (Constant arg)

examples:

```
(Constant 3) ---> 3
             ---> 3
             ---> 3
                  etc.
```

Countup --- a structure that returns nonnegative integers in succession, starting with its argument, when queried.

format: (Countup n)

examples:

```
(Countup 3) ---> 3
            ---> 4
            ---> 5
                 etc.
(Countup 8) ---> 8
            ---> 9
            ---> 10
                 etc.
```

C-group (Copy-group) --- a structure that returns a number of copies of a given argument.

format: (C-group start length)

examples:

```
(C-group 2 3) --> (2 2 2)
              --> (2 2 2)
                  etc.

(C-group (Countup 1) 2) --> (1 1)
                        --> (2 2)
                        --> (3 3)
                            etc.
```

S-group (Successor-group) --- a structure that returns a given-length run of successive integers, starting with a given value.

format: (S-group start length)

examples:

```
(S-group 2 3) --> (2 3 4)
              --> (2 3 4)
                  etc.
(S-group 5 4) --> (5 6 7 8)
              --> (5 6 7 8)
                  etc.
(S-group (Countup 1) 2) --> (1 2)
                        --> (2 3)
                        --> (3 4)
                        etc.
```

P-group (Predecessor group) --- a structure that returns a given-length downward progression of nonnegative integers, starting with a given value.

format: (P-group start length)

examples:

```
(P-group 8 4) --> (8 7 6 5)
              --> (8 7 6 5)
                 etc.
(P-group 2 4) --> undefined (would run to
                 negative numbers).
```

Y-group (Symmetry group) --- a structure that returns a given group of nonnegative integers, symmetric about the center.

format: (Y-group [start] [length] actual)

examples:

```
(Y-group 1 5 (1 8 3 8 1)) --> (1 8 3 8 1)
                          --> (1 8 3 8 1)
                             etc.
(Y-group ((Countup 1) 8 (Countup 1)))
          --> (1 8 1)
          --> (2 8 2)
          --> (3 8 3)
             etc.
```

Cycle --- a structure that cycles through its actual parameter's value, returning one top-level element each time queried.

format: (Cycle actual)

examples:

```
(Cycle (2 1 8)) --> 2
                --> 1
                --> 8
                --> 2
                --> 1
                  etc.


(Cycle (3 (Countup 1))) --> 3
                        --> 1
                        --> 3
                        --> 2
                        --> 3
                        --> 3
                        --> 3
                        --> 4
                          etc.
```

Tuple --- a structure that returns its actual parameter's value each time queried.

format: (Tuple actual)

examples:

```
(Tuple (1 8 4)) --> (1 8 4)
                --> (1 8 4)
                  etc.
```

```
(Tuple (2 2))  ---> (2 2)
               ---> (2 2)
                   etc.
```

THE HIERARCHY OF IDEAL TYPES

These non-atomic ideal-types (or Platonic classes) fall into a hierarchy of categories, each of which captures an important organizing notion for the Seek-Whence world. The realizations of the types at different levels of the system have differing attributes but always reflect this basic organization.

Briefly, the categories can be distinguished as follows:

repeater type -- These are one-parameter generate types; given the single parameter (and the state), the next value can be generated.

members: Constant, Countup

generate type -- Given a typename and the start and length parameters (e.g. ,(C-group 1 3)), the actual value (e.g., (1 1 1)) can be generated by the associated "generating function".

Any generate type possesses a process as described below.

members: C-group, S-group, P-group

process type -- Possesses a "process", a method of determining whether or not some actual group is a representative of the class without reference to any information external to the group and the class in question.

member: Y-group

fence type -- Has no generator, no process; virtually any collection of neighboring terms can be called a group by virtue of these types. Typically, such groups exist because of external pressure from neighboring terms or groups rather than internal cohesion. In effect, the terms are "fenced off" into a group by their neighbors.

members: Cycle, Tuple

At the bottom of the hierarchy are the fence types, the least restrictive types. The name derives from the fact that groups are usually identified as being of this type when the system:

1) cannot classify them as being higher-level types and

2) can set up a "fence", identifying the group as a group.

For example, in the parsed sequence 1 5 8 3 2 5 8 3 3 5 8 3 4 5 8 3..., the terms "5 8 3" are grouped, not because of any mutual attraction or shared characteristic, but simply because of the interleaved 1, 2 , 3,... and the fact that the group repeats. It is important to note that in order to see the repetition of the group, it is necessary to identify it as a group, and such recognition of repetition in effect confirms the budding notion that a group is there to be found. The group would probably be represented as "(Tuple (5 8 3))".

An entire sequence can have a fence-type representation:

4 7 4 7 ... can be represented as (Cycle (4 7)), with an understood repetition.

At the next-highest level of the hierarchy are the process types. The only entry here is Y-group, a symmetry group. The characteristic of this class is that it possesses a "process", a method for identifying representatives of the class, if not for generating them. The form "(Y-group 1 5)", a Y-group of length 5 starting with a 1 (we have given this form the optional start and length parameters), is not sufficient to generate a unique symmetry group, but is sufficient to determine that (1 7 4 7 1) is such a Y-group whereas (1 8 2 5 1), (1 7 1), and (2 0 0 0 1) are not.

The generate types, the next-highest in our hierarchy, can use two parameters, the starting value and length, to generate a particular grouping representative of the given type. For example,

(C-group 1 5) <—> (1 1 1 1 1), a constant group of five 1's;

(S-group 2 3) <—> (2 3 4), a successorship group of length three, starting with 2;

(P-group 9 4) <—> (9 8 7 6), a predecessorship group of length four starting at 9.

Each repeater type takes one parameter. The form "(Constant 2)" represents a structure that always returns 2 as its value, while "(Countup 3)" represents a structure that always returns a 3 upon first request, then a 4, a 5, and so on.

COMBINING IDEALS

Ideal types can be combined to create structures which encapsulate fairly intricate patterns. In the examples below, each line again represents one hit of the given form. Shared structures are indicated by the word "shared", with an arrow pointing to the first instance of the structure to be shared.

```
(S-group 1 (Countup 2))  --> 1 2
                         --> 1 2 3
                         --> 1 2 3 4
                       etc.,
```

thus giving the sequence:

1 2 1 2 3 1 2 3 4 . . .

```
(C-group (S-group 1 (Countup 1) 2) --> (1 1)
                                   --> ((1 2) (1 2))
                                   --> ((1 2 3) (1 2 3))
                                     etc.,
```

giving:

```
1 1 1 2 1 2 1 2 3 1 2 3 . . .
```

```
(Cycle( 8 (Countup 1) shared))      --> 8
                                    --> 1
                                    --> 2
                                    --> 8
                                    --> 3
                                    --> 4
                                      etc.,
```

giving:

```
8 1 2 8 3 4 8 5 6 . . .
```

```
(Tuple (2 (Countup 1) 2))     --> 2 1 2
                              --> 2 2 2
                              --> 2 3 2
                                etc.,
```

giving:

```
2 1 2 2 2 2 2 3 2 2 4 2 2 5 2 . . .
```

```
(Tuple  (2 (Countup 1) shared))    --> 2 1 2
                                   --> 2 2 2
                                   --> 2 3 2
                                     etc.,
```

giving:

```
2 1 2 2 2 2 2 3 2 2 4 2 2 5 2 . . .
```

The difference between the latter two representations is subtle but can be important. In the last one, the sameness of the "bracketing" 2's is made explicit. Notice how this can affect generalizations of the sequence:

(Tuple (5 (Countup 1) 2)) is a possible generalization of the first representation because the bracketing integers are seen as distinct, having no necessary sameness. Querying it three times will give us:

```
--> 5 1 2
--> 5 2 2
--> 5 3 2
  etc..
```

In the second form, we generalize to 5 as follows:

```
(Tuple  (5 (Countup 1) shared))    --> 5 1 5
                                   --> 5 2 5
                                   --> 5 3 5
                                     etc..
```

The ability to combine the Platonic notions as demonstrated gives the system the flexibility and expressive power needed to model sequence patterns and create a hypothesis.

2. THE HYPOTHESIS

The major goal of the Seek-Whence system is to formulate a hypothesis -- a structure that describes and can extrapolate the perceived pattern. The hypothesis, which is derived from the information at hand -- the sequence terms seen and any relationships the program has been able to establish among them -- is expressed in terms of the Platonic classes described above. For example,

1 1 2 1 2 3 1 2 3 4 can be expressed as the form:

(S-group 1 (Countup 1)), while

2 1 2 2 2 3 2 4 ... can be expressed as the form:

(Cycle (2 (Countup 1))).

Should a hypothesis fail to predict properly, the tendency of Seek-Whence will be:

a) to generalize parameters, maintaining the Platonic class structure;

b) to slip to a less strict class ("vertical" slippage) or to a related class ("lateral" slippage).

In the future, the system will have an implicit imperative to modify the hypothesis so that in all instances the strictest appropriate class is used in the representation. For example, while "1 1 1" can be viewed as a Tuple, it is also a C-group and should generally be so characterized. There are of course times when "1 1 1" *should* be viewed as a Tuple; for this reason there will be no prohibition against doing so, but it is very unlikely to be the first view adopted.

Both generalization and specification are required in Seek-Whence and both require knowledge of the grouping types described above and any relations among the Platonic types. Such relations will be stored in the platoplasm as "Ideal-relations" and will include lateral links (between C-group and S-group, say, for "groups" of length one) as well as vertical ones (as between Countup and S-group) in the plato-class hierarchy.

3. THE CYTOPLASM -- THE BASE

The cytoplasm has the role of "real world" in our "Socratic model", in which the platoplasm houses the analogues of Plato's Ideals and the socratoplasm is the analogue of Socrates' "perceived world". It houses the lowest-level structures in the system -- the sparks, bonds, glints and gloms -- and is the site of much upward-thrusting, relatively uncritical activity. Suggestions for pattern formulation bubble up from cytoplasm-level (hereinafter shortened to "cyto-level") activity to be tested at higher, more "cognitive" levels. We believe that the probabilistic, undirected cyto-level activity mimics low-level human perception processes to some extent. Groupings are continually being generated and regenerated at this level. Just as people cannot prevent themselves from reinventing an idea, reperceiving a pattern, or reperforming an action, but compensate for such repetition by an ability to notice that they are cycling or looping in their behavior, we will leave it to higher-level processes to notice and handle any unproductive looping in cyto-level activity.

The cyto-level should bombard the upper levels with suggestions, noted similarities, and groupings of terms. It is up to the processes above to curb this enthusiasm and to consider the suggestions more carefully and critically.

CYTO-LEVEL STRUCTURES

The four data types residing in the cytoplasm are Sparks, Bonds, Glints, and Gloms. The former two classes are for finding, proposing, and later evaluating glom groupings. The latter two classes, the Glints and Gloms, are used for representing the sequence terms and any term groupings of current interest.

GLINTS AND GLOMS

In Seek-Whence, glints are the cyto-level representations of the integers entered at the keyboard. Each glint is a structure with several fields: class, name, print-value, position, span, pred (or left-nbr), succ (or right-nbr), and bonds-in, the last one being optional. For example, if the terms "1 2 2 3" had been entered, the second 2 might be represented as follows:

class:	Glints
name:	glint3
print-value:	2
position:	3
span:	1
left-nbr:	glint2
right-nbr:	glint4

The "span" field is really unnecessary in glints, but is a consequence of the fact that the Glints class is a subclass of the Gloms. It indicates that this glom is of length 1. This glint's left-nbr, its neighbor to the left, would be the glint representing the preceding 2, called "glint2" here. Similarly, its right-nbr, its neighbor to the right, would be the glint representing the succeeding 3. The other fields are self-explanatory.

When a sequence term is entered, the system creates a glint for it and then lists that glint as a manifestation, or cyto-level analogue, of the appropriate ideal-atom in the platoplasm. In our example, glint3 would become a manifestation of ideal2 because its value is 2 and it represents an input integer.

The glint is then examined by cyto-level tasks as described below to determine how it is related to other cyto-level structures.

The class "Glints" is a distinguished subclass of the class "Gloms". Each glint is indestructible -- an "atomic" glom. Non-glint gloms are cohesive units, made of adjacent atoms bound by "bonds" of one type or another. It should be noted that chains of atoms linked by bonds are not necessarily converted to gloms; some bond types (e.g., one indicating that gloms (8 1 4) and (1 2 3) are of the same length) are generally not considered strong enough to cause glomming, but are facts of note preserved for use by higher-level processes.

Non-glint gloms have as fields: class, name, type, print-value, start-position, span, positions-covered, subgloms, structure, and bonds-in. The last two are optional, and are filled in when appropriate by cyto-level processes. For example, in the sequence segment "8 2 2 2 8", the three 2's might be represented jointly as a glom, as follows:

class:	Gloms
name:	glom7
type:	(Same print-value group)
print-value:	(2 2 2)
start-position:	2
span:	3
positions-covered:	(2 4)
subgloms:	(glint2 glint3 glint4)

Such gloms are ephemeral and can disappear at any time. Disappearance by dissolving (being destroyed as a unit, but with all subgloms surviving intact), bursting (being destroyed as a unit and having all non-glint subgloms burst as well -- leaving only the underlying glints), or glomming (being combined with another glom) is fluid and continual. The cytoplasm might be viewed as a soup bubbling with gloms, the bubbles which rise to the top being the system's current view of the sequence. If neighboring bubbles have enough mutual attraction (strong enough bonds) they will combine; otherwise they will either exist independently or burst to permit new bubbles to take their place.

BONDING AND GLOMMING

The identification and creation of useful gloms is the primary function of the tasks operating at the cytoplasm level. To see how this is done, we must start at the bottom and follow the process of "pushing up" gloms.

SPARKS AND BONDS

Sparks and Bonds, two more cyto-level classes (the others being the Glints and Gloms discussed above), are used during the early stages of group discovery. A spark is created between two gloms when a Sparkler task pulls those gloms at random from the cytoplasm and determines in a very cursory way that the two structures might be amenable to bonding. The Sparkler simply looks for gloms that are not subgloms of each other. It does not look for any common features -- this is the work of other tasks. For example, the glints "1" and "1" might very well be bondable, since they have the same print-value. Gloms "(1 2)" and "(1 2)" might be bondable for the same reason, or because they have the same "span" (length in sequence terms covered). The glint "1" might be bonded to the glom "(2 3)" by reason of adjacent successorship -- 2 is the successor of 1, and the structures in question are adjacent. However, no glom can be bonded to one of its own subgloms, so the glom "(1 (1 1))" cannot be bonded to the subglom "(1 1)" in any way.

BEYOND SPARKLING

When a spark is created between two gloms, a horde of "Testers" is placed on the taskrack. When invoked (chosen to run) at some later time, each Tester chooses some spark, not necessarily the one whose creation caused the tester's creation. The spark's members (the two gloms between which the spark is flying) are tested to ascertain if they are currently in existence (recall that gloms are ephemeral). If both gloms exist, their bond-fields — the characteristics such as print-value or span which are important enough to be used as a rationale for bonding -- are intersected, and these fields' values are tested for similarities. The system uses several types of bonds -- sameness, successorship, predecessorship, adjacency, and meeting (e.g., "(8 1 4)" and "(4 7)" "meet" at 4) -- grouped into families, to link gloms. The most important of these are, not surprisingly, sameness and successor-predecessorship. If a bonding test is passed, a "Bonder" task is created with the intent of performing the actual bonding. One Bonder task will be created for each bonding test passed by the two gloms, so several Bonders might actually be created for any given glom pair. For example, gloms "(1 2)" and "(1 2)" might engender both "same print-value" and "same span" Bonders.

BONDING

When invoked, a Bonder

1) checks to see that both gloms still exist, and

2) checks to see that they are not already bonded in this way.

If these conditions are satisfied, then the Bonder creates a Bond-class structure, which we refer to simply as a <u>bond</u>. This bond, which exists in the cytoplasm, links the two gloms and has a <u>strength</u> associated with it. Bond strength is derived from the bond type (e.g., "sameness"), any bond modifiers (adjacent terms are more strongly bonded than non-adjacent ones, for example) and the glom characteristic (e.g., "print-value") that is the subject of the bond. Creation of a bond causes the release of more Sparklers, stimulating the system to carry out more low-level search and bond creation, and causes the release of some Glom-scouts -- tasks designed to look for and push up glom groupings.

GLOMMING

Bonds are created in order to provide some basis for the grouping of sequence terms (glints) and term groups (gloms). The act of bonding simply reflects the fact that two gloms are related in some way. Glomming, however, is performed only when the bonds among two or more gloms are sufficiently strong that the system should view the items comprising the bond-chain as a unit. The system distinguishes between the "bond-fields" of a glom and its "glom-fields". <u>Bond-fields</u> are those characteristics of gloms that are to be compared for the purpose of bonding. Typically, the print-value and span are useful bond-fields. Thus, two gloms such as "(1 2 3)" and "(7 8 9)" or "(1 2 3)" and "(8 1 4)" will generally be bonded. But, although knowledge of the fact that two gloms have the same span is "interesting", it is generally not compelling enough to warrant glomming them in and of itself. In an early version of this system, we did allow such gloms. The result was a plethora of uninteresting gloms that seemed to get in the way of the system's real work. In fact, this was one of the main reasons for introducing the "bond-field"/"glom-field" distinction. <u>Glom-fields</u> are the glom characteristics that are important enough to use for glomming purposes. Only print-value is used as a glom-field in the current system. The system can make chosen glom characteristics more salient by designating them as bond-fields or glom-fields, or less salient by removing these designations; in practice this ability is not yet used.

As was noted earlier, bond creation causes the release of Glom-scouts onto the taskrack. These tasks look for glommable bond-chains. They also serve to introduce a good example of a terraced scan in Seek-Whence. The tasks introduced between this point in the dissertation and our discussion of "Plato-evaluator" tasks perform increasingly extensive tests on target gloms, screening the gloms as potential representatives of various Platonic classes. If a glom passes one test, it is targeted for further evaluation. Should a glom fail a test, it may be re-evaluated by other tasks. Gloms that are not discernably Platonic are either ignored or destroyed.

When invoked, a Glom-scout chooses a cyto-element (a glint or glom which is not a subglom of any other glom) and attempts to group it with its neighbors. Actually, three quicktests tests are made for any bond-family in which the glom is involved:

1) is it groupable? (bonded to any neighbors in this way?)

2) is it coverable? (bonded into a symmetry group?)

3) is it fenceable? (are there remote gloms to which it is bonded?)

Note that these tests are the precursors of the plato-level notions of generate, process, and fence classes. Any tests passed cause creation of a Glomtester task to make a more extensive test of the glom. The Glomtester's weight (urgency) is dependent on the test ("groupable" being strongest) and the bond-type involved (sameness being stronger than successorship, and so on). For example, if the terms "1 8 8 5 2" have been entered, the system may notice several relationships among various terms. The two neighboring 8's might be seen as a budding "sameness group" because of their adjacent sameness. However, the remote successorship between the 1 and the 2 might also be noted and used to propose a "successorship fence" group, one that would separate the given segment into gloms "(1 8 8 5)" and "(2)". Such groupings are potentially very important, especially if the sequence is:

1885 2885 3885 . . .

or the like, but are not as immediately compelling as such groupings as the pair of 8's. Therefore, a Glomtester for a sameness group is given a higher weight than a predecessor fence Glomtester. The system is thus biased toward noticing certain similarities first, yet it is not compelled to do so.

When invoked, a Glomtester task must first be certain that the glom it is supposed to test is still in the cytoplasm. If so, it must then determine the extent of the evolving glom. The thrust here is to get maximally-sized gloms.

For example,

1) in "2 1 1 1 3", with the second "1" targeted and "sameness" the bond-type, the Glomtester would suggest that "1 1 1" be grouped.

2) in "9 1 2 5 9", with the first "9" targeted and "fence" the reason, "9 1 2 5" would be suggested.

3) in "5 3 1 6 1 3 4", with "6" targeted and "symmetry" the reason, "3 1 6 1 3" would be suggested.

4) in "1 2 3 2 3", with the first "2" the target and the "pred-succ" bond family the reason, "1 2 3 2 3" would be suggested.

The Glomtester either rejects the group as a glom or creates a Glommer task to refine the group and perform the final glomming.

When invoked, the Glommer will do a bit of "bookkeeping". It creates the new glom and makes it an active focus -- a site of increased system activity. It also creates a Glom-inspector task to continue pushing the glom up to higher levels.

4. PLATO-CYTO RELATIONS

The Seek-Whence system has an imperative to find analogues of its plato-classes. At the low levels of processing discussed thus far in this dissertation, that drive has been realized, in a procedural and uncritical way, by the nature of the system's tasks. Above these levels, some declarative knowledge is used; some manifest reference is made directly to the ideal-types to begin, if not rejecting gloms, then favoring those that seem the purest analogues of the ideals. When found, these special gloms will be "dubbed". All others will be put on a track towards destruction. Plato-scouts perform the first step in this process.

A Glom-inspector determines which, if any, plato-classes might find the given glom "interesting" -- which classes might possibly consider it a "manifestation" of themselves. If there are any such classes, the Glom-inspector then creates a Plato-scout task, giving it the glom in question and the names of the "interested" (candidate) plato-classes. If the glom still exists when the Plato-scout is invoked, the scout begins its work.

During the glomming stage, maximally-sized groups of gloms -- all "chains" that consist of gloms related to their neighbors by some element of a bond family -- are collected. For example, "1 2 3 2 3" could be a "pred-succ-family" glom. The Plato-scout stage will now focus on "purifying" these groups.

Recall that all but the fence-type plato-classes (Tuple and Cycle) possess a "process" -- a function which, when given a number string, determines whether or not the string is an instantiation of the class. A Plato-scout is given a glom and a list of candidate plato-classes. It applies the process function for each candidate class to the glom's print-value. If the glom passes the test, then the glom is pure and will be dubbed a manifestation of the candidate class. A glom can be dubbed more than once. The glom "(1 1 1)", for example, might be dubbed as both a C-group ("copy-group") and a Y-group ("symmetry-group").

If the plato-class is a fence type (and so has no process), and if the glom is "flat" -- has only glints as subgloms -- a "pass by default" occurs. This permits flat potential Tuple and Cycle gloms to be dubbed as such.

Any glom that does not pass even one of its process tests is cause for the creation of a Plato-evaluator task. When invoked, the Plato-evaluator examines the glom a bit further, looking for dubbable gloms of secondary purity (non-flat) and for pure subcollections of gloms within the proposed target glom. Thus "1 2 3 2 3" might well be sent to a Plato-evaluator, which might break it into the two pure successorship groups "1 2 3" and "2 3", and send on the gloms for these two groups to be dubbed. If the Plato-evaluator has no success and the glom has not already been dubbed as a manifestation of some other platotype, the scene is set for its destruction. A Burster task is created to destroy the glom and all its non-glint subgloms.

DUBBING THE PURE

If some glom is deemed "pure" by a Plato-scout, the Plato-scout calls for the glom to be "dubbed". This is a two-step process:

1) the glom's structure field is modified to indicate

   a) new structure: (platotype [start-val] [length] [value])

      e.g.,(C-group 1 3) or (Cycle (2 9 8 1))

   b) purity: pure <--> an exact manifestation; flat; e.g., (1 1 1)

      secondary <--> not flat ; e.g., ((1 5) (1 5) (1 5))

2) the Platonic ideal has this glom added to its list of manifestations.

Dubbing causes creation of a Template-scout process, indicating that the glom is strong enough to warrant a check to determine whether or not its structure, as described in the structure field added during dubbing, might yield a pattern for the entire sequence.

Those gloms that do not pass any of the tests leading to dubbing are targeted for destruction by a Burster or a Dissolver task. A Dissolver is a task that destroys a glom, leaving its highest-level subgloms to float independently in the cytoplasm. For example, if the glom "((1 1) (2 2))" were to be dissolved, the underlying gloms "(1 1)" and "(2 2)" would survive, but would, of course, no longer be glommed with each other. A Burster task is even more destructive of glom structure. If a Burster were set on the glom "((1 1) (2 2))", all levels of glomming would be destroyed, leaving only the glints "1", "1", "2", "2" in the cytoplasm.

Once a Burster or Dissolver has been created on a glom, the glom's only escape route is to become invisible in the cytoplasm by glomming with another glom. Thus, after reaching the point of perusal by a Plato-evaluator, a glom will either a) be dubbed; b) be destroyed and have some subgloms dubbed; or c) be targeted for destruction. The Plato-evaluator creates no other tasks. Our terraced scan has come to an end.

5. REVIEW AND PREVIEW

Thus far in this thesis we have discussed all the major cyto-level tasks and structures. Before moving on to a discussion of other levels, it might be well to get an overview of what remains and how it relates to what we have already done.

When people work on sequence or Bongard problems, they usually progress through several stages. At first, they see and recognize new terms as the terms are revealed. Then they make linkages between new and previously-encountered terms, and begin to make tentative groupings of terms in an effort to "come up with something". This essentially data-driven activity is modeled in the cyto-level processing that we have just discussed. Actual terms and real, undisputable relationships (e.g., "adjacent successor") are used as the basis for creating, rather haphazardly and nondeterministically, the ephemeral structures,"held tremblingly in the hand", known to Seek-Whence as "gloms".

The next step in human sequence-solution activity is to answer the question, "What is it I've seen?", or better still, "What is it I *think* I've seen?". The corresponding processing level of Seek-Whence, the template level, makes a similar attempt to realize or identify what the system "perceives" that it has seen. In the process of doing this, it tries to create a "template" for the sequence -- a first rough approximation of the developing sequence pattern-description. This is a stage where we try "to get a handle" on the pattern for internal processing purposes. People operating at this stage will often say something like, "Wait -- I think I've got it . . . no, maybe not." The description is a tentative one, not believed too firmly, but nonetheless a sort of crystallization of current perception. The happiest possible outcome from this stage is a parenthesization of the sequence in accord with the developing and now more firmly held and more explicit pattern description. In Seek-Whence, this happy outcome means the creation of a hypothesis -- the more-firmly-held description -- and the creation of gnoths -- the parenthesization.

A Seek-Whence hypothesis is the closest analogue the system has to a verbalization of the sequence pattern. A human sequence solver, perhaps after one or more false starts, will eventually announce triumphantly, "I think I've got it!". At this point, or certainly by the time the description is verbalized, the subject's pattern description has probably crystallized completely. This description is (usually) firmly held, is predictive, and can be communicated clearly to others -- either by some encapsulation method (e.g.,"three 1's and a

2") or by reciting the terms in a patterned or sing-song manner. Seek-Whence hypotheses (which are described in detail later in this paper) have similar features; the system has not yet begun to sing them, though.

As mentioned earlier, we view the gnoths -- collectively, our parenthesization of the sequence -- as existing in a place we call the "socratoplasm", somewhere between the "real world" of the cytoplasm and the "ideal world" of the platoplasm. If we view the cytoplasm as data-driven and its structures as "real", and the platoplasm as theory-driven with "ideal" structures, then the socratoplasm is what we will call "perception-driven" and its structures "perceived". In the socratoplasm, Seek-Whence must reconcile theory with reality, and thus must in effect answer the question, "Does what I think I've seen make sense?". The gnoths will always agree with the hypothesis to some extent, but may fail to be fully consistent with it. Similarly, there may also be some temporary disagreement between the gnoths and the gloms that they in effect "represent". This rather unpleasant-sounding state of affairs is a consequence of the necessary state of flux at this level. If the hypothesis is changed -- if, for example, the system now wants the segment "1 2 2" parenthesized as "1 (2 2)", whereas it used to be parenthesized as "(1 2) 2" -- the system will have to propagate that change down through the various processing levels. The socrato-level is the level possessing the vocabulary in which to express those necessary changes. It is the level at which reformulation begins to be brought about.

Now that we have some foreshadowing of future developments, it is time to return to our more systematic discussion of Seek-Whence processing. We left off at the point when template-level processing was about to begin, the stage of "casting around" for an appropriate formulation of the sequence pattern description.

6. TEMPLATE CREATION -- ONE MOLD TO FIT ALL

Whenever a glom is dubbed as a manifestation of some plato-type, a Template-scout process is placed on the taskrack. It and other processes involved in template creation and evaluation operate at an intermediate level between the "real", data-driven cyto-level and the "perceived", perception-driven socrato-level. The human analogue is the stage during which a person's eyes move back and forth across the terms, as the person waits for an idea to emerge. This is a stage in which people can literally observe themselves work, yet be unable to explain verbally what is happening, what they are "thinking". People working on Bongard problems experience this stage in an especially clear and forceful way.

In Seek-Whence, the template-level processes attempt to come up with a template or descriptor of the sequence. This is a preliminary step to devising a hypothesis -- that is, a predictive model of the sequence, an encapsulation of its structure. Templates and hypotheses have similar forms, but templates are far less complete and exact, lacking the predictive ability and expressive power of hypotheses. A good, working template will eventually give rise to a hypothesis.

A template is formed when the structure of some particular dubbed glom is found to explain, at least roughly, all the sequence terms seen thus far. For example, the template form "(S-group 1 n)" suffices to explain "1   1 2   1 2 3" since it "fits" all the term groupings, even though there is no built-in notion or even any recognition that "n" means "countup" here. The same template would suffice equally well for "1 2   1 2 3 4 5   1   1 2 3". The ability to notice "cross-glom" properties, such as n <--> countup, is left to higher-level processes.

Once a template is created, it puts a pseudo-glom called the <u>template glom</u> over the highest-level gloms in the cytoplasm (those that are not subgloms of any other glom) to prevent the disappearance of the gloms that engendered and now reflect it. A pseudo-glom cannot combine with real gloms, and it prevents its subgloms from glomming activity as well. Some cyto-level activity can still continue -- bonding being done as freely as ever, for example -- but no new templates are considered for the lifetime of the given template.

The template will be checked by a Template-evaluator task and either be passed, or rejected and abandoned. If it is passed, it will probably be the basis for hypothesis and gnoth creation. This means that until a hypothesis exists, all new terms will be "filtered" past the template, checked for agreement with it. Should a term not fit the template, a review is set up, with resulting modification or rejection of the template. The filtering process is the first major top-down action performed by the system; the template level has taken control. This is *not* to say that cyto-level activity ceases or slows; the cyto-level processes continue in their accustomed way. What is added is direction from above: instructions to make or dissolve gloms, to create units of a particular form.

TEMPLATE DIFFICULTIES

The process of devising a template is not as easy as it might first appear. For example, suppose that the sequence terms "1 1 2 1 2 3" were entered and the first "1 2" were glommed, dubbed as "(S-group 1 2)", and targeted by a Template-scout. When invoked, the scout would set out a Template-applier task to determine if the entire sequence seen is of that form. The applier would attempt to view the sequence as a repetition of "(S-group 1 2)" and would, of course, fail because of the initial "1" and the trailing "3". The applier does not give up immediately, but rather checks to see if loosening a parameter or two in its representation would help. In this case, changing the form from the original "(S-group 1 2)" to "(S-group 1 n)" -- where "n" means "any nonnegative integer" -- will do the trick. The accepted template will then be "(S-group 1 n)".

If the Template-applier fails, it creates a Template-resolver task for one final attempt. When invoked, the resolver looks at glints rather than gloms to determine whether or not the sequence can be re-viewed to fit into the given mold. For example, if the segment "1 1 2 1 2 3" were glommed as (1 1) 2 (1 2 3), the Template-applier would fail because of the first glom. A Template-resolver working with the template "(S-group 1 n)", however, might be able to find the appropriate S-groups by looking at the sequence terms rather than the gloms. If the Template-resolver is successful, it "blasts" (does an immediate burst of) all gloms and has the glints reglommed to fit the template. This is a fairly radical action in that it ignores all the cyto-generated glom units, but it does provide some potential for destroying "locked-in" gloms, ones the system created and can never seem to burst. If the proposed template does not work at term level, it is forgotten and the engendering glom dissolved.

In practice, the Template-resolver is seldom invoked because the system can usually devise a template early on which is good enough to push up a hypothesis. Once that has happened, the higher levels take over the job of resolving problems. It is in the spirit of Seek-Whence processing to give each level a little more capability than it should need to use -- the ability to handle, albeit lamely, situations that would be better handled by higher-level processes.

The Template-reviewer process is in this category. It is invoked after a template has been created (but no hypothesis exists) and when new terms fail to fit the template. It can try some very simple fixes and can either:

1) call for modification of the template and restart the creation and evaluation processes;

2) leave the template alone;

3) target it for abandoning .

The creation and acceptance of a template causes increased activity in the system, in effect "raising the temperature" in the system. Most importantly, it sets off two tasks, a Gnoth-maker and a Hypothesizer. This action pushes processing up into the next level, the socrato-level, where more considered operations are performed on the fluid but less ephemeral structures of the socratoplasm.

7. THE SOCRATOPLASM -- IN THE MIDDLE

The socratoplasm is the "perceived world" of Seek-Whence, the place where perceptions developed at the cyto-level are noticed, catalogued, and dealt with. It can be viewed as a battleground between the "ideal" plato-notions and the "real" cyto-glimmerings -- that is, between the semantic and the syntactic -- or -- to put it one last way -- between the cognitive and the subcognitive. In any case, it is the system's playground, where perceptions can be modified and manipulated; in short, it is where slipping occurs.

For emphasis, we should note once again that operations carried out at the socrato-level inevitably cause cyto-level activity. This is very desirable. Such low-level activity may result in the noticing of a special bond or the creation of a new glom which might eventually engender a better parse.

A single cyto-level task is too low-level to control its own or the system's processing directly (although in aggregate these tasks are very influential). In contrast, the socrato-level can and does support tasks which say, in essence, "Enough! I have a hypothesis. Let's have the next term to check it out", or better still, "I think the answer is .... Tell me if I'm wrong."

As was previously noted, the acceptance of a template signals the system's readiness to consider creation of a hypothesis -- an encapsulation of the sequence's structure. Although this goal may not yet be attainable, a correct hypothesis not forthcoming, the highest-level processes should now be introduced into the fray. At this point, malleable, manipulable, relatively non-ephemeral structures are needed so that any necessary slipping can be noticed and carried out. Moreover, a reformulation vocabulary must be developed so that the system can express clearly and succinctly the actions it needs to take. Thus, the structures we call "gnoths" are created.

GNOTHS

Each gnoth, a member of the class Gnoths, is viewed in three different ways:

1) it is an actualization of a Platonic class;
2) it has an underlying glom collection from which it derives its structure;
3) it represents one "hit" of the current hypothesis (if there is one).

When a Gnoth-maker task, set off by the system after template creation, is invoked, it creates one gnoth for each subglom of the template-glom and notifies the associated plato-classes of their existence.

For example, in the sequence "1 1 2 1 2 3", where we might have gloms:

glom2: (1)

glom7: (1 2)

glom4: (1 2 3)

and template "(S-group 1 n)", the Gnoth-maker would create three gnoths:

gnoth1

class: Gnoths	
name: gnoth1	
frame: 1	this gnoth holds the first hypothesis "hit"
plato-class: S-group	
glom: glom10	(where glom10 has glom2 as subglom)
	the gnoth's "pseudo-glom"
range: (1 1)	the sequence terms it "covers"

gnoth2

class: Gnoths

name: gnoth2

frame: 2

plato-class: S-group

glom: glom11 (where glom11 has glom7 as subglom)

range: (2 3)

gnoth3

class: Gnoths

name: gnoth3

frame: 3

plato-class: S-group

glom: glom12 (where glom12 has glom4 as subglom)

range: (4 6)

Each gnoth places a pseudo-glom (called a "gnoth-glom") over its glom collection (which contains just one glom initially). A gnoth-glom, like a template-glom, cannot glom with other cyto-elements and serves to prevent the haphazard disappearance of glom structures important to the system. In this case, since the underlying glom collection gives the gnoth its character, that collection must be preserved until the gnoth itself must change. Cyto-level bonding activity can continue but now the gnoth oversees the fate of its gloms.

Cyto-level tasks are somewhat myopic, able to view the sequence only in a restricted, localized way. They have no overview of the sequence. The structures -- the gloms -- created at the cyto-level reflect this myopic view. In contrast, the hypothesis and platonic-level processes can be said to have no "underview" of the sequence, no direct contact with reality as it exists in the cytoplasm. Gnoths are designed to bridge the gap between these levels, to provide a place where inconsistencies between the high-level and low-level views can be worked out.

8. HYPOTHESES -- ENCAPSULATING PATTERNS

The overall purpose of the system is to develop a reasonable hypothesis: a clean, predictive model of the rule underlying the sequence. When a template is accepted, a Hypothesizer task is set off along with a Gnoth-maker, described above. When invoked, the Hypothesizer is responsible for devising a hypothesis for the sequence, based on the template and the existing gnoths (if any) and gloms. If, for some reason, there is a faulty template (or none at all), the Hypothesizer can take the fall-back position of declaring the sequence to be a Tuple, the weakest of all plato-classes.

Because the Hypothesizer's model, like those developed by humans, may turn out, as more terms arrive, to fail to be predictive, or may be judged "clumsy" or "ugly", it must also be easy to change. Thus, hypotheses must be not only predictive and clean, but also amenable to reformulation -- "slippable".

Naturally, Seek-Whence must be able to notice when reformulation is called for, to know why it should be done, to know what changes to make, and to know how to carry out these changes. It becomes obvious, then, that hypothesis structure is critical, in that it can make or break the system's ability to carry out these tasks.

The predictive nature of a hypothesis is a semantic rather than a syntactic requirement, and so poses few constraints on hypothesis form. The other two goals -- clean representation and slippable form -- do give us something to work towards. A hypothesis must have sufficient expressive power to represent the observed regularity accurately. It should have a clean visual appearance so that it can be understood by humans -- who will, after all, be investigating its validity. It should be modular, so that the reformulation so fluidly and naturally done by humans can be carried out equally smoothly by the system.

HYPOTHESIS FORM

The form we have chosen for hypotheses is, not surprisingly, closely tied to the ideals in the platoplasm -- a natural and direct consequence of having the system view its world in terms of those concepts. It also closely resembles S--e-Whence diagrams. The fragment "1 1 1", for example, may well be viewed as a C-group (Constant group). A hypothesis would express this in the form "(C-group 1 3)", a list consisting of the Platonic class name, the start-value and the (top-level) length of the grouping.

The sequence segment "4 5 6 4 5 6" could be expressed:

(C-group (S-group 4 3) 2),

indicating a C-group of length 2, each of whose entries is the S-group (successor-group) starting with 4 and of length 3.

The segment "4 5 6   5 6 7" could be:

(S-group (Countup 4) 3).

Each "hit" or evaluation of this form would yield a length-3 successor group. The first group would start with 4, the next with 5, etc..

The segment "1   5 8 4   2   5 8 4   3   5 8 4" might be expressed:

(Cycle ((Countup 1) (Tuple (5 8 4)))).

The segment "1 1 2 1" could be:

(Cycle ((Countup 1) 1)) <--> (1 1) (2 1) (3 1) ...

OR

(Cycle (1 (S-group 1 2))) <---> (1 (1 2)) (1 (1 2)) .... .

The segment "1 2 1" might be:

(S-group 1 2) <--> (1 2) (1 2)...

OR

(Y-group [1] [3] (1 2 1)) <--> (1 2 1) (1 2 1) ... .

These forms are constructed by the system as it attempts to build a hypothesis for the pattern presented. The Hypothesizer process will take such a form and from it construct a Seek-Whence hypothesis -- a data structure with several fields, capabilities, and functions.

HYPOTHESIS FEATURES

First and apparently simplest, the hypothesis can display its form, much as was shown in the last section. It can also predict the next term to be expected following that form. In addition, it has a <u>validity</u> associated with it -- a number that grows as new, correctly-predicted terms are encountered. The most crucial field, however, and the one that supports the others, is simply called the hypothesis' <u>box</u>. The box is the structure which, when "hit", produces the next run of terms predicted by the hypothesis. The box can be reset to start again, asked to list a number of terms, or asked to predict the next term, given the sequence's currently known terms. The hypothesis' box is a member of the class "Boxes" and as such lives in the socratoplasm, the middle level of the Seek-Whence world, along with members of the classes "Printstructures" and "Gnoths". Gnoths, as we have seen earlier, are the central representative structures in the socratoplasm; boxes and the closely-related printstructures are not as visible, serving a more private purpose. The next section details the operation of boxes and is not central to the flow of our discussion.

BOXES AND PRINTSTRUCTURES

Each box is a repository of information about an underlying printstructure and through that printstructure branches out, tree-like, to represent in an active way structures with such forms as:

(C-group 1 3) or (C-group (S-group 2 3) 2) [see Figure 3].

Boxes can be "hit", prodded for their next value. When implementing box hits, I wanted to be sure that hit propagation down the box tree could be done in a fully parallel manner, with no reliance on the return of any particular value before any other. The following implementation will work in this fashion, although the current version of the program treats box hits as indivisible operations, rather than as a task series.

When a box is hit, it calls upon its underlying printstructure to feed it a value. Each printstructure has a collection of <u>fire-boxes</u>, subboxes which must be hit to give it a value. When the printstructure "fires" — that is, hits its fire-boxes, each box must return a value. Thus, a hit on a top-level box propagates down through the tree of printstructures and boxes below it until the most deeply-nested structures return their values. These are passed up and the upwards-bubbling proceeds until the top-level answer appears in the top box's "pstruc-val" field.

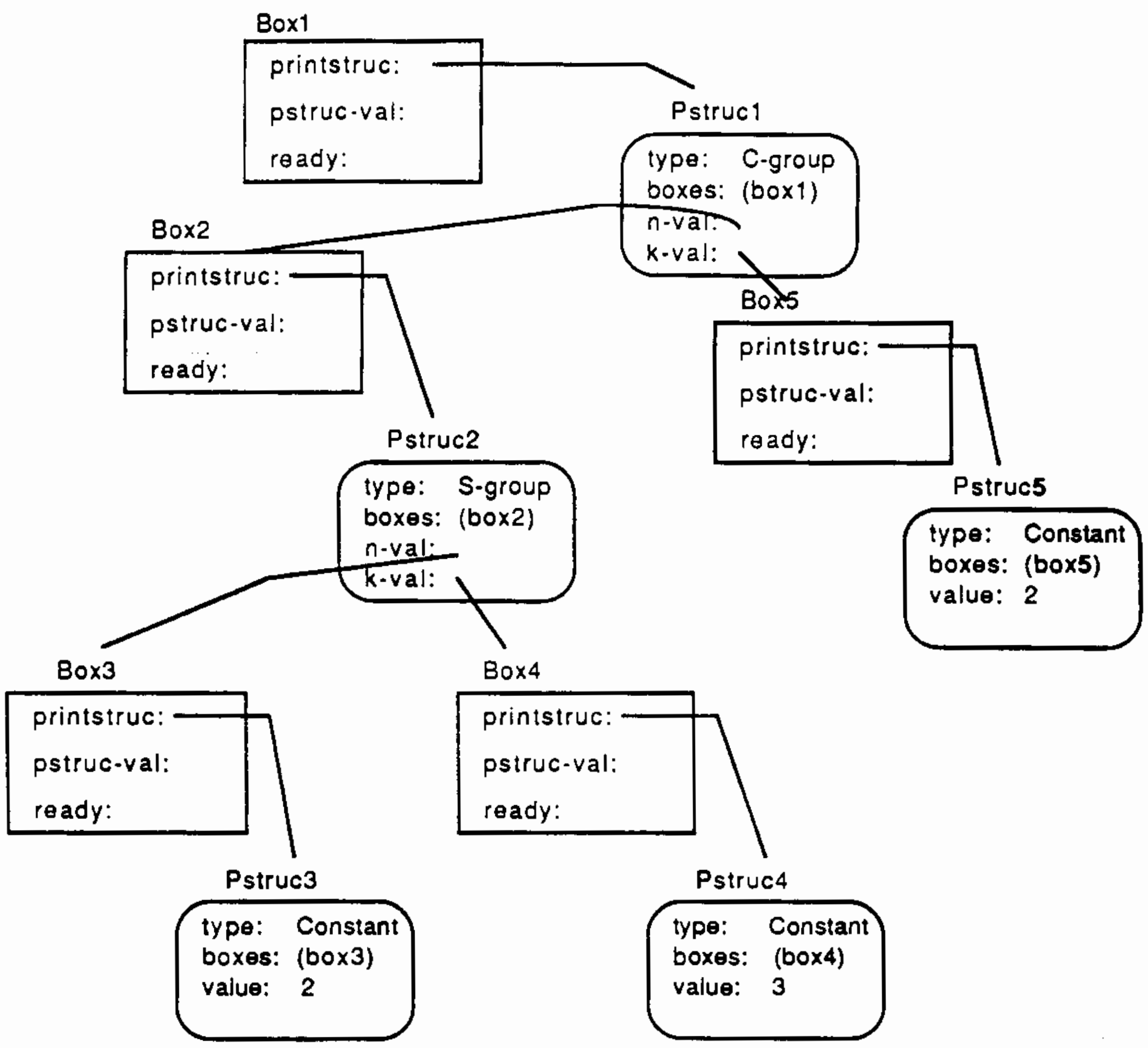


Figure 3 -- A box tree for the form (C-group (S-group 2 3) 2)

In Figure 3, for example, the "fire-boxes" for Pstruc1 -- a C-group printstructure -- are its "n-val" and "k-val" boxes, namely Box2 and Box5. In order for Box2 to fire, though, it must in turn receive a value from its subordinate printstructure, Pstruc2 -- an S-group printstructure. When Box2 is duly filled, its "ready" field will be set to "true", and it will report the value in its "pstruc-val" field to Pstruc1. It is possible for two or more boxes to share the same underlying printstructure. This happens, for example, in the sequence:

"8 1 2 8 3 4 8 5 6...",

which can be described by the form "(Cycle (8 (Countup 1) shared))". Modeling this form requires the creation of three boxes: one for the "Constant 8", one for the first "Countup 1", and one for the second "Countup 1" (referred to as "shared" in the given form). The "shared" distinguishes this situation from the one implied by the form "(Cycle(8 (Countup 1) (Countup 1)))", which also requires three boxes, and which corresponds to the sequence "8 1 1 8 2 2 8 3 3 ...".

In our first form, only one Countup printstructure is created. When that printstructure fires in response to a hit on the first Countup box, it feeds both Countup boxes, making both boxes "ready". Later on, when the second Countup box is hit, the <u>same</u> printstructure will fire, again feeding both boxes, but this time with the <u>next</u> value in sequence. In contrast, the second form causes creation of different printstructures for the two Countup boxes. Those printstructures are hit independently, once each in a turn around the Cycle.

A simpler example of the same phenomenon can be seen using Figure 3. If the form modeled had been "(C-group (S-group 2 3) shared)" -- rather than "(C-group (S-group (2 3)) 2)" -- so that the sameness of the 2's were to be modeled explicitly, our diagram in Figure 3 would have been slightly different. Pstruc3 would have "(box3 box5)" in its "boxes" field, and there would be no need for Pstruc5. Box5 would point to Pstruc3 as its "printstruc".

In order to handle the details of firing and box-filling, each printstructure type (C-group, S-group, etc.) has a <u>firer</u> associated with it, a process which knows how to fire the relevant fireboxes of the printstructure and what to do with the results. When a printstructure is shared by two or more boxes, each box must be filled whenever the printstructure fires. Those boxes must then record the fact that they already have a value -- set their "ready" fields to "true" -- so that they can report this value until the next time they are hit. Boxes can also be reset to start from the beginning of the pattern described, or asked to show a number of terms. One proposed project for refinement of Seek-Whence is to create a box-tree editor. We or the system could then change the box tree associated with a form. This would make hypothesis modification cleaner and more sophisticated than the current technique, which is to scrap the old box tree and make a new one.

We feel that the chosen implementation of hypotheses goes a long way toward meeting our goals. It gives us an active structure capable of realizing any well-formed hypothesis form. It accurately represents pattern structures, and shared substructures can be represented explicitly in the box tree. Thus it is expressive. It is modular so that slipping -- reformulation -- is supported.

D. THE END OF STAGE ONE

Once the hypothesis is in place and the gnoths corresponding to it are "up" (created by the Gnoth-maker), the system has reached the culmination of its stage-one processing. From now on, activity will take place at all levels of the system simultaneously. The new goal will be confirmation of a predictive model for the sequence.

Virtually all the structures created before the gnoths and hypothesis operate at a level that we feel is generally ignored by most AI systems. We have developed a set of gnoth operations, a language in which we can express several ways for gnoths to combine and split, to share terms, and generally to interact with each other. This is the level that AI programs tend to take as a starting point. We have attempted to implement a rich "subcognitive" level to illustrate our belief that such a substrate is critically important to truly intelligent systems, marking a step up from formal symbol manipulation. Many AI programs have been created to do very sophisticated things, but few if any are able to do simple, childlike things. Both abilities are important. A program able to combine fluidly reformulatable, structural concepts such as ours with the knowledge of a sophisticated domain would be an achievement indeed, both knowledgeable and flexible.

CHAPTER THREE

SEEK-WHENCE: STAGE TWO -- REFORMULATION

A. INTRODUCTION

The current version of Seek-Whence was designed as an illustration of the plausibility of our approach, so I spent much time developing the paradigm and implementing the lower levels of the system described in the previous chapter. The highest levels are not as completely implemented, but do serve to illustrate the potential of our approach. Several sequence problems have been solved by the system. These include:

1 1 1 ...

1 2 3 4 ...

1 1 1 2 2 2 3 3 3 ...

3 4 3 4 3 4 ...

3 7 3 7 3 7 ...

3 7 3 3 7 3 3 7 3 ...

16 15 14 17 16 15 18 17 16 ...

16 15 14 15 14 13 14 13 12 ... (as well as possible, given a non-infinite pattern)

1 2 1 2 3 1 2 3 4 ...

We will use the last of these in a running example of Seek-Whence processing throughout the remainder of this dissertation. A discussion of what the current version of the system cannot do is given in Chapter Five, along with some speculations as to why and some goals for the future.

B. BACKGROUND

The defining characteristic of stage two is the looming presence of the hypothesis. Without it, the system suffers from a "blind men and elephants" problem -- trying to make global sense from multiple local perspectives. With the hypothesis, the system has a "point of view", a predictive model of the sequence to which it can cling until contradictory evidence is encountered.

1. THE HUMAN APPROACH

We have presented sequence patterns to people singly and in groups. Almost invariably, and justifiably, once they have developed a hypothesis they insist on its correctness until it is proved incorrect by the production of a term that simply will not fit. For example, when shown:

1 2 2 3

many people hypothesize:

(1) (2 2) (3 3 3) (4 4 4 4)...

or, in Seek-Whence terminology:

(C-group (Countup 1) shared).

If we say, "Nope, not it" and then present another 3, the usual reaction is "Yeah?", uttered with an inflection of challenge and the hint of a suggestion that the presenter has actually forgotten the pattern. It is only when the next term is presented, a 4, making the initial sequence:

1 2 2 3 3 4

that they really believe another formulation is required. Then follows a variable-length period of review and reorganization, which is in turn followed by the generation of a new firmly-held hypothesis (or, in difficult cases, resignation).

This "show me" attitude and the belief in a favorite hypothesis are modeled in Seek-Whence. The system maintains one hypothesis at any given time, rather than a list of possibilities. It is able to do this and still function reasonably well because of its ability to "slip" from an old hypothesis to a new one. The hypothesis is, in effect, surrounded by cloud of potential hypotheses, close variants into which it can be transformed whenever appropriate. Underlying this ability are links among the Platonic concepts and information about the cyto-level environment favored by each Platonic concept.

2. PLATONIC RELATIONS (no pun intended)

The Platonic concepts of Seek-Whence, C-group, Tuple, Countup, and the like, are to be connected by a variety of links reflecting the concepts' interrelationships. This network of connections, in conjunction with a philosophy for their use, constitutes the "Slipnet" which is so essential to the system's reformulation ability. In the current version, we have implemented a small number of undifferentiated slipping links, called s-links, for this purpose.

The system's slipping network -- which is all within the platoplasm -- is supplemented by another, " level-spanning", network which relates each concept to its own lower-level realizations. This network includes the lists each Platonic concept maintains of its manifestations and actualizations. As described earlier, the manifestations of a concept are cyto-level structures which have been dubbed as representatives of the concept, which model it up to the expressive ability of that level. The actualizations are socrato-level structures which have similarly been identified as representatives of the concept at that level. Also included in the level-spanning network are lists of pulling and pushing bonds, bonds which the concept can use to group or separate sequence terms. Level-spanning links are little used as yet.

S-LINKS

As currently implemented, the s-links have direction and "slipperiness". For example, S-group has s-links to Countup, C-group, Y-group, Cycle, and Tuple. Associated with each s-link is a number between 0 (non-slippery) and 1 (perfectly slippery), which indicates my estimate of the system's proclivity to move from the given concept to the neighbor. The s-link from S-group to Countup has slipperiness 0.1, reflecting the fact that it is difficult to slip to a stricter class. Slipperiness from S-group to Tuple is 0.4, since Tuple can serve as a generic grouping mechanism if no satisfactory stricter class is appropriate. The slipperiness values can be changed during processing, although the current system does not do so. A richer collection of linkage types and a fuller description of the Slipnet notion is given in [Hofstadter 84].

PULL-PUSH BONDS

In addition to the s-links, each concept preserves information about the types of bonds it finds most useful in grouping sequence terms. For example, the C-group concept, because it involves copy or sameness groups, favors adjacent sameness bonds most strongly, but also likes to see gloms having the same span (number of sequence terms covered). Bonds which a Platonic class might use to hold groups together are listed as "pull-bonds"; those it tends to use to separate groups are listed as "push-bonds". Each so-designated bond type is given a strength from 1 to 10, strengths which again could be, but in practice are not, changed by the system.

3. FREEZE-DRIED HYPOTHESES

When a hypothesis has been deemed inadequate, it is "freeze-dried" -- its form is extracted and is kept on a list of old hypotheses, along with the number of terms of the sequence it explained. The old hypotheses serve as a check against cycling in the system. When Seek-Whence has trouble coming up with a hypothesis, it, like most humans, keeps coming back to the same incorrect hypotheses again and again. This, we feel, is not a bad feature, since people are guilty of the same "foolish" behavior. It would be disastrous, however, should it go unnoticed. Gray Clossman and others in the Fluid Analogies Research Group (FARG) at Michigan have thought quite deeply about the importance of "self-noticing" or "self-watching" [Hofstadter 85]. No doubt the Copycat project in progress there will have a more sophisticated approach to the problem than the small effort presented here. In any case, freeze-dried hypotheses at least flag cyclic behavior at this level of granularity. On the other hand, we do not want to prevent cycles at low levels for several reasons. People experience them. Although we may find them quite annoying at times, they are often quite useful in forcing us to consider once again a correct notion which we had rejected for some "high-level" but incorrect reason. Seek-Whence has thrashed about more than once, clinging to some Platonic class or glom, while underlying layers push up another, correct, notion over and over again. Knowing when to permit these notions to take over and when to squelch them is a most difficult problem. Our current solution has been, when no progress has been made for quite some time, to blast away all gnoths and gloms, leaving only the glints and their bonds to push up an inspiration. A mathematics student and friend was the inspiration for this approach. After struggling unsuccessfully for hours with a problem set, she would toss all her papers away, walk around the room, confront the problem sheet and say, in a very cheerful voice, "Oh, look -- a problem set! I wonder what the questions are. Shall we try some? I bet they'll be _fun_." Sometimes it worked and sometimes....

C. CHANGING A HYPOTHESIS

There are actually two reasons for changing a hypothesis:

1) it fails to predict ;

2) it is predictive but its form is less than satisfactory.

We term hypothesis changes made for the former reason "medical reformulations" to distinguish them from the "cosmetic reformulations" made in response to the latter. The current version of Seek-Whence supports the more critical medical reformulations but has only made a beginning at handling the cosmetic ones. Because our discussion of medical reformulation will of necessity be rather lengthy, we will cover the cosmetic reforms first.

1. COSMETIC REFORM

Once a hypothesis has been formulated, it becomes important to refine it. An "ugly", though correct, parse can be very dissatisfying to humans; there is generally strong agreement on which of several candidate parses is "best" in this heuristic sense. For example, given the sequence

2 1 2 2 2 2 2 2 3 2 2 4 2 2 5 2...

most successful solvers will come up with the parse:

(2 1 2) (2 2 2) (2 3 2) ....

More than one person has parsed it as:

2 (1 2 2) (2 2 2) (3 2 2)....,

becoming annoyed at the presenter for posing a problem with such a tricky, ugly parse, "with that 2 sticking out in front."

In some instances, alternative parses are equally acceptable, but will generalize differently. For example, such sequences as:

(5 1 5) (5 2 5) (5 3 5). . . and

(5 1 6) (5 2 6) (5 3 6). . .

are both considered generalizations of the sequence

(4 1 5) (4 2 5) (4 3 5)....

The difference is that in the first generalization the countup between the bracketing 4 and 5 in the original (last-listed) sequence is either not noticed or not considered salient, while in the second generalization it is maintained. For an interesting study of the problems of analogy and generalization, see [Hofstadter 82c].

Hypothesis refinement is as yet only minimally supported in

Seek-Whence. It is to be carried out by internal gnoth reformers, processes that modify the internal structure of the gnoths. Such modification will be done for either of two reasons:

1) to relieve internal pressure within a gnoth, pressure deriving from those bonds within the gnoth that would push it apart;
2) to make the gnoth's structure conform more closely to the reigning hypothesis.

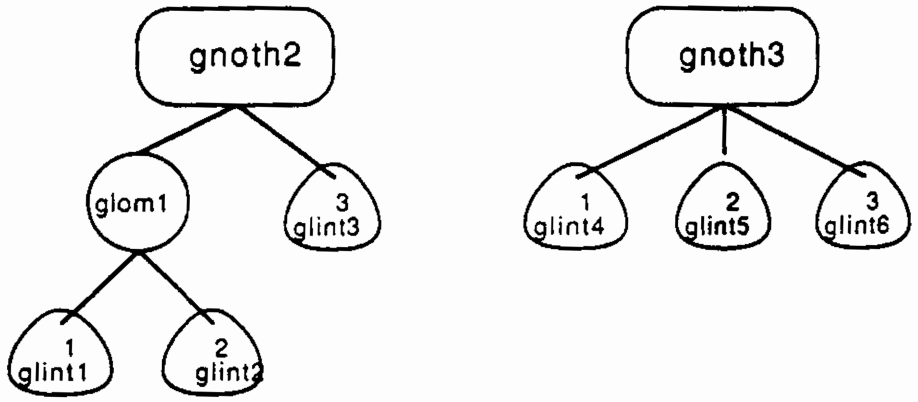
The first of these describes "bottom-up" pressures, such as an unwieldy structure or poor parenthesization. An example of this would be the 2 1 2 2 2 2 2 2 3 2. . . case cited above, where the first structure -- holding a lone 2 -- would seem rather out of place. The second is a "top-down" attempt to insure that the gnoths model the reigning hypothesis as closely as possible. The driving force behind this attempt is the goal of structural equivalence between each gnoth and the hypothesis.

GNOTH-HYPOTHESIS EQUIVALENCE

We have stated that each gnoth is to represent one hit of the hypothesis. But is it sufficient that the gnoth give the same terms as a hypothesis hit? Or do we want the same terms with the same parenthesization? Or might we also want the gnoth to obey the same underlying form (that is, have the same parenthesization for the same reason)? In the following sections we will describe these three levels of representation, which we call "term equivalence", "parse equivalence", and "structural equivalence". We use the term frame of a hypothesis to mean an abstractly-viewed hit of the hypothesis: the collection of Seek-Whence forms that would produce the given hit.

TERM EQUIVALENCE

Term equivalence, the weakest of the three types of representation, requires that each gnoth govern precisely the same terms as one frame of the hypothesis. For example, if the hypothesis is: (S-group 1 3), then both gnoths shown in the following diagram are term-equivalent to it.



When asked for its value, gnoth2 produces ((1 2) 3), while gnoth3 yields (1 2 3). Both gnoths produce the three terms 1, 2, 3 in that order, so both satisfy the requirement for term-equivalence.

PARSE EQUIVALENCE

Parse equivalence, the next level, requires that the gnoth print its value with the same parenthesization as the corresponding hypothesis frame. In the above example, gnoth3 is parse-equivalent to the given hypothesis while gnoth2 is not.

STRUCTURAL EQUIVALENCE

The third and strongest level of equivalence is structural equivalence. In order to display structural equivalence with the hypothesis, a gnoth must be parse-equivalent to it and the gnoth's form must be the same as the corresponding frame of the hypothesis. But what form should a gnoth assume if it is to reflect the hypothesis accurately? There are two distinguishable possibilities, which we call deep structure and shallow structure.

For example, suppose that we have a fairly complicated hypothesis such as "(C-group (S-group (Countup 1) 3) 2)", derived from input terms:

1 2 3 1 2 3 2 3 4 2 3 4 3 4 5 3 4 5, and parsed as:

1 2 3 1 2 3   2 3 4 2 3 4   3 4 5 3 4 5.

Viewed at the term level, the first hit of this hypothesis generates "1 2 3 1 2 3".

The shallow-structure (or deeply-hit) form of the first frame of our hypothesis would be:

(C-group (1 2 3) 2).

The corresponding deep-structure (or shallowly-hit) form is:

(C-group (S-group 1 3) 2).

More structural detail is retained in the deep-structure form, with only the lowest-level structures replaced by constants or runs. In the shallow-structure form, all but the top-level structures are so replaced.

DEEP VS. SHALLOW STRUCTURE

For comparison, the first three deep-structure and shallow-structure frames of our hypothesis "(C-group (S-group (Countup 1) 3) 2)" are:

<u>shallow</u>	<u>deep</u>
(C-group (1 2 3) 2)	(C-group (S-group 1 3) 2)
(C-group (2 3 4) 2)	(C-group (S-group 2 3) 2)
(C-group (3 4 5) 2)	(C-group (S-group 3 3) 2)

Because the deep-structure form presents more structural detail and represents a "one-step-down" view of the hypothesis, we chose it as our goal. Once a hypothesis is made, the system gives each gnoth its target form, the deep-structure equivalent of the hypothesis frame to which it corresponds. When the gnoth's form matches this given one, the gnoth is said to exhibit structural equivalence with the hypothesis. At that time, the gnoth should be completely "happy", having no further goals.

FORM POLISHING

In summary, all gnoths must always maintain term equivalence with the hypothesis. Their goal will be to achieve structural equivalence by reforming into the deep-structure form of one hypothesis frame. Along the way they will achieve the middle state of parse equivalence, indicated by the fact that the gnoth's "parse-print", the parenthesized printing of its value, matches that of the hypothesis frame.

IMPORTANCE TO GENERALIZATION

The form polishing described above will be essential to an ability to generalize sequences in reasonable ways and make analogies between sequence descriptions. Also required will be the ability to notice structural samenesses, such as the (Countup 1) in the hypothesis "(C-group (Countup 1) (Countup 1))", which yields the terms:

1 2 2 3 3 3.

These are among the future high-level goals of the Seek-Whence project, unimplemented as yet.

2. MEDICAL REFORM

Medical reformulation, which is supported in the current version of Seek-Whence, is done when the hypothesis has been demonstrated to be invalid. It involves a review of the old hypothesis and the underlying structures supporting it, a decision as to which Platonic type should hold sway, a re-evaluation of the bonds noticed by the system, the use of bonds in the environment of the chosen Platonic type to engender gnoth reformulations, and finally (it is hoped) the construction of a new, predictive hypothesis.

GNOTH-SETTER

The system stores hypothesis-confirming terms in a catchall gnoth, a special gnoth that simply serves as a repository for non-troublesome terms. When an unexpected term is encountered, the system immediately sets the hypothesis' validity to 0, releases sparks to encourage low-level activity, and places a Gnoth-setter task on the taskrack. When invoked, the Gnoth-setter carefully fills out gnoths in accordance with the old hypothesis and calls for the system to reconsider its parse. For example, if the old hypothesis were "(S-group 1 3)", and two gnoths, each holding "1 2 3", were already in existence, the catchall gnoth might be holding "1 2 3 4". The first three terms in the segment "1 2 3 4" are in the catchall because they were predicted by the hypothesis; the "4" is the last term entered -- the troublesome one. The Gnoth-setter would therefore create two new gnoths, one to hold the initial "1 2 3" from the catchall and the other to hold the trailing 4.

Each gnoth is marked with the frame and equivalence type (term or parse, depending on agreement with the hypothesis' parenthesization) appropriate for it. Any non-fitting terms are collected together in a final gnoth and the catchall is destroyed. In the example above, the two pre-existing gnoths and the first of the newly-created ones would be marked as parse-equivalent to the old hypothesis.

In our running example ("1 2 1 2 3"), entry of the first two terms -- "1 2 " -- causes the system to hypothesize (Countup 1). When the next term entered is "1", a Gnoth-setter puts out three gnoths, one for each term. The first two are in accord with the discredited hypothesis and are in fact parse-equivalent to it. The last one simply holds the non-fitting term.

Now, with "all the cards on the table", the Gnoth-setter calls for reconsideration to begin.

3. RECONSIDERATION

The goal of reconsideration is the construction of a new and valid hypothesis. This is not a mechanical, program-directed reconstruction, however, but rather a "homing in" on a new formulation from a tightening spiral of possibilities generated by independent but interacting processes.

a. DETERMINATION OF THE REIGNING TYPE

The first step taken during reconsideration is a bookkeeping measure, saving the form of the old hypothesis and destroying its box, the home of its active representation. This leaves the system with no active structure to govern or filter processing, only a "freeze-dried" form to remind it of its most recent perspective. The system then decides whether to stay with the <u>reigning class</u> -- the Platonic class at the highest level of the (former) hypothesis -- or to slip to a new one. This decision is made on several considerations.

First, if a reigning class -- such as Constant -- is very strict in the sense that it is difficult to generalize without moving to a new class altogether, slipping is chosen immediately. Otherwise, some deeper investigation is made. The old hypotheses are checked to determine the number of recent hypotheses of this class -- how many "tries" the class has had since it seized power. All bonds are assessed in the environment of this class -- assigned a strength which depends on the class in question as well as on the type of the bond. (Bond assessment is described in some detail in the next section.) The result of this assessment is a rough measurement of the existing "bond tension", the strength of the bonds favoring modification of the current gnoths. Strong bond tension implies strong pressure to change some aspect of the current parse -- either to abandon the current reigning class or to modify the gnoths' structure within the framework of that class.

ASSESSING BOND PULLS

Bond assessment is a relatively straightforward procedure designed to assign strengths to all existing bonds under the assumption that a particular Platonic class holds sway. If, for example, S-group is the reigning class, adjacent successor bonds are given large positive values to indicate that they are strong pulling bonds while sameness bonds are given negative values to indicate that they tend to push gloms apart. Should C-group be in ascendancy, sameness bonds become strong whereas successorship bonds are made negative. The information required for the system to assign these values is in the platoplasm, with each Platonic concept listing both pulling and pushing bond types and their strengths.

Procedurally, each gnoth is processed in turn. Its internal bonds, those among the gloms it covers, are noted, and their strengths in the current environment -- that of the reigning class -- are assigned. Its external bonds, those between its elements and those of other gnoths, are similarly assessed. These values become instrumental in determining the "happiness" of the gnoth -- its inclination to stand pat. The collective happiness of all the gnoths is used as a measure of the success of the reigning class in organizing the system's perception of the sequence.

SLIP OR STAY

The pressure to stay with the reigning class is the sum of what is termed "gnoth-stabilities", a less anthropomorphic and more functionally defined term for the "happiness" mentioned above. The stability of a gnoth is the difference between the bond forces holding it together and those acting to tear it apart. "Holding" bonds are internal pulls and external pushes. "Tearing" bonds are external pulls and internal pushes. In our "1 2 1 2 3" example, just after the 3 is introduced, we should have (S-group 1 2) as the now-discredited hypothesis and three gnoths as shown in Figure 1.

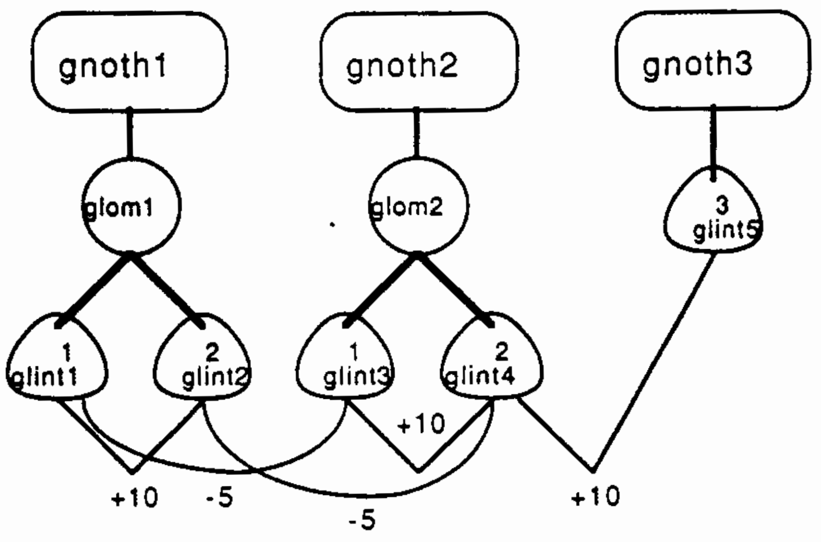
In Figure 1, the adjacent-successor bond between glint1 and glint2 has strength +10 because S-group is the reigning Platonic type and S-group favors such bonds. This particular bond functions as an "internal pull" for gnoth1 since it has a positive value and both members, glint1 and glint2, are within that gnoth. In contrast, the adjacent successor bond between glint4 and glint5 also has value +10, but functions as an "external pull" between gnoth2 and gnoth3. Thus, the former bond tends to uphold the status quo, tends to make gnoth1 "happy", while the latter bond causes some unhappiness for both gnoth2 and gnoth3.

The remote sameness bond (with strength -5) between glint2 and glint4 functions as an "external push", tending to keep the parent gnoths, gnoth1 and gnoth2, apart. Therefore, it contributes to the stability or "happiness" of both gnoths involved.

In this particular example, there are no "internal push" bonds.

STABILITY

To calculate a gnoth's stability, we first add the strengths of the bonds holding it together. For gnoth1 in Figure 1, with S-group reigning, this sum would be +10 +2.5 +2.5 = 15. The +10 comes from the internal pull applied by the adjacent successor bond between glint1 and glint2. The 2.5's represent half the strength of the two external push bonds under gnoth1. These are the remote sameness bond between glint1 and glint3, and the remote sameness bond between glint2 and glint4. Strength-halving is done so that external bond values are not counted twice, once for each gnoth involved.



+10 -- adjacent successor value bond

-5 -- remote same value bond

Figure 1 -- Measuring gnoth stability

Once the holding strength is calculated, we subtract the sum of the tearing-bond strengths acting on the gnoth to come up with its stability. Gnoth1 has no tearing bonds (no internal pushes or external pulls), and so its stability is: (+10 +2.5 +2.5) -(0) = 15.

Similarly, with S-group reigning, gnoth2 has three "holding" bonds -- the internal pull between glint3 and glint4 from their "adjacent successor" bond, the external push between glint1 and glint3 ("remote same"), and the external push between glint2 and glint4 (again, "remote same"). In addition, gnoth2 has one "tearing" bond -- the external pull between glint4 and glint5 ("adjacent successor"), of strength 10. Thus, we have gnoth2 stability:

$(+10 + 2.5 + 2.5) - (5) = 10.$

Finally, since gnoth3 has only one bond -- a "tearing" external pull of strength 10, its stability is:

$(0) - (5) = -5.$

We then add the individual gnoth stabilities to find a total system stability, in this case, of $15 + 10 - 5 = +20$.

We note that some of the tearing pressure is due to unresolved bond pulls favoring the reigning type -- if not its specific realization in the current hypothesis -- and so may be considered inappropriate for our purposes. Nevertheless, we are tapping a measure of internal consistency. That is, if we assume an environment of this class and still find much bond tension (much gnoth unhappiness), we may quickly abandon the type, at least for a while.

In order for a reigning class to be abandoned, however, some other class has to demonstrate strength in its own right. Those classes that "neighbor" the current reigning class -- those connected to it by s-links in the platoplasm -- are the primary "pretenders to the throne". If one of them can show sufficient strength (sufficient "slipping pressure", as described below), it may supplant the current "monarch".

The slipping pressure from the reigning class to a neighbor is evaluated by:

1) adding two quantities -- the sum of all pulling-bond strengths, and the absolute value of the sum of all pushing-bond strengths -- taken over all existing bonds, and assessed in the environment of the neighboring class, and then
2) multiplying the sum by the slipperiness of the link between the monarch and the neighbor -- the proclivity to slip in that direction.

In effect, the system tries to estimate the gnoth stability in an "alternative universe" -- the environment dominated by the neighboring class -- as well as the likelihood of moving from the current universe to the alternative one. A very "close" neighbor of the current monarch who presents fairly strong prospects for stability would be a strong candidate for ascendancy to the throne, whereas a "distant" neighbor -- one connected to the reigning class by a non-slippery s-link -- whose stability prospects are low would be a weak candidate.

Slipping-pressure estimates are calculated for each class that is an s-link neighbor of the reigning class. If the largest of these values is greater than the "staying pressure" -- the current stability -- then a slip to the corresponding class will be made and the system will have a new reigning class.

For example, in our "1 2 1 2 3 " case, the slipping pressure from S-group to Y-group ("symmetry" group -- for, say, a parse: 1 2 1 2 3 2) is:

0.4 [the s-link slipperiness] * (10 + 10) = 8,

where the 10's are the strengths, in a Y-group environment, of the "remote same" bonds between glint1 and glint3 and between glint2 and glint4.
Thus Y-group, with a slipping pressure of 8, cannot seize the throne from the reigning S-group, whose staying pressure is 20.

A similar value is calculated for each neighboring class, and if the largest of these values is greater than the staying pressure, a slip to that class will be made.

b. REFORMULATOR

At this point, a reigning class has been established -- or reconfirmed -- and so a Reformulator process is placed on the taskrack. When invoked, this process will attempt to find salient bonds and will set out Gnoth-operator tasks designed to act upon the bond pulls or pushes in order to change the gnoths.

The Reformulator's first act is to determine a threshold bond strength. Bonds or bond groups exerting pressures below this threshold will be ignored. Currently, the new threshold is set to either 1 more than the existing threshold value or, if none exists, 80% of the strongest pull-bond strength for the reigning type. (This value was chosen arbitrarily, with some vague remembrance of Winston's grouping algorithm in his "blocks-world" program [Winston 75]. It has remained because it seems to have done no harm as yet.) Because new bonds may have been established since the Reformulator's creation time, its next act is to assess all bond pulls in the environment of the reigning class, as described above.

Then begins the process of finding strong pulls and/or pushes, and turning them into gnoth operations -- actions that modify gnoths. If the Reformulator finds no actions to be taken or if it has completed its recommendations, it hangs a Bond-assessor task on the taskrack (to determine system "happiness") and terminates.

SELECTION OF NEIGHBOR-PULLS

All inter-gnoth moves involve the rightmost (at some level) glom of some gnoth and the leftmost (again, at some level) glom of the gnoth's neighbor to the right. This is a consequence of the sequential nature of our domain. We obviously cannot rearrange the order of sequence terms (even though such an operation might make a "more interesting" sequence); we can only readjust our groupings. (For a study of a less restricted pulling environment, see [Hofstadter 83].)

For example, given neighboring gnoths as shown in Figure 2 below, our system will be interested in the "lasts" of gnoth3:

(glom15 glom10 glom7 glint4),

and the "firsts" of gnoth4:

(glom8 glom3 glom1 glint5).

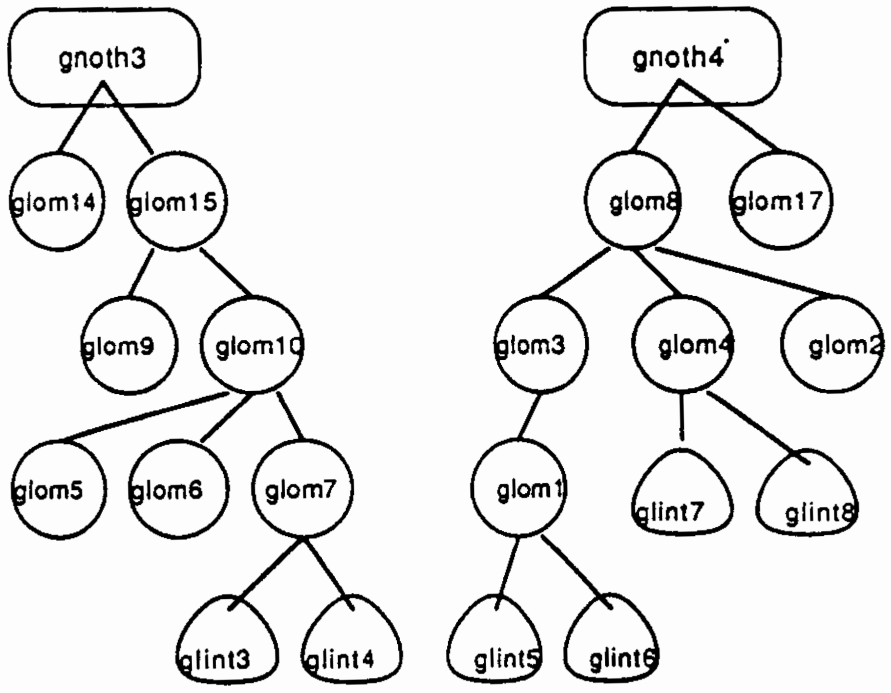


Figure 2 -- Neighboring gnoths

The first moves considered are those at the highest level, under the theory that if a glom wants to move, its subgloms should follow. It is also possible that some glom feels relatively content but one of its subgloms is attracted to a glom in the neighbor gnoth. In such a case, the subglom should be popped out and over to the neighbor. Should both glom and subglom feel a pull, the glom move should take precedence since it is structurally more important. Subsequently, internal gnoth operations — actions which modify the internal structure of a gnoth -- could be used to move the subglom if it still feels the need to leave its parent glom.

SELECTION ORDER

In our Figure 2 example, neighbor-driven reformulation would be explored in the following order:

level 1:	glom15 <--> glom8	(Assess the pull between the topmost
level 2:	glom10 <--> glom8	gloms, then between level two gloms
	glom15 <--> glom3	and those at levels one and two.)
	glom10 <--> glom3	
level 3:	glom7 <--> (glom8 glom3)	(assess pull with each in the list)
	(glom15 glom10) <--> glom1	
	glom7 <--> glom1	
level 4:	glint4 <--> (glom8 glom3 glom1)	
	(glom15 glom10 glom7) <--> glint5	
	glint4 <--> glint5	

As soon as some reformulation is strong enough -- the bond pulls and pushes supporting it exceed the threshold -- the Reformulator creates an appropriate gnoth operation or program of operations and sets a Gnoth-operator on the taskrack to carry it out. The Reformulator will not suggest any further moves, since any others would occur at a lower structural level and therefore would be less important to the system. Should any lower-level moves be important, they will eventually be discovered by some future Reformulator.

CONVERSION OF BOND-PULLS INTO GNOTH OPERATIONS

When there is sufficient strength of pull between two gloms from neighboring gnoths, a gnoth operation must be devised to bring the two gloms together. Simply shifting one glom into the other's gnoth may not be sufficient, because the decisive pull on it may be coming from a deeply-nested glom, one several levels down from the top. In Figure 2 for example, glint4 may be pulled toward glom3. In our "1 2 1 2 3" example, at the time described in Figure 1, the last term -- the "3" -- is pulled by its predecessor -- a "2" -- which is nested within a glom whose print-value is "(1 2)".

The system must decide which of the two attracting gloms is to move and which is to stay put. This is determined by an analysis of the bonds holding the gloms in their respective gnoths. Single gloms are the most likely to move, leaving an empty gnoth behind, a shell which the system destroys.

Once the direction of the move is determined, the total move must be constructed. As will be discussed below, gnoth operations can be quite destructive of a gnoth's internal structure, bursting gloms until the target gloms below are reached. When a gnoth operation is performed, at least some of this structural damage must be repaired; we do not want the destruction of important nesting structures to be a side-effect of reformulation.

Finally, the strength of the operation is calculated. This strength -- the difference between the gloms' mutual attraction and the pull exerted by other gloms to hold them in place -- must exceed the system-determined threshold, or else the move would not have been generated. The strength is used by Seek-Whence to weight competing alternatives when necessary.

4. THE GNOTH OPERATIONS

Gnoth operations fall into two categories: external or inter-gnoth operations, and internal or intra-gnoth operations. The external operations are: SHIFT-LEFT, SHIFT-RIGHT, and SPLIT. The internal operations are: CAPTURE, ENCLOSE, FRACTURE, MERGE, and NO-OP.

All of these operations require a bit of careful manipulation. As was described earlier, each gnoth has an associated "pseudo-glom", a glom that cannot interact with others, serving as a cap to prevent the disappearance, through natural glomming, of useful gloms and glom groups. The pseudo-gloms of any gnoths involved in gnoth operations must be destroyed to permit the true gloms below to interact with each other. Similarly, if a very deeply nested glom is to be involved in an operation, all gloms containing it must be destroyed so that it can rise to the top of the cytoplasm and become available.

Naturally, all this glom-bursting destroys the encasing gnoth's structure. This is permitted because neither we nor the system can know whether the destruction is the primary purpose of the operation or just a side-effect of its real intent. Any proposer of gnoth operations that wishes to preserve some of the original structure must make the effort to do so. The burst gloms cannot, of course, be brought back, but functionally similar (not identical, because the gnoth operation did change something) ones can be created.

When a gnoth operation is completed, a capping procedure puts a new pseudo-glom in place above the gnoth's gloms. Often, Plato-scout tasks are placed on the taskrack to peruse the gnoth's newly-created gloms, searching among them for any new manifestations of the Platonic concepts.

EXTERNALS

The formats for the external operations are:

(SHIFT-LEFT <left-gnoth> <right-gnoth> <glomlist>),

where glomlist is a list of the gloms (which must be neighbors in order) to be transferred from right-gnoth to left-gnoth;

(SHIFT-RIGHT <left-gnoth> <right-gnoth> <glomlist>),

where glomlist serves an analogous purpose, this time from left-gnoth to right-gnoth;

(SPLIT <gnoth> <splitlist>),

where splitlist is a list of gloms currently under the given gnoth. A new gnoth is to be formed using the splitlist gloms as its top level.

SHIFT EXAMPLE

initial state:

```
gnoth2: [ (1 1)  (2 2) ]      gnoth3: [ (3 3)  (4 4)  (4 4) ]
          glom3  glom5                  glom7 glom10 glom15
```

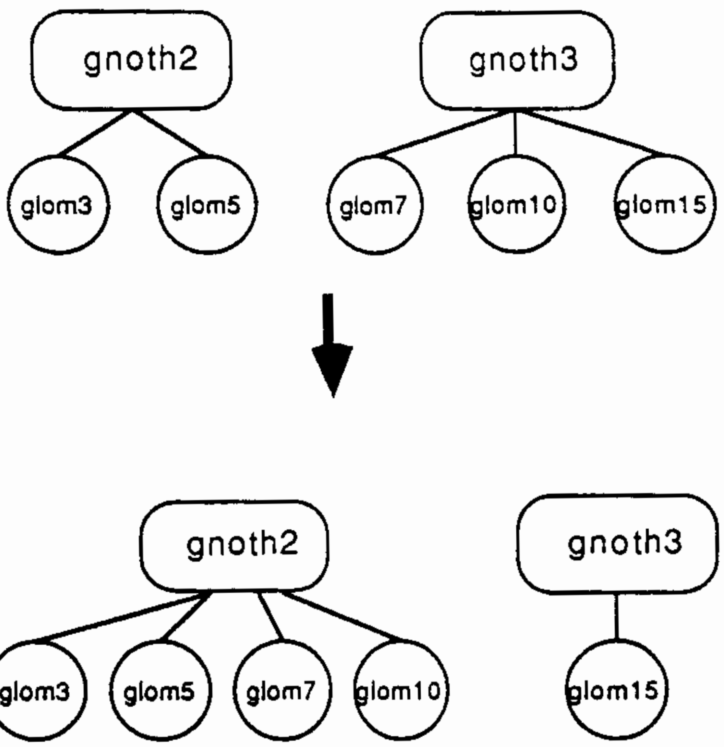
operation:

(SHIFT-LEFT gnoth2 gnoth3 (glom7 glom10))

final state:

```
gnoth2: [ (1 1)  (2 2)  (3 3)  (4 4) ]   gnoth3: [ (4 4) ]
         glom3 glom5 glom7 glom10                 glom15
```

SHIFT DIAGRAM



(SHIFT-LEFT gnoth2 gnoth3 (glom7 glom10))

SPLIT EXAMPLE

initial state:

<u>gnoth2</u>: [(1 2) (2 3) (2 3)]

glom2 glom6 glom10

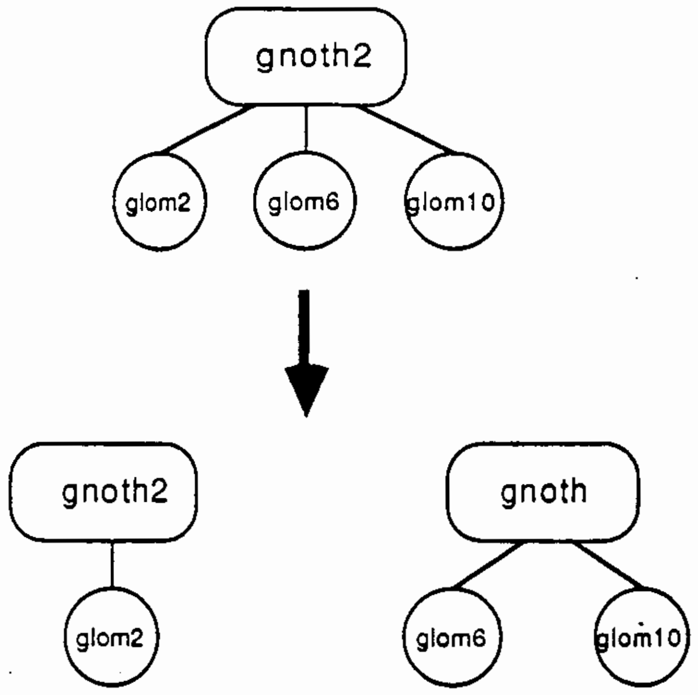
operation:

(SPLIT gnoth2 (glom6 glom10))

final state:

<u>gnoth2</u>: [(1 2)]   <u>gnoth(new)</u>: [(2 3) (2 3)]

SPLIT DIAGRAM



(SPLIT gnoth2 (glom6 glom10))

INTERNALS

The formats for the internal gnoth operations are:

(CAPTURE-LEFT <gnoth> <glom> <captive>),

where the given glom within the given gnoth is to swallow its neighbor, captive, whole. Actually, the glom is destroyed and a new one created with the captive as its leftmost subglom and also containing all the original glom's subgloms.

(CAPTURE-RIGHT <gnoth> <glom> <captive>),

analogous to the operation above;

(ENCLOSE <gnoth> <encloselist>),

where encloselist is a list of neighboring gloms within the gnoth to be covered by a new glom, dubbed to be of type "enclose";

(FRACTURE <gnothname>),

where the given gnoth is to have all of its top-level gloms (the direct subgloms of its pseudo-glom) dissolved, bringing their subgloms to the top-level;

(MERGE <gnoth> <glomlist>),

where glomlist is a list of neighbor gloms in order within the given gnoth. The listed gloms are all uncovered and their subgloms glommed into a "merge"-type glom, which becomes a top-level glom in the gnoth;

(NO-OP <gnoth>),

which causes the gnoth to be "uncapped" -- have its pseudo-glom suspended -- and remain that way until the Capper task it sets out is invoked and recaps the gnoth. This "slow-recap" permits natural glomming to occur within the gnoth, and between gnoths should two neighboring gnoths be uncapped simultaneously. The Capper finds all current gloms whose ranges overlap with the original range of the gnoth (before it was uncapped) and claims them for the gnoth. Should two different gnoths claim the same glom -- one formed, perhaps, by combining gloms from the two gnoths -- the gnoth that recaps first will get the glom and the extended range.

CAPTURE EXAMPLES

initial state:

```
        gnoth1: [ (1 2) 3 ]
                  glom1 glint3
```

operation:

(CAPTURE-RIGHT gnoth1 glom1 glint3)

final state:

```
        gnoth1: [(1 2 3)]
                   glom2
```

initial state:

```
        gnoth1: [ ((1 2)  (2 3)  ( (2 3)  (2 3) ) )  (1 2) ]
                   glom5 glom7  glom8 glom9    glom18
                                <-- glom10 ->
                <---------   glom15  -------->
```

operation:

(CAPTURE-LEFT gnoth1 glom10 glom7)

final state:

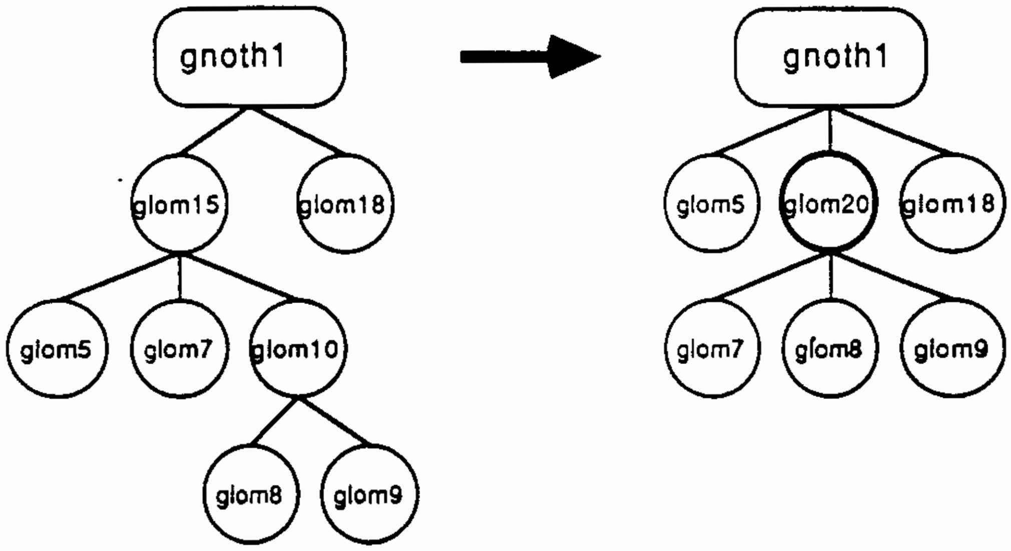
```
        gnoth1: [ (1 2)  (  (2 3)  (2 3)  (2 3)  )  (1 2) ]
                 glom5   glom7 glom8 glom9    glom18
                       <------ glom20 ------>
```

CAPTURE DIAGRAM

The operation:

(CAPTURE-LEFT gnoth1 glom10 glom7)

will cause glom 10 to "swallow" its neighbor to the left, glom7, within gnoth1.



(CAPTURE-LEFT gnoth1 glom10 glom7)

ENCLOSE EXAMPLES

initial state:

<u>gnoth1</u>: [1 2 3 1]

glints 1,2,3,4

operation:

(ENCLOSE gnoth1 (glint1 glint2 glint3))

final state:

<u>gnoth1</u>: [(1 2 3) 1]

glom1 glint4

initial state:

```
gnoth3: [ (2 2)  ( (3)  ( (3 3)  (4 4) )  )  ( (4 4)   (5 5)  ) ]
          glom1    glom4 glom6 glom5        glom7 glom8
                         <- glom10 ->      <--- glom9 --->
                   <------ glom12 ------>
```

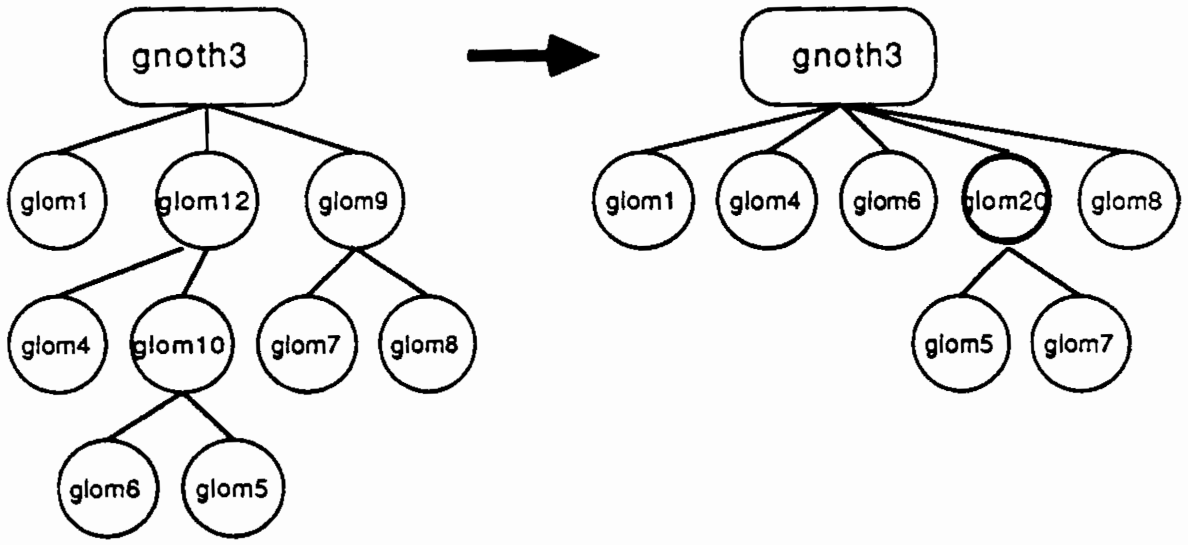
operation:

```
(ENCLOSE gnoth3 (glom5 glom7) )
```

final state:

```
gnoth3: [ (2 2)  (3)   (3 3)   ( (4 4)  (4 4)   )  (5 5)  ]
          glom1 glom4 glom6   glom5 glom7    glom8
                              <-- glom20 -->
```

ENCLOSE DIAGRAM



(ENCLOSE gnoth3 (glom5 glom7))

FRACTURE EXAMPLE

initial state:

```
        gnoth2: [ (1 2)   ( (3 3)  (4 4) ) ]
                 glom1     glom4 glom3
                          <----glom5---->
```

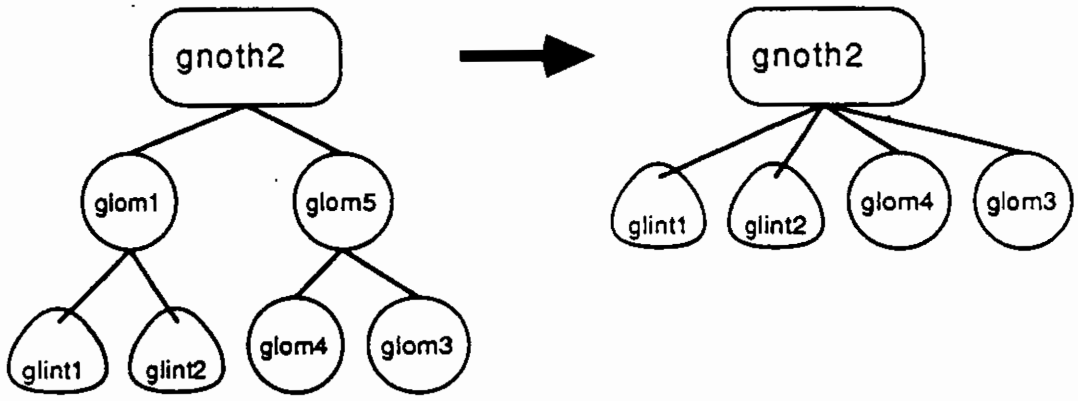
operation:

(FRACTURE gnoth2)

final state:

```
        gnoth2: [ 1  2  (3 3)  (4 4) ]
                        glom4  glom3
```

FRACTURE DIAGRAM



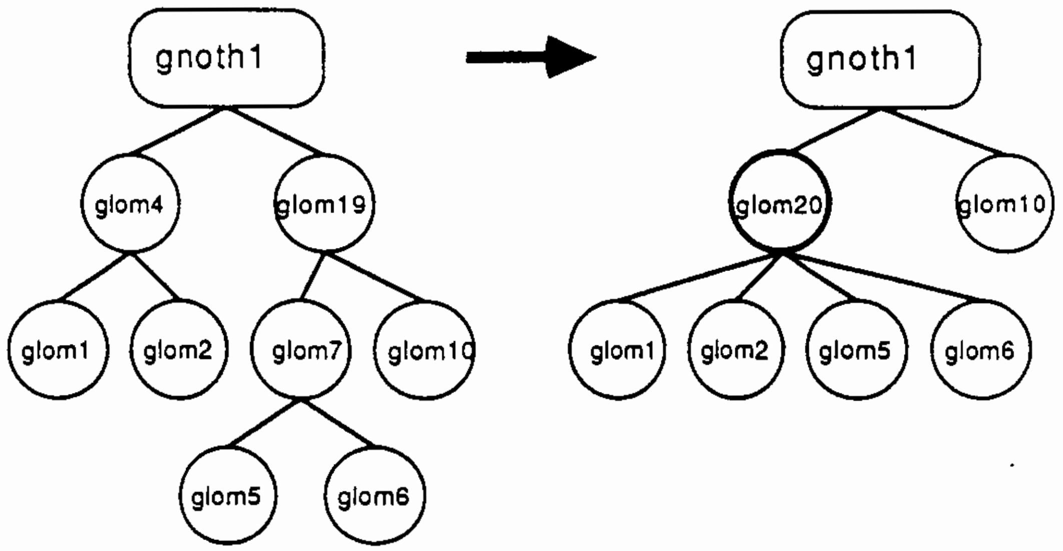
(FRACTURE gnoth2)

MERGE EXAMPLES

```
initial state:

        gnoth1: [ (1 1)  (1 1 1) ]
                 glom1   glom2

operation:

        (MERGE gnoth1 (glom1 glom2))

final state:    gnoth1: [ (1 1 1 1 1) ]
                           glom3
```

```
initial state:

        gnoth1: [ ( (1 2 3) (3 4) )   ( ( (5 6) (4 5)  )  (6 7 8)  ) ]
                     glom1  glom2          glom5  glom6     glom10
                  <--- glom4 --->        <-- glom7 --->
                                      <---------- glom19 ---------->

operation:

        (MERGE gnoth1 (glom4 glom7) )

final state:

        gnoth1: [ ( (1 2 3) (3 4)  (5 6)  (4 5)  ) (6 7 8) ]
                   glom1  glom2  glom5  glom6     glom10
                 <---------- glom20 ---------->
```

MERGE DIAGRAM



(MERGE gnoth1 (glom4 glom7))

OPERATIONS IN SERIES

The gnoth operations described above can be used to reformulate the gnoth-based parse of the sequence (as opposed to the hypothesis-based parse) when applied in series. Two examples follow.

In our first example, we start out with three gnoths which parenthesize the sequence segment " 1 2 3 3 4 3 4 5" as shown initially. After several operations, a more "reasonable" final parenthesization emerges.

Initial state:

```
        gnoth1            gnoth2                 gnoth3
        [(1 2 3)]         [((3 4) (3 4))]        [5]
         glom1             glom2 glom3           glint8
                          <--- glom4 --->
(SHIFT-LEFT gnoth2 gnoth3 glint8)
        [(1 2 3)]         [((3 4) (3 4)) 5]      [](disappears)
(CAPTURE-RIGHT  gnoth2 glom3 glint8 )
        [(1 2 3)]         [(3 4) (3 4 5)]
                          glom2 glom5
(FRACTURE gnoth1 )
        [1 2 3]           [(3 4) (3 4 5)]
(ENCLOSE gnoth1 (glint1 glint2) )
        [(1 2) 3]         [(3 4) (3 4 5)]
(SPLIT gnoth2 (glom5))
        [(1 2) 3]         [(3 4)]                [(3 4 5)](emerges)
(SHIFT-LEFT gnoth2 gnoth3 (glint6))
        [(1 2) 3]         [(3 4) 3]              [4 5]
(ENCLOSE gnoth3 (glint7 glint8) )
        [(1 2) 3]         [(3 4) 3]              [(4 5)]
```

In our second example, we once again have three gnoths which exchange gloms and are reformulated internally to come up with a new, more coherent parenthesization -- this time of the sequence segment "1 2 3 3 4 5 6 4 5 6 7 8".

Initial state:

```
          gnoth1              gnoth2                     gnoth3

          [(1 2 3)]           [(3 4)  (5 6)  (4 5)]      [(6 7 8)]

           glom1              glom2 glom3 glom4           glom5

(SHIFT-RIGHT gnoth2 gnoth3 (glom4) )

          [(1 2 3)]           [(3 4)  (5 6)]             [(4 5)  (6 7 8)]

(MERGE gnoth3 (glom4 glom5) )

          [(1 2 3)]           [(3 4)  (5 6)]             [(4 5 6 7 8)]

                                                           glom6

(FRACTURE gnoth2)

          [(1 2 3)]           [3 4 5 6]                  [(4 5 6 7 8)]

(ENCLOSE gnoth2 (glint4 glint5 glint6 glint7) )

          [(1 2 3)]           [(3 4 5 6)]                [(4 5 6 7 8)]

                                glom7
```

BONDS INTO GNOTH OPERATIONS

The conversion of bond pulls and pushes into gnoth operations simply requires that care be taken about who is attracting whom and how deeply nested each of the participants is in its original gnoth.

When more than a straightforward top-level move is to be required, a Reformulator must create a PROGRAM or series of moves designed to put the proper glom in its proper place and repair as much resulting gnoth-tearing as possible. Some examples may help explain exactly what is done.

HIGH-LEVEL MOVES

In Figure 2 (p.106), if glom8 is to be pulled away from gnoth4 by glom15, this high-level attraction is translated into the move:

```
(SHIFT-LEFT gnoth3 gnoth4 (glom8) ).
```

If, however, glom1 is to be pulled away by glom7, we have a more complicated situation.

DEEPER MOVES

In the case of such deeper moves, a PROGRAM must be generated. In the glom7 - glom1 example, a translation of the result is:

```
(PROGRAM ((SHIFT-LEFT gnoth3 gnoth4 (glom1))  [move glom1 over]
    (CAPTURE-RIGHT gnoth3 glom10 glom1)        [swallow it]
    (ENCLOSE gnoth4 siblings-of-glom1)         [replace glom3]
    (ENCLOSE gnoth4 newglom3&sibs-of-glom3)    [replace glom8]
    (ENCLOSE gnoth3 sibs-of-glom10&newglom10) [replace glom15]
    ))
```

A REAL MOVE

In most cases, such deep nesting is not encountered. In the case of "1 2 1 2 3" [Figure 1], the 2 <--> 3 pull is resolved via:

```
(PROGRAM (   ((1 2)) ((1 2)) (3)
     (SHIFT-LEFT gnoth2 gnoth3 (glint5))       --> ((1 2)) ((1 2) 3)
     (CAPTURE-RIGHT gnoth2 glom2 glint5)       --> ((1 2)) ((1 2 3))
     (ENCLOSE gnoth2 nil)                     [no repair necessary]
     (ENCLOSE gnoth3 nil)                     [no repair necessary]
    ))
```

Reformulator processes are responsible for creating such "PROGRAM"s as described above. For the most part, once the initial move or two have been supplied, the remainder of the PROGRAM is designed simply and mechanically to repair any concomitant structural damage. Such damage is usually caused by the need to burst a glom in order to get at one of its subgloms, perhaps even one nested several levels below it. The damage is repaired by re-enclosing the remaining gloms at each intermediate level -- those not directly involved in the operation -- and setting out Plato-scouts on the newly-enclosed gloms. This last step is taken to determine whether any "interesting" new structures have been created. It should be emphasized that PROGRAM construction is a mechanical action, performed by a task that exists at a high enough level to possess the necessary vocabulary. The Reformulator's activity in writing a PROGRAM is no more intelligent than a Glommer's or a Bonder's, or that of any other Seek-Whence task. Whatever "intelligence" the Seek-Whence system possesses is an emergent phenomenon arising from the performance of all of these mechanical tasks in parallel.

DIVESTING PUSHES

In addition to neighbor pulls, there is a second potentially strong agent for reformulation -- what we call a "divesting push". There may be no real pull between one glom in a gnoth and the neighboring gnoth, but the glom's current home may not want it. This sort of unilateral decision to push out a glom and either foist it off on the neighbor or create a new gnoth to hold it could be the foundation for much useful reformulation. Divesting pushes are not implemented in the current system, causing some weakness in its performance when handling Tuples, for example. More will be said about this in the "Problems" chapter.

5. CARRYING OUT REFORMS

A Gnoth-operator task is charged with carrying out the operation or PROGRAM given it by a Reformulator at the time of its creation. It must first check to see that all the structures relevant to its operation are still in existence, having survived the system's activity while the Gnoth-operator was hanging on the taskrack. If the relevant structures do still exist, the Gnoth-operator carries out the operations; if not, it will simply terminate. When a Gnoth-operator does in fact operate, its last action is to decrease the system's bond-strength threshold by 1. The effect of this threshold reduction is to encourage the system to make more reformulations by allowing weaker bonds to be considered, in effect "heating up" the environment. Reformulators, by adding 1 to the threshold, have the opposite effect, cooling things down. Eventually, the system will settle as the Reformulators find fewer and fewer relevant operations to suggest to Gnoth-operators, reflecting the fact that the gnoths are stabilizing.

BOND-ASSESSOR

A Bond-assessor task is created each time a Reformulator decides that it has finished finding interesting gnoth operations at some particular level. The Bond-assessor's job is to look at all current bonds and determine whether or not there is reason to continue reformulation. If there are sufficiently strong bonds to warrant further reforms, the Bond-assessor places a Reformulator on the taskrack. If not, it creates a Gnoth-caster task instead and terminates.

CASTING GNOTHS

When invoked, a Gnoth-caster attempts to describe each gnoth in terms of the reigning class. In more sophisticated versions of the program, there will be provision for casting gnoths in terms of more complicated but still incomplete forms -- such as " (C-group (S-group m n) (Cycle (1 4)))" or " (Y-group [3] ((C-group m n) 8 shared))", where "m" and "n" have no numerical value. This will be necessary when more complicated Seek-Whence descriptions are required to parse target sequences.

Since each gnoth is supposed to represent one frame of the hypothesis, such casting must be possible *if* the class is right and the gnoths are correctly formed.

If all the gnoths can be cast, or if all but the last can and it shows promise, the Gnoth-caster then attempts to create a more general form common to all the castings. For example, if the term groupings generated by the gnoths are: (3 4) (3 4) (3), then the form "(S-group 3 2)" would be generated.

In our slightly more complicated running example, given (1 2) (1 2 3), the form "(S-group 1 (Countup 2))" is generated.

The casting process is quite mechanical, as currently implemented, and so errors or poor castings are possible. A final-test -- to weed out any surviving bad casts -- is made of a cast when it becomes a hypothesis candidate.

TESTING HYPOTHESIS CANDIDATES

The casting form returned, if any, now becomes a hypothesis candidate. A "box", or predictive model, is made for it and is tested to see whether it can accurately "postdict" the known terms of the sequence. If so, the candidate is instantiated as the new hypothesis for the system, which can now sit back in the "certainty" that its new model is the correct one for the given sequence. At this point, the system typically calls for the next term in order to test its new hypothesis.

D. FAILURE AND SLIP-SCOUTS

If the Gnoth-caster is unable to cast all the gnoths in terms of the reigning class, or if it cannot generalize the casts to come up with a candidate, or if the candidate fails to postdict the sequence properly, the reformulation effort has failed. In each such case, a "Slip-scout" process is placed on the taskrack.

Slip-scouts are only skeletally implemented in the current system, a partial explanation for its floundering in many cases when initial reformulation fails. When invoked, a Slip-scout will make a more detailed study of the potential for slipping to another reigning class, and the probability of a class change will increase. The Slip-scout will look at all existing bonds to find frequently-occurring types and will be especially sensitive to the possibility of an interleaving of two or more independent subsequences. It will use the knowledge of which classes favor what bonds to help suggest a new reigning class, or perhaps a subclass within a reigning Cycle or Tuple.

This seems to be the point where Simon and Kotovsky [Simon 63] began their program -- looking for a cycle. If so, we have now almost completed the substrate necessary for a system to support heuristics of their sort in a fluid, non-mechanistic way.

CHAPTER FOUR

COMPARISONS WITH OTHER WORK

A. INTRODUCTION

Inevitably, because the domain we have chosen is that of integer sequences and because we are interested in exploring the process of induction, our work must be compared with that of several predecessors. These include Pivar and Finkelstein, Simon and Kotovsky, Persson, and Dietterich. There are also comparisons and contrasts to be made with work by Evans and by Lenat.

B. COMPARISON WITH PIVAR & FINKELSTEIN

Pivar and Finkelstein [Pivar 64] were interested in "the problem of programming a computer to perform induction on certain general kinds of data in a manner superior to the majority of human beings" (p. 125). Their program was capable of building models of certain types of sequences and of extrapolating from these models more quickly and more accurately than most people. The program could recognize certain well-known sequences, such as the primes, and could devise models with exceptions for non-fitting terms. The target sequence types were cyclic, constant skip, or an intertwining of the two. Thus, the program could "solve" (represent as a LISP function) such sequences as:

2 4 6 8 . . .

2 1 3 2 5 3 7 4 11 5 . . . (primes and positive integers intertwined)

1 4 9 16 25 . . . (squares of positive integers)

However, the *process* of induction, as done by people, was not explored. Their program relied heavily on finite-differencing methods to model polynomial and other highly mathematical sequences, in effect substituting the "black box" of differencing for that of induction.

In fact, they note a difference in thrust between their program and that of Simon and Kotovsky:

> "The program was written as a result of seeing a previous program developed by Simon. Simon's program was developed for the purpose of simulating the observed behavior of people when trying to solve problems of predicting letter sequences from an intelligence test. The program PERTEST, on the other hand, was oriented towards the automation of inductive thinking rather than the simulation of human beings; therefore, we developed somewhat simpler though perhaps more mathematical ways of dealing with the problem." (p. 131).

We feel that in trying to "automate" the process, they were, in fact, looking for a shortcut, a way of obtaining the result of inductive thought -- in this case, a model of the sequence — without having to go through or understand the inductive process itself. In contrast, our major interest is in the process of induction. Sequences of interest to us tend to represent *patterns,* such as:

1 2 1 2 3 1 2 3 4 . . ., or

1 1 2 3 1 2 2 3 1 2 3 3 1 1 2 3 . . .,

rather than nth-degree polynomials or every third Fibonacci number. We want to explore inductive processes that might be similar to those used by humans as they notice and represent patterns; we do not simply want to extrapolate sequences. To paraphrase the mathematician Atiyah (on the NOVA program "Mathematical Mystery Tour"), "we are not simply in the business of getting answers; we want to *understand*". This, then, would seem to put us in the company of Simon and Kotovsky, but there are distinctions to be drawn here as well.

C. RELATION TO SIMON - KOTOVSKY

In their 1963 paper [Simon 63], Simon and Kotovsky presented convincing evidence to support their theory that:

1) people build a mental model of a sequence from the terms they have seen, and

2) they use this model to extrapolate the sequence, to generate successive terms.

In addition, they demonstrated that the most salient features noticed before and during model-construction were sameness and successorship-predecessorship. We heartily agree with all these points. Our differences with Simon and Kotovsky are matters of direction and emphasis and can be described along several dimensions.

Simon and Kotovsky were primarily interested in demonstrating that people do build and use mental models which are developed through a process of induction. In contrast to their work, we simply assume that this is the case. However, we believe that it is important to explore model construction far more deeply.

The Simon-Kotovsky program was presented several terms of a target sequence in a list and proceeded by looking first for periodicity in the data [Simon 63]. Then, once a period was discovered, equal and successor relations between neighboring terms of a period were explored, to finalize the pattern description. In fact, all fifteen of their target sequences were cyclical with fixed-length period. For example, problem 9 was the sequence:

u r t u s t u t t u ___

The resulting formulation was judged either correct or incorrect.

Our approach differs in a subtle but important way; the Seek-Whence system is presented terms of a sequence one at a time. This apparently small difference is the visible tip of a veritable iceberg of processing differences between the two systems. In Seek-Whence, each new term not only inspires a good deal of noticing of samenesses, successorships, and the like but also drives the system to revise its model of the sequence. That is, the processes of model construction and revision go on in parallel with those of noticing. In contrast, to quote Simon and Kotovsky [Kotovsky 73]:

> "The Ss' [human subjects'] behavior departs in one respect from the model. Periodicity is determined by noticing I and N [identity and next -- same and successor] relations. In the computer program, information about relations that are noticed at this stage is not retained, but is regenerated during the second stage, when the pattern description is being built up. Ss clearly retain much or all of this information, and use it while building the pattern description. Thus, the current program separates the two phases of problem-solving activity -- detection of periodicity and pattern description -- more sharply than do the Ss." (p. 410).

Because of the way in which Seek-Whence goes about its modeling job, it is very likely to come up with early formulations of the sequence that are "wrong" in that they will be contradicted by future terms. When this happens, when a contradictory term is entered, the system must react to the failure of its model. It does so by attempting to reformulate the model on the basis of the new evidence (the new term). Thus, Seek-Whence's formulation changes during the course of processing, based upon the "evidence" -- sequence terms -- it has seen so far. We feel that this approach more accurately models human induction, a view supported by the Kotovsky quote above.

Finally, the requirements imposed on the system by its use of reformulation include the need for a different type of model. The Simon-Kotovsky model had to express accurately a description of the sequence.

But, because the description was developed only once and then simply checked for correctness, it could be essentially static in nature. Our model, or *hypothesis*, as we call it, must be modifiable and reactive to failure. The system does not simply go back and apply a machine to the "new" sequence consisting of the old one with one more term at the end in order to generate a new hypothesis. Rather, it analyzes the current hypothesis in the light of the new term's evidence and attempts to change the hypothesis' form to encompass the new term.

In summary, Seek-Whence is directly concerned with the inductive, model-building aspect of the extrapolation of patterned (as opposed to mathematical) sequences. This requires the noticing of relationships among terms and term groupings simultaneously with model creation. Our system, then, needs a different sort of model than did Simon and Kotovsky's. Our model is not simply an end-product defining an extrapolation, but a structure with expressive fluidity, one that is reformulatable on the basis of new evidence, one that evolves as the sequence terms are presented one by one.

D. COMPARISON WITH PERSSON

In 1966, Staffan Persson wrote a series of programs -- "machines", as he called them -- to solve sequence-extrapolation problems [Persson 66]. His main interest appears to have been in extrapolating and identifying "noisy" sequences with underlying generating polynomials, making his domain much like that of Pivar and Finkelstein. This similarity of domain was parallelled by a similarity of approach. Persson, like Pivar and Finkelstein, relied heavily on differencing. He also devised a special machine to extrapolate intertwined sequences. Here again, though, the cycles investigated were always of fixed length.

Persson's interest in error-correction was realized by having the program interpolate correct terms based on the values of the surrounding ones. For example, given as input the segment:

9 16 21 24 blank 24 21 16 9

Persson's program attempts to come up with an explanatory polynomial. Its result is: $-x^2 + 10x + 0$, which it then finds wanting because of the "blank" at the fifth term. It rechecks the polynomial and, finding it explanatory in all other cases (and having been forewarned that there might be one error in the input data), uses the polynomial to interpolate the missing term, a 25, and then extrapolate the sequence [Persson 66, p.128]

Persson recognized that computers solving sequence-extrapolation problems by such methods might be seen as having more capability than they actually possess:

> "At first glance, sequence-extrapolation will seem to require application of genuine induction, i.e., to start out from a pattern, represented by an input-sequence, and eventually arrive at a more general representation from which the input-sequence may be deduced. However, true inductive reasoning is not necessarily required. In many cases, apparent inductive behavior should rather be described as 'deduction disguised as induction'." (sec. 4.)
>
> "... the risk of confusing 'inductive power' with efficient algorithms for exploring very narrow domains must also be realized." (p. 66)

In fact, Persson mentions [Persson 66, pp. 66-7] both Pivar and Finkelstein [Pivar 64], and Simon and Kotovsky [Simon 63] as having claimed inductive behavior in programs which are actually purely deductive in design. We agree with this criticism, and believe that none of the systems thus far discussed addressed the central issue of modeling inductive reasoning.

E. DIETTERICH AND MICHALSKI

"Given a sequence of events (or objects), each characterized by a set of attributes, the problem considered is to discover a rule characterizing the sequence and able to predict a plausible continuation." [Dietterich 85, Abstract]

Clearly, given the above quote and the preceding discussion of Seek-Whence, our interests lie very close to those of Thomas Dietterich and Ryszard Michalski. The questions they ask, the domain explored, and even some of the terminology they use -- e.g.,"structural descriptions", "conceptual clustering", "constructive induction" -- bear a striking resemblance to our own. They, too, obviously reject the idea that sequence pattern induction is a solved problem. However, we and they take a very different approach to processing. They rely on a logic-based formulation and an algorithmic solution technique. We employ structural pattern descriptions and a "terraced scan" [Hofstadter 83; 84] in order to approximate the actual processes of induction.

"SPARC/E", the program discussed in [Dietterich 85], is an advisor to a human who is playing the card game "Eleusis". In this game the dealer, with a card-pattern-generating law in mind, puts down a card. In turn, each player places on the table a card they believe to be in the class of possible next terms. If a player is correct, the card is left on the "main line"; if incorrect, the card is placed on the "side line" below the last correct (main line) card. The positive evidence on the main line in conjunction with the negative evidence on the side lines is used by players in their formulation of a description of the underlying rule. The player who can first formulate the dealer's rule is the winner.

For example, the dealer might put down the Ace of Spades, with the pattern "alternate black and red cards" in mind. If the first player puts down the deuce of Spades (thinking "sequential spades"), the dealer will put the deuce on the side line below the Ace. Should the next player put down the Ace of Clubs, it too will be placed on the side line. If, finally, a player puts down the Ace of Hearts, it will be placed on the main line next to the Ace of Spades. Play will continue until one of the participants guesses the "correct" rule.

The Eleusis advisor program will eventually be called in by its user to analyze a given situation and to try to come up with the "best" generating rule for that situation. Given the board we have described, it might guess "alternating red and black Aces", for instance.

The descriptors for playing cards are initially just suit and rank. Other descriptors, such as color or primeness of rank, can be added later by the user and employed by the system in its analysis. This addition of attributes is one of the four ways in which a game situation can be transformed "in order to facilitate the discovery of sequence-generating rules" [Dietterich 85, p. 200]. The others are:

segmenting -- dividing the sequence into non-overlapping segments, each of which can be described separately;

splitting -- dividing the original into separate subsequences (seeing the original as what we have been calling "interleaved" sequences);

blocking -- creating overlapping segments, called "blocks", and giving attributes to each separately.

In order to devise its rule, the program uses the card descriptions given it as positive and negative evidence in parametrizing each of three different potential models of the sequence (decomposition, periodic, and disjunctive normal form). This model construction is done in stages, using five "rings" or processing levels. Each model is then tidied up as it is passed back up through the rings, assessed for plausibility, and the winning rule or rules are presented to the user as potential organizing notions [Dietterich 85, p. 223]

SPARC/E can solve some fairly intricate problems, situations with rules such as: " strings of the same color ... strings must always have odd length" [Dietterich 85, p.225], or "a higher-rank card in the next 'higher' suit (recall that the suits are cyclically ordered) or a lower-rank card in the next 'lower' suit" [Dietterich 85, p.227].

In spite of the impressive performance of SPARC/E in what is, to us, a very appropriate domain, we have some serious differences of opinion with Dietterich and Michalski on the structure of computer systems designed to perform in inductive domains.

The underlying structures and processing techniques in SPARC/E are logic-based. For example, in the case of the DNF (disjunctive normal form) model, a logical description of the cards on the table is constructed in disjunctive normal form and is fed into the A^q algorithm. This algorithm constructs a "cover" -- a logical description that includes all positive instances and excludes all negative ones -- having the fewest conjunctive terms. The result is passed back up through the processing rings to be presented as a candidate rule. This process has more of a "black box" flavor than we would like; it skirts the central issue (to us) of the process of induction.

Moreover, in SPARC/E processing, all three potential models are always used to construct pattern descriptions; virtually the entire processing structure is brought to bear on each problem, regardless of its "difficulty". We would prefer a system that uses the evidence presented to select a model and to work with that model until it proves fruitless or another seems more appropriate.

Notice that in SPARC/E, an entire situation is given to the system, whereas Seek-Whence continually reacts to new evidence. In SPARC/E, the entire system would have to be restarted for a new game situation; there is no sense of flow or continuity. This means that another central issue, that of reformulation, does not enter into SPARC/E processing. A game analysis, a rule or collection of potential rules, is either "right"or "wrong"; there is no reaction to new data, no response if the generated rules are incorrect.

In spite of these criticisms -- or, more accurately, differences of opinion on what is important -- we have a great deal of respect for Michalski and his group. They have had some real success in constructing useful programs, such as Michalski's soybean-disease classifier, while still maintaining an interest in the core issues of learning and induction. We attempt to concentrate on the "core", but have so far built only a toy.

Dietterich and Michalski have developed some very appealing notions. These include the distinction between "attribute descriptions" -- those which "specify only global properties of an object" -- and "structural descriptions" -- those which "portray objects as composite structures consisting of various components" [Dietterich 83, p. 42]. Certainly, as they note, Patrick Winston's "blocks-world" program [Winston 75] was a ground-breaker in the use of structural descriptions.

The pattern descriptions constructed by Seek-Whence are also structural descriptions. In addition, they can be summarized neatly in their "freeze-dried" form, and so can become part of an attribute-based description. That is, once a concept has been formulated, it can be "captured" in an attribute-description framework. The freeze-dried summary of the concept's structure could be recorded as one of many attributes, and the enclosing frame used in a purely syntactic way. However, any time the concept was used in a semantic way, its underlying structure could be "reconstituted" so that it could have its very critical structural component.

Another appealing Dietterich-Michalski notion, and one that we believe Seek-Whence addresses directly, is that of "constructive induction".

> "Constructive induction is any form of induction that generates new descriptors not present in the input data. It is important for learning programs to be able to perform constructive induction, since it is well known that many AI problems cannot be solved without a change of representation."
> [Dietterich 83, p. 47]

Certainly in Seek-Whence we at least make a valiant attempt to employ a form of constructive induction to come up with a structural description of the input sequence pattern. Eventually, we hope to keep these descriptions (or at least their "freeze-dried" summaries) around to help in the solution of new pattern problems, thus supporting a pattern-remembering system.

F. SOME RELATED SYSTEMS

In addition to the work described above, there have been other programs related to Seek-Whence in spirit, if not in domain. These include Thomas Evans' ANALOGY program [Evans 68] and Douglas Lenat's AM and EURISKO [Lenat 82; 83 a,b,c; 84].

1. EVANS AND ANALOGY

Evans' ANALOGY program was designed to solve pictorial analogy problems, many of which were taken from examinations given to college-bound high-school students by the American Council on Education. They are of the form "A is to B as C is to which of (#1, #2, # 3, # 4, # 5)?", where #1 ... #5 are five candidate pictures. The testee is to choose the candidate that, in its relation to picture C, is most like B's relation to A.

The program was written in two major pieces (primarily because the whole system could not fit into the available computer). Data structures describing the figures in each picture and their positions were fed to the first part of the program. This information was used to form relationships between pictures A and B, as well as between picture C and each of the five candidate pictures. The vocabulary used in describing the relationships consisted of some fixed notions (e.g., "above", "left-of") along with any descriptors the user might decide to add for a particular run (e.g., "shaded", "overlap").

Once the descriptions were made, the system had to choose the "C to candidate" description that was most like the "A to B" description. This was accomplished by assigning weights (importance) to the various types of transformations and formulating "rules" to describe how picture A could be transformed into B, and how C could be transformed into each of the candidates. The A : B rule set was then compared to each C : candidate set. Each A : B rule was "reduced", if possible, to fit a given C : candidate rule. Then the rules were assigned weights based on the transformations they used, the weights were assessed, and the winning candidate — the one with the highest score — was chosen. The program accomplished its task with varying degrees of success, dependent to a great extent on the adequacy of the supplied descriptors to capture the salient relationships in a given problem.

The **ANALOGY** program was an impressive piece of work, but we believe that it is a mistake to attribute to the program powers of "induction" and "theory formation". Here, as in the Pivar and Finkelstein sequence program, we again have a program that can do very well -- probably better than humans -- in a well-defined domain that is really smaller than it would appear at first glance.

Although Evans claimed that the program could probably handle fifteen out of the thirty problems typically given on an ACE exam, we are not given systematic evidence to support that claim. All the problems solved by the system were numbered 12 or lower. It could not handle the only "problem 20" given it. Moreover, the problems were taken from different exams, rather than systematically from one exam. This in itself might simply mean that the program has the inductive power of a sixth-grader rather than that of a high-schooler. However, there are very simple analogy problems from the same ACE exams that the system cannot do [Evans 68, p.325] indicating perhaps a less than human inductive ability, or at least one very different from humans'.

In summary, then, we do not believe that the Evans program can be credited with achieving inductive "concept formation" [Lenat 83a, p.35] because the "concepts" formulated are too brittle, too "attribute-based" (to use the Michalski terminology). We echo the Persson comment (made about Pivar and Finkelstein's sequence-extrapolation program) that the processing technique employed here is really "deduction disguised as induction". Nonetheless, the ANALOGY program is remarkable for its ability to operate in a "core" domain, one that has potential for leading us to central issues in intelligence. It would be a treat to see the program redone in the light of recent thinking about induction, concept formation, and analogy. The domain is one to which artificial intelligence researchers should return "until we get it right".

2. LENAT AND HEURISTICS

Douglas Lenat is deeply concerned with inductive thought. He has explored what he calls "theory formation" in several domains through his programs AM and EURISKO. In particular, he is interested in the development and use of heuristics in discovering and exploring new concepts.

Certainly, both AM and EURISKO have been enormously successful programs. AM is famous for its rediscovery of arithmetic operations, prime numbers, and some important conjectures in number theory. Next to this, what does Seek-Whence have to offer? The answer: roots.

The "accretion model of theory formation", developed by Lenat for the EURISKO system [Lenat 83a], is a program of seven steps to be followed by the system in forming theories about some underlying domain. The model maps out broad sweeps of territory for the system to cover. For example, step 2 of the seven is "to try to notice regularities, patterns, and exceptions to patterns, in the data" [Lenat 83a, p. 37]. Lenat himself recognizes that his program, being concerned with the "big picture", can only approximate a solution to the problems posed in fully implementing step2:

> "Step 2 in the model innocuously requests the learner to be observant for recognizable patterns. That assumes that he/she/it has a large store of known patterns to recognize, or is working in a world where an adequate set can be learned very quickly. "...the process of 'recognizing' blends continuously into 'analogizing'."
> (p.38)

Domains in which Lenat can best employ his heuristics methods have such characteristics as: many objects and operators and many types of both; several types of relations among objects and among operators; lots of heuristics but few algorithms to follow in exploring the domain. These domains should have been little explored previously, and should provide a way to conduct or simulate experiments [Lenat 83b, pp. 91-94]. He advocates studying difficult or complicated domains, ones that are "lush with structure" [Lenat 83c, p.285].

In contrast, the Seek-Whence domain has few objects and is simple in structure. Nonetheless, it represents a complex, if not complicated (to use our terminology from Chapter One), domain in the sense that the central problems of inductive thought can be encountered here. It may be that the broad sweeps and structurally rich domains Lenat favors can be served adequately by an attribute-based representation system, because the concepts grow tall rather than deep. With the accretion model, new ideas are built upon old ones, giving a tower-like effect as the system explores "interesting" ideas to the fullest. Rich underlying structures are not necessary to the type of upward-thrusting concept generation that goes on in these programs; the approximations offered by attribute-based representations are good enough to permit good upward progress. When, however, we stop to explore deeply the small portions -- the nooks and crannies -- of the broadly-swept territory, we need to capture underlying structural descriptions. It may very well be that there is, at present, not enough computing power in a single system to be both broad and deep.

But AM developed its ideas from first principles, from very primitive roots. How can it *not* be deep as well as broad? The answer to this is that AM was accretive. It formulated many ideas, some good and some less fruitful. In a sense, it is akin to a story-generating program as opposed to a story-understander. It could construct whatever ideas it liked; someone -- in fact, Lenat himself -- was bound to notice the "winners". An analogous understander would have to find a way to represent concepts with which it was presented without losing any important facets. The difference between programs of these two types is like the difference between the charges:

"Find something interesting." , and

"Here is an interesting idea. Do you get it?".

Neither problem is particularly easy; they are just different, each with its own difficulties.

Finally, one problem we attempted to address in Seek-Whence was identified very clearly in [Lenat 83c]: "The carrying along of multiple representations simultaneously, and the concomitant need to shift from one to another, has not been much studied, or attempted, in AI to date...." (p. 283) We hope that our efforts to implement a system that supports reformulation will be the first step in attacking this problem.

CHAPTER FIVE

PERFORMANCE, PROBLEMS, AND FUTURE DIRECTIONS

A. IMPLEMENTATION AND PERFORMANCE

The Seek-Whence program currently consists of approximately 5400 lines of Franz Lisp code. Because it is still under development, the program is running interpreted rather than compiled. This, combined with the fact that it runs on a VAX 11/750 which also serves an entire small-college computing operation, slows Seek-Whence down a bit. Nonetheless, successful runs are generally completed in under ten minutes of real time. Unsuccessful runs take a bit longer (potentially forever), as the program thrashes about for a solution.

1. SYSTEM PERFORMANCE

In order to get some perspective on the current program's strengths and weaknesses, let us go through the "Blackburn dozen" -- the twelve sequences we presented to twenty-five college students -- and analyze the system's performance on those problems.

(1) 1 1 2 1 2 3 1 2 3 4 . . .

The program thrashes hopelessly on this one, although it readily solves "1 2 1 2 3 1 2 3 4 . . ."[see Appendix]. The problem seems to be that the initial C-group interferes with the system's ability to find the lengthening S-groups. When it does find them, it seems unable to push the correct notion beyond the template level. Lingering high-level interest in C-groups and low-level rediscovery of C-groups combine to cause this unhappy state of affairs.

(2) 1 2 3 4 . . .

Fortunately, the program can solve this one -- and quite readily, in just under one minute.

(3) 2 1 2 2 2 2 2 3 2 2 4 2 . . .

This is hopeless as yet; we have not even attempted it. There is far too much interference -- terms having multiple potential roles. (For example, in the segment "1 2 2", the middle 2 could be part of a C-group or part of a Countup.) This particular sequence is one of our favorite examples. It has been, and continues to be, a distant goal.

(4) 1 2 2 3 3 3 4 4 4 4 . . .

Turnabout is fair play. Here, the initial S-groups -- (1 2) and (2 3) -- interfere with the budding C-group notion. The central problem here is analogous to that in sequence (1) -- the correct notion is discovered, but cannot seem to break through into a hypothesis. Not surprisingly, in view of the sequence (1) commentary, the system can solve the sequence problems "2 2 3 3 3 4 4 4 4 . . ." and "1 1 2 2 2 3 3 3 3 . . .".

(5) 1 8 5 8 1 8 5 8 . . .

In this sequence, the program finds the Y-group "1 8 5 8 1" and doggedly clings to it. We stopped it after a fifteen-minute attempt, since it seemed to make little progress. Note that it can, however, solve the sequence "1 8 5 8 1 1 8 5 8 1 . . ..

(6) 2 1 2 2 2 3 2 4 2 5 . . .

Again, there is too much interference here, combined with an alternation of terms. This is beyond the current system.

(7) 2 3 1 2 3 2 2 2 3 3 3 3 2 3 4 4 4 4 . . .

This is far beyond the current system. It combines interference, interleaving, and growing group lengths -- all features that make a sequence problem more difficult.

(8) 1 2 2 3 3 4 4 5 . . .

The system solves this within three minutes.

(9) 1 2 3 3 4 4 5 5 5 6 6 6 . . .

This will prove difficult for a while yet. The subtle pattern of growth in group lengths will tax the system's representation scheme.

(10) 9 1 9 2 9 3 9 4 ...

The only problem here is the alternation of terms of which at least one is non-constant. This will probably be the next sequence solved by the system.

(11) 1 8 1 2 1 8 1 2 3 2 1 8 1 2 3 4 3 2 1 8 1 2 3 4 ...

This sequence is hard. There is interference between the groups, which grow at both ends. The expressive power is available, but the system gets bogged down in spurious relationships.

(12) 1 8 5 5 8 1 1 8 5 5 8 1 ...

The system solves this, but can take up to twenty minutes to do so. It finds a Y-group, but often it is the Y-group "(1 8 (5 (5 8 1 1 8 5) 5) 8 1)", rather than the one we would like. The need for "cosmetic reform" becomes evident in cases such as this.

In summary, then, the current Seek-Whence program can solve only three of the Blackburn dozen -- problems 2, 8, and 12. With slight extension, it should solve problem 10 as well. It will have to cling less forcefully to its original formulation in order for it to solve problems 1, 4, 5, and 6. The system's interference handling will need improvement before it can handle problems 3, 7, and 11. The solution of problem 9 will probably require that group lengths be used as manifestations (they are not, currently). In addition, the system will need the ability to use its representational power more effectively.

2. HUMAN PERFORMANCE

When we presented these sequence problems to our human subjects [Meredith 83], we permitted them to take as much time as they wanted on each sequence. A subject could "pass" on a particular sequence if it proved insoluble. The subject could not return to a passed sequence.

We kept a record of the number of people who passed on each of the sequences. We also timed the subjects, in order to determine which sequences took the longest time to parse. We assume that these will tend to be the most difficult for human solvers.

Problem (7) was definitely the most difficult for our subjects. Seven people passed on it (no more than two people passed on any other sequence), and those who did solve it took far more time on it than on any other sequence. Problem (11) was also clearly more time-consuming than most others. The "easiest" problems were (2) and (10), followed by (8), then (4) and (5), then (6) and (12), and then (1), (3), and (9).

We find it heartening that the problems Seek-Whence has been able to solve, and those which we feel it is closest to solving, are among the easier problems for humans, while those our system finds difficult are also difficult for humans.

B. PROBLEMS

The original goals set for the Seek-Whence program were and still are:

1) to discover non-mathematically-sophisticated patterns in sequences of nonnegative integers ;
2) to represent those patterns as concepts constructed from eight "primitive" concepts -- Constant, Countup, C-group, S-group, P-group, Y-group, Cycle, and Tuple;
3) to be able to reformulate the pattern descriptions fluidly, by the technique of "slipping", when the description is non-predictive or non-optimal.

Each of these goals has been met to some extent, but more work will be required to implement a system that realizes them in full. From our discussion above, it becomes clear that the Seek-Whence program

1) fails to notice interleaved sequences of any complexity;

2) is unable to handle interference well;

3) clings too tenaciously to its first organizing notion.

In the following sections, we will discuss these and other problems and will present our current thoughts as to how to solve them.

1. IMPLEMENTATION FAUX PAS

As in any fairly substantial system written over a period of years, there are no doubt some inconsistencies and quirks in the current implementation of Seek-Whence. The present system was programmed by one person, and so reflects the weaknesses and idiosyncracies of a particular style. These include a fairly conservative, but readable, expr-based approach to Lisp programming and some disregard for "neatness" in cleaning up old, unwanted structures.

Seek-Whence is unabashedly "ad hoc". There has been no focus on separating domain-dependent from domain-independent processing, structures, or approaches. The only excuse for this is that the program is a proto-effort in the development of a generic processing structure and approach. People with similar ideas have been programming and continue to program systems for Jumbo (word unscrambling), Letter Spirit (style extrapolation), and Copycat (letter-sequence analogies). When all the systems are completed, we will hopefully be able to abstract out common, domain-independent features which will be generally useful. This is a "high-risk, high-gain" strategy. We hope it works.

If all of our problems were ones of programming style, we would be delighted. Unfortunately, there are some more fundamental worries, not the least of which is that there are some non-difficult sequences that the system cannot parse.

2. UNCONQUERED SEQUENCES

Although Seek-Whence does a good job in analyzing the simplest of sequences and can do some medium-difficulty ones, it fails on some not-very-hard ones. It is unable to handle independent interleaved sequences when the components are any more complicated than constants. That is, it can do "3 7 3 7 3 7..." but it cannot as yet do "1 2 10 3 4 10 5 6 10...".

A major reason for this problem is the way bonds are used by the system. Currently, bonds are used only in a bottom-up fashion, to push up gloms. However, there is knowledge in the platoplasm of the bond types favored by the various Platonic classes. For example, the existence of many "adjacent sameness" bonds might be a clue that C-group is a strong candidate as an organizing notion, because C-groups are closely associated with such bonds. As yet, the system makes no direct use of this information. It is important to note that such information must be used cautiously, since it may lead to false conclusions. In the sequence "2 1 2 2 2 2 2 3 2 2 4 2 . . .", for example, there are many adjacent samesses between 2's, but the "C-group" notion is not involved in the correct parse.

Knowledge about manifestations and actualizations, which could be useful in suggesting alternative organizing notions or in indicating the existence of interleaved sequences, is virtually unused by the current system. Slip-scouts, described later, will begin to make some use of this information.

The Seek-Whence system cannot analyze sequences that display a good deal of interference -- such as "2 1 2 2 2 2 2 3 2 2 4 2. . ." or "1 1 1 1 2 1 1 3 1. . .". People seem to overcome interference by looking for a place in the sequence where there is little confusion -- a place where the interference is minimal. Seek-Whence may need to look more closely at terms that have few bonds and use these a guideposts for organizing the sequence. This strategy, like looking for interleaved sequences, is a relatively high-level one, suitable at the Slip-scout level and beyond.

The system's inclination to cling to early organizing notions is related to the other two problems, and probably stems from the same root causes. In addition, we may have to tinker with our slipping mechanisms, to see if we can get a bit more movement away from failed ideas.

3. LOW-LEVEL MYOPIA

The low-level processes of Seek-Whence operate with a micro-level vocabulary, dealing with localized structures and providing no overview of the sequence pattern as a whole. This naturally leads to the phenomenon that we call "low-level myopia". There can be some micro-level rigidity as a result, with the lower-level processes clinging to certain favorite groupings (usually gloms formed early in the processing). This can get in the way of pushing up neatly balanced structures -- we can get " ((1 2) 3)" handed up instead of a preferred "(1 2 3)" -- but it is not a devastating problem. Its effects will be mitigated when divesting pushes, cosmetic reform, and "form-polishing" are implemented.

4. HIGH-LEVEL HAUGHTINESS

The higher levels of Seek-Whence seem to suffer as well from some basic rigidity. Once the high levels take over, the imposition of top-down, model-driven processing does not appear to leave quite enough room for lower-level coercion of change. This leads the system to stick with a formulation type or platonic class longer than it should, to be optimally effective. It becomes too difficult for lower-level processes to push up a notion with sufficient force to stage a "coup".

We have often watched in frustration as a good notion has come up repeatedly to become a template, and then to disappear, never to reach hypothesis status. We plan to investigate this unfortunate phenomenon, which we call the "Little Prince Problem":

Low levels: "See my pretty bond-chain?"

High levels: "Not now -- I'm trying to parse this sequence."

DIVESTING PUSHES

A "divesting push" will occur when a gnoth contains a glom that causes it "unhappiness" in the sense of decreasing its stability, but the neighboring gnoth does not have any particular attraction for the glom either. In this case, the parent gnoth may push the glom out to the neighbor or may simply call for the creation of an intervening gnoth to hold the unwanted glom. These pushes will permit gnoths to work on conforming to the hypothesis, or suggesting weaknesses in it. Implementation of divesting pushes will be a first step in giving more credence to low-level suggestions, thereby decreasing the degree of "high-level haughtiness". They will also serve as a safety valve for the current reigning class, by increasing gnoth stability without calling for a new monarch.

5. COORDINATION PROBLEMS

Although Seek-Whence relies on independent, parallel processes to carry out its work, there is nonetheless some need for coordination of results. For instance, the hypothesis and the gnoths must be in agreement (at least to some extent) on the current view or parse of the sequence. Devising a technique for insuring this coordination has been a major problem, and one which we are not certain is solved at present. The levels of hypothesis-gnoth equivalence give us something of a handle on the problem, but it would be nice not to have to worry about it at all. That is, it would be nice simply to change either the gnoths or the hypothesis and be certain that the other would automatically fall into agreement. We have not yet devised such a mechanism, nor are we sure that one exists.

6. HERKY-JERKY

One goal of Seek-Whence was fluid reformulation, the ability to move easily from one concept representation to another. The current system is only partially successful in meeting this goal. Its reforms, at the highest level, can seem a little rough. Instead of the smooth transition we want, we get something more akin to the jerky motion felt when one rides to the top of the Gateway Arch in St. Louis -- one gets there, but the ride is not as continuous as one would like it to be. This may point to the need for another or level or two of processing to ease the transitions, or it may simply require more care in programming. Below, we suggest the possibility that a richer system of linkages in the platoplasm might help mitigate this problem.

DIFFERENTIATING PLATO-LINKS

The platoplasm's link system currently consists of undifferentiated "slipping links" -- the s-links. It is very likely that in using differentiated links, we would be able to give the system a more rational collection of slipping alternatives and the ability to apply more constraints on slippage possibilities in particular situations. That is, instead of having to consider slippage possibilities on the relatively gross grounds of s-link slipperiness in conjunction with "absolute bond pulls", the system may be able to use a finer-grained decision strategy. We therefore need to investigate more deeply what types of links belong in the platoplasm and how best to incorporate them into the system's processing. This is a very big question in an abstract sense, but implementation in Seek-Whence should not be too difficult, and may go a long way toward conquering the "herky-jerky" problem.

C. THE FUTURE

We plan to revise and extend the Seek-Whence system in several ways in the future, and at many levels of abstraction. There are some relatively minor details that need to be addressed, some major additions to be made, and ultimately we will have to redo the system in a more structured, domain-independent fashion.

1. MINOR DETAILS

Some of the minor reforms will be fairly simple to include, but one or two will require some careful thought before implementation can be considered.

GREASING PLATO-LINKS

As was previously mentioned, it is possible that various platonic classes will be "closer" to a given class at different times. This means that the s-links between concepts should have different slipperiness values at different times. The current system does not provide any mechanism for changing s-link slipperiness, nor does it explore the notion of "relative closeness" in any way. It would be interesting to investigate this question a bit further in later versions of the program. This is an example of an addition that will be fairly easy to implement once we decide exactly what we want to do.

CHANGING PLATONIC BOND STRENGTHS

Similarly, the degree to which a given platonic class favors certain types of bonds may change during the course of processing. Changing the bond strengths would not be hard to implement, but the central question -- not a particularly easy one -- would be how to have the system decide when it should be done and how much to change the strengths.

ADDING AND REMOVING BOND-FIELDS

An interesting problem is the central one of "salience". What features of a sequence are of central importance? What should be used to describe it? We have built into Seek-Whence the capacity to use any field of a glom for bonding or glomming purposes, but as was mentioned earlier, we currently use only "value" for glomming and "value" or " span" for bonding. Building in a real capacity to add to or subtract from these fields is critical in accurately parsing some sequences -- such as " 1 2 2 3 3 3 4 4 4 4..." -- where the length of a group and its content or position in the sequence are intimately connected. We certainly hope to build this capacity into future versions of the system.

BOX STRUCTURE EDITOR

A nice little project associated with Seek-Whence, but outside of the mainstream of its processing, is the construction of a "box-tree" editor. The system could use this to model its own reformulation actions by editing a hypothesis' box to reflect a new modification of the hypothesis. The current (heavy-handed) technique is to completely scrap and replace the box.

2. MAJOR GOALS

We have some major plans for future revisions of Seek-Whence, in addition to the "fix-ups" mentioned above. These deal with broader issues within the domain of our project, issues with perhaps more "global" significance.

FORM POLISHING

"Form polishing" is the term we use to cover the notions of cosmetic reform -- reformulation done to improve the look of a hypothesis -- and internal gnoth reformulation in order to achieve structural equivalence with the hypothesis. A gnoth displays structural equivalence with the hypothesis when its actual, glom-based form agrees with the deep-structure form given it as a model. The deep-structure form is that of one frame of the hypothesis -- the frame corresponding to the gnoth. These reforms will probably not be easy to carry out, because they are not central to having a "correct" parse of the sequehce, but rather the "best" parse, and for the "right" reason. That is, form-polishing is more heuristic than is parenthesization of the sequence, and so its implementation will probably be even less deterministic than normal Seek-Whence processing.

USING MANIFESTATIONS -- SLIP-SCOUTS

One of our major goals for the future will be to implement "Slip-scouts", processes that will begin to use information that the system has gathered about the sequence, but has as yet not used. Slip-scouts will be looking at bonds, manifestations, and actualizations, in order to suggest ways in which the sequence could be parsed. They will be especially sensitive to interleaved independent sequences, such as "1 2 10 3 4 10 5 6 10 ...", and will suggest parses with deeper nesting of structures than is required for the simpler types of sequences. The addition of Slip-scouts is extremely important if the system is going to move on to parse more difficult sequences, and so will be one of the first goals we attack.

FINER-GRAINED REFORMS

When reformulation is required, we now use a rather heavy-handed approach -- reform at the top. What the system now needs is the ability to perform finer-grained reformulations, perhaps retaining the reigning class as monarch, but adding some "epicycles" to the hypothesis. The reigning class may be the right one, but because there are deeply-nested structures which the system does not perceive as such, there may be a good deal of "unhappiness" in the system -- the stability may be low. Rather than toss the monarch out, the system should sometimes investigate other reforms, reforms geared toward finding a deeper explanatory structure.

LEARNING

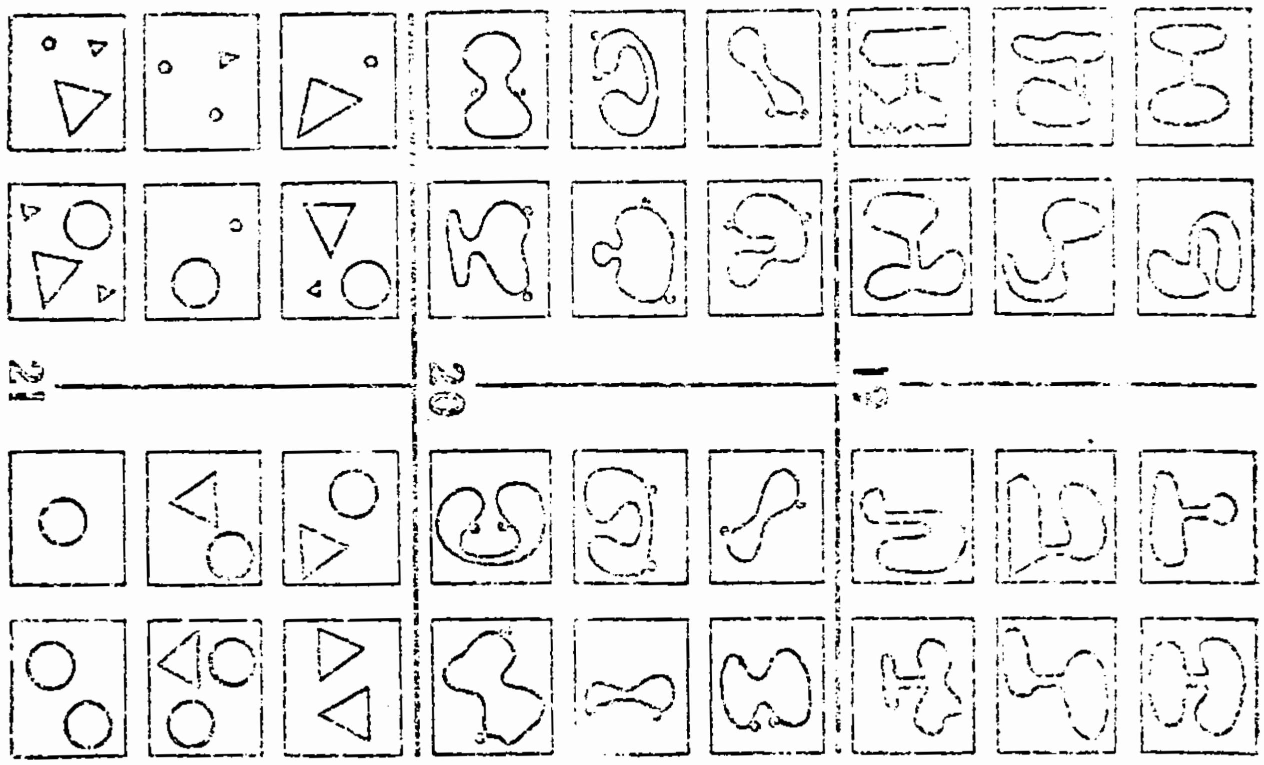
There are two essential requirements for a successful "inductive learning" program. First, it must discover that which it is to learn. Second, it must remember what it has discovered. The Seek-Whence program has made some progress in the area of discovery. Unfortunately, as currently structured, Seek-Whence does not "remember" a parsed sequence in order to aid in parsing another, or for purposes of comparison.

We would like to build upon our idea of "freeze-dried" hypotheses to implement a facility whereby old, remembered hypotheses could, in essence, offer themselves up as models for parsing new sequences. That is, the old hypotheses could be loosely "plugged in" at various levels of the system, and when a similar structure is created could interrupt the proceedings to present themselves as potential models. This, and the ability to do Bongard-like analogy and generalization problems with collections of sequences, are more removed, but potential goals for future research.

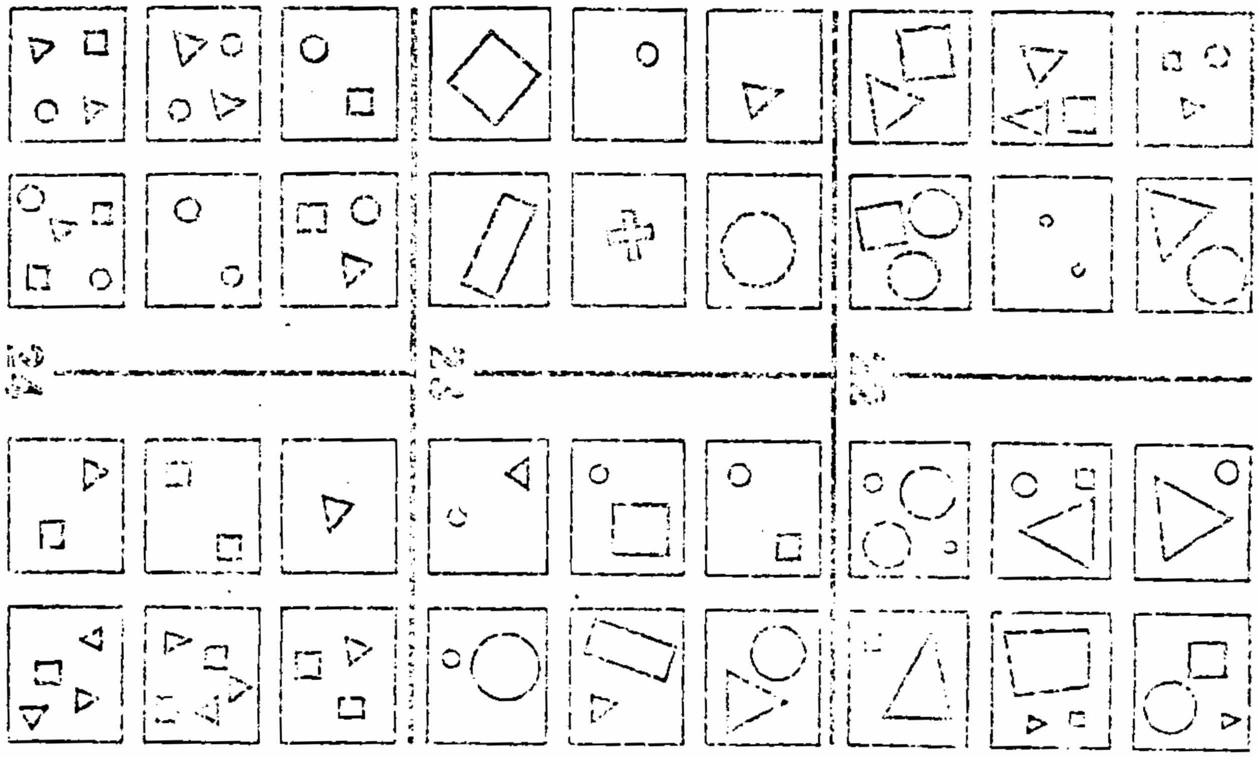
D. CONCLUSION

Seek-Whence is not a perfect program. It suffers from problems at several levels and of several types. Nonetheless, it does serve as an example of a new approach to the programming of "intelligent" systems, a sample of a new paradigm. The hallmarks of this approach are: concepts with underlying levels of representation; a representation scheme that encourages fluid reformulation; the ability to accept and react to evidence; and a nondeterministic, parallel system organization. We believe that these are important notions, ones that should be explored further and in many domains. They may prove useful -- and even critical -- in the development of systems that possess "common sense" and the ability to relate concepts in unexpected and novel ways.

APPENDIX



21
20
19



24
23
22

The program's equivalent of the form

(C-group (Countup 1) shared):

```
-> (build-box
        '(C-group (same pstruc1) (same pstruc1))
          '((pstruc1  (Countup 1))))

box5
-> (show-box 'box5)
        (1)
-> (show-box 'box5)
        (2 2)
-> (show-box 'box5)
        (3 3 3)
-> (show-box 'box5)
        (4 4 4 4)
```

The program's equivalent of the form

```
(Tuple 1 3 ((Countup 1) 8 shared))):


-> (build-box
        '(Tuple 1 3 ((same pstruc2) 8 (same pstruc2)))
              '( (pstruc2  (Countup 1))))

box12
-> (show-box 'box12)
        (1 8 1)
-> (show-box 'box12)
        (2 8 2)
-> (show-box 'box12)
        (3 8 3)
-> (show-box 'box12)
        (4 8 4)
```

The program's equivalent of the form

```
(Cycle 1 3 ((Countup 1) 8 shared)):


-> (build-box
        '(Cycle 1 3 ((same pstruc3) 8 (same pstruc3)))
              '( (pstruc3  (Countup 1))))

box19
-> (show-box 'box19)
        1
-> (show-box 'box19)
        8
-> (show-box 'box19)
        2
-> (show-box 'box19)
        3
-> (show-box 'box19)
        8
-> (show-box 'box19)
        4
-> (show-box 'box19)
        5
-> (show-box 'box19)
        8
-> (show-box 'box19)
        6
```

The Sequence 3 7 3 7 3 7

```
-> (startup)

please enter a term: 3
doing task Sparkler-plus on (glint1 glint1 2)
doing task Dissolver

please enter a term: 7
doing task Sparkler
spark1 --- between glint1glint2
doing task Tester
doing task Dissolver on (glint1)
doing task Sparkler
spark2 --- between glint1glint2
doing task Sparkler-plus on (glint2 glint1 10)
spark3 --- between glint2glint1
doing task Sparkler-plus on (glint2 glint2 2)
doing task Sparkler-plus on (glint2 glint1 10)
spark4 --- between glint2glint1
doing task Tester
doing task Tester
doing task Sparkler-plus on (glint2 glint1 2)
spark5 --- between glint2glint1
doing task Tester
doing task Tester

please enter a term: 3
doing task Sparkler-plus on (glint3 glint1 2)
spark6 --- between glint3glint1
doing task Dissolver on (glint2)
doing task Sparkler
doing task Sparkler-plus on (glint3 glint2 10)
spark7 --- between glint3glint2
doing task Sparkler-plus on (glint3 glint3 2)
doing task Sparkler-plus on (glint3 glint2 2)
spark8 --- between glint3glint2
doing task Tester
doing task Tester
doing task Sparkler
doing task Sparkler-plus on (glint3 glint2 10)
spark9 --- between glint3glint2
doing task Tester
doing task Tester
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
bond1 --- between glint1glint3
doing task Sparkler
doing task Sparkler
spark10 --- between glint3glint1
doing task Glom-scout
```

```
Same -cover proposed --> glint1
Same -fence proposed --> glint1
doing task Glomtester on (Same cover glint1)
doing task Glomtester on (Same fence glint1)
;  The system now gloms the first two terms,
;        giving a parse of  (3 7) 3.

doing task Glommer on
 (Same print-value fence (glint1 glint2)).
Glommer for Same print-value fence
 members: (glint1 glint2)
doing task Sparkler-plus on (glom1 glom1 10)
doing task Glom-inspector on (glom1)
doing task Glommer on
 (Same print-value cover (glint1 glint2 glint3))
failed to glom (glint1 glint2 glint3)
doing task Plato-scout on (Cycle glom1)
doing task Tester
doing task Sparkler
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
doing task Template-scout on (glom1)
doing task Template-applier on
 (glom1 (Cycle 3 2 (3 7)))
create-template-glom (Cycle 3 2 (3 7))(glom1)
top-down glom glom2

;  A template is made.

template made : (form (Cycle 3 2 (3 7))
doing task Template-evaluator
check-cycle template (Cycle 3 2 ((3 7)))

;  A hypothesis is created.

doing task Hypothesizer
(Cycle 3 2 (3 7))
doing task Glom-scout
doing task Sparkler
spark11 --- between glint1glint2
doing task Sparkler-plus on (glom1 glint3 10)
spark12 --- between glom1glint3
doing task Sparkler
doing task Gnoth-maker
top-down glom glom3
gnoths constructed
doing task Tester
doing task Tester
doing task Call-term
```

```
please enter a term: show-hypothesis
(Cycle 3 2 (3 7))


;  The next term will confirm the hypothesis.

please enter a term: 7
doing task Hfilter
new term being hypothesis-filtered
 through (Cycle 3 2 (3 7))
top-down glom glom4

I have a guess!

;  The system ventures a guess.

hypothesis: (Cycle 3 2 (3 7))
     3 7 3 7

;  It is correct -- this time.

enter no if wrong, ok if right ok
bye
```

The Sequence 3 7 3 3 7 3

```
please enter a term: 3

please enter a term: 7
spark1 --- between glint1glint2
spark2 --- between glint1glint2
spark3 --- between glint2glint1
spark4 --- between glint2glint1
spark5 --- between glint2glint1

please enter a term: 3
spark6 --- between glint3glint1
spark7 --- between glint3glint2
spark8 --- between glint3glint2
spark9 --- between glint3glint2
bond1 --- between glint1glint3
spark10 --- between glint3glint1
Same -cover proposed --> glint1
Same -fence proposed --> glint1
Glommer for Same print-value fence
 members: (glint1 glint2)
failed to glom (glint1 glint2 glint3)
create-template-glom (Cycle 3 2 (3 7))(glom1)
top-down glom glom2

;  A template is created after three terms.

template made : (form (Cycle 3 2 (3 7))
 state working coverage (1 2) glom glom2)
check-cycle template (Cycle 3 2 ((3 7)))
spark11 --- between glint1glint2
spark12 --- between glom1glint3
top-down glom glom3
gnoths constructed

;  We ask the system to "show" us its structures.

please enter a term: show
terms of the sequence:
3 7 3

bonds:
bond1 Same print-value (remote) -- (glint1 glint3)

gloms:
glom1 (Same print-value fence) --> (3 7) terms 1 to 2
glom3 pseudo --> ((3 7))  terms 1 to 2
```

```
gnoths:

class: Gnoths
name: gnoth1
frame: 0
plato-class: Cycle
glom: glom3
notes: nil
form: (((Cycle 3 2 (3 7)) pure))
state: stable
range: (1 2)

;  A hypothesis was made.  We ask to see it

please enter a term: show-hypothesis
(Cycle 3 2 (3 7))

;  The next term will deny the hypothesis.

please enter a term: 3
new term being hypothesis-filtered
 through (Cycle 3 2 (3 7))
spark13 --- between glint4glint2
spark14 --- between glint4glint3
spark15 --- between glint4glint3
spark16 --- between glint4glint1
spark17 --- between glint4glint3
set-out -- validity: 0
top-down glom glom4
groups: ((glint1 glint2))
glom: (glint1 glint2)
top-down glom glom5
top-down glom glom6
top-down glom glom71
top-down glom glom8
gnoths: (gnoth1 gnoth2 gnoth3)

;  The system will continue to let "Cycle" reign.

slip-check: stayval: 0.0
       best: nil
spark18 --- between glint3glint4
bond2 --- between glint3glint4
spark19 --- between glint3glint4
spark20 --- between glint4glint3
bond3 --- between glint1glint4
spark21 --- between glint4glint1
casts: ((Cycle 3 1 (3)) (Cycle 7 1 (7))
 (Cycle 3 1 (3))) (Cycle 3 3 (3 7 3))
```

```
;  A new hypothesis is made.

new hypoth candidate (Cycle 3 3 (3 7 3))
spark22 --- between glint1glint2

please enter a term: show-seq
3 7 3 3

;  The next term will confirm the new hypothesis.

please enter a term: 7
spark23 --- between glint5glint1
new term being hypothesis-filtered
 through (Cycle 3 3 (3 7 3))
top-down glom glom9

I have a guess!

hypothesis: (Cycle 3 3 (3 7 3))
3 7 3 3 7

;  The hypothesis is correct.

enter no if wrong, ok if right ok
bye
```

The Sequence 1 2 1 2 3 1 2 3 4

```
-> (startup)

please enter a term: 1
doing task Sparkler-plus on (glint1 glint1 2)
doing task Dissolver
doing task empty-task

please enter a term: 2
doing task Sparkler
spark1 --- between glint1glint2
doing task Tester
doing task Sparkler-plus on (glint2 glint1 10)
spark2 --- between glint2glint1
doing task Sparkler-plus on (glint2 glint1 10)
spark3 --- between glint2glint1
doing task Dissolver on (glint1)
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
bond1 --- between glint1glint2
doing task Tester
doing task Glom-scout
Pred -group proposed --> glint2
doing task Glom-scout
Pred -group proposed --> glint2
doing task Glomtester on (Pred group glint2)
doing task Tester
doing task Glommer on
 (Pred print-value group (glint1 glint2))
Glommer for Pred print-value group members: (glint1
glint2)
doing task Glomtester on (Pred group glint2)
Glomtester failed glint2
doing task Sparkler
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Glom-inspector on (glom1)
doing task Sparkler-plus on (glint2 glint1 2)
spark4 --- between glint2glint1
doing task Tester
doing task Plato-scout on ((S-group P-group) glom1)
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Plato-evaluator on (P-group glom1)
doing task Sparkler-plus on (glint2 glint2 2)
doing task Sparkler
doing task Template-scout on (glom1)
create-template-glom (S-group 1 2)(glom1)
top-down glom glom2
```

```
;  A template is made.

template: (form (S-group 1 2)
doing task Sparkler
spark5 --- between glint1glint2
doing task Template-evaluator on nil
doing task Sparkler-plus on (glom1 glom1 10)
doing task Gnoth-maker
top-down glom glom3
gnoths constructed
doing task Sparkler

;  Now comes the first hypothesis.

doing task Hypothesizer
(Countup 1)
doing task Tester
doing task Call-term

please enter a term: show-hypothesis
(Countup 1)

please enter a term: show
terms of the sequence:
1 2

bonds:
bond1 Succ print-value (adjacent) -- (glint1 glint2)

gloms:
glom1 (Pred print-value group) --> (1 2) terms 1 to 2
glom3 pseudo --> ((1 2))  terms 1 to 2

gnoths:

class: Gnoths
name: gnoth1
frame: 0
plato-class: S-group
glom: glom3
notes: nil
form: (((S-group 1 2) pure))
state: stable
range: (1 2)

;  The next term denies the hypothesis

please enter a term: 1
doing task Hfilter
new term being hypothesis-filtered
 through (Countup 1)
```

```
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Sparkler-plus on (glint3 glint1 10)
spark6 --- between glint3glint1
doing task Sparkler-plus on (glint3 glint2 10)
spark7 --- between glint3glint2
doing task Sparkler-plus on (glint3 glom3 2)
doing task Sparkler-plus on (glom1 glint3 10)
spark8 --- between glom1glint3
doing task Sparkler-plus on (glint3 glint2 10)
spark9 --- between glint3glint2
doing task Tester
doing task Sparkler-plus on (glint3 glom3 10)
doing task Bonder on
 (Pred print-value (adjacent) glint2 glint3)
bond2 --- between glint2glint3
doing task Sparkler-plus on (glint3 glint3 2)
doing task Sparkler-plus on (glint3 glom3 2)
doing task Tester
doing task Dissolver on (glom3)
glom3 is not in cytoplasm
doing task Tester
doing task Sparkler-plus on (glint3 glint1 10)
spark10 --- between glint3glint1
doing task Sparkler-plus on (glint3 glint2 10)
spark11 --- between glint3glint2
doing task Tester
doing task Tester
doing task Sparkler-plus on (glom1 glint1 10)
doing task Sparkler-plus on (glint3 glint3 2)
doing task Bonder on
 (Pred print-value (adjacent) glint2 glint3)
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
bond3 --- between glint1glint3
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
doing task Glom-scout
doing task Glom-scout
doing task Bonder on
 (Pred print-value (adjacent) glint2 glint3)
doing task Sparkler
spark12 --- between glint1glint3
doing task Sparkler
doing task Glom-scout
Same -cover proposed --> glint3
Same -fence proposed --> glint3
doing task Sparkler
spark13 --- between glint1glint2
```

```
doing task Tester
doing task Glomtester on (Same cover glint3)
doing task Tester
doing task Glomtester on (Same fence glint3)
doing task Sparkler
doing task Tester
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
doing task Sparkler
doing task Sparkler
spark14 --- between glint1glint3
doing task Sparkler-plus on (glom1 glom1 10)
doing task Sparkler
doing task Bonder on
 (Pred print-value (adjacent) glom1 glint3)
bond4 --- between glom1glint3
doing task Sparkler
spark15 --- between glint2glint1
doing task Glom-scout
Same -cover proposed --> glint3
Same -fence proposed --> glint3
doing task Glomtester on (Same fence glint3)
doing task Tester
doing task Bonder on
 (Same print-value (remote) glint1 glint3)
doing task Glomtester on (Same cover glint3)
doing task Sparkler
doing task Gnoth-setter
top-down glom glom4
top-down glom glom5
top-down glom glom6
top-down glom glom7
top-down glom glom8
gnoths: (gnoth1 gnoth2 gnoth3)
doing task Glom-scout
doing task Tester
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Plato-scout on
 ((C-group S-group P-group Y-group Cycle Tuple)
         glom6)
doing task Template-scout on (glom6)
doing task Template-scout on (glom6)
doing task Sparkler
doing task Template-scout on (glom6)
doing task Sparkler
doing task Template-scout on (glom6)
doing task Sparkler
spark16 --- between glint1glint2
doing task Template-scout on (glom6)
doing task Template-scout on (glom6)
doing task Sparkler
doing task Reformulator

;  Changing to a new reigning class.
;  Countup -->  S-group

doing task Bond-assessor on (S-group 8.0)

;  Reformulation is performed.

doing task Gnoth-operator
((PROGRAM ((SHIFT-RIGHT gnoth1 gnoth2 (glint1))
 (ENCLOSE gnoth1 nil))))
top-down glom glom9
doing task Glom-scout
doing task Reformulator
doing task Tester
doing task Bonder on
 (Succ print-value (adjacent) glint1 glint2)
doing task Sparkler
doing task Sparkler
doing task Bond-assessor on (S-group 8.0)
doing task Reformulator
doing task Bond-assessor on (S-group 9.0)
doing task Reformulator
doing task Bond-assessor on (S-group 10.0)
doing task Gnoth-caster
casts: ((S-group 1 2) (S-group 1 1))
(S-group 1 2)
new hypoth candidate (S-group 1 2)
doing task Call-term

;  A second hypothesis has been devised.

please enter a term: show-hypothesis
(S-group 1 2)

please enter a term: show-parse
((1 2) (1))

please enter a term: show
terms of the sequence:
1 2 1

bonds:
bond1 Succ print-value (adjacent) -- (glint1 glint2)
bond2 Pred print-value (adjacent) -- (glint2 glint3)
bond3 Same print-value (remote) -- (glint1 glint3)
```

```
gloms:
glom8 pseudo --> (1)  terms 3 to 3
glom9 pseudo --> (1 2)  terms 1 to 2

gnoths:

external-bonds: ((bond2 0) (bond3 -5))
internal-bonds: ((bond1 10))
equivalence-type: parse
groups: nil
class: Gnoths
name: gnoth2
frame: 1
plato-class: S-group
glom: glom9
notes: nil
form: (S-group 1 2)
state: stable
range: (1 2)
external-bonds: ((bond2 0) (bond3 -5))
internal-bonds: nil
class: Gnoths
name: gnoth3
frame: 2
plato-class: S-group
glom: glom8
notes: nil
form: (S-group 1 1)
state: stable
range: (3 3)

;  The next term confirms the hypothesis,
;  although it is incorrect.

please enter a term: 2
doing task Hfilter
new term being hypothesis-filtered through (S-group 1
2)
top-down glom glom10

;  The system ventures a guess.

I have a guess!

hypothesis: (S-group 1 2)
(1 2)(1 2)(1 2)

;  It is wrong this time.

enter no if wrong, ok if right nope
```

```
please enter a term: show-seq
1 2 1 2

;  A new term is entered.

please enter a term: 3
doing task Hfilter
new term being hypothesis-filtered through
(1 2)1
doing task Sparkler-plus on (glint5 glint4 10)
spark17 --- between glint5glint4
doing task Sparkler-plus on (glint5 glom8 2)
doing task Sparkler-plus on (glint4 glint1 10)
doing task Sparkler-plus on (glint4 glint3 10)
spark18 --- between glint4glint3
doing task Dissolver on (glom8)
glom8 is not in cytoplasm
doing task Sparkler-plus on (glint4 glint3 10)
spark19 --- between glint4glint3
doing task Sparkler-plus on (glint5 glint1 10)
spark20 --- between glint5glint1
doing task Sparkler-plus on (glint5 glom10 2)
doing task Sparkler-plus on (glint5 glint3 10)
spark21 --- between glint5glint3
doing task Sparkler-plus on (glint5 glint5 2)
doing task Sparkler-plus on (glint5 glint2 10)
spark22 --- between glint5glint2
doing task Dissolver on (glom10)
glom10 is not in cytoplasm
doing task Sparkler-plus on (glint4 glint2 10)
spark23 --- between glint4glint2
doing task Sparkler-plus on (glint4 glom8 2)
doing task Sparkler-plus on (glint5 glint2 10)
spark24 --- between glint5glint2
doing task Gnoth-setter
top-down glom glom11
top-down glom glom12
top-down glom glom13
gnoths: (gnoth2 gnoth3 gnoth5)

;  We will stay with the reigning class -- S-group.

slip-check: stayval: 15.0
       best: (Y-group 4.0)
doing task Tester
doing task Bonder on
 (Same print-value (remote) glint2 glint4)
bond5 --- between glint2glint4
doing task Tester
doing task Tester
doing task Sparkler-plus on (glint5 glint3 10)
spark25 --- between glint5glint3
doing task Tester
doing task Bonder on
 (Succ print-value (remote) glint2 glint5)
bond6 --- between glint2glint5
doing task Sparkler
doing task Bonder on
 (Succ print-value (adjacent) glint3 glint4)
bond7 --- between glint3glint4
doing task Sparkler
doing task Sparkler-plus on (glint5 glom10 10)
doing task Sparkler-plus on (glint5 glom10 2)
doing task Tester
doing task Sparkler-plus on (glint5 glom8 2)
doing task Reformulator
doing task Sparkler-plus on (glint5 glint4 10)
spark26 --- between glint5glint4

;  More reformulation is performed.

doing task Gnoth-operator on
 ((SHIFT-RIGHT gnoth3 gnoth5 (glint3)))
top-down glom glom14
doing task Tester
doing task Bond-assessor on (S-group 8.0)
doing task Sparkler-plus on (glint5 glint4 10)
spark27 --- between glint5glint4
doing task Sparkler
doing task Tester
doing task Sparkler
spark28 --- between glint4glint5
doing task Bonder on
 (Succ print-value (adjacent) glint4 glint5)
bond8 --- between glint4glint5
doing task Sparkler-plus on (glint5 glint1 10)
spark29 --- between glint5glint1
doing task Tester
doing task Tester
doing task Sparkler
doing task Sparkler-plus on (glint4 glint4 2)
doing task Bonder on
 (Succ print-value (adjacent) glint3 glint4)
doing task Sparkler-plus on (glint5 glint5 2)
doing task Sparkler-plus on (glint4 glom9 2)
doing task Bonder on
 (Succ print-value (adjacent) glint4 glint5)
doing task Sparkler-plus on (glint4 glom8 10)
doing task Bonder on
 (Succ print-value (remote) glint2 glint5)
doing task Sparkler-plus on (glint5 glom9 2)
doing task Reformulator
doing task Tester
doing task Sparkler
doing task Bonder on
 (Succ print-value (adjacent) glint4 glint5)
doing task Sparkler
doing task Tester
doing task Sparkler-plus on (glint5 glom9 2)
doing task Tester
doing task Bonder on
 (Succ print-value (adjacent) glint4 glint5)
doing task Bond-assessor on (S-group 8.0)
doing task Reformulator
doing task Tester
doing task Sparkler
spark30 --- between glint5glint1
doing task Bond-assessor on (S-group 9.0)
doing task Tester
doing task Reformulator
doing task Bond-assessor on (S-group 10.0)
doing task Gnoth-caster
casts: ((S-group 1 2) (S-group 1 3))
(S-group 1 (Countup 2))
new hypoth candidate (S-group 1 (Countup 2))
doing task Call-term

;  A third hypothesis is formulated.

please enter a term: show-hypothesis
(S-group 1 (Countup 2))

please enter a term: show-parse
((1 2) (1 2 3))

please enter a term: show
terms of the sequence:
1 2 1 2 3

bonds:
bond1 Succ print-value (adjacent) -- (glint1 glint2)
bond2 Pred print-value (adjacent) -- (glint2 glint3)
bond3 Same print-value (remote) -- (glint1 glint3)
bond5 Same print-value (remote) -- (glint2 glint4)
bond6 Succ print-value (remote) -- (glint2 glint5)
bond7 Succ print-value (adjacent) -- (glint3 glint4)
bond8 Succ print-value (adjacent) -- (glint4 glint5)

gloms:
glom9 pseudo --> (1 2)  terms 1 to 2
glom14 pseudo --> (1 2 3)  terms 3 to 5
```

```
gnoths:

external-bonds:
 ((bond2 0) (bond5 -5) (bond6 0) (bond3 -5))
internal-bonds: ((bond1 10))
equivalence-type: parse
groups: nil
class: Gnoths
name: gnoth2
frame: 1
plato-class: S-group
glom: glom9
notes: nil
form: (S-group 1 2)
state: stable
range: (1 2)

external-bonds:
 ((bond6 0) (bond5 -5) (bond2 0) (bond3 -5))
internal-bonds: ((bond8 10) (bond7 10))
groups: nil
class: Gnoths
name: gnoth5
frame: 2
plato-class: S-group
glom: glom14
notes: nil
form: (S-group 1 3)
state: stable
range: (3 5)


;  The next term confirms the hypothesis.

please enter a term: show-seq
1 2 1 2 3

please enter a term: 1
doing task Hfilter
new term being hypothesis-filtered through (S-group 1
(Countup 2))
top-down glom glom15

I have a guess!

hypothesis: (S-group 1 (Countup 2))
(1 2)(1 2 3)(1 2 3 4)


;  This time the guess is correct.

enter no if wrong, ok if right ok
bye
```

GLOSSARY

actualization -- A gnoth that exhibits the properties of some Platonic class is an actualization of that class at the socratoplasm level.

attribute-based description -- a concept representation scheme that views a concept as a unit with only global properties, rather than as a structure (see "structural description").

bond -- a cytoplasm-level structure that defines a relationship (e.g., sameness, successorship) between two gloms (or glints).

box -- the active portion of the structural representation of a Seek-Whence concept, and a repository of information about the value of that structure.

bursting -- an operation that destroys a glom and its subgloms, leaving only underlying glints behind.

catchall gnoth -- a rightmost or "trailer" gnoth that simply holds input terms that agree with the hypothesis without parenthesizing them.

cosmetic reform -- the reformulation of a predictive hypothesis for aesthetic reasons -- to give it a cleaner form -- or to make its structure conform more closely to that dictated by the reigning hypothesis.

cytoplasm -- the lowest level of the Seek-Whence world: home of bonds, glints, and gloms.

dissolving -- an operation that destroys a glom, freeing its top-level subgloms into the cytoplasm.

divesting push -- a unilateral move by a gnoth to rid itself of an internal glom that decreases its stability, whether or not a neighboring gnoth has any attraction for the glom.

dubbing -- the marking of a glom as a manifestation of a particular Platonic class. For example, when the system recognizes that the glom (1 1 1) has the properties of a C-group, it will be "dubbed" as a C-group manifestation.

frame -- an abstractly-viewed hit of a hypothesis: the collection of Seek-Whence forms that would produce the given hit.

freeze-dried hypothesis -- the form of a hypothesis without its active, structural description.

glint -- Seek-Whence's cytoplasm-level representation of an input-sequence term.

glom -- a cytoplasm-level structure representing a plausibly groupable collection of neighboring glints (and/or gloms).

gnoth -- a socratoplasm-level structure representing a logical grouping of terms in the system's parenthesization or parse of a sequence.

gnoth operation -- one of several well-defined actions -- SHIFT-LEFT, SHIFT-RIGHT, SPLIT, CAPTURE, ENCLOSE, FRACTURE, MERGE, NO-OP -- for modifying a gnoth or neighboring gnoths.

gnoth-hypothesis equivalence -- the representation by a gnoth of one frame of a hypothesis. There are three levels of equivalence -- term, parse, and structural (see pp. 94-97).

hit -- a query of a Seek-Whence diagram or of a box for its next value -- a term or grouping of terms.

hypothesis -- a reformulatable structure that models and can extrapolate a sequence pattern, and is constructed from one or more of the eight primitive Platonic concepts.

hypothesis filtering -- a process whereby new input terms are checked for conformity with the reigning hypothesis. Should a new term not conform to the hypothesis, reformulation begins.

ideal types -- the Platonic concepts -- ideal atoms and ideal groups.

manifestation -- A glom that exhibits the properties of some Platonic class is a manifestation of that class at the cytoplasm level.

medical reform -- the reformulation, using the evidence presented by a new term or terms, of a hypothesis because it fails to be predictive.

parenthesization -- an expression of a perceived sequence parse, made by putting gnoths over certain gloms and glom collections. For example, the parenthesization (1 2) (1 2 3) is achieved by putting the first two terms into one gnoth and the last three into another.

parse -- a patterned view of a sequence.

Platonic class (concept) -- an idealized version of an integer, or one of the eight primitives (Constant, Countup, C-group, S-group, P-group, Y-group, Cycle, Tuple) from which Seek-Whence concepts are constructed.

platoplasm -- the highest level of the Seek-Whence world, which houses the Platonic concepts and information about them.

PROGRAM -- a series of gnoth operations proposed by a Reformulator task in order to modify the system's parenthesization of a sequence.

pseudo-glom -- an inert glom, in that it cannot combine with other gloms, generally used as a cap to prevent the disappearance of a given glom cluster (one glom or a collection of neighboring gloms).

reformulation -- the conversion of one concept into another, related, concept in a "reasonable" way; a synonym for slippage.

s-link -- A "slipping link" between two Platonic classes. The slipperiness of such a link indicates the system's proclivity to slip from one class to another.

Seek-Whence diagram -- a set of primitive node types and a structural representation technique used to give a visual sense of our concept representation scheme and of the effects of reformulation.

Slipnet -- a repository of the information about the Platonic concepts and their interrelationships needed for reformulation.

slipperiness -- (see "s-link")

socratoplasm -- the middle level of the Seek-Whence world, which houses the gnoths.

structural description -- a concept representation that portrays a concept as having separately-describable components, rather than as a single entity with only global attributes (see "attribute-based description").

task -- an uninterruptible (and generally small) segment of a computational process. Tasks are capable of creating or modifying structures, setting off other tasks, or querying the user.

template -- a "proto-hypothesis", developed as the first rough statement of an emerging formulation.

terraced scan -- a technique for progressively deepening the exploration of several pathways in parallel, whereby the most plausible pathways are explored more extensively than the less plausible ones.

BIBLIOGRAPHY

1. Anderson, James A., and Hinton, Geoffrey E., Models of information processing in the brain, in: G.E. Hinton and J. Anderson (Eds.) *Parallel Models of Associative Memory* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1981) 9-48.

2. Anderson, John R., *Cognitive Skills and Their Acquisition* (Lawrence Erlbaum, Hillsdale NJ, 1981).

3. Anderson, John R., Acquisition of proof skills in geometry, in: R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983) 191-219.

4. Anzai, Y., and Simon, H., The theory of learning by doing, *Psychological Review* 36(2) (1979) 124-140.

5. Berkeley, E.C. and Bobrow, D. (Eds.), *The Programming Language LISP: Its Operation and Applications* (Information International, Inc. Cambridge MA, 1964).

6. Bierre, Pierre, The professor's challenge, *The AI Magazine* 5(4) (Winter, 1985) 60-70.

7. Bobrow, Daniel G., and Collins, A. (Eds.), *Representation and Understanding* (Academic Press, New York, 1975).

8. Bongard, Mikhail, *Pattern Recognition* (Spartan Books, New York, 1970).

9. Brachman, Ronald J., What's in a concept: structural foundations for semantic networks, *International Journal of Man-Machine Studies* 9 (1977) 127-152.

10. ____________, On the epistemological status of semantic networks , in: N. V. Findler(Ed.), *Associative Networks : Representation and Use of Knowledge by Computers* (Academic Press, New York, 1979) 3-50.

11. ____________ and Schmolze, James, An overview of the KL-ONE knowledge representation system, *Cognitive Science* 9(2) (1985).

12. Buchanan, Bruce G., and Mitchell, T. M., Model-directed learning of production rules, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed Inference Systems* (Academic Press, New York, 1978).

13. Carbonell, J.G., Learning by analogy: formulating and generalizing plans from past experience, in: R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA,1983) 137-161.

14. Clossman, Gray A., A model of the encoding of perceptual features according to the concepts implicit in a set of associations, internal memo, Fluid Analogies Research Group, University of Michigan, Ann Arbor, 1985.

15. Darden, Lindley, Reasoning by analogy in scientific theory construction, in: R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House, University of Illinois at Urbana-Champaign (June, 1983) 32-40.

16. Davis, Philip J., and Hersh, R., *The Mathematical Experience* (Houghton Mifflin, Boston, 1981).

17. DeJong, Gerald, An approach to learning from observation, in: R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House, University of Illinois at Urbana-Champaign (June, 1983) 171-176.

18. Dennett, Daniel C., *Brainstorms* (Bradford Books, Montgomery VT, 1978).

19. Dietterich, Thomas G., M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1979.

20. ______________ and Michalski, R.S., A comparative review of selected methods for learning from examples, in: R.S. Michalski, J.G.Carbonell, and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983) 41 - 81.

21. ______________ and Michalski, R.S., Discovering patterns in sequences of events, *Artificial Intelligence* 25(2) (1985) 187-232.

22. Evans, Thomas G., A program for the solution of a class of geometric-analogy intelligence-test questions, in: M. Minsky (Ed.), *Semantic Information Processing* (MIT Press, Cambridge, MA, 1968) 271-353.

23. Feigenbaum, E. A. and Feldman, J. (Eds.), *Computers and Thought* (McGraw-Hill, New York, 1963).

24. Fredkin, Edward, Techniques using LISP for automatically discovering interesting relations in data, in: E.C. Berkeley and D. Bobrow (Eds.), *The Programming Language LISP: Its Operation and Applications*, (Information International, Inc. Cambridge MA, 1964) 108-124.

25. Gentner, Dedre, Structure-mapping: a theoretical framework for analogy, *Cognitive Science* 7 (1983) 155-170.

26. Gregg, L. W. (Ed.), *Knowledge and Cognition* (Lawrence Erlbaum, New York, 1974).

27. Groner, R., Groner, M., and Bischof, W. (Eds.) *Methods of Heuristics* (Lawrence Erlbaum, Hillsdale NJ, 1983).

28. Hinton, Geoffrey E., and Anderson, J. (Eds.), *Parallel Models of Associative Memory* (Lawrence Erlbaum, Hillsdale NJ, 1981).

29. Hofstadter, Douglas R., *Gödel, Escher, Bach: an Eternal Golden Braid* (Basic Books, New York, 1979).

30. ______________, Clossman, G. A. and Meredith, M. J., Shakespeare's plays weren't written by him, but by someone else of the same name, Technical Report No. 96, Department of Computer Science, Indiana University, Bloomington, July, 1980.

31. Hofstadter, D. R. and Dennett, D.C., *The Mind's I* (Basic Books, New York, 1981).

32. Hofstadter, D. R., Clossman, G. A. and Meredith, M. J., SW: A computer model of perception, abstraction, and induction, internal memo, Department of Computer Science, Indiana University, Bloomington, 1982a.

33. Hofstadter, D. R:, Artificial intelligence: subcognition as computation, Technical Report No. 132, Department of Computer Science, Indiana University, Bloomington, November, 1982b.

34. ____________, On Seeking Whence, unpublished manuscript, 1982c.

35. ____________, The architecture of Jumbo, in: R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House, University of Illinois at Urbana-Champaign (June, 1983) 161-170.

36. ____________, The Copycat project: an experiment in nondeterminism and creative analogies, A.I. Memo 755, Massachusetts Institute of Technology, The Artificial Intelligence Laboratory, January, 1984.

37. ____________, *Metamagical Themas: Questing for the Essence of Mind and Pattern* (Basic Books, New York, 1985a).

38. ____________, Clossman, G., Rogers, D., Mitchell, M., Huber, G. and Leban, R., Research on fluid analogies, *Fluid Analogies Research Group Charter*, Department of Psychology, University of Michigan, Ann Arbor, April, 1985b.

39. Holland, John H., Escaping brittleness, in: R. S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House, University of Illinois at Urbana-Champaign (June, 1983) 92-95.

40. Kotovsky, Kenneth, and Simon, H. A., Empirical tests of a theory of human acquisition of concepts for sequential patterns, *Cognitive Psychology* 4 (1973) 399-424.

41. Kuhn, T., *The Structure of Scientific Revolutions*, (University of Chicago Press, 1962).

42. Langley, Pat, Bradshaw, G. L., and Simon, H., Rediscovering chemistry with the BACON system, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983) 307-329.

43. ____________, Learning to search: from weak methods to domain-specific heuristics, *Cognitive Science* 9 (1985) 217-260.

44. Lenat, Douglas B., The nature of heuristics, *Artificial Intelligence* 19(2) (1982) 189-249.

45. ____________, Theory formation by heuristic search: the nature of heuristics II: background and examples, *Artificial Intelligence* 21(1,2) (1983a) 31-59.

46. ____________, EURISKO: a program that learns new heuristics and domain concepts: the nature of heuristics III: program design and results, *Artificial Intelligence* 21(1,2)(1983b) 61-98.

47. Lenat, Douglas B., The role of heuristics in learning by discovery: three case studies, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA ,1983c) 243-306.

48. ______________ and Brown, John Seely, Why AM and EURISKO appear to work, *Artificial Intelligence* 23(3) (1984), 269-294.

49. Meredith, Marsha J., Reynolds, Paul and Wehking, Audrey, Pattern perception experiment, presentation to the Illinois State Academy of Sciences, Computer Science Section, April, 1983.

50. Meredith, Marsha J., The code for Seek-Whence, Technical Report, Department of Computer Science, Indiana University (forthcoming), 1986.

51. Michalski, Ryszard S., Carbonell, J.G. and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983a).

52. Michalski, Ryszard S. (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House,University of Illinis at Urbana-Champaign (June, 1983b).

53. ______________, A theory and methodology of inductive learning, *Artificial Intelligence* 20(2) (1983c) 111-161.

54. ______________ and Stepp, R.E., Learning from observation: conceptual clustering, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983d) 331-363.

55. Minsky, Marvin (Ed.), *Semantic Information Processing* (MIT Press, Cambridge, MA, 1968).

56. ______________, A framework for representing knowledge, in: P. H. Winston (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975) 211-277.

57. ______________, Jokes and the logic of the cognitive unconscious, in: R. Groner, M. Groner and W. F. Bischoff (Eds.), *Methods of Heuristics* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1983) 171-193.

58. ______________, The society of mind, lecture presented at Southern Illinois University at Edwardsville, April, 1986.

59. Mitchell, Thomas M., Utgoff, Paul and Banerji, Ranan, Learning by experimentation: acquiring and refining problem-solving heuristics, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA,1983) 163-190.

60. Moore, J. and Newell, A., How can Merlin understand?, in: L. Gregg (Ed.), *Knowledge and Cognition* (Lawrence Erlbaum, Potomac, MA, 1973).

61. Norman, Donald A. (Ed.), *Perspectives on Cognitive Science* (Ablex, Norwood NJ, 1981).

62. Pearl, Judea (Ed.), *Search and Heuristics* (North-Holland, Amsterdam, 1983).

63. Persson, Staffan, Some sequence extrapolating programs: a study of representation and modeling in inquiring systems, Report No. STAN-CS-66-050, Department of Computer Science, Stanford University, Stanford, CA, 1966.

64. Pivar, M. and Finkelstein, M., Automation, using LISP, of inductive inference on sequences, in: E.C. Berkeley and D.Bobrow (Eds.), *The Programming Language LISP: Its Operation and Applications*, (Information International, Inc. Cambridge MA,1964) 125-136.

65. Reddy, Raj, Erman, L., Hayes-Roth, F., Lesser, V. and Shockey, L., Working papers in speech recognition -- IV -- the HEARSAY II system, Carnegie-Mellon University Computer Science Department Technical Report, February, 1976.

66. Rogers, David, personal memorandum, 1986.

67. Schank, Roger C., and Colby, K. M. (Eds.), *Computer Models of Thought and Language* (Freeman, San Francisco, 1973).

68. Schank, Roger C., Language and memory, *Cognitive Science* 4(3) (1980) 243-284.

69. ______________, *Dynamic Memory: A theory of reminding and learning in computers and people* (Cambridge University Press, Cambridge, 1982).

70. ______________, *The Cognitive Computer* (Addison-Wesley, Reading MA, 1984).

71. Searle, John, Minds, brains, and programs, *The Behavioral and Brain Sciences* 3 (September, 1980) 417-457.

72. Simon, Herbert A. and Kotovsky, K., Human acquisition of concepts for sequential patterns, *Psychological Review* 70(6) (1963) 534-546.

73. Simon, Herbert A., Complexity and the representation of patterned sequences of symbols, *Psychological Review* 79(5) (1972) 369-382.

74. ______________, *Models of Discovery* (Reidel, Dordrecht, 1977).

75. Stepp, R. E., and Michalski, R. S., Conceptual Clustering of Structured Objects: A Goal-Oriented Approach, *Artificial Intelligence*, 28(1) (1986), 43-69.

76. Ulam, Stanislaw, *Adventures of a Mathematician* (Charles Scribner's, New York, 1976).

77. Waterman, D. A., and Hayes-Roth, F. (Eds.), *Pattern-directed Inference Systems* (Academic Press, New York, 1978).

78. Wickelgren, Wayne A., *How to Solve Problems* (W. H. Freeman and Company, San Francisco, 1974).

79. Winston, Patrick H., Learning structural descriptions from examples, in: P. H. Winston (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975) 157-209.

80. ______________, Learning and reasoning by analogy, *Communications of the Association for Computing Machinery*, 23(12) (December,1980).

81. Winston, P. H., Learning by augmenting rules and accumulating censors, in: R. S.Michalski (Ed.), *Proceedings of the International Machine Learning Workshop*, Allerton House, University of Illinois at Urbana-Champaign (June, 1983) 2-11.