AD-A155 378

Stanford Artificial Intelligence Laboratory
Memo AIM-286

July 1976

Computer Science Department
Report No. STAN-CS-76-570

AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search

by

Douglas B. Lenat

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION IS UNLIMITED (A)

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

DTIC FILE COPY

DTIC
ELECTE
JUN 17 1985
S D
G

85 06 13 140

Stanford Artificial Intelligence Laboratory
Memo AIM-286

July 1976

Computer Science Department
Report No. STAN-CS-76-570

AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search

by

Douglas B. Lenat

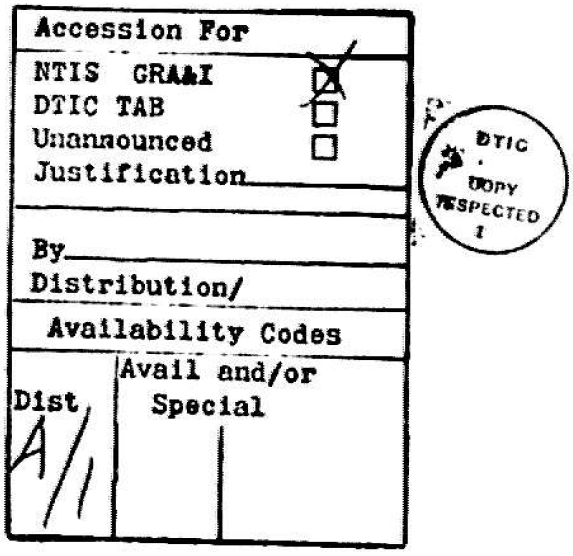
ABSTRACT

A program, called "AM", is described which models one aspect of elementary mathematics research: developing new concepts under the guidance of a large body of heuristic rules. "Mathematics" is considered as a type of intelligent behavior, not as a finished product.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-76-C-0206 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22161.



Acknowledgments

I owe a great debt of thanks to many people, both for the input of new ideas and for the evaluation, channelling, and pruning of my own.

Let me begin by alphabetically thanking my committee: Bruce Buchanan, Ed Feigenbaum, Cordell Green, Don Knuth, and Allen Newell. Interacting with each of them has been an exciting experience, and my thesis has greatly benefited from their guidance.

The following individuals have each informally supplied some ideas or comments that appear within this thesis. They all have earned my gratitude, and have significantly improved the experence you are about to have, that of reading this thesis: Danny Bobrow, Don Cohen, Paul Coehn, Avra Cohn, Randy Davis, Bob Floyd, Carl Hewitt, Earl Sacerdoti, Richard Weyrauch, and Terry Winograd. Let me also thank SAIL, SRI, and SUMEX for providing a sophisticated computing environment in which to work.

Around this point in the *Acknowledgements*, most theses have some sort of tribute to the candidate's wife. Until I was in the throes of this research, I never fully appreciated the importance of such support. So let me sincerely acknowledge the indispensable aid I received from Merle, my wonderful wife, who put up with inverted schedules and who gave me the confidence to tackle this problem and the enthusiasm to keep going.

Table of Contents

1. Overview
   - 1.1. Abstract of this Thesis 1
   - 1.2. Five-page Summary of the Project 2
     - Detour: Analysis of a discovery
     - What AM does: Syntheses of discoveries
     - Results
     - Motivation [optional]
     - Conclusions
   - 1.3. Ways of viewing AM as some common process 6
     - AM as Hill-climbing
     - AM as Heuristic Search
     - AM as a Mathematician
     - AM as a Thesis [optional]

2. An Example: Discovering Prime Numbers
   - 2.1. Discussion of the AM Program 14
     - Representation
     - Agenda and Heuristics
   - 2.2. What to get out of – and NOT get out of – this example 17
   - 2.3. Deciphering the Example 18
   - 2.4. The Example Itself 20
   - 2.5. Recapping the Example 27

3. Control Structure
   - 3.1. AM's Search 28
   - 3.2. Constraining AM's Search 30
   - 3.3. The Agenda 32
     - Why an Agenda?
     - Details of the Agenda scheme

4. Heuristic Rules
   - 4.1. Syntax of the Heuristics 35
     - Syntax of the Left-hand Side
     - Syntax of the Right-hand Side
   - 4.2. Heuristics Suggest New Tasks 38
     - An Illustration: "Fill in Generalizations of Equality"
     - The Ratings Game
   - 4.3. Heuristics Create New Concepts 42
     - An Illustration: Discovering Primes
     - The Theory of Creating New Concepts
     - Another Illustration: Squaring a number

4.4. Heuristics Fill in Entries for a Specific Facet 47
An Illustration: "Fill in Examples of Set-union"
Heuristics Propose New Conjectures
An Illustration: "All primes except 2 are odd"
Another illustration: Discovering Unique Factorization
4.5. Gathering Relevant Heuristics 54
Domain of Applicability
Rippling
Ordering the Relevant Heuristics
4.6. AM's Starting Heuristics 58
Heuristics Grouped by the Knowledge They Embody
Heuristics Grouped by How Specific They Are

5. AM's Concepts
5.1. Motivation and Overview 61
A Glimpse of a Typical Concept
The main constraint: Fixed set of facets
BEINGs Representation of Knowledge
5.2. Facets 66
Generalizations/Specializations
Examples/Isa's
In-Domain-of/In-Range-of
Views
Intuitions
Analogies
Conjec's
Definitions
Algorithms
Domain/Range
Worth
Interest
Suggest
Fillin/Check
Other Facets which were Considered
5.3. AM's Starting Concepts 105
Diagram of Initial Concepts
Summary of Initial Concepts
Rationale behind Choice of Concepts

6. Results
6.1. What AM Did 114
Linear Task-by-task Summary of a Good Run
Two-Dimensional Behavior Graph
AM as a Computer Program

6.2. Experiments with AM 125
Must the Worth numbers be finely tuned?
How finely tuned is the Agenda?
How valuable is tacking reasons onto each task?
What if certain concepts are eliminated/added?
What if certain heuristics are tampered with?
Can AM work in a new domain: Plane Geometry?

7. Evaluating AM
7.1. Judging Performance 135
AM's Ultimate Discoveries
The Magnitude of AM's Progress
The Quality of AM's Route
The Character of the User-System Interactions
AM's Intuitive Powers
Experiments on AM
How to Perform Experiments on AM
Future Implications of this Project
Open Problems: Suggestions for Future Research
Comparison to Other Systems
7.2. Capabilities and Limitations of AM 153
Current Abilities
Current Limitations
Limitations of the Agenda scheme
Limiting Assumptions
Choice of Domain
Limitations of the Model of Math Research
Ultimate powers and weaknesses
7.3. Final Conclusions 169

Appendix 1. Glossary of Technical Terms 165
Glossary of Math Terms
Glossary of AI Terms

Appendix 2. AM's Concepts 172
Initial Concepts
Index of Initial Concepts
Anything, Any-concept, Active, Predicate, Object-equality, Constant-predicate, Constant-True, Constant-False, Operation, Compose, Insert, Set-insert, Oset-insert, List-insert, Bag-insert, Delete, Set-Delete, Bag-Delete, List-Delete, Oset-Delete, Intersect, List-Intersect, Oset-Intersect, Set-Intersect, Bag-Intersect, Union, List-Union, Oset-Union, Set-Union, Bag-Union, Difference, List-Diff, Oset-Diff, Set-Diff, Bag-Diff, Coalesce, Canonize, Parallel-replace2, Parallel-replace, Repeat2, Repeat, Parallel-join2, Parallel-join, Reverse-ord-pair, Last-element, First-element, All-but-the-first-element, All-but-the-last-element, Member, Projection1, Projection2, Identity, Restrict, Invert-an-operation, Inverted-op, Relation, Logical-combination, Object, Conjecture, Atom-obj, Truth-value, Structure, Structure-of-Structures, Ord-Structure, Unord-Structure, Multiple-elements-structure, No-multiple-elements-structure, Empty-structure, Nonempty-structure, Sets, Bags, Lists, Ordered-pairs, Osets,
Concepts never fully implemented

Concepts and Heuristics as coded in LISP
The 'Compose' Concept
The 'Oneto' Concept
Concepts created by AM

Appendix 3. AM's Heuristics 226
Heuristics for dealing with Anything
Heuristics for dealing with Any-concept
Heuristics for any facet of Any-concept
Heuristics for the Examples facets of Any-concept
Heuristics for the Conjecs facet of Any-concept
Heuristics for the Analogies facet of Any-concept
Heuristics for the Genl/Spec facets of Any-concept
Heuristics for the View facet of Any-concept
Heuristics for the In-dom/ran-of facets of Any-concept
Heuristics for the Definition facet of Any-concept
Heuristics for dealing with any Active concept
Heuristics for dealing with any Predicate
Heuristics for dealing with any Operation
Heuristics for dealing with any Composition
Heuristics for dealing with any Insertions
Heuristics for dealing with the operation Coalesce
Heuristics for dealing with the operation Canonize
Heuristics for dealing with the operation Substitute
Heuristics for dealing with the operation Restrict
Heuristics for dealing with the operation Invert
Heuristics for dealing with Logical combinations
Heuristics for dealing with Structures
Heuristics for dealing with Ordered-structures
Heuristics for dealing with Unordered-structures
Heuristics for dealing with Multiple-eles-structures
Heuristics for dealing with Sets

Appendix 4. Maximally-Divisible Numbers 277
A Meaningful Question
Special Case: $n = 2^a 3^b$
Special Case: $n = 2^a 3^b 5^c$
The General Case
An even stronger claim
AM and Ramanujan

Appendix 5. Traces of AM in Action 287
Prose Traces
A 'Nice' Task-by-task Trace
An 'Unadulterated' Trace

Appendix 6. Bibliography 337
Documentation

Chapter 1. Overview

Indeed, you can build a machine to draw demonstrative conclusions for you, but I think you can never build a machine that will draw plausible inferences.

-- Polya

1.1. Abstract of this Thesis

A program, called "AM", is described which models one aspect of elementary mathematics research: developing new concepts under the guidance of a large body of heuristic rules. "Mathematics" is considered as a type of intelligent behavior, not as a finished product.

The local heuristics communicate via an agenda mechanism, a global list of tasks for the system to perform and reasons why each task is plausible. A single task might direct AM to define a new concept, or to explore some facet of an existing concept, or to examine some empirical data for regularities, etc. Repeatedly, the program selects from the agenda the task having the best supporting reasons, and then executes it.

Each concept is an active, structured knowledge module. A hundred very incomplete modules are initially provided, each one corresponding to an elementary set-theoretic concept (e.g., union). This provides a definite but immense "space" which AM begins to explore. AM extends its knowledge base, ultimately rediscovering hundreds of common concepts (e.g., numbers) and theorems (e.g., unique factorization).

This approach to plausible inference contains great powers and great limitations.

1.2. Five-page Summary of the Project

Scientists often face the difficult task of formulating nontrivial research problems which are solvable. In any given branch of science, it is usually easier to tackle a specific given problem than to propose interesting yet managable new questions to investigate. For example, contrast *solving* the Missionaries and Cannibals problem with the more ill-defined reasoning which led to *inventing* it.

This thesis is concerned with creative theory formation in mathematics: how to propose interesting new concepts and plausible hypotheses connecting them. The experimental vehicle of my research is a computer program called AM[1] Initially, AM is given the definitions of 115 simple set-theoretic concepts (like "Delete", "Equality"). Each concept is represented internally as a data structure with a couple dozen slots or facets (like "Definition", "Examples", "Worth"). Initially, most facets of most concepts are blank, and AM uses a collection of 250 heuristics – plausible rules of thumb – for guidance, as it tries to fill in those blanks. Some heuristics are used to select which specific facet of which specific concept to explore next, while others are used to actually find some appropriate information about the chosen facet. Other rules prompt AM to notice simple relationships between known concepts, to define promising new concepts to investigate, and to estimate how interesting each concept is.

1.2.1. Detour: Analysis of a discovery

Before discussing how to *synthesize* a new theory, consider briefly how to *analyze* one, how to construct a plausible chain of reasoning which terminates in a given discovery. One can do this by working backwards, by reducing the creative act to simpler and simpler creative acts. For example, consider the concept of prime numbers. How might one be led to define such a notion? Notice the following plausible strategy:

> **"If f is a function which transforms elements of A into elements of B, and B is ordered, then consider just those members of A which are transformed into *extremal* elements of B. This set is an interesting subset of A."**

When f(x) means "divisors of x", and the ordering is "by length", this heuristic says to consider those numbers which have a minimal[2] number of factors – that is, the primes. So this rule actually *reduces* our task from "proposing the concept of prime numbers" to the more elementary problems of "discovering ordering-by-length" and "inventing divisors-of".

But suppose we know this general rule: **"If f is an interesting function, consider its inverse."** It

---

[1] The original meaning of this mnemonic has been abandoned. As Exodus states: I AM that I AM.

[2] The other extreme, numbers with a MAXIMAL number of factors, was also proposed by AM as worth investigating. This led AM to many interesting questions. See Appendix 4.

reduces the task of discovering divisors-of to the simpler task of discovering multiplication[3]. Eventually, this task reduces to the discovery of very basic notions, like substitution, set-union, and equality. To explain how a given researcher might have made a given discovery, such an analysis is continued until that inductive task is reduced to "discovering" notions which the researcher already knew, which were his conceptual primitives.

1.2.2. What AM does: Syntheses of discoveries

> *This leads to the paradox that the more original a discovery the more obvious it seems afterwards. The creative act is not an act of creation in the sense of the Old Testament. It does not create something out of nothing; it uncovers, selects, re-shuffles, combines, synthesizes already existing facts, faculties, skills. The more familiar the parts, the more striking the new whole.*
>
> -- Koestler

Suppose a large collection of these heuristic strategies has been assembled (e.g., by analyzing a great many discoveries, and writing down new heuristic rules whenever necessary). Instead of using them to *explain* how a given idea might have evolved, one can imagine starting from a basic core of knowledge and "running" the heuristics to *generate* new concepts. We're talking about reversing the process described in the last section: not how to *explain* discoveries, but how to *make* them.

Such syntheses are precisely what AM does. The program consists of a large corpus of primitive mathematical concepts, each with a few associated heuristics[4]. AM's activities all serve to expand AM itself, to enlarge upon a given body of mathematical knowledge. To cope with the enormity of the potential "search space" involved, AM uses its heuristics as judgmental criteria to guide development in the most promising direction. It appears that the process of inventing worthwhile new[5] concepts can be guided successfully using a collection of a few hundred such heuristics.

Each concept is represented as a frame-like data structure with 25 different facets or slots. The types of facets include: **Examples, Definitions, Generalizations, Domain/Range, Analogies, Interestingness,** and many others. Modular representation of concepts provides a convenient scheme for organizing the heuristics; for example, the following strategy fits into the *Examples* facet of the *Predicate* concept: **"If, empirically, 10 times as many elements *fail* some predicate P, as *satisfy* it, then some *generalization* (weakened version) of P might be more interesting than P".** AM considers this suggestion after trying to fill in examples of each

---

[3] Plus noticing that multiplication is associative and commutative.

[4] Situation/action rules which function as local "plausible move generators". Some suggest tasks for the system to carry out, some suggest ways of satisfying a given task, etc.

[5] Typically, "new" means new to AM, not to Mankind; and "worthwhile" can only be judged in hindsight.

predicate[6].

AM is initially given a collection of 115 core concepts, with only a few facets filled in for each. Its sole activity is to choose some facet of some concept, and fill in that particular slot. In so doing, new notions will often emerge. Uninteresting ones are forgotten, mildly interesting ones are kept as parts of one facet of one concept, and very interesting ones are granted full concept-module status. Each of these new modules has dozens of blank slots, hence the space of possible actions (blank facets to fill in) grows rapidly. The same heuristics are used both to suggest new directions for investigation, and to limit attention: both to sprout and to prune.

1.2.3. Results

The particular mathematical domains in which AM operates depend upon the choice of initial concepts. Currently, AM begins with nothing but a scanty knowledge of concepts which Piaget might describe as *prenumerical*: Sets, substitution, operations, equality, and so on. In particular, AM is not told anything about proof, single-valued functions, or numbers.

From this primitive basis, AM quickly discovered[7] elementary numerical concepts (corresponding to those we refer to as natural numbers, multiplication, factors, and primes) and wandered around in the domain of elementary number theory. AM was not designed to *prove* anything, but it did *conjecture* many well-known relationships (e.g., the unique factorization theorem).

AM was not able to discover any "new-to-Mankind" mathematics purely on its own, but *has* discovered several interesting notions hitherto unknown to the author. A couple bits of new mathematics have been *inspired* by AM.[2] A synergetic AM–human combination can sometimes produce better research than either could alone.[8] Although most of the concepts AM proposed and developed were already very well known, AM defined some of them in novel ways (e.g., prime pairs were defined by restricting addition to primes; that is, for which primes p,q,r is it possible that p+q=r?[9]).

Everything that AM does can be viewed as testing the underlying body of heuristic rules. Gradually, this knowledge becomes better organized, its implications clearer. The resultant body of detailed heuristics may be the germ of a more efficient programme for educating

---

[6] In fact, after AM attempts to find examples of SET-EQUALITY, so few are found that AM decides to generalize that predicate. The result is the creation of a new predicate which means "Has-the-same-length-as" -- i.e., a rudimentary precursor to natural numbers.

[7] "Discovering" a concept means that (1) AM recognized it as a distinguished entity (e.g., by formulating its definition) and also (2) AM decided it was worth investigating (either because of the interesting way it was formed, or because of surprising preliminary empirical results).

[8] This is supported by Gelernter's experiences with his geometry program: While lecturing about how it might prove a certain theorem about isosceles triangles, he came up with a new, cute proof. Similarly, Guard and Eastman noticed an intermediate result of their SAM resolution theorem prover, and wisely interpreted it as a nontrivial result in lattice theory (now known as SAM's lemma).

[9] The answer is that either p or q must be 2, and that the other two primes are a prime pair -- i.e., they differ by two.

math students than the current dogma[10].

Another benefit of actually constructing AM is that of *experimentation*: one can vary the concepts AM starts with, vary the heuristics available, etc., and study the effects on AM's behavior. Several such experiments were performed. One involved adding a couple dozen new concepts from an entirely new domain: plane geometry. AM busied itself exploring elementary geometric concepts, and was almost as productive there as in its original domain. New concepts were defined, and new conjectures formulated. Other experiments indicated that AM was more robust than anticipated; it withstood many kinds of "de-tuning". Others demonstrated the tremendous impact that a few key concepts (e.g., Equality) had on AM's behavior. Several more experiments and extensions have been planned for the future.

1.2.4. Motivation [optional]

> *We need a super-mathematics in which the operations are as unknown as the quantities they operate on, and a super-mathematician, who does not know what he is doing when he performs these operations.*
>
> -- Eddington

Although the motivation for carrying out this research of course preceded the effort, I have delayed until this section a discussion of why this is worthwhile, why it was attempted.

First there was the inherent interest of getting a handle on scientific creativity. AM is partly a demonstration that some aspects of creative theory formation can be demystified, can be modelled as simple rule-governed behavior.

Related to this is the potential for learning from AM more about the processes of concept formation. This was touched on previously, and several experiments already performed on AM will be detailed later.

Third, AM itself may grow into something of pragmatic value. Perhaps it will become a useful tool for mathematicians, for educators, or as a model for similar systems in more "practical" fields. Perhaps in the future we scientists will be able to rely on automated assistants to carry out the "hack" phases of research, the tiresome legwork necessary for "secondary" creativity.

Historically, the domain of AM came from a search for a scientific field whose activities had no specific goal, and in which natural language abilities were unnecessary. This was to test out the BEINGs [Lenat 75b] ideas for a modular representation of knowledge.

---

[10] Currently, an educator takes the very best work any mathematician has ever done, polishes it until its brilliance is blinding, then presents it to the student to induce upon. Many individuals (e.g., Knuth and Polya) have pointed out this blunder. A few (e.g., Papert at MIT, Adams at Stanford) are experimenting with more realistic strategies for "teaching" creativity. See the references by these authors in the bibliography.

It would be unfair not to mention the usual bad reasons for this research: the "Look ma, no hands" syndrome, the AI researcher's classic maternal urges, ego, the usual thesis drives, etc.

1.2.5. Conclusions

AM is forced to judge *a priori* the value of each new concept, to lose interest quickly in concepts which aren't going to develop into anything. Often, such judgments can only be based on hindsight. For similar reasons, AM has difficulty formulating new heuristics which are relevant to the new concepts it creates. Heuristics are often merely compiled hindsight. While AM's "approach" to empirical research may be used in other scientific domains, the main limitation (reliance on hindsight) will probably recur. This prevents AM from progressing indefinitely far on its own.

This ultimate limitation was reached. AM's performace degraded more and more as it progressed further away from its initial base of concepts. Nevertheless, AM demonstrated that selected aspects of creative discovery in elementary mathematics could be adequately represented as a heuristic search process. Actually constructing a computer model of this activity has provided an experimental vehicle for studying the dynamics of plausible empirical inference.

1.3. Ways of viewing AM as some common process

This section will provide a few metaphors: some hints for squeezing AM into paradigms with which the reader might be familiar. For example, the existence of heuristics in AM is functionally the same as the presence of domain-specific information in any knowledge-based system.

Consider assumptions, axioms, definitions, and theorems to be syntactic rules for the language that we call Mathematics. Thus theorem-proving, and the whole of textbook mathematics, is a purely syntactic process. Then the heuristic rules used by a mathematician (and by AM) would correspond to the semantic knowledge associated with these more formal methods.

Just as one can upgrade natural-language-understanding by incorporating semantic knowledge, so AM is only as successful as the heuristics it knows.

Four more ways of "viewing" AM as something else will be provided: (i) AM as a hill-climber, (ii) AM as a heuristic search program, (iii) AM as a mathematician, and (iv) AM as a thesis.

1.3.1. AM as Hill-climbing

Let's draw an analogy between the process of developing new mathematics and the familiar process of hill-climbing. We may visualize AM as exploring a space using a measuring or "evaluation" function which imparts to it a topography.

Consider AM's core of very simple knowledge. By compounding its known concepts and methods, AM can explore beyond the frontier of this foundation a little wherever it wishes. The incredible variety of alternatives to investigate includes all known mathematics, much trivia, countless deadends, and so on. The only "successful" paths near the core are the narrow ridges of known mathematics (plus perhaps a few as-yet-undiscovered isolated peaks).

How can AM walk through this immense space, with any hope of following the few, slender trails of already-established mathematics (or some equally successful new fields)? AM must do hill-climbing: As new concepts are formed, decide how promising they are, and always explore the currently most-promising new concept. The evaluation function is quite nontrivial, and this thesis may be viewed as an attempt to study and explain and duplicate the judgmental criteria people employ. Preliminary attempts[11] at codifying such "mysterious" emotive forces as intuition, aesthetics, utility, richness, interestingness, relevance... indicated that a large but not unmanageable collection of heuristic rules should suffice.

The important visualization to make is that with proper evaluation criteria, AM's planar mass of interrelated concepts is transformed into a three-dimensional relief map: the known lines of development become mountain ranges, soaring above the vast flat plains of trivia and inconsistency below.

Occasionally an isolated hill is discovered near the core;[12] certainly whole ranges lie undiscovered for long periods of time[13], and the terrrain far from the initial core is not yet explored at all.

1.3.2. AM as Heuristic Search

As the title of this section -- and this thesis -- proclaims, AM is a kind of "heuristic search" program. That must mean that AM is exploring a particular "space," using some informal evaluation criteria to guide it.

The flavor of search which is used here is that of progressively enlarging a tree. Certain "evaluation-function" heuristics are used to decide which node of the tree to expand next, and other guiding rules are then used to produce from that node a few interesting successor nodes. To do mathematical research well, I claim that it is necessary and sufficent to have good methods for proposing new concepts from existing ones, and for deciding how interesting each "node" (partially-studied concept) is.

AM is initially supplied with a few facts about some simple math concepts. AM then

---

[11] These took the form of informal simulations. Although far from controlled experiments, they indicated the feasability of attempting to create AM, by yielding an approximate figure for the amount of informal knowledge such a system would need.

[12] E.g., Conway's numbers, as described in [Knuth 74].

[13] E.g., non-Euclidean geometries weren't thought of until 1848.

explores mathematics by selectively enlarging that basis. One could say that AM consists of an active body of mathematical concepts, plus enough "wisdom" to use and develop them effectively. For "wisdom", read "heuristics". Loosely speaking, then, AM is a heuristic search program. To see this more clearly, we must explain what the nodes of AM's search space are, what the successor operators or links are, and what the evaluation function is.

AM's space can be considered to consist of all nodes which are consistent, partially-filled-in concepts. Then a primitive "legal move" for AM would be to (i) enlarge some facet of some concept, or (ii) create a new, partially-complete concept. Consider momentarily the size of this space. If there were no constraint on what the new concepts can be, and no informal knowledge for quickly finding entries for a desired facet, a blind "legal-move" program would go nowhere – slowly! One shouldn't even call the activity such a program would be doing "math research."

The heuristic rules are used as little "plausible move generators". They suggest which facet of which concept to enlarge next, and they suggest specific new concepts to create. The only activities which AM will consider doing are those which have been motivated for some specific good[14] reason. A global *agenda of tasks* is maintained, listing all the activities suggested but not yet worked on.

AM has a definite algorithm for rating the nodes of its space. Many heuristics exist merely to estimate the worth of any given concept. Other heuristics use these worth ratings to order the tasks on the global agenda list. Yet AM has no specific goal criteria: it can never "halt", never succeed or fail in any absolute sense. AM goes on forever[15].

Consider Nilsson's descriptions of depth-first searching and breadth-first searching ([Nilsson 71]). He has us maintain a list of "open" nodes. Repeatedly, he plucks the top one and expands it. In the process, some new nodes may be added to the Open list. In the case of depth-first searching, they are added at the top; the next node to expand is the one most recently created; the Open-list is being used as a push-down stack. For breadth-first search, new nodes are added at the bottom; they aren't expanded until all the older nodes have been; the Open-list is used as a queue. For heuristic search, or "best-first" search, new nodes are evaluated in some numeric way, and then "merged" into the already-sorted list of Open nodes.

This process is very similar to the agenda mechanism AM uses to manage its search. This will be discussed in detail in Chapter 3. Each entry on the agenda consists of three parts: (i) a plausible task for AM to do, (ii) a list of reasons supporting that task, and (iii) a numeric estimate of the overall priority this task should have. When a task is suggested for some reason, it is added to the agenda. A task may be suggested several times, for different reasons. The global priority value assigned to each task is based on the combined value of its reasons. The control structure of AM is simply to select the task with the highest priority, execute it, and select a new one. The agenda mechanism appears to be a very well-suited data structure for managing a "best-first" search process.

---

[14] Of course, AM thinks a reason is "good" if -- and only if -- it was told that by a heuristic rule; so those rules had better be plausible, preferably the ones actually used by the experts.

[15] Technically, forever is about 100,000 list cells and a couple cpu hours.

Similar control structures were used in LT [Newell, Shaw, & Simon 57], the predictor part of Dendral [Buchanan et al 69], SIMULA-67 [Dahl 68], and KRL [Bobrow & Winograd 77]. The main difference is that in AM, symbolic reasons are used (albeit in trivial token-like ways) to decide whether – and how much – to boost the priority of a task when it is suggested again.

There are several difficulties and anomalies in forcing AM into the heuristic search paradigm. In a typical heuristic search (e.g., Dendral [Feigenbaum et al 71], Meta-Dendral [Buchanan et al 72], most game-playing programs [Samuel 67]), a "search space" is defined implicitly by a "legal move generator". Heuristics are present to constrain that generator so that only plausible nodes are produced. The second kind of heuristic search, of which AM is an example, contains no "legal move generator". Instead, AM's heuristics are used as plausible move generators. Those heuristics themselves implicitly define the possible tasks AM might consider, and *all* such tasks should be plausible one. In the first kind of search, removing a heuristic widens the search space; in AM's kind of search, removing a heuristic *reduces* it.

Another anomaly is that the operators which AM uses to enlarge and explore the space of concepts are themselves mathematical concepts (e.g., some heuristic rules result in the creation of new heuristic rules; "Compose" is both a concept and an operation which results in new concepts). Thus AM should be viewed as a mass of knowledge which enlarges *itself* repeatedly. Typically, computer programs keep the information they "discover" quite separate from the knowledge they use to make discoveries[16]

Perhaps the greatest difference between AM and typical heuristic search procedures is that AM has no well-defined target concepts or target relationships. Rather, its "goal criterion" – its sole aim – is to maximize the interestingness level of the activities it performs, the priority ratings of the top tasks on the agenda. It doesn't matter precisely which definitions or conjectures AM discovers – or misses – so long as it spends its time on plausible tasks. There is no fixed set of theorems that AM should discover, so AM is not a typical *problem-solver*. There is no fixed set of traps AM should avoid, no small set of legal moves, and no winning/losing behavior, so AM is not a typical *game-player*.

For example, no stigma is attached to the fact that AM never discovered real numbers[17]; it was rather surprising that AM managed to discover *natural* numbers! Even if it hadn't done that, it would have been acceptable[18] if AM had simply gone off and developed ideas in set theory.

---

[16] Of course this is often because the two kinds of knowledge are very different: For a chess-player, the first kind is "good board positions," and the second is "strategies for making a good move." Theorem-provers are an exception. They produce a new theorem, and then use it (almost like a new operator) in future proofs. A program to learn to play checkers [Samuel 67] has this same flavor, thereby indicating that this 'self-help' property is not a function of the task domain, not simply a characteristic of mathematics.

[17] There are many "nice" things which AM didn't -- and can't -- do: e.g., devising geometric concepts from its initial simple set-theoretic knowledge. See the discussion of the limitations of AM, Section 7.2.

[18] Acceptable to whom? Is there really a domain-invariant criterion for judging the quality of AM's actions? See the discussions in Section 7.1.

1.3.3. AM as a Mathematician

Before diving into the innards of AM, let's take a moment to discuss the totality of the mathematics which AM carried out. Like a contemporary historian summarizing the work of the Babylonian mathematicians, we shan't hesitate to use current terms and criticize by current standards.

AM began its investigations with scanty knowledge of a few set-theoretic concepts (sets, equality of sets, set operations). Most of the obvious set-theory relations (e.g., de Morgan's laws) were eventually uncovered; since AM never fully understood abstract algebra, the statement and verification of each of these was quite obscure. AM never derived a formal notion of infinity, but it naively established conjectures like "a set can never be a member of itself", and procedures for making chains of new sets ("insert a set into itself"). No sophisticated set theory (e.g., diagonalization) was ever done.

After this initial period of exploration, AM decided that "equality" was worth generalizing, and thereby discovered the relation "same-size-as". "Natural numbers" were based on this, and soon most simple arithmetic operations were defined.

Since addition arose as an analog to union, and multiplication as a repeated substitution followed by a generalized kind of unioning[19] it came as quite a surprise when AM noticed that they were related (namely, $N*N=2*N$). AM later re-discovered multiplication in three other ways: as repeated addition, as the numeric analog of the Cartesian product of sets, and by studying the cardinality of power sets[20]. These operations were defined in different ways, so it was an unexpected (to AM) discovery when they all turned out to be equivalent. These surprises caused AM to give the concept 'Times' quite a high Worth rating.

Exponentiation was defined as repeated multiplication. Unfortunately, AM never found any obvious properties of exponentiation, hence lost all interest in it.

Soon after defining multiplication, AM investigated the process of multiplying a number by itself: squaring. The inverse of this turned out to be interesting, and led to the definition of square-root. AM remained content to play around with the concept of *integer*-square-root. Although it isolated the set of numbers which had no square root, AM was never close to discovering rationals, let alone irrationals.

Raising to fourth-powers, and fourth-rooting, were discovered at this time. Perfect squares and perfect fourth-powers were isolated. Many other numeric operations and kinds of numbers were isolated: Odds, Evens, Doubling, Halving, etc. Primitive notions of numeric inequality were defined but AM never even discovered Trichotomy.

The associativity and commutativity of multiplication indicated that it could accept a BAG

---

[19] Take two bags A and B. Replace each element of A by the bag B. Remove one level of parentheses by taking the union of all elements of the transfigured bag A. Then that new bag will have as many elements as the product of the lengths of the two original bags.

[20] The size of the set of all subsets of S is 2^S. Thus the power set of AUB has length equal to the product of the lengths of the power sets of A and B individually (assuming A and B are disjoint).

of numbers as its argument. When AM defined the inverse operation corresponding to Times, this property allowed the definition to be: "any *bag* of numbers (>1) whose product is x". This was just the notion of factoring a number x. Minimally-factorable numbers turned out to be what we call primes. Maximally-factorable numbers were also thought to be interesting.

Prime pairs were discovered in a bizarre way: by restricting addition (its arguments and its values) to Primes.[21] AM conjectured the fundamental theorem of arithmetic (unique factorization into primes) and Goldbach's conjecture (every even number >2 is the sum of two primes) in a surprisingly symmetric way. The unary representation of numbers gave way to a representation as a bag of primes (based on unique factorization), but AM never thought of exponential notation. [22] Since the key concepts of remainder, greater-than, gcd, and exponentiation were never mastered, progress in number theory was arrested.

When a new base of *geometric* concepts was added, AM began finding some more general associations. In place of the strict definitions for the equality of lines, angles, and triangles, came new definitions of concepts we refer to as Parallel, Equal-measure, Similar, Congruent, Translation, Rotation, plus many which have no common name (e.g. the relationship of two triangles sharing a common angle). A cute geometric interpretation of Goldbach's conjecture was found[23]. Lacking a geometry "model" (an analogic representation like the one Gelernter employed), AM was doomed to failure with respect to proposing only plausible geometric conjectures.

Similar restrictions due to poor "visualization" abilities would crop up in topology. The concepts of continuity, infinity, and measure would have to be fed to AM before it could enter the domains of analysis. More and more drastic changes in its initial base would be required, as the desired domain gets further and further from simple finite set theory and elementary number theory.

---

[21] That is, consider the set of triples p,q,r, all primes, for which p+q=r. Then one of them must be "2", and the other two must therefore form a prime pair.

[22] A tangential note: All of the discoveries mentioned above were made by AM working by itself, with a human being observing its behavior. If the level of sophistication of AM's concepts were higher (or the level of sophistication of its users were lower), then it might be worthwhile to develop a nice user--system interface. The user in that case could -- and ought to -- work right along with AM as a co-researcher.

[23] Given all angles of a prime number of degrees, (0,1,2,3,5,7,11,..,179 degrees), then any angle between 0 and 180 degrees can be approximated (to within 1 degree) as the sum of two of those angles.

1.3.4. AM as a Thesis [optional]

> *Walking home along a deserted street late at night, the reader may imagine himself to feel in the small of his back a cold, hard object; and to hear the words spoken behind him, 'Easy now. This is a stick-up. Hand over your money.' What does the reader do? He attempts to generate the utterance. He says to himself, now if I were standing behind someone holding a cold, hard object against his back, what would make me say that? What would I mean by it? The reader is advised that he can only arrive at the deep structure of this book, and through the deep structure the semantics, if he attempts to generate the book for himself. The author wishes him luck.*
>
> -- Linderholm

Don't be scared by the weight of the document you're now holding. If you flip to page 165, you'll see that the last two-thirds are just appendices.

Each chapter is of roughly equal importance, which explains the huge variation in length. Start looking over Chapter 2 right away: it contains a detailed example of what AM does. Since you're reading this sentence now, we'll assume that you want a preview of what's to come in the rest of this document.

Chapter 3 covers the top-level control structure of the system, which is based around the notion of an 'agenda' of tasks to perform. In Chapter 4 the low-level control structure is revealed: AM is really guided by a mass of heuristic rules of varying generality. Chapter 5 contains more than you want to know about the representation of knowledge in AM. The diagram showing some of AM's starting concepts (page 105) is worth a look, even out of context.

Most of the results of the project are presented in Chapter 6. In addition to simply 'running' AM, several experiments have been conducted with it. It's awkward to evaluate AM, and therefore Chapter 7 is quite long and detailed.

The appendices provide material which supplements the text. Appendix 2 contains a description of all the initial concepts, some examples of how they were coded into Lisp, and a partial list of the concepts AM defined and investigated along the way. Appendix 3 exhibits all 242 heuristics that AM is explicitly provided with. Appendix 4 is essentially a math article, about the major discovery that AM motivated: maximally-divisible numbers. Finally, Appendix 5 contains traces of AM in action: a long prose description, a long task-by-task description, and a long undoctored transcript excerpt. Appendix 1 hasn't been mentioned yet, and forms the subject of the remainder of this section.

This thesis – and its readers – must come to grips with a very interdisciplinary problem. For the reader whose background is in Artificial Intelligence, most of the system's actions – the "mathematics" it does – may seem inherently uninteresting. For the mathematician, the word "LISP" signifies nothing beyond a speech impediment (to Artificial Intelligence types it

also connotes a programming impediment). If I don't describe "LISP" the first time I mention it, a large fraction of potential readers will never realize that potential. If I *do* stop to describe LISP, the other readers will be bored.

In an attempt not to lose readers due to jargon, two glossaries of terms have been compiled. Appendix 1.1 (p. 165) contains capsule descriptions of the mathematical terms, ideas, and notations used in this thesis. Appendix 1.2 renders the analogous service for Artificial Intelligence jargon and computer science concepts.

Chapter 2. An Example: Discovering Prime Numbers

This chapter will present an example of AM in action, an excerpt from the output of AM, as it investigates some concepts.

After a brief discussion of AM's control structure in Section 2.1, the reader will be told what the point of this example is – and is *not*. Section 2.3 provides a few eleventh-hour hints at decoding the example.

The excerpt itself follows in Section 2.4. It skips the first half of the session, and picks up at a point just after AM has defined the concept "Divisors-of". Soon afterward, AM defines Primes, and begins to find interesting conjectures related to them. The excerpt goes on to show how AM conjectured the fundamental theorem of arithmetic and Goldbach's conjecture. AM derived the notion of partitioning a collection of n objects into smaller bundles, but failed to find any interesting conjectures about that process. Instead, AM was side-tracked into the (probably) fruitless investigation of numbers which can be represented as the sum of two primes in one unique way.

The final section of this chapter will recap this example the way a math historian might report it.

2.1. Discussion of the AM Program

2.1.1. Representation

AM is a program which expands a knowledge base of mathematical concepts. Each concept is stored as a particular kind of data structure, namely as a collection of properties or "facets" of the concept. For example, here is a miniature example of a concept[1]:

[1] The right arrow ("→") in the box on the next page is the symbol for "implies". "Nos." is an abbreviation for "Numbers". The vertical bar "|" is a symbol for the predicate "divides evenly into"; the hook "¬" is a symbol for the predicate "the negation of". "⊕" indicates exclusive or, and the symbol "∀" is read "for all". Please consult the glossary, Appendix 1.1, for fuller discussion of these, plus other math terms like "Prime pairs".

NAME: Prime Numbers

DEFINITIONS:
ORIGIN: Number-of-divisors-of(x) = 2
PREDICATE-CALCULUS: Prime(x) = (∀z)(z|x → z=1 ∨ z=x)
ITERATIVE: (for x>1): For i from 2 to Sqrt(x), ~(i|x)

EXAMPLES: 2, 3, 5, 7, 11, 13, 17
BOUNDARY: 2, 3
BOUNDARY-FAILURES: 0, 1
FAILURES: 12

GENERALIZATIONS: Nos., Nos. with an even no. of divisors, Nos. with a prime no. of divisors

SPECIALIZATIONS: Odd Primes, Prime Pairs, Prime Uniquely-addables

CONJECS: Unique factorization, Goldbach's conjecture, Extremes of Number-of-divisors-of

INTU'S: ***A metaphor to the effect that Primes are the building blocks of all numbers***

ANALOGIES:
Maximally-divisible numbers are converse extremes of Number-of-divisors-of
Factor a non-simple group into simple groups

INTEREST: Conjectures tying Primes to TIMES, to Divisors-of, to closely related operations

WORTH: 800

"Creating a new concept" is a well-defined activity: it involves setting up a new data structure like the one above, and filling in entries for some of its facets or slots. Filling in a particular facet of a particular concept is also quite well-defined, and is accomplished by executing a collection of relevant heuristic rules. This process will be described in great detail in later chapters.

2.1.2. Agenda and Heuristics

An agenda of plausible tasks is maintained by AM. A typical task is "Fill-in examples of Primes". The agenda may contain hundreds of entries such as this one. AM repeatedly selects the top task from the agenda and tries to carry it out. This is the whole control structure! Of course, we must still explain how AM creates plausible new tasks to place on the agenda, how AM decides which task will be the best one to execute next, and how it carries out a task.

If the task is "Fill in new Algorithms for Set-union", then *satisfying* it would mean actually synthesizing some new procedures, some new LISP code capable of forming the union of any two sets. A heuristic rule is *relevant* to a task iff executing that rule brings AM closer to satisfying that task. Relevance is determined a priori by where the rule is stored. A rule tacked onto the Domain/range facet of the Compose concept would be presumed relevant to the task "Check the Domain/range of InsertDelete".

Once a task is chosen from the agenda, AM gathers some heuristic rules which might be relevant to satisfying that task. They are executed, and then AM picks a new task. While a rule is executing, three kinds of actions or effects can occur:

(i) Facets of some concepts can get filled in (e.g., examples of primes may actually be found and tacked onto the "Examples" facet of the "Primes" concept). A typical heuristic rule which might have this effect is:

> **To fill in examples of X, where X is a kind of Y (for some more general concept Y),**
> **Check the examples of Y; some of them may be examples of X as well.**

For the task of filling in examples of Primes, this rule would have AM notice that Primes is a kind of Number, and therefore look over all the known examples of Number. Some of those would be primes, and would be transferred to the Examples facet of Primes.

(ii) New concepts may be created (e.g., the concept "primes which are uniquely representable as the sum of two other primes" may be somehow be deemed worth studying). A typical heuristic rule which might result in this new concept is:

> **If some (but not most) examples of X are also examples of Y (for some concept Y),**
> **Create a new concept defined as the intersection of those 2 concepts (X and Y).**

Suppose AM has already isolated the concept of being representable as the sum of two primes in only one way (AM actually calls such numbers "Uniquely-prime-addable numbers"). When AM notices that some primes are in this set, the above rule will create a brand new concept, defined as the set of numbers which are both prime and uniquely prime addable.

(iii) New tasks may be added to the agenda (e.g., the current activity may suggest that the following task is worth considering: "Generalize the concept of prime numbers"). A typical heuristic rule which might have this effect is:

> **If very few examples of X are found,**
> **Then add the following task to the agenda: "Generalize the concept X".**

Of course, AM contains a precise meaning for the phrase "very few". When AM looks for primes among examples of already-known kinds of numbers, it will find dozens of non-examples for every example of a prime it uncovers. "Very few" is thus naturally

implemented as a statistical confidence level[2].

The concept of an agenda is certainly not new: schedulers have been around for a long time. But one important feature of AM's agenda scheme *is* a new idea: attaching -- and using -- a list of quasi-symbolic[3] reasons to each task which explain why the task is worth considering, why it's plausible. *It is the responsibility of the heuristic rules to include reasons for any tasks they propose.*[4] For example, let's reconsider the heuristic rule mentioned in (iii) above. It really looks more like the following:

> **If very few examples of X are found,**
> **Then add the following task to the agenda: "Generalize the concept X", for the following reason: "X's are quite rare; a slightly less restrictive concept might be more interesting".**

If the same task is proposed by several rules, then several different reasons for it may be present. In addition, one ephemeral reason also exists: "Focus of attention". Any tasks which are similar to the one last executed get "Focus of attention" as a bonus reason. AM uses all these reasons, e.g. to decide how to rank the tasks on the agenda. The "intelligence" AM exhibits is not so much "what it does", but rather the *order* in which it arranges its agenda[5]. AM uses the list of reasons in another way: Once a task has been selected, the quality of the reasons is used to decide how much time and space the task will be permitted to absorb, before AM quits and moves on to a new task. This whole mechanism will be detailed in Section 3.3.2, on Page 33.

2.2. What to get out of -- and NOT get out of -- this example

The purpose of the example which begins on page 20 is to convey a bit of AM's flavor. After reading through it, the reader should be convinced that AM is *not* a theorem-prover, nor is it *randomly* manipulating entries in a knowledge base, nor is it *exhaustively* manipulating or searching. AM is carefully growing a network of data structures representing mathematical concepts, by repeatedly using heuristics both (a) for guidance in choosing a task to work on next, and (b) to provide methods to satisfy the chosen task.

---

[2] The ratio of examples found to non-examples stumbled over lies between .001 and .05. Philosophers outraged by this may be somewhat appeased by knowledge that large changes in the precise numbers very rarely alter AM's behavior.

[3] Each reason is an English sentence. While AM can tell whether two given reasons coincide, it can't actually do any internal processing on them. If this lack of intelligence had proved to be a limiting problem, then more work would have been expended on giving AM some such abilities.

[4] An alternative scheme, perhaps even a bit more human-like, would be to (perhaps only occasionally) allow a burst of poorly-motivated tasks to be proposed, and then use some pruning criteria to weed out the obvious losers. During this time, AM could type out to the user (who otherwise would be closely monitoring its activities) a cute anthropomorphic phrase like "I'm now sitting back and puffing on my pipe, lost in contemplation."

[5] For example, alternating a randomly-chosen task and the "best" task (the one AM chose to do) only slows the system down by a factor of 2, yet it totally destroys its credibility as a rational researcher (as judged by the human user of AM). This is one conclusion of experiment 2 (see Section 8.2.2, page 129).

The following points are important but can't be conveyed by any lone example:

(i) Although AM appears to have reasonable natural language abilities, this is a typical AI illusion: most of the phrases AM types are mere tokens, and the syntax which the user must obey is unnaturally constrained. For the sake of clarity, I have "touched up" some of the wording, indentation, syntax, etc. of what AM actually outputs, but left the spirit of each phrase intact. As the reader becomes more familiar with AM, future examples can be "unretouched". If he wishes, he may glance at Appendix 5.3, which shows some actual listings of AM in action.

(ii) The reader should be skeptical of the generality of the program; is the knowledge base "just right" (i.e., finely tuned to elicit this one chain of behaviors)? The answer is "*No*"[6]. The whole point of this project is to show that a relatively small set of general heuristics can guide a nontrivial discovery process. Each activity, each task, was proposed by some heuristic rule (like "look for extreme cases of X") which was used time and time again, in many situations. It was not considered fair to insert heuristic guidance which could only "guide" in a single situation.

This kind of generality can't be shown convincingly in one example. Nevertheless, even within this small excerpt, the same line of development which leads to decomposing numbers (using $TIMES^{-1}$) and thereby discovering unique factorization, also leads to decomposing numbers (using ADD^{-1}) and thereby discovering Goldbach's conjecture. The same heuristic which caused AM to expect that unique factorization will be useful, also caused AM to suspect that Goldbach's conjecture will be useless.

Let me reemphasize that the "point" of this example is *not* the specific mathematical concepts, nor the particular chains of plausible reasoning AM produces, nor the few flashy conjectures AM spouts, but rather an illustration of the *kinds* of things AM does.

2.3. Deciphering the Example

Recall that in general, each task on the agenda will have several reasons attached to it. In the example excerpt, the reasons for each task are printed just after the task is chosen, and before it's executed.

AM numbers its activities sequentially. Each time a new task is chosen, a counter is incremented. The first task in the example excerpt is labelled ** TASK 65 **, meaning that the example skips the first 64 tasks which AM selects and carries out. The reason simply is that the development of simple concepts related to divisibility will probably be more intelligible and palatable to the reader, than AM's early ramblings in finite set theory.

---

[6] The design of AM was finely tuned so that the answer to this question would be "No". Ponder that one!

In the example itself, several irrelevant tasks have been excised[7]. About half of those omitted tasks were interesting in themselves, but all of them were tangential or unrelated to the development shown. The reader can tell by the global task numbering how many were skipped. For example, notice that the excerpt jumps from Task 67 to Task 79.

To help gauge AM's abilities, the reader may be interested to know that AM defined "Natural Numbers" during Task 44, and "TIMES" was defined during Task 57. AM started with no knowledge of numbers, and only scanty knowledge of sets and set-operations. Task 3, e.g., was to fill in examples of Sets.

The concepts that AM talks about are self-explanatory – by and large. Below are discussed some nonstandard ones.

BAG is a kind of list structure, a bunch of elements which are unordered, but one in which multiple copies of the same element are permitted. One may visualize a paper bag filled with cardboard letters. Technically, we shall say that a set is *not* considered to be a bag. A bag is denoted by enclosure within parentheses, just as sets are within braces. So the bag containing X and four Y's might be written (X Y Y Y Y), and would be considered indistinguishable from the bag (Y Y Y X Y).

Number will mean (typically) a positive integer.

$TIMES^{-1}$ is a particular relation. For any number x, $TIMES^{-1}(x)$ is a set of bags. Each bag contains some numbers which, when multiplied together, equal x. For example, $TIMES^{-1}(18)$ = { (18) (2 9) (2 3 3) (3 6) }. Checking, we see that multiplying, e.g., the numbers in the bag (2 3 3) together, we do get 2x2x3=18. $TIMES^{-1}(x)$ contains all possible such bags (containing natural numbers >1).

ADD^{-1} is a relation analogous to $TIMES^{-1}$. For any number x, $ADD^{-1}(x)$ is also a set of bags. Each bag contains a bunch of numbers which, when added together, equal x. For example, $ADD^{-1}(4)$ = { (4) (1 1 1 1) (1 1 2) (1 3) (2 2) }. $ADD^{-1}(x)$ contains all possible such bags (containing numbers >0); it finds all possible *partitions* of x.

Divisors-of is a more standard relation. For any number x, Divisors-of(x) is the set of all positive numbers which divide evenly into x. For example, Divisors-of(18) = {1 2 3 6 9 18}.

The definitions for most of the mathematical terms used in the excerpt can be found in the Glossary (Appendix 1.1). Whenever there is a conflict between "computer science jargon" and "math jargon", I have opted for the latter. So, e.g., all "functions" are necessarily single-valued for each member of their domain.

AM is an *interactive* computer program. It prints out phrases indicating what it is doing at each moment, and a human being, referred to as the *User*, watches AM's activities. At any moment, the user can interrupt AM to ask questions or re-direct its energies. In the excerpt, the user never re-directs AM, although he does occasionally ask AM to rename some concept and to explain its behavior (e.g., just after ** TASK 106 **). For the benefit of the reader, I have bold-faced all the user's comments.

---

[7] This is fair, despite the results of Experiment 2 (see Section 6.2.2) because the remaining tasks clump together in twos, threes, etc; they are uninterrupted lines of research (e.g., Tasks 65-67), separated by very large gaps (e.g., the jump from Task 67 to 79).

My personal notes and comments occur right-justified, in italics.

2.4. The Example Itself

** Task 65: **
Filling in examples of the following concept: "Divisors-of".

3 Reasons: (1) No known examples for Divisors-of so far.
(2) TIMES, which is related to Divisors-of, is now very interesting.
(3) Focus of attention: AM recently defined Divisors-of.

26 examples found, in 9.2 seconds. e.g.: Divisors-of(6)={1 2 3 6}.

** Task 66: **
Considering numbers which have very small sets of Divisors-of (i.e., very few divisors).

2 Reasons: (1) Worthwhile to look for extreme cases.
(2) Focus of attention: AM recently worked on Divisors-of.

Filling in examples of numbers with 0 divisors.
0 examples found, in 4.0 seconds.
Conjecture: no numbers have 0 divisors.

Filling in examples of numbers with 1 divisors.
1 examples found, in 4.0 seconds. e.g.: Divisors-of(1)={1}.
Conjecture: 1 is the only number with 1 divisor.

Filling in examples of numbers with 2 divisors.
24 examples found, in 4.0 seconds. e.g.: Divisors-of(13)={1 13}.
No obvious conjecture. This kind of number merits more study.
Creating a new concept: "Numbers-with-2-divisors".

AM had previously derived the notion of singleton, doubleton, etc. Above, AM was actually defining, e.g., the set of numbers which gets mapped into a doubleton of numbers, by the operation Divisors-of. I took the liberty of translating this into the above form, which uses, e.g., the number "2".

Filling in examples of numbers with 3 divisors.
11 examples found, in 4.0 seconds. e.g.: Divisors-of(49)={1 7 49}.
All numbers with 3 divisors are also Squares. This kind of number merits more study.
Creating a new concept: "Numbers-with-3-divisors".

How did AM notice that conjecture? It took a random example of Numbers-with-3-divisiors, say 49. Then it asked what other known concepts "49" was an example of. The two answers were: Odd-numbers and Perfect-squares. AM then tested these conjectures on the other ten examples just found. The only surviving conjecture was that all numbers-with-3-divisors are also perfect-squares.

** Task 67: **
Considering the square-roots of Numbers-with-3-divisors.

2 Reasons: (1) Numbers-with-3-divisors are unexpectedly also perfect Squares.
(2) Focus of attention: AM recently worked on Numbers-with-3-divisors.

All square-roots of Numbers-with-3-divisors seem to be Numbers-with-2-divisors.
e.g., Divisors-of(Square-root(169)) = Divisors-of(13) = {1 13}.

Formulating the converse to this statement. Empirically, it seems to be true.
The square of each Number-with-2-divisors seems to be a Number-with-3-divisors.
This is very unusual. It is not plausibly a coincidence. (Chance of coincidence is < .001)

Boosting interestingness factor of the concepts involved:
Interestingness factor of "Divisors-of" raised from 300 to 400.
Interestingness factor of "Numbers-with-2-divisors" raised from 100 to 600.
Interestingness factor of "Numbers-with-3-divisors" raised from 200 to 700.

USER: Call the set of numbers with 2 divisors "Primes".

** Task 68: **
Considering the squares of Numbers-with-3-divisors.

2 Reasons: (1) Squares of Numbers-with-2-divisors were interesting.
(2) Focus of attention: AM recently worked on Numbers-with-3-divisors.

•
•
•

This gap in the sequencing — from task 67 to task 79 — eliminates some tangential and boring tasks. See page 19 for an explanation.

•
•
•

** Task 79: **
Examining $TIMES^{-1}(x)$, looking for patterns involving its values.

2 Reasons: (1) $TIMES^{-1}$ is related to the newly-interesting concept "Divisors-of".
(2) Many examples of $TIMES^{-1}$ are known, to induce from.

Looking specifically at $TIMES^{-1}(12)$, which is { (12) (2 6) (2 2 3) (3 4) }.
13 conjectures proposed, after 2.0 seconds.
e.g., "$TIMES^{-1}(x)$ always contains a bag containing only even numbers".
Testing the conjectures on other examples of $TIMES^{-1}$.

5 false conjectures deal with even numbers.
AM will sometime consider the restriction of $TIMES^{-1}$ to even numbers.

Only 2 out of the 13 conjectures are verified for all 26 known examples of $TIMES^{-1}$:

Conjecture 1: $TIMES^{-1}(x)$ always contains a singleton bag.
e.g., $TIMES^{-1}(12)$, which is { (12) (2 6) (2 2 3) (3 4) }, contains (12).
e.g., $TIMES^{-1}(13)$, which is { (13) }, contains (13).

Creating a new concept, "Single-times".
Single-times is a relation from Numbers to Bags-of-numbers.
Single-times(x) is all bags in $TIMES^{-1}(x)$ which are singletons.
e.g., Single-times(12)={ (12) }.
e.g., Single-times(13)={ (13) }.

Conjecture 2: $TIMES^{-1}(x)$ always contains a bag containing only primes.
e.g., $TIMES^{-1}(12)$, which is { (12) (2 6) (2 2 3) (3 4) }, contains (2 2 3).
e.g., $TIMES^{-1}(13)$, which is { (13) }, contains (13).

Creating a new concept, "Prime-times".
Prime-times is a relation from Numbers to Bags-of-numbers.
Prime-times(x) is all bags in $TIMES^{-1}(x)$ which contain only primes.
e.g., Prime-times(12)={ (2 3 3) }.
e.g., Prime-times(13)={ (13) }.

** Task 80: **
Considering the concept "Prime-times".

2 Reasons: (1) Conjecs about Prime-times will tell much about Primes and TIMES^{-1}.
(2) Focus of attention: AM recently defined Prime-times.

Looking specifically at Prime-times(48), which is { (2 2 2 2 3) }.
4 conjectures proposed, after .6 seconds.
e.g., "x is never inside any member of Prime-times(x)".
Testing them on other examples of Prime-times.

Only 1 out of the 4 conjectures are verified for all 23 known examples of Prime-times:

Conjecture 1: Prime-times(x) is always a singleton set.
That is, Prime-times is a function, not just a relation.
e.g., Prime-times(48), which is { (2 2 2 2 3) }, is a singleton set.
e.g., Prime-times(47), which is { (47) }, is a singleton set.
This holds for all 17 known examples of Prime-times. (Chance of coincidence is .0001)
This fails for 2 of the boundary cases (extreme numbers): 0 and 1.
Conjecture is amended: Each number >1 is the product of a unique bag of primes.
I suspect that this conjecture may be very useful.[8]

USER: Call this conjecture "Unique factorization conjecture".

•
•
•

To show that AM isn't really always right on the mark, the next sequence of tasks includes a crime of omission (ignoring the concept of Partitions) and a false start (worrying about numbers which can be represented as the sum of two primes in precisely one way). Notice the skip here; 3 tasks have been omitted.

[8] How did AM know this? One of the (unfortunately few!) meta-heuristics in AM said the following: "When using the 'look at the inverse of extreme items under the operation f' rule, Tack the following note onto the interest facet of the new concept which is created: 'Conjectures involving this concept and f (or f^{-1}) are natural, interesting, and probably useful.' " Now the concept PRIMES was defined using the 'extrema' heuristic rule, with f=Divisors-of. When PRIMES was first created, the meta-rule we just presented tacked the following note onto Primes.interest: *'Conjectures involving Primes and division (or multiplication) are natural, interesting, and probably useful.'* Thus the unique factorization conjecture triggers this feature, whereas Goldbach's conjecture wouldn't.

** Task §4: **
Examining $ADD^{-1}(x)$, looking for patterns involving its values.

2 Reasons: (1) ADD^{-1} is analogous to the newly-interesting concept "$TIMES^{-1}$".
(2) Many examples of ADD^{-1} are known, to induce from.

Looking specifically at $ADD^{-1}(6)$, which is { (1 1 1 1 1 1) (1 1 1 1 2) (1 1 1 3) (1 1 2 2) (1 1 4) (1 2 3) (1 5) (2 2 2) (2 4) (3 3) (6)}.
17 conjectures proposed, after 3.9 seconds.
e.g., "$ADD^{-1}(x)$ always contains a bag of primes".
Testing them on other examples of ADD^{-1}.
Only 11 out of the 17 conjectures are verified for all 19 known examples of ADD^{-1}:
3 out of the 11 conjectures were false until amended.

Conjecture 1: $ADD^{-1}(x)$ never contains a singleton bag.

Conjecture 2: $ADD^{-1}(x)$ always contains a bag of size 2 (also called a "pair" or a "doubleton").
e.g., $ADD^{-1}(6)$ contains (1 5), (2 4), and (3 3).
e.g., $ADD^{-1}(4)$ contains (1 3), and (2 2).

Creating a new concept, "Pair-add".
Pair-add is a relation from Numbers to Pairs-of-numbers.
Pair-add(x) is all bags in $ADD^{-1}(x)$ which are doubletons (i.e., of size 2).
e.g., Pair-add(12)={ (1 11) (2 10) (3 9) (4 8) (5 7) (6 6) }.
e.g., Pair-add(4)={ (1 3) (2 2) }.

Conjecture 3: $ADD^{-1}(x)$ always contains a bag containing only 1's.
•
•
•

Conjecture 10: $ADD^{-1}(x)$ always contains a pair of primes.
This conjecture is false. Conjecture is amended:
"$ADD^{-1}(x)$ usually (but not always) contains a pair of primes."
e.g., $ADD^{-1}(10)$ contains (3 7), and (5 5).
e.g., $ADD^{-1}(4)$ contains (2 2).
e.g., $ADD^{-1}(11)$ does not contain a pair of primes.

Creating a new concept, "Prime-add".
Prime-add is a relation from Numbers to Pairs-of-numbers.
Prime-add(x) is all bags in $ADD^{-1}(x)$ which are pairs of primes.
e.g., Prime-add(12)={ (5 7) }.
e.g., Prime-add(10)={ (3 7) (5 5) }.
e.g., Prime-add(11) = { }
•
•
•

** Task 106: **
Considering the set of numbers for which Prime-add is defined (has non-empty value).

1 Reason: (1) Prime-add often has non-empty value. Worth isolating that case.

Warning: no task on the agenda has an interestingness value above 200!!!

Creating a new concept "Prime-addable".
Prime-addable is a kind of Number. x is Prime-addable if Prime-add(x) is non-empty.
Will spend 5.0 seconds filling in examples of Prime-addable.
18 examples found. Here are some of them: 4 5 6 7 8 9 10 12 13 14 17 16 18.
Empirically, all even numbers are also in this set (ignoring boundary cases: 0 2 4 6).
So conjectured. Danger: must examine boundary cases: the numbers 0, 2, 4, and 6.
Two exceptions noticed. The only exceptions are the smallest boundary cases: 0, 2.
Conjecture is amended: All even numbers >2 are the sum of two primes.

Warning: I expect this conjecture will be cute but useless.

USER: Why?

Because the concept of "Primes" originally arose as extreme cases of Divisors-of. The more closely an operation X is related to the concept Divisors-of, the more natural will be any conjecture involving both that operation X and Primes. E.g., conjectures involving both Primes and Times will be natural and useful. But this conjecture, which involves Primes and ADD^{-1}, will be cute but useless, since the relation "ADD^{-1}" is unrelated to the relation "Divisors-of".
End of explanation.

USER: Call this conjecture "Goldbach's conjecture".

** Task 107: **
Considering the set of numbers for which the relation Prime-add is single-valued.

3 Reasons: (1) Prime-add often has singleton value. Worth isolating that case.
(2) Restricted to this set, Prime-add would be a function.
(3) Focus of attention: AM recently worked on Prime-add.

Creating a new concept "Uniquely-prime-addable".
"Uniquely-prime-addable" is a kind of Number.
x is Uniquely-prime-addable if Prime-add(x) is a singleton.
Will spend 10.0 seconds filling in examples of Uniquely-prime-addable.
11 examples found. Here are some of them: 4 5 7 8 9 12 13.
No obvious conjecture derived empirically.

Will forget "Uniquely-prime-addable numbers", if no Ties found in near future.

2.5. Recapping the Example

Let's once again eavesdrop on a mathematician, as he describes to a colleague what AM did.

This example was preceded by the momentous discoveries of multiplication and division. Several interesting properites of these operations were noticed. The first task which was illustrated (** Task 65 **) involves exploring the concept of "divisors of a number" (meaning all positive integers which divide evenly into the given number). After tiring of finding examples of this relation, AM investigates extreme cases: that is, it wonders which numbers have very few or very many divisors.

AM thus discovers Primes in a curious way. Numbers with 0 or 1 divisor are essentially nonexistent, so they're not found to be interesting. AM notices that numbers with 3 divisors always seem to be squares of numbers with 2 divisors (primes). This raises the interestingness of several concepts, including primes. Soon (** TASK 79 **), another conjecture involving primes is noticed: Many numbers seem to factor into primes. This causes a new relation to be defined, which associates to a number x, all prime factorizations of x. The first question AM asks about this relation is "is it a function?". This question is the full statement of the unique factorization conjecture: the fundamental theorem of arithmetic. AM recognized the value of this relationship, and assigned it a high interestingness rating.

In a similar manner, though with lower hopes, it noticed some more relationships involving primes, including Goldbach's conjecture. AM quite correctly predicted that this would turn out to be cute but of no future use mathematically.

The last activity mentioned (** TASK 107 **) shows AM examining a rather nonstandard concept: "numbers which can be written as the sum of a pair of primes, in only one way". These are termed "uniquely-prime-addable" numbers. It was mildly unfortunate that AM gave up on this concept before noticing that p+2 is uniquely-prime-addable, for any prime number p, and that in fact these are the only odd uniquely-prime-addable numbers. The session was repeated once, with a human user telling AM explicitly to continue studying this concept. AM did in fact construct "Uniquely-prime-addable-odd-numbers", and then notice this relationship. Here we see an example of unstable equilibrium: if pushed slightly this way, AM will get very interested and spend a lot of time working on this kind of number. Since it doesn't have all the sophistication (i.e., compiled hindsight) that we have, it can't know instantly whether what it's doing will be fruitless.

Chapter 3. Control Structure

'Objectively' given, 'important' problems may arise [in math]. But even then the mathematician is essentially free to take it or leave it and turn to something else, while an 'important' problem in [any other science] is usually a conflict, a contradiction, which 'must' be resolved. The mathematician has a wide choice of which way to turn, and he enjoys a very considerable freedom in what he does.

-- von Neumann

AM is one of those awkward programs whose representations only make sense if you already understand how they will be operated on. A discussion of AM's control structure (this chapter and the next) must thus precede a discussion of concepts and how they are represented (Chapter 5). Section 2.1 gave the reader a sufficient knowledge of AM's "anatomy" to follow these chapters. Thus armed with a cursory knowledge of the "statics" of AM, we shall proceed to describe in detail its "dynamics".

Section 3.1 will give the reader a feeling for the immensity of AM's search space. This is the "problem". The next section will give the top-level "solution": the flow of control is governed by a job-list, an agenda of plausible tasks. Section 3.3 will present some details of this global control scheme.

Chapter 4 deals with the way AM's heuristics operate; this could be viewed as the "low-level" or *local* control structure of AM. Chapter 5 contains some detailed information about the actual concepts (and heuristics) AM starts with, and a little more about their design and representation. The reader is also directed to Appendix 5, which presents several detailed examples of AM "in action".

3.1. AM's Search

To develop mathematics, one must always labor to substitute ideas for calculations.

-- Dirichlet

Let's first spend a paragraph reviewing how concepts are stored. AM contains a collection

of data structures, called *concepts*. Each concept is meant to coincide intuitively with one mathematical idea (e.g., Sets, Union, Trichotomy). As such, a concept has several aspects or parts, called *facets* (e.g., Examples, Definitions, Domain/range, Worth). If you wish to think of a concept as a "frame", then its facets are "slots" to be filled in. Each facet of a concept will either be totally blank, or else will contain a bunch of *entries*. For example, the Algorithms facet of the concept Union may point to several equivalent LISP functions, each of which can be used to form the union of two sets[1]. Even the "heuristic rules" are merely entries on the appropriate kind of facet (e.g., the entries on the Interest facet of the Structure concept are rules for judging the interestingness of Structures[2]).

At any moment, AM contains a couple hundred concepts, each of which has only *some* of its facets filled in. AM starts with 115 concepts, and grows to about 300 concepts before running out of time/space. Most facets of most concepts are totally blank. AM's basic activity is to select some facet of some concept, and then try to fill in some entries for that slot[3]. Thus the primitive kind of "*task*" for AM is to deal with a particular facet/concept pair. A typical task looks like this:

Check the entries on the "Domain/range" facet of the "Bag-Insert" concept

If the average concept has ten or twenty blank facets, and there are a couple hundred concepts, then clearly there will be about 20x200=4000 "fill-in" type tasks for AM to work on, at any given moment. If several hundred facets have recently been filled in, there will be that many "check-entries" type tasks available. Executing a task happens to take around ten or twenty cpu seconds, so over the course of a few hours only a small percentage of these tasks can ever be executed.[4]

Since most of these tasks will never be explored, what will make AM appear smart – or stupid – are its choices of which task to pick at each moment.[5] So it's worth AM's spending a nontrivial amount of time deciding which task to execute next. On the other hand, it had better not be *too* much time, since a task does take only a dozen seconds.[6]

One question that must be answered is: What percentage of AM's legal moves (at any

---

[1] The reasons for having multiple algorithms is that sometimes AM will want one that is fast, sometimes AM will be more concerned with economizing on storage, sometimes AM will want to "analyze" an algorithm, and for that purpose it must be a very un-optimized function, etc.

[2] A typical such rule is: "A structure is very interesting if all its elements are mildly interesting in precisely the same way."

[3] This is not quite complete. In addition to filling in entries for a given facet/concept pair, AM may wish to check it, split it up, reorganize it, etc.

[4] The precise "18 seconds average" figure is not important. All heuristic-search programs suffer this same handicap: As the depth to which they've searched increases, the percentage of nodes (at or above that level) which have been examined decreases exponentially (assuming the branching factor b is strictly larger than unity).

[5] This is true of all heuristic search programs. The branchier the search, the more it applies.

[6] The answer is that AM spends this "deciding" time not just before a task is *picked*, but rather each time a task is added to the agenda. A little under 1 cpu second is spent, on the average, to place the task properly on the agenda, to assign it a meaningful numeric priority value. So "action time" is roughly one order of magnitude larger than "deciding time".

typical moment) would be considered intelligent choices, and what percentage would be irrational? The answer comes from empirical results. The percentages vary wildly depending on the previous few tasks. Sometimes, AM will be obviously "in the middle" of a sequence of tasks, and only one or two of the legal tasks would seem plausible. Other times, AM has just completed an investigation by running into dead-ends, and there may be hundreds of tasks it could choose and not be criticized. The median case would perhaps permit about 6 of the legal tasks to be judged reasonable.

It is important for AM to locate one of these currently-plausible tasks, but it's not worth spending much time deciding which of *them* to work on next. AM still faces a huge search: find one of the 6 winners out of a few thousand candidates.

Its choice of tasks is made even more important due to the 10-second "cycle time" – the time to investigate/execute one task. A human user is watching, and ten seconds is a nontrivial amount of time to him. He can therefore observe, perceive, and analyze each and every task that AM selects. Even just a few bizarre choices will greatly lower his opinion of AM's intelligence. The trace of AM's actions is what counts, not its final results. So AM can't draw much of its apparent intelligence from the *speed* of the computer.

Chess-playing programs have had to face the dilemma of the trade-off between "intelligence" (foresight, inference, processing,...) and total number of board situations examined. In chess, the characteristics of current-day machines, language power *vs.* speed, and (to some extent) the limitations of our understanding of how to be sophisticated, have to date unfortunately still favored fast, nearly-blind[7] search. Although machine speed and LISP slowness may allow blind search to win over symbolic inference for *shallow* searches, it can't provide any more than a constant speed-up factor for an exponential search. Inference is slowly gaining on brute force,[8] and must someday triumph.

Since the number of "legal moves" for AM at any moment is in the thousands, it is unrealistic to consider "systematically"[9] walking through the entire space that AM can reach. In AM's problem domain, there is so much "freedom" that symbolic inference finally *can* win over the "simple but fast" exploration strategy[10].

3.2. Constraining AM's Search

---

[7] i.e., using a very simple static evaluation function.

[8] E.g., see [Berliner 74]. There, searching is used mainly to verify plausible moves (a convergent process), not to discover them (a bushier search).

[9] e.g., exhaustively, or using $\alpha\beta$ minimaxing, etc.

[10] This is the author's opinion, partially supported by the results of AM. Paul Cohen disagrees, feeling that machine speed should be the key to an automated mathematician's success.

There exist too many combinations to consider all combinations of existing entities; the creative mind must only propose those of potential interest.

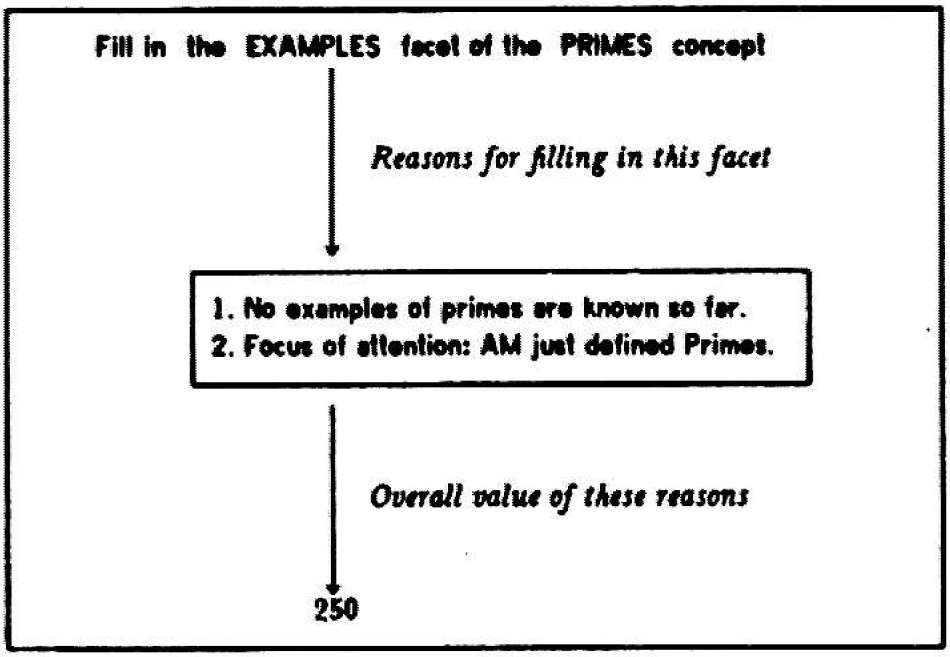
-- Poincare'

A great deal of heuristic knowledge is required to constrain the necessary processing effectively, to zero in on a good task to tackle next. This is done in two stages.

1. A list of plausible facet/concept pairs is maintained. Nothing can get onto this list unless there is some reason why filling in (or checking) that facet of that concept would be worthwhile.
2. All the plausible tasks on this "job list" are ranked by the number and strength of the different reasons supporting them. Thus the facet/concept pairs near the top of the list will all be very promising tasks to work on.

The first of these constraints is akin to replacing a *legal* move generator by a *plausible* move generator. The second kind of constraint is akin to using a heuristic evaluation function to select the best move from among the plausible ones.[11]

The job-list or *agenda* is a data structure which is a natural way to store the results of these procedures. It is (1) a list of all the plausible tasks which have been generated, and (2) it is kept ordered by the numeric estimate of how worthwhile each task is. A typical entry on the agenda might look like this:



[11] Past AI programs (e.g., [Samuel 67]) have indicated that constraining generation (1) is more important than sophisticated ordering of the resultant candidates (2). This was confirmed by the experiments performed on AM.

The actual top-level control structure is simply to pluck the top task from the agenda and execute it. That is, select the facet/concept pair having the best supporting reasons, and try to fill in that facet of that concept.

While a task is being executed, some new tasks might get proposed and merged into the agenda. Also, some new concepts might get created, and this, too, would generate a flurry of new tasks.

After AM stops filling in entries for the facet specified in the chosen task, it removes that task from the agenda, and moves on to work on whichever task is the highest-rated at that time.

The reader probably has a dozen good questions in mind at this point (e.g., How do the reasons get rated?, How do the tasks get proposed?, What happens after a task is selected?,...). The next section should answer most of these. Some more judgmental ones (How dare you propose a numeric calculus of plausible reasoning?! If you slightly de-tune all those numbers, does the system's performance fall apart?...) will be answered in Chapter 7.

3.3. The Agenda

Creative energy is used mainly to ask the right question.

-- Halmos

3.3.1. Why an Agenda?

This subsection provides motivation for the following one, by arguing that a job-list scheme is a natural mechanism to use to manage the task-selection problem AM faces. If that seems obvious to you, feel free to skip ahead to section 3.3.2, page 33.

Recall that AM must zero in on one of the best few tasks to perform next, and it repeatedly makes this choice. At each moment, there might be thousands of directions to explore (plausible tasks to consider).

If all the legal tasks were written out, and reasons were thought up to support each one, then perhaps we could order them by the strength of those reasons, and thereby settle on the "best" task to work on next. In order to appear "smart" to the human user, AM should *never* execute a task having no reasons attached.

Some magical function will be assumed to exist, which provides a numeric rating, a priority value, for any given task. The function looks at a given facet/concept pair, examines all the associated reasons supporting that task, and computes an estimate of how worthwhile it would be for AM to spend some time now working on that facet of that concept.

So AM will maintain a list of those legal tasks which have some good reasons tacked onto them, which justify why each task should be executed, why it is plausible. At least implicitly, AM has a numeric rating for each task. The obvious control algorithm is to choose the task with the highest rating, and work on that one next.

Assuming the tasks on this list are kept ordered by this numeric rating, then AM can just repeatedly pluck the highest task and execute it. While it's executing, some new tasks might get proposed and added to the list of tasks. Reasons are kept tacked onto each task on this list, and form the basis for the numeric priority rating.

Give or take a few features, this notion of a "job-list" is the one which AM uses. It is also called an *agenda*.[12] "A task on the agenda" is the same as "a job on the job-list" is the same as "a facet/concept pair which has been proposed" is the same as "an active node in the search space". Henceforth, I'll use the following all interchangeably: task, facet/concept pair, node, job. This should break up the monotony[13].

The flavor of agenda-list used here is similar to the control structure of HEARSAY-II [Lesser/Fennell/Erman/Reddy 75]. Vast numbers of tasks are proposed and added to the job-list. Occasionally, when some new data arrives, some task is repositioned

3.3.2. Details of the Agenda scheme

At each moment, AM has many plausible tasks (hundreds or even thousands) which have been suggested for some good reason or other, but haven't been carried out yet. Each task is at the level of working on a certain facet of a certain concept: filling it in, checking it, etc. Recall that each task also has tacked onto it a list of symbolic reasons explaining why the task is worth doing.

In addition, a number (between 0 and 1000) is attached *to each reason*, representing some absolute measure of the value of that reason (at the moment). One global formula[14] combines all the reasons' values into a single priority value for the task as a whole. This overall rating is taken to indicate how worthwhile it would be for AM to bother executing that task, how interesting the task would probably turn out to be. The "intelligence" of AM's selection of task is thus seen to depend on this one formula. Yet experiments show that its precise form is not important. We conclude that the "intelligence" has been pushed down into the careful assigning of reasons (and *their* values) for each proposed task.

---

[12] Borrowed from Kaplan's term for the job-list present in KRL (see [Bobrow & Winograd 77]). For an earlier general discussion of agendas, see [Knuth 68].

[13] and cover my sloppiness. Seriously, thanks to English, each of these terms will conjure up a slightly different image: a "job" is something to do, a "node" is an item in a search space, "facet/concept pair" reminds you of the format of a task.

[14] Here is that formula: $Worth(J) = |[SQRT(SUM\ R_i^2)]| \times [\ 0.2 \times Worth(A) + 0.3 \times Worth(F) + 0.5 \times Worth(C)]$, where J = job to be judged = (Act A, Facet F, Concept C), and $\{R_i\}$ are the ratings of the reasons supporting J. For the sample job pictured in the box below, A=Fillin, F=Examples, C=Sets, $\{R_i\}=\{100,100,200\}$. The formula will be repeated -- and explained -- in Section 4.2, on page 40.

A typical entry on the agenda might look like this:

TASK: Fill-in examples of Sets
PRIORITY: 300
REASONS:
100: No known examples for Sets so far.
100: Failed to fillin examples of Set-union, for lack of examples of Sets
200: Focus of attention: AM recently worked on the concept of Set-union

Notice the similarity of this to the initial few lines which AM types just after it selects a job to work on.

The flow of control is simple: AM picks the task with the highest priority value, and tries to execute it. As a side effect, new jobs occasionally get added to the agenda while the task is being executed.

The global priority value of the task also indicates how much time and space this task deserves. The sample task above might rate 20 cpu seconds, and 200 list cells. When either of these resources is used up, AM terminates work on the task, and proceeds to pick a new one. These two limits will be referred to in the sequel as "*time/space quanta*" which are allocated to the chosen task. Whenever several techniques exist for satisfying some task, the remaining time/space quanta are divided evenly among those alternatives; i.e., each method is tried for a small time. This policy of parceling out time and space quanta is called "activation energy" in [Hewitt 76] and called "resource-limited processes" in [Norman & Bobrow 75]. In the case of filling in examples of sets, the space quantum (200 cells) will be used up quickly (long before the 20 seconds expire).

There are two big questions now:

1. Exactly how is a task proposed and ranked?
   - How is a plausible new task first formulated?
   - How do the supporting reasons for the task get assigned?
   - How does each reason get assigned an absolute numeric rating?
   - Does a task's priority value change? When and how?
2. How does AM execute a task, once it's chosen?
   - Exactly what can be done during a task's execution?

The next chapter will deal with both of these questions. A detailed discussion of difficulties and limitations of these ideas can be found in Section 7.2, on page 156.

Chapter 4. Heuristic Rules

Assume that somehow AM has selected a particular task from the agenda -- say "Fill-in Examples of Primes". What precisely does AM do, in order to execute the task? How *are* examples of primes filled in?

The answer can be compactly stated as follows:

"AM selects relevant heuristics, and executes them."

This really just splits our original question into two new ones: (i) How are the relevant heuristics selected, and (ii) What does it mean for heuristics to be executed (e.g., how does executing a heuristic rule help to fill in examples of primes?).

These two topics (in reverse order) are the two major subjects of this chapter. Although several examples of heuristics will be given, the complete list is relegated to Appendix 3. [1]

The first section explains what heuristic rules look like (their "syntax", as it were). The next three sections illustrate how they can be executed to achieve their desired results (their "semantics").

Section 4.5 explains where the rules are stored and how they are accessed at the appropriate times.

Finally, the initial body of heuristics is analyzed. The informal knowledge they contain is categorized and described. Unintentionally, the distribution of heuristics among the concepts is quite nonhomogeneous; this too is described in Section 4.6.

4.1. Syntax of the Heuristics

Let's start by seeing what a heuristic rule looks like. In general (see [Davis & King 75] for historical references to production rules), it will have the form

If *<situational fluent>*
Then *<actions>*

As an illustration, here is a heuristic rule, relevant when checking examples of anything:

[1] There they are condensed and phrased in English. The reader wishing to see examples of the heuristics as they actually were coded in LISP should glance at Appendix 2.3.

If the current task is to Check Examples of any concept X,
and (For some Y) Y is a generalization of X,
and Y has at least 10 examples,
and all examples of Y are also examples of X,
Then print the following conjecture: X is really no more specialized than Y,
and add it to the Examples facet of the concept named "Conjectures",
and add the following task to the agenda: "Check examples of Y", for the reason: "Just as Y was no more general than X, one-of Generalizations(Y) may turn out to be no more general than Y", with a rating for that reason computed as the average of: ||Examples(Generalizations(Y))||, ||Examples(Y)||, and Priority(Current task).

As with production rules, and formal grammatical rules, each of AM's heuristic rules has a left-hand-side and a right-hand-side. On the left is a test to see whether the rule is applicable, and on the right is a list of actions to take if the rule applies. The left-hand-side will also be called the IF-part, the predicate, the preconditions, left side, or the situational fluent of the rule. The right-hand-side will sometimes be referred to as the THEN-part, the response, the right side, or the actions part of the rule.

4.1.1. Syntax of the Left-hand Side

The situational fluent is a LISP predicate, a function which always returns True or False (in LISP, it actually returns either the atom T or the atom NIL). This predicate may investigate facets of any concept (often merely to see whether they are empty or not), use the results of recent tests and behaviors (e.g., to see how much cpu time AM spent trying to work on a certain task), etc.

The left side is a conjunction of the form P1 ∧ P2 ∧... All the conjuncts, except the very first one, are arbitrary LISP predicates. They are only constrained to obey two commandments:

1. *Be quick!* (return either True or False in under 0.1 cpu seconds)
2. *Have no side effects!* (destroying or creating list structures or Lisp functions, resetting variables)

Here are some sample conjuncts that might appear inside a left-hand side (but *not* as the very first conjunct):

- More than half of the current task's time quantum is already exhausted,...
- There are some known examples of Structures,...
- Some generalization of the current concept (the concept mentioned as part of the current task) has an empty Examples facet,...
- The space quantum of the current task is gone, but its time allocation is less than 10% used up,...
- A task recently selected had the form "Restructure facet F of concept X", where F is any facet, and X is the current concept,...

- The user has used this system at least once before,...
- It's Tuesday,...

The very first conjunct of each left-hand side is special. Its syntax is highly constrained. It specifies the domain of applicability of the rule, by naming a particular facet of a particular concept to which this rule is relevant.

AM uses this first conjunct as a fast "pre-precondition", so that the only rules whose left-hand sides get evaluated are already known to be somewhat relevant to the task at hand. In fact, AM physically attaches each rule to the facet and concept mentioned in its first conjunct.[2] This will be discussed in more detail in Section 4.5, "Gathering relevant heuristics". This first conjunct will always be written out as follows, in this document (where A, F, and C are specified explicitly):

> The current task (the one just selected from the agenda) is of the form **"Do action A to the F facet of concept C"**

This can be viewed as the "syntax" of the very first conjunct on each rule's left-hand side. Here are two typical examples of allowable first conjuncts:

- The current task (the one last selected from the agenda) is of the form **"Check the Domain/range facet of concept X"**, where X is any operation
- The current task is of the form **"Fillin the examples facet of the Primes concept"**

These are the only guidelines which the left-hand side of a heuristic rule must satisfy. Any LISP predicate which satisfies these constraints is a syntactically valid left-hand side for a heuristic rule. It turned out later that this excessive freedom made it difficult for AM to inspect and analyze and synthesize its own heuristics; such a need was not foreseen at the time AM was designed.

Because of this freedom, there is not much more to say about the left-hand sides of rules. As the reader encounters heuristics in the next few sections, he should notice the (unfortunate) variety of conjuncts which may occur as part of their left-hand sides.

4.1.2. Syntax of the Right-hand Side

"Running" the left-hand-side means evaluating the series of conjoined little predicates there, to see if they all return True. If so, we say that the rule "triggers". In that case, the right-hand-side is "run", which means executing all the actions specified there. A single heuristic rule may have a list of several actions as its right-hand-side. The actions are executed in order, and we then say the rule has finished running.

Only the right-hand-side of a heuristic rule is permitted to have side effects. The right side of a rule is a series of little LISP functions, each of which is called an *action*.

---

[2] Sometimes, I will mention where a certain rule is attached; in that case, I can omit explicit mention of the first conjunct. Conversely, if I include that conjunct, I needn't tell you where the rule is stored.

Semantically, each action performs some processing which is appropriate in some way to the kinds of situations in which the left-hand-side would have triggered. The final value that the action function returns is irrelevant.

Syntactically, there is only one constraint which each function or "action" must satisfy: Each action has one of the following 3 side-effects, and no other side-effects:

1. It suggests a new task for the agenda.
2. It causes a new concept to be created.
3. It adds (or deletes) a certain entry to a particular facet of a particular concept.

To repeat: the right side of a rule contains a list of actions, each of which is one of the above three types. A single rule might thus result in the creation of several new concepts, the addition of many new tasks to the agenda, and the filling in of some facets of some already-existing concepts.

These three kinds of actions will now be discussed in the following three sections.

4.2. Heuristics Suggest New Tasks

This section discusses the "proposing a new task" kind of action.

Here is the basic idea in a nutshell: The left-hand-side of a rule triggers. Scattered among the "things to do" in its right-hand-side are some suggestions for future tasks. These new tasks are then simply added to the agenda list.

4.2.1. An Illustration: "Fill in Generalizations of Equality"

If a new task is suggested by a heuristic rule, then that rule must specify how to assemble the new task, how to get reasons for it, and how to evaluate those reasons. For example, here is a typical heuristic rule which proposes a new task to add to the agenda. It says to generalize a predicate if it is very rarely[3] satisfied:

If the current task was (Fill-in examples of X),
and X is a predicate,
and more than 100 items are known in the domain of X,
and at least 10 cpu seconds were spent trying to randomly instantiate X,
and the ratio of successes/failures is both >0 and less than .05
Then add the following task to the agenda: (Fill-in generalizations of X), for the following reason:

---

[3] The most suspicious part of the situational fluent (the IF-part) is the number ".05". Where did it come from? Hint: if all humans had f fingers, this would probably be 0.05 in base f. Seriously, one can change this value (to .01 or to .25) with virtually no change in AM's behavior. This is the conclusion of experiment 3 (see Section 5.2.3). Such empirical justification is one important reason for actually writing and running large programs like AM.

"X is rarely satisfied; a slightly less restrictive concept might be more interesting".
This reason's rating is computed as three times the ratio of nonexamples/examples found.

Even this is one full step above the actual LISP implementation, where "X is a predicate" would be coded as "(MEMBER X (EXAMPLES PREDICATE))". The function EXAMPLES(X) rummages about looking for already-existing examples of X. Also, the LISP code contains information for normalizing all the numbers produced, so that they will lie in the range 0-1000.

Let's examine an instance of where this rule was used. At some point, AM chose the task "Fillin examples of List-equality". One of the ways it filled in examples of this predicate was to run it on pairs of randomly-chosen lists, and observe whether the result was True or False[4]. Say that 244 random pairs of lists were tried, and only twice was this predicate satisfied. Sometime later, the IF part of the above heuristic is examined. All the conditions are met, so it "triggers". For example, the "ratio of successes to failures" is just 2/242, which is clearly greater than zero and less than 0.05. So the right-hand-side (THEN-part) of the above rule is executed. The right-hand side initiates only one action: the task "Fillin generalizations of List-equality" is added to the agenda, tagged with the reason "List-equality is rarely satisfied; a slightly less restrictive concept might be more interesting", and that reason is assigned a numeric rating of 3x(242/2) = 363.

Notice that the heuristic rule above supplied a little function to compute the value of the reason. That formula was: "three times the ratio of examples/nonexamples found".[5] Functions of this type, to compute the rating for a reason, satisfy the same constraints as the left-hand-side did: the function must be very fast and it must have no side effects. The "intelligence" that AM exhibits in selecting which task to work on ultimately depends on the accuracy of these local rule evaluation formulae. Each one is so specialized that it is "easy" for it to give a valid result; the range of situations it must judge is quite narrow. Note that these little formulae were hand-written, individually, by the author. AM wasn't able to create new little reason-rating formulae.

The reason-rating function is evaluated at the moment the job is suggested, and only the numeric result is remembered, not the original function. In other words, we tack on a list of reasons and associated numbers, for each job on the agenda. The agenda *doesn't* maintain copies of the reason-rating functions which gave those numbers. This simplification is used merely to save the system some space and time.

Let's turn now from the reason-rating formulae to the reasons themselves. Each reason supporting a newly-suggested job is simply an English sentence (an opaque string, a token). AM cannot do much intelligent processing on these reasons. AM is not allowed to inspect parts of it, parse it, transform it, etc. The most AM can do is compare two such tokens for equality. Of course, it is not to hard to imagine this capability extended to permit AM to

---

[4] The True ones became examples of List-equality, and the pairs of lists which didn't satisfy this predicate became known as non-examples (failures, foibles,..). A heuristic similar to this "random instantiation" one is illustrated in Section 4.4, on page 48.

[5] In actuality, this would be checked to ensure that the result lies between 0 and 1000.

syntactically analyze such strings, or to trivially compute some sort of "difference" between two given reasons.[6] Each reason is assumed to have some semantic impact on the user, and is kept around partly for that purpose.

Each reason will have a numeric rating (a number between 0 and 1000) assigned to it locally, by the heuristic rule which proposed the task for that reason. One global formula will then combine all the reasons' ratings into one single priority value for the task.

4.2.2. The Ratings Game

In general, a task on the agenda list will have several reasons in support of it. Each reason consists of an English phrase and a numeric rating. How can a task have more than one reason? There are two contributing factors: (i) A single heuristic rule can have several reasons in support of a job it suggests, and (ii) When a rule suggests a "new" task, that very same task may already exist on the agenda, with quite distinct reasons tacked on there. In that case, the new reason(s) are added to the already-known ones.

One global formula looks at all the ratings for the reasons, and combines them into a single priority value for the task as a whole. Below is that formula, in all its gory detail:

$$\text{Worth}(J) = \| \text{SQRT}(\text{SUM } R_i^2) \| \times [\ .2 \times \text{Worth}(A) + .3 \times \text{Worth}(F) + .5 \times \text{Worth}(C)]$$

Where J = job to be judged = (Act A, Facet F, Concept C)
and $\{R_i\}$ are the ratings of the reasons supporting J.

For example, consider the job J = (Check examples of Primes). The act A would be "Check", which has a numeric worth of 100. The facet F would be "Examples", which has a numeric worth of 700. The concept C would be "Primes", which at the moment might have Worth of 800. Say there were four reasons, having values 200, 300, 200, and 500. The double lines "||...||" indicate normalization, which means that the final value of the square-root must be between 0 and 1, which is done by dividing the result of the Square-root by 1000 and then truncating to 1.0 if the result exceeds unity.

In this case, we first compute $\text{Sqrt}(200^2 + 300^2 + 200^2 + 500^2) = \text{Sqrt}(420{,}000)$, which is about 648. After normalization, this becomes 0.648. The expression in square brackets in the formula[7] is actually computed as the dot-product of two vectors[8]; in this case it is the dot-product of (100 700 800) and (.2 .3 .5), which yields 630. This is multiplied by the normalized Square-root value, 0.648, and we end up with a final priority rating of 408.

The four reasons each have a fairly low priority, and the total priority of the task is

---

[6] It is in fact trivial to IMAGINE it. Of course DOING it is quite a bit less trivial. In fact, it probably is the toughest of all the "open research problems" I'll propose.

[7] Namely, [0.2xWorth(A) + 0.3xWorth(F) + 0.5xWorth(C)].

[8] Namely, <Worth(A), Worth(F), Worth(C)> and < .2, .3, .5 >. The dot-product of <a1 a2 a3...> and <b1 b2 b3...> is defined as (a1 x b1) + (a2 x b2) + (a3 x b3) +...

therefore not great. It is, however, higher than any single reason multiplied by 0.648. This is because there are many *distinct* reasons supporting it. The global formula uniting these reasons' values does not simply take the largest of them (ignoring the rest), nor does it simply add them up.

The above formula was intended originally as a first pass, an *ad hoc* guess, which I expected I'd have to modify later. Since it has worked successfully, I have not messed with it. There is no reason behind it, no justification for taking dot-products of vectors, etc. I concluded, and recent experiments tend to confirm, that the particular form of the formula is unimportant; only some general characteristics need be present:

1. The priority value of a task is a monotone increasing function of each of its reasons' ratings. If a new supporting reason is found, the task's value is increased. The better that new reason, the bigger the increase.
2. If an already-known supporting reason is re-proposed, the value of the task is *not* increased (at least, it's not increased very much). Like humans, AM is fooled whenever the same reason reappears in disguised form.
3. The priority of a task involving concept C should be a monotone increasing function of the overall worth of C. Two similar tasks dealing with two different concepts, each supported by the same list of reasons and reason ratings, should be ordered by the worth of those two concepts.

I believe that all of these criteria are absolutely essential to good behavior of the system. Several of the experiments discussed later bear on this question (See Section 6.2, page 125). Note that the messy formula given on the last page does incorporate all 3 of these constraints. In addition, there are a few features of that formula which, while probably not necessary or even desirable, the reader should be informed of explicitly:

1. The task's value does not depend on the order in which the reasons were discovered. This is not true psychologically of people, but it is a feature of the particular priority-estimating formula initially selected.
2. Two reasons are either considered identical or unrelated. No attempt is made to reduce the priority value because several of the reasons are overlapping semantically or even just syntacticaly. This, too, is no doubt a mistake.
3. There is no need to keep around all the individual reasons' rating numbers. The addition of a new reason will demand only the knowledge of the *number* of other reasons, and the old priority value of the task.
4. A task with no reasons gets an absolute zero rating. As new reasons are added, the priority slowly increases toward an absolute maximum which is dependent upon the overall worth of the concept and facet involved.

There is one topic of passing interest which should be covered here. Each possible Act A (e.g., Fillin, Check, Apply) and each possible facet F (e.g., Examples, Definition, Name(s)) is assigned a fixed numeric value (by hand, by the author). These values are used inside the formula on the last page, where it says 'Worth(A)' and 'Worth(F)'. They are fairly resistant to change, but certain orderings should be maintained for best results. E.g., "Examples" should be rated higher than "Specializations", or else AM may whirl away on a cycle of specialization long after the concept has been constrained into vacuousness. As for the Acts, their precise values turned out to be even less important than the Facets'.

Now that we've seen how to compute this priority value for any given task, let's not forget what it's used for. The overall rating has two functions:

(i) The tasks on the agenda list are ordered by their ratings, and AM always chooses the top task. Thus this rating determines which task to execute next. This is not an ironclad policy: In reality, AM prints out the top few tasks, and the user has the option of interrupting and directing AM to work on one of those other tasks instead of the very top one.

(ii) Once a task is chosen, its overall rating determines how much time and space AM will expend on it before quitting and moving on to a new task. The precise formulae are unimportant. Roughly, the 0-1000 rating is divided by ten to determine how much time to allow, in cpu seconds. The rating is divided by two to determine how much space to allow, in list cells.

4.3. Heuristics Create New Concepts

Recall that a heuristic rule's actions are of three types:

1. Suggest new tasks and add them to the agenda.
2. Create a new concept.
3. Fill in some entries for a facet of a concept.

This subsection discusses the second activity.

Here is the basic idea in a nutshell: Scattered among the "things to do" in the right-hand-side of a rule are some requests to create specific new concepts. For each such request, the heuristic rule must specify how to construct it. At least, the rule must specify ways of assembling enough facets of the new concept to disambiguate it from all the other known concepts. Typically, the rule will explain how to fill in the Definition of – or an Algorithm for – the new concept. After executing these instructions, the new concept will "exist", and a few of its facets will be filled in, and a few new jobs will probably exist on the agenda, indicating that AM might want to fill in certain other facets of this new concept in the near future.

4.3.1. An Illustration: Discovering Primes

Here is a heuristic rule that results in a new concept being created:

If the current task was (Fill-in examples of F),
and F is an operation from domain space A into range space B,
and more than 100 items are known examples of A (in the domain of F),
and more than 10 range items (in B) were found by applying F to these domain items,
and at least 1 of these range items is a distinguished member (esp: extremum)[9] of B,
Then (for each such distinguished member 'b'∈B) create the following new concept:

---

[9] This is handled as follows: AM takes the given list of range items. It eliminates any which are not interesting (according to Interesta(B)) or extreme (an entry on B.Exs-Bdy, the boundary examples of B). Finally, all those extreme range items are moved to the front of this list. AM begins walking down this list, creating new concepts according to the rule. Sooner or later, a timer (or a storage-space-watcher) will terminate this costly activity. Only the frontmost few range items on the list will have generated new concepts. So "especially" really just means priority consideration.

Name: F-inverse-of-b
Definition: λ (x) (F(x) is b)
Generalization: A
Worth: Average(Worth(A), Worth(F), Worth(B), Worth(b), ||Examples(B)||)
Interest: Any conjecture involving both this concept and either F or Inverse(F)

In case the user asks, the reason for doing this is: "Worthwhile investigating those A's which have an unusual F-value, namely, those whose F-value is b"
The total amount of time to spend right now on all of these new concepts is computed as: Half the remaining cpu time in the current task's time quantum.
The total amount of space to spend right now on each of these new concepts is computed as: The remaining space quantum for the current task.

Although some examples of F-inverse-of-b might be easily obtained (or already known) at the moment of its creation, the above rule doesn't specifically tell AM how to fill in that facet. The very last line of the heuristic indicates that a few cpu seconds may be spent on just this sort of activity: filling in facets of the new concept which, though not explicitly mentioned in the rule, are easy to fill in now. Any facet X which didn't get filled in "right now" will probably cause a new task to be added to the agenda, of the form: "Fillin facet X of concept F-inverse-of-b". Eventually, AM would choose that task, and spend a large quantum of time working on that single facet.

Heuristics for the new concept are quite hard to fill in. This was one of AM's most serious limitations, in fact (see Chapter 7). Above, we see a trivial kind of "heuristic schema" or template, which gets instantiated to provide one new, specialized heuristic about the new concept. That new heuristic tells how to judge the interestingness of any conjecture which crops up involving this new concept. Whenever such conjectures get proposed, they are evaluated by calling on just such heuristics.

Now let's look at an instance of when this heuristic was used. At one point, AM was working on the task "Fill-in examples of Divisors-of".

This heuristic's IF-part was triggered because: Divisors-of is an operation (from Numbers to Sets of numbers), and far more than 100 different numbers are known, and more than 10 different sets of factors were found altogether, and some of them were distinguished by being extreme kinds of sets: empty-sets, singletons, doubletons and tripletons.

After its left side triggered, the right side of the heuristic rule was executed. Namely, four new concepts were created immediately. Here is one of them:

Name: Divisors-of-Inverse-of-Doubleton
Definition: λ (x) (Divisors-of(x) is a Doubleton)
Generalization: Numbers
Worth: 100
Interest: Any conjecture involving both this concept and either Divisors-of or Times

This is a concept representing a certain class of numbers, in fact the numbers we call *primes*. The heuristic resets a certain variable, so that in case the user interrupts and asks *Why?*, AM informs him:

"This concept was created because it's worthwhile investigating those numbers which have an extreme divisors-of value; in this case, numbers which have only two divisors".

AM was willing to spend half the remaining quantum of time allotted to "Fillin examples of Divisors-of" on these four new concepts[10].

The heuristic rule is applicable to any operation, not just numeric ones. For example, when AM was filling in examples of Set-Intersection, it was noticed that some pairs of sets were mapped into the extreme kind of set Empty-set. The above rule then had AM define the concept of *Disjointness*: pairs of sets having empty intersection.

4.3.2. The Theory of Creating New Concepts

All the heuristic rule must do is to fill in enough facets so that the new concept is disambiguated from all the others, so that it is "defined" clearly. Should AM pause and fill in lots of facets at that time? After all, several pieces of information are trivial to obtain at this moment, but may be hard to reconstruct later (e.g., the reason why C was created). On the other hand, filling in anything without a good reason is a bad idea (it uses up time and space, and it won't dazzle the user as a brilliant choice of activity).

So the universal motto of AM is to fill in facets of a new concept if – and only if – that filling-in activity will be much easier at that moment than later on.

In almost all cases, the following facets[11] will be specified explicitly in the heuristic rule, and thus will get filled in right away: Definitions, Algorithms, Domain/range, Worth, plus a tie to

---

[10] Some trivial details: One-eighth of the remaining time is spent on each of these 4 concepts: Numbers-with-0-divisors, Numbers-with-1-divisor, Numbers-with-2-divisors, Numbers-with-3-divisors. The original time/space limits were in reality about 25 cpu seconds and 800 list cells, and at the moment this heuristic was called, only about 10 seconds and 600 cells remained, so e.g. the concept Primes was allotted only 1.2 cpu seconds to "get off the ground". This was no problem, as it used far less than that. The heuristic rule states that each of the four new concepts may use up the full remaining space allocation (600 cells), and, e.g., Primes needed only a fraction of that initially.

[11] The reader may wish to glance ahead to Section 5.2, page 67 to note the full range of facets that any concept may possess: what their names are, and the kind of information that is stored in each.

some related concept (e.g., if the new concept is a generalization of Equality, then we can trivially fill in an entry on its Specializations facet: "Equality".)

On the other hand, the following facets will *not* be trivial to fill in: Conjectures, Examples, Generalizations, Specializations, and Interestingness. For example, filling in the Specializations facet of a new concept may involve creating some new concepts; finding some entries for its Conjectures facet may involve a great deal of experimenting; finding some Examples of it may involve twisting its definition around or searching. None of these is easier to do at time of creation than any other time, so it's deferred until some reason for doing it exists.

For each such "time-consuming" facet F, of the new concept C, one new task gets added to the agenda, of the form "Fill in entries for facet F of concept C", with reasons of the form "Because C was just created," and also "No entries exist so far on C.F"[12]. Most of the tasks generated this way will have low priority ratings, and may stay near the bottom of the agenda until/unless they are re-suggested for a new reason.

Using the Primes example, from the last subsection, we see that a new task like "Fillin specializations of Primes" was suggested with a low rating, and "Fillin examples of Primes" was suggested with a mediocre[13] rating. The ratings of these tasks increase later on, when the same tasks are re-proposed for new reasons.

4.3.3. Another Illustration: Squaring a number

Let's take another simple (but not atypical) illustration of how new concepts get created. (The reader may skip this subsection; it contains more details about how AM actually sets up new concepts.)

Assume that AM has recently discovered the concept of multiplication, which it calls "TIMES," and AM decides that it is very interesting. A heuristic rule exists which says:[14]

> **If a newly-interesting operation F(x,y) takes a pair of N's as arguments,**
> **Then create a new concept, a specialization of F, called F-itself, taking just one N as argument, defined as F(x,x), with initial worth Worth(F).**

In the case of F = TIMES, we see that F takes a pair of numbers as its arguments, so the heuristic rule would have AM create a new concept called TIMES-itself, defined as TIMES-

---

[12] C.F is an abbreviation for facet F of concept C

[13] Not as low a rating as the task just mentioned. Why? Each possible facet has a worth rating which is fixed once and for all. As an illustration, we mention that the facet Examples is rated much higher than Specializations. Why is this? Because looking for examples of a concept is often a good expenditure of time, producing the raw data on which empirical induction thrives. On the other hand, each specialization of the new concept C would itself be a brand new concept. So filling in entries for the Specializations facet would be a very explosive process.

[14] By glancing back at the Primes example, two subsections ago, page 42, you can imagine what this rule actually looked like. There is nothing to be gained by stretching it out in all its glory, hence I've taken the liberty condensing it, inserting pronouns, etc.

Itself(x) = TIMES(x,x). That is, create the new concept which is the operation of *squaring a number*.

What would AM do in this situation? The global list of concepts would be enlarged to include the new atom "TIMES-Itself", and the facets of this new concept would begin to be filled in. The following facets would get filled in almost instantly:

NAME: TIMES-Itself

DEFINITIONS:
ORIGIN: λ(x,y) [TIMES.DEFN(x,x,y)]

ALGORITHMS: λ(x) [TIMES.ALG(x,x)]

DOMAIN/RANGE: Number → Number

GENERALIZATIONS: TIMES

WORTH: 600

The name, definition, domain/range, generalizations, and worth are specified explicitly by the heuristic rule.

The lambda expression stored under the definition facet is an executable LISP predicate, which accepts two arguments and then tests them to see whether the second one is equal to TIMES-Itself of the first argument. It performs this test by calling upon the predicate stored under the definition facet of the TIMES concept. Thus TIMES-Itself.Defn(4,16) will call on TIMES.Defn(4,4,16), and return whatever value *that* predicate returns (in this case, it returns True, since 4x4 does equal 16).

A trivial transformation of this definition provides an algorithm for computing this operation. The algorithm says to call on the Algorithms facet of the concept TIMES. Thus TIMES-Itself.Alg(4) is computed by calling on TIMES.Alg(4,4) and returning *that* value (namely, 16).

The worth of TIMES was 600 at the moment TIMES-Itself was created, and this becomes the worth of TIMES-Itself.

TIMES-Itself is by definition a specialization of TIMES, so the SPECIALIZATIONS facet of TIMES is incremented to point to this new concept. Likewise, the GENERALIZATIONS facet of TIMES-Itself points to TIMES.

Note how easy it was to fill in these facets now, but how difficult it might be later on, "out of context". By way of contrast, the task of, e.g., filling in *Specializations* of TIMES-Itself will be no harder later on than it is right now, so we may as well defer it until there's a good

reason for it. This task will probably be added to the agenda with so low a priority that AM will never get around to it, unless some new reasons for it emerge.

The task "Fill-in examples of TIMES-itself" is probably worthwhile doing soon, but again it won't be any harder to do at a later time than it is right now. So it is not done at the moment; rather, it is added to the agenda (with a fairly high priority).

Incidentally, the reader may be interested to know that the next few tasks AM selected (in reality) were to create the inverse of this operation (i.e., integer square-root), and then to create a new kind of number, the ones which can be produced by squaring (i.e., perfect squares). Perfect squares were deemed worth having around because Integer-square-root is *defined* precisely on that set of integers.

4.4. Heuristics Fill in Entries for a Specific Facet

The last two subsections dealt with how a heuristic rule is able to propose new tasks and create new concepts. This section will illustrate how a rule can find some entries for a given facet of a specific concept.

Typically, the facet/concept involved will be the one mentioned in the current task which was chosen from the agenda. If the task "Fillin Examples of Set-union" were plucked from the agenda, then the "relevant" heuristics would be those useful for filling in entries for the Examples facet of the Set-union concept.

There is an important class of exceptions to this, however: conjectures. Some rules will specify plausible relationships to look for; if found, they constitute a new conjecture. For example, the reader will see in Section 4.4.4, on page 52, that the unique factorization theorem is proposed merely as an observation of the form "The range of operation F is not just B but rather the more specialized concept BB". The particular case of the unique factorization theorem leads to this statement: "The range of Prime-factorings[15] is not just 'Sets' but rather 'Singletons'." In fact, this whole conjecture is recorded by merely replacing <Number→Set> by <Number→Singleton> as an entry on the Domain/range facet of the concept Prime-factorings.

The reader may be surprised to learn that the only kind of conjecture AM can make is of that form (add a new entry to some facet of some concept)[16]. Apparently, this is sufficient to plausibly notice and state most interesting conjectures. Good definitions make the statements of theorems short and simple.[17]

---

[15] Prime-factorings(x), also called Prime-times(x), is the set of all bags-of-primes whose product is x; i.e., all ways of factoring x into primes.

[16] That's why "conjecturing" is classified under the "add-an-entry" type of heuristic rule action.

[17] Exercise for the doubting reader: State the unique factorization theorem in purely set-theoretic terms. Seriously, one important way that definitions are invented is to see what bulky construct in a theorem can be collapsed into a single term. Typically one hopes that the term will be used elsewhere, of course.

We'll take these two kinds of "filling in entries" one at a time: first the standard "find an entry for the facet of the concept mentioned in the current task", followed by the interesting but rarer activity of "looking for a conjecture".

4.4.1. An Illustration: "Fill in Examples of Set-union"

Recall that a task is typically of the form "Fill in facet F of concept C". How can executing relevant heuristic rules satisfy such a task? This subsection illustrates how a heuristic rule might be executed to find some entries for the facet designated by the current task.

A typical heuristic, attached to the concept Activity, says:

> **If the current task is to fill in examples of the activity[18] F,**
> **One way to get them is to run F on randomly chosen examples of the domain of F.**

Of course, in the LISP implementation, this situation-action rule is not coded quite so neatly. It would be more faithfully translated as follows:

If CURRENT-TASK matches (FILLIN EXAMPLES F←anything)),
and F isa Activity,
Then carry out the following procedure:
1. **Find the domain of F, and call it D;**
2. **Find examples of D, and call them E;**
3. **Find an algorithm to compute F, and call it A;**
4. **Repeatedly:**
   - **4a. Choose any member of E, and call it E1.**
   - **4b. Run A on E1, and call the result X.**
   - **4c. Check whether <E1,X> satisfies the definition of F.**
   - **4d. If so, then add <E1 → X> to the Examples facet of F.**
   - **4e. If not, then add <E1 → X> to the Non-examples facet of F.**

Let's take a particular instance where this rule would be useful. Say the current task is "Fillin examples of Set-union". The left-hand-side of the rule is satisfied, so the right-hand-side is run.

Step (1) says to locate the domain of Set-union. The facet labelled Domain/Range, on the Set-union concept, contains the entry (SET SET → SET), which indicates that the domain is a pair of sets. That is, Set-union is an operation which accepts (as its arguments) a pair of sets, and returns (as its value) some new set.

Since the domain elements are sets, step (2) says to locate examples of sets. The facet labelled Examples, on the Sets concept, points to a list of about 30 different sets. This includes {Z}, {A,B,C,D,E}, {}, {A,{{B}}},...

Step (3) involves nothing more than accessing some randomly-chosen entry on the Algorithms facet of Set-union. One such entry is a recursive LISP function of two arguments, which halts when the first argument is the empty set, and otherwise pulls an

---

[18] "Activity" is a general concept which includes operations, predicates, relations, functions, etc.

element out of that set and SET-INSERT's it into the second argument, and then recurses on the new values of the two sets. For convenience, we'll refer to this algorithm as UNION.

We then enter the loop of Step (4). Step (4a) has us choose one pair of our examples of sets, say the first two {Z} and {A,B,C,D,E}. Step (4b) has us run UNION on these two sets. The result is {A,B,C,D,E,Z}. Step (4c) has us grab an entry from the Definitions facet of Set-union, and run it. A typical definition is this formal one:

```
(λ (S1 S2 S3)
      (AND
            (For all x in S1, x is in S3)
            (For all x in S2, x is in S3)
            (For all x in S3, x is in S1 or x is in S2)
      )
) )
```

It is run on the three arguments S1={Z}, S2={A,B,C,D,E}, S3={A,B,C,D,E,Z}. Since it returns "True", we proceed to Step (4d). The construct <{Z}, {A,B,C,D,E} → {A,B,C,D,E,Z}> is added to the Examples facet of Set-union.

At this stage, control returns to the beginning of the Step (4) loop. A new pair of sets is chosen, and so on.

But when would this loop stop? Recall that each task has a time and a space allotment (based on its priority value). If there are many different rules all claiming to be relevant to the current task, then each one is allocated a small fraction of those time/space quanta. When either of these resources is exhausted, AM would break away at a "clean" point (just after finishing a cycle of the Step (4) loop) and would move on to a new heuristic rule for filling in examples of Set-union.

This concludes the demonstration that a heuristic rule really can be executed to produce the kinds of entities requested by the current task.

4.4.2. Heuristics Propose New Conjectures

We saw in the sample excerpt (Chapter 2) that AM occasionally notices some unexpected relationship, and formulates it into a precise conjecture. Below is an example of how this is done. As you might guess from the placement of this subsection,[19] the mechanism is our old friend the heuristic rule which fills in entries for certain facets.

In fact, a conjecture evolves through four stages:

1. A heuristic rule looks for a particular kind of relationship. This will typically be of the form "X is a Generalization of Y", or "X is an example of Y", or "X is the same as Y", or "F1.Defn(X,Y)" where F1 is an active concept AM knows about, or

[19] or recall from the opening remarks of Section 4.4

"F1.Defn(Y,X)"[20].

2. Once found, the relationship is checked, using supporting contacts. A great deal of empirical evidence must favor it, and any contradictory evidence must be "explained away" somehow.
3. Now it is believed, and AM prints it out to the user. It is added as a new entry to the Conjecs facet of both concepts X and Y. It is also added as an entry to the Examples facet of the Conjecture concept.
4. Eventually, AM will get around to the task "Check Examples of Conjecture", or to the task "Check Conjecs of X". If AM had any concepts for proving conjectures, they would then be invoked. In the current LISP implementation, these are absent. Nevertheless, several "checks" are performed: (*i*) see if any new empirical evidence (pro or con) has appeared recently; (*ii*) see if this conjecture can be strengthened; (*iii*) check it for extreme cases, and modify it if necessary; (*iv*) Modify the worth ratings of the concepts involved in the conjecture.

The left-hand-side of such a heuristic rule will be longer and more complex than most other kinds, but the basic activities of the right-hand-side will still be filling in an entry for a particular facet.

The entries filled in will include: (*i*) a new example of Conjectures, (*ii*) a new entry for the Conjec facet of each concept involved in the conjecture, (*iii*) if we're claiming that concept X is a generalization of concept Y, then "X" would be added to the Generalizations facet of Y, and "Y" added to the Specializations facet of X, (*iv*) if X is an Example of Y, "X" is added to the Examples facet of Y, and "Y" is added to the ISA facet of X.

The right-hand-side may also involve adding new tasks to the agenda, creating new concepts, and modifying entries of particular facets of particular concepts. As is true of all heuristic rules, both sides of this type of conjecture-perceiving rule may run any little functions they want to: any functions which are quick and have no side effects (e.g., FORALL tests, PRINT functions, accesses to a specified facet of some concept).

4.4.3. An Illustration: "All primes except 2 are odd"

As an illustration, here is a heuristic rule, relevant when checking examples of any concept:

[20] These last two say that F1(X)=Y, and that F1(Y)=X, respectively.

```
If the current task is to Check Examples of X,
    and (For some Y) Y is a generalization of X,
    and Y has at least 10 examples,
    and all examples of Y (ignoring boundary cases) are also examples of X,
Then print the following conjecture: X is really no more specialized than Y,
    and add it to the Examples facet of Conjectures,
    and if the user asks, inform him that the evidence for this was that all ||Examples(Y)|| Y's
            (ignoring boundary examples of Y's) turned out to be X's as well,
    and Check the truth of this conjecture on boundary examples of Y,
    and add "X" to the Generalizations facet of Y,
    and add "Y" to the Specializations facet of X,
    and (if there is an entry in the Generalizations facet of Y) add the following task to the
            agenda "Check examples of Y", for the reason: "Just as Y was no more
            general than X, one-of Generalizations(Y) may turn out to be no more
            general than Y", with a rating for that reason computed as:
                0.4x||Examples(Generalizations(Y))|| +
                0.3x||Examples(Y)|| +
                0.3xPriority(Current task).
```

Let's take a particular instance where this rule would be useful. Say the current task is "Check examples of Odd-primes". The left-hand-side of the rule is run, and is satisfied when the generalization Y is the concept "Primes". Let's see why this is satisfied.

One of entries of the Generalization facet of Odd-primes is "Primes". AM grabs hold of the 30 examples of primes (located on the Examples facet of Primes), and removes the ones which are tagged as boundary examples ("2" and "3"). A definition of Odd-prime numbers is obtained (Definitions facet of Odd-primes), and it is run on each remaining example of primes (5, 7, 11, 13, 17, ...). Sure enough, they all satisfy the definition. So all primes (ignoring boundary cases) appear to be odd. The left-hand-side of the rule is satisfied.

At this point, the user sees a message of the form "Odd-primes is really no more specialized than Primes". If he interrupts and asks about it, he is told that the evidence for this was that all 30 primes (ignoring boundary examples of primes) turned out to be Odd-primes.

Of the boundary examples (the numbers 2 and 3), only the integer "2" fails to be an odd-prime, so the the user is notified of the finalized conjecture: "All primes (other than '2') are also odd-primes". This is added as an entry on the Examples facet of the concept named 'Conjectures.'

Before beginning all this, the Generalizations facet of Odd-primes pointed to Primes. Now, this rule has us add "Primes" as an entry on the Specializations facet of Odd-primes. Thus Primes is both a generalization and a specialization of Odd-primes (to within a single stray exception), and AM will be able to treat these two concepts as if they were merged together. They are still kept separate, however, in case AM ever needs to know precisely what the difference between them is, or in case later evidence shows the conjecture to be false[21].

---

[21] When space is exhausted, one emergency measure AM takes is to destructively coalesce a pair of concepts X,Y where X is both a generalization of and a specialization of Y, even if there are a couple "boundary" exceptions.

The final action of the right-hand-side of this rule is to propose a new task (if there exist some generalizations of the concept Y, which in our case is "Primes"). So AM accesses the Generalizations facet of Primes, which is "(Numbers)". A new task is therefore added to the agenda: "**Check examples of Primes**", with an associated reason: "Just as Primes was no more general than Odd-primes, so Numbers may turn out to be no more general than Primes". The reason is rated according to the formula given in the rule; say it gets the value 500.

To make this example a little more interesting, let's suppose that the task "**Check examples of Primes**" already existed on the agenda, but for the reason "Many examples of Primes have been found, but never checked", with a rating for the reason of 100, and for the task as a whole of 200. The global task-rating formula then assigns the task a new overall priority of 600, because of the new, fairly good reason supporting it.

When that task is eventually chosen, the heuristic rule pictured above (at the beginning of this subsection) will trigger and will be run again, with X=Primes and Y=Numbers. That is, AM will be considering whether (almost) all numbers are primes. The left-hand-side of the heuristic rule will quickly fail, since, e.g., "6" is an example of Numbers which does not satisfy the definition of Primes.

4.4.4. Another illustration: Discovering Unique Factorization

Below is a heuristic rule which is a key agent in the process of "noticing" the fundamental theorem of arithmetic[22]. (The reader may skip this subsection; it contains more details about how AM actually proposed conjectures).

> **If F(a) is unexpectedly a B,**
> **Then maybe (Vx) F(x) is a B.**

Below, the same rule is given in more detail. The first conjunct on the IF-part of the heuristic rule indicates that it's relevant to checking examples of any given operation F. A typical example is selected at random, say F(x)=y. Then y is examined, to see if it satisfies any more stringent properties than those specified in the Domain/range facet of F. That is, the Domain/range facet of F contains an entry of the form A→B; so if x is an A, then all we are guaranteed about y is that it is an example of a B. But now, this heuristic is asking if y isn't really an example of a much more specialized concept than B. If it is (say it's an example of a BB), then the rest of the examples of F are examined to see if they too satisfy this same property. If all examples appear to map from domain set A into range set BB (where BB is much more restricted than the set B specified originally in the Domain/range facet of F), then a new conjecture is made: the domain/range of F is really A→BB, not A→B. Here is that rule, in crisper notation:

---

[22] The unique factorization conjecture: any positive integer n can be represented as the product of prime numbers in precisely one way (to within reorderings of those prime factors). Thus 28 = 2x2x7, and we don't distinguish between the factorization (2 2 7) and (2 7 2)

If the current task is to Check Examples of the operation F,
and F is an operation from domain A into range B,
and F has at least 10 examples,
and a typical one of these examples is "<x→y>" (so x∈A and y∈B),
and (Foreome Specialization BB of B), y is a BB.
and all examples of F (ignoring boundary cases) turn out to be BB's,
Then print the following conjecture: "F(a) is always a BB, not simply a B",
and add it to the Examples facet of Conjectures concept,
and add "<A → BB>" as a new entry to the Domain/range facet of F, replacing "<A→B>",
and if the user asks, inform him that the evidence for this was that all ||Examples(F)|| examples of F (ignoring boundary examples) turned out to be BB's,
and check the truth of this conjecture by running F on boundary examples of A.

Let's see how this rule was used in one instance. In Task 79 in the sample excerpt in Chapter 2, AM defined the concept Prime-times, which was a function transforming any number n into the set of all factorizations of n into primes. For example, Prime-times(12)={(2 2 3)}, Prime-times(13)={(13)}. The domain of F=Prime-times was the concept Numbers. The range was Sets. More precisely, the range was Sets-of-Bags-of-Numbers, but AM didn't know that concept at that time.

The above heuristic rule was applicable. F was Prime-times, A was Numbers, and B was Sets. There were far more than 10 known examples of Prime-times in action. A typical example was this one: <21 → {(3,7)}>. The rule now asked that {(3,7)} be fed to each specialization of Sets, to see if it satisfied any of their definitions. The Specializations facet of Sets was acccessed, and each concept pointed to was run (its definition was run) on the argument "{(3,7)}". It turned out that Singleton and Set-of-doubletons were the only two specializations of Sets satisfied by this example. At this moment, AM had narrowed down the potential conjectures to these two:

1. Prime-times(x) is always a singleton set.
2. Prime-times(x) is always a set of doubletons.

Each example of Prime-times was examined, until one was found to refute each conjecture (for example, <8→{(2,2,2)}> destroys conjecture 2). But no example was able to disprove conjecture 1. So the heuristic rule plunged forward, and printed out to the user "A new conjecture: Prime-times(n) is always a singleton-set, not simply a set". The entry <Numbers→Singleton-sets> was added to the Domain/range facet of Prime-times, replacing the old entry <Numbers→Sets>.

Let's digress for a moment to discuss the robustness of the system. What if this heuristic were to be excised? Could AM still propose unique factorization? The answer is yes, there are other ways to notice it. If AM has the concept of a Function[23], then a heuristic rule like the one in the previous subsection (page 50) will cause AM to ask if Prime-times is not merely a relation, but also a Function.

---

[23] A single-valued relation. That is, for any domain element x, F(x) contains precisely one member. It is never empty (i.e., undefined), nor is it ever larger than a singleton (i.e., multiple-valued).

The past few sections should have convinced the reader that isolated heuristic rules really can do all kinds of things: propose new tasks, create new concepts, fill in entries for specific facets (goal-driven), and look for conjectures (data-driven empirical induction). The rules appear fairly general[24] – though that must be later verified empirically. They are redundant in a pleasing way: some of the most "important", well-known, and interesting conjectures can (apparently) be derived in many ways. Again, we'll have to check this experimentally.

4.5. Gathering Relevant Heuristics

Each concept has facets which contain some heuristics. Some of these are for filling in, some for checking, some for deciding interestingness[25], some for noticing new conjectures, etc.

AM contains hundreds of these heuristics. In order to save time (and to make AM appear more rational), each heuristic should only be tried in situations where it might apply, where it makes sense.

How is AM able to zero in on the relevant heuristic rules, once a task has been selected from the agenda?

4.5.1. Domain of Applicability

The secret is that each heuristic rule is stored somewhere *a propos* to its "domain of applicability". This "proper place" is determined by the first conjunct in the left-hand side of the rule.

What does this mean? Consider this heuristic:

If the current task is to fill in examples of the operation F, <====
and some examples of the domain of F are known,
Then one way to get examples of F is to run F on randomly chosen examples of the domain of F.

This is a reasonable thing to try – but only in certain situations. Should it be tried when the current task is to check the Worth facet of the Sets concept? No, it would be irrational. Of course, even if it were tried then, the left-hand-side would fail very quickly. Yet *some* cpu time would have been used, and if the user were watching, his opinion of AM would

---

[24] ie, applicable in many situations. It would be worse than useless if a rule existed which could lead to a single discovery like "Fibonacci series" but never lead to any other discoveries. The reasons for demanding generality are not only "fairness", but the insights which occur when it is observed that several disparate concepts were all motivated by the same general principle (eg, "looking for the inverse image of extrema").

[25] The reader has already seen several heuristics useful for filling in and checking facets; here is one for judging interestingness: an entry on the Interest facet of Compose says that a composition AoB is more interesting if the range of B equals the domain of A, than if if they only partially overlap.

decrease.[26]

That particular heuristic has a precise domain of applicability: AM should use it whenever the current task is to fill in examples of an operation, and only in those kinds of situations.

The key observation is that a heuristic typically applies to *all examples of a particular concept C*. In the case we were considering, C=Operation. Intuitively, we'd like to tack that heuristic onto the Examples facet of the concept Operation, so it would only "come to mind" in appropriate situations. This is in fact precisely where the heuristic rule *is* stored.

Initially, the author identified the proper concept C and facet F for each heuristic H which AM possessed, and tacked H onto C.F[27]. This was all preparation, completed long before AM started up. Each heuristic was tacked onto the facet which uniquely indicates its domain of applicability. The first conjunct of the IF-part of each heuristic indicates where it is stored and where it is applicable. Notice the little arrow (<==) pointing to that conjunct above.[28].

While AM is running, it will choose a task dealing with, say, facet F of concept C. AM must quickly locate the heuristic rules which are relevant to satisfying that chosen task. AM simply locates all concepts which claim C as an example. If the current task were "Check the Domain/range of UnionoUnion"[29], then C would be UnionoUnion. Which concepts claim C as an example? They include Compose-with-Self, Composition, Operation, Active, Any-concept, and Anything. AM then collects the heuristics tacked onto facet F (in this case, F is Domain/range) of each of those concepts. All such heuristics will be relevant. In the current case, some relevant heuristics might be garnered from the Domain/range facet of the concept Operation. Any heuristic which can deal with the Domain/range facet of *any* operation can certainly deal with UnionoUnion's Domain/range. A typical rule on Operation.Domain/range.Check[30] would be this one:

> **If a Dom/ran entry of F is of the form <D D D...D → R>, where R is a generalization of D,**
> **Then test whether the range might not be simply D.**

Suppose one entry on UnionoUnion.Dom/ran was '<Nonempty-sets Nonempty-sets Nonempty-sets → Sets>'. Then this last heuristic rule would be relevant, and would have AM ask the plausible question: Is the union of three nonempty sets always nonempty? The

---

[26] This notion of worrying about a human user who is observing AM run in real time may appear to be quite language- and machine-dependent. An increase in speed of a couple orders of magnitude would radically alter the qualitative appearance of AM. In Chapter 7, however, the reader will grasp how difficult it is to objectively rate a system like AM. For that reason, all measures of judgment must be respected. Also, to the actual human being using the system this really is one of the most important measures.

[27] Recall that C.F is an abbreviation for facet F of concept C.

[28] In the LISP implementation, these first conjuncts are omitted, since the *placement* of a heuristic serves the same purpose as if it had some "pre-preconditions" (like these first conjuncts) to determine relevance quickly.

[29] This operation is defined as UnionoUnion(x,y,z) = (x U y) U z. It accepts 3 sets as arguments, and returns a new set as its value.

[30] the 'Check' subfacet of the 'Domain/range' facet of the 'Operation' concept.

answer is affirmative, empirically, so AM modifies that Domain/range entry for Union∘Union. AM would ask the same question for Intersect∘Intersect. Although the answer then would be '*No*', it's still a rational inquiry. If AM called on this heuristic rule when the current task was "Fillin specializations of Bags", it would clearly be an irrational act. The domain of applicability of the rule is clear, and is precisely fitted to the slot where the rule is stored (tacked onto Operation.Domain/range).

To recap the basic idea: when dealing with a task "Do act A on facet F of concept C", AM must locate all the concepts X claiming C as an example. AM then gathers the heuristics tacked onto X.F.A, for each such general concept X. All of them – and only they – are relevant to satisfying that task.

So the whole problem of locating relevant heuristics has been reduced to the problem of efficiently finding all concepts of which C is an example (for a given concept C). This process is called "*rippling away from C in the ISA direction*", and forms the subject of the next subsection.

4.5.2. Rippling

Given a concept C, how can AM find all the concepts which claim C as an example?

The most obvious scheme is to store this information explicitly. So the Examples facet of C would point to all known examples of C, and the Isa facet of C would point to all known concepts claiming C as one of their examples. Why not just do this? Because one can substitute a modest amount of processing time (via chasing links around) for the vast amount of storage space that would be needed to have "everything point to everything".

Each facet contains only enough pointers so that the entire graph of Exs/Isa and Spec/Genl links could be reconstructed if needed. Since "Genl"[31] is a transitive relation, AM can compute that Numbers is a generalization of Mersenne-primes, if the facet Mersenne-primes.Genl contains the entry "Odd-primes", and Odd-primes.Genl contains a pointer to "Primes", and Primes.Genl points to "Numbers". This kind of "*rippling*" activity is used to efficiently locate all concepts related to a given one X. In particular, AM knows how to "ripple upward in the Isa direction", and quickly[32] locate all concepts which claim X as one of their examples.

It turns out that AM cannot simply call for X.Isa, then the Isa facets of those concepts, etc.,

---

[31] "Genl" is an abbreviation for the Generalizations facet of a concept; similarly, "Spec" means Specializations, Exs means Examples, etc. "Isa" is the converse facet to Exs; i.e., A ∈ B Exs iff B ∈ A Isa. Saying "Genl is transitive" just means the following: if A is a generalization of B, and B of C, then A is also a generalization of C.

[32] With about 200 known concepts, with each Isa facet and each Genl facet pointing to about 3 other concepts, about 25 links will be traced along in order to locate about a dozen final concepts, each of which claims the given one as an example. This whole rippling process, tracing 25 linkages, uses less than .01 cpu seconds, in compiled Interlisp, on a KI-10 type PDP-10.

because Isa is not transitive[33]. For the interested reader, the algorithm AM uses to collect Isa's of X is given below.[34]

1. All generalizations of the given concept X are located. AM accesses X.Genl, then the Genl facets of *those* concepts, etc.
2. The "Isa" facet of each of those concepts is accessed.
3. AM locates all generalizations of these newly-found higher-level concepts. This is the list of all known concepts which claim X as one of their examples.

In regular form, one might express this rippling recipe more compactly as: $Genl^{*}(Isa(Genl^{*}(X)))$. There is not much need for a detailed understanding of this process, hence it will not be delved into further in this thesis. This section probably already contains more than anyone would want to know about rippling.[34]

4.5.3. Ordering the Relevant Heuristics

Now that all these relevant heuristics have been assembled, in what order should AM execute them?[35] It is important to note that the heuristics tacked onto very general concepts will be applicable frequently, yet will not be very powerful. For example, here is a typical heuristic rule which is tacked onto the Examples facet of the very general concept Any-concept:

> **If the current task is to fill in examples of any concept X,**
> **Then one way to get them is to symbolically instantiate[36] a definition of X.**

It takes a tremendous amount of inference to squeeze a couple awkward examples of Intersect&Intersect out that concept's definition. Much time could be wasted doing so[37].

---

[33] If x isa y, and y isa z, then x is (generally) NOT a z. This is due to the intransitivity of "member-of". Generalization is transitive, on the other hand, because "subset-of" is transitive.

[34] For the very interested reader, it is explained in great detail in file RIPPLE[dia,dbl] at SAIL. This file has been permanently archived at SAIL.

[35] The discussion below assumes that the heuristics don't interact with each other; i.e., that each one may act independently of all others. The validity of this simplification is tested empirically (see Chapter 6) and discussed theoretically (see Chapter 7) later.

[36] "Symbolic instantiation" is a euphemism for a bag of tricks which transform a declarative definition of a concept into particular entities satisfying that definition. The only constraint on the tricks is that they not actually run the definition. One such trick might be: if the definition is recursive, merely find some entity that satisfies the base step. AM's symbolic instantiation tricks are too hand-crafted to be of great interest, hence this will not be covered any more deeply here. The interested reader is directed to the pioneering work by [Lombardi & Raphael 64], or the more recent literature on these techniques applied to automatic program verification (e.g., [Moore 75]).

[37] Incidentally, this illustrates why no single heuristic should be allowed to monopolize the processing of any one task.

Just as general heuristics are weak but often relevant, specific heuristics are powerful but rarely relevant. Consider this heuristic rule, which is attached to the very specific concept Compose-with-Self:

> **If the current task is to fill in examples of the composition FoF,**
> **Then include any fixed-points of F.**

For example, since Intersect(*phi*,X) equals *phi*, so must IntersectoIntersect(*phi*,X,Y).[38] Assuming that such examples exist already on Intersect, this heuristic will fill in a few examples of IntersectoIntersect with essentially no processing required. Of course the domain of applicability of this heuristic is minuscule.

As we expected, the narrower its domain of applicability, the more powerful and efficient a heuristic is, and the less frequently it's useful. Thus in any given situation, where AM has gathered many heuristic rules, it will probably be best to execute the most specific ones first, and execute the most general ones last.

Below are summarized the three main points that make up AM's scheme for finding relevant heuristics in a "natural" way and then using them:

1. Each heuristic is tacked onto the most general concept for which it applies: it is given as large a domain of applicability as possible. This will maximize its generality, but leave its power untouched. This brings it closer to the "ideal" tradeoff point between these two quantities.

2. When the current task deals with concept C, AM ripples away from C and quickly locates all the concepts of which C is an example. Each of them will contain heuristics relevant to dealing with C.

3. AM then applies those heuristics in order of increasing generality. You may wonder how AM orders the heuristics by generality. It turns out that the rippling process automatically gathers heuristics in order of increasing generality. In the LISP system, each rule is therefore executed as soon as it's found. So AM nevers wastes time gathering heuristics it won't have time to execute.

4.6. AM's Starting Heuristics

This section will briefly characterize the collection of 242 heuristic rules which AM was originally given. A complete listing of those rules is found in Appendix 3; the rule numbers below refer to the numbering given in that appendix.

---

[38] *phi* is another name for the empty set, also written {}. This last sentence thus says that since {} ∩ X = {}, then ({} ∩ X) ∩ Y must also equal {}.

4.6.1. Heuristics Grouped by the Knowledge They Embody

Many heuristics embody the belief that mathematics is an empirical inquiry. That is, one approach to discovery is simply to perform experiments, observe the results, thereby gather statistically significant amounts of data, induce from that data some new conjectures or new concepts worth isolating, and then repeat this whole process again. Some of the rules which capture this spirit are numbers 21, 43-57, 91, 136-139, 146-148, 153-154, 212-216, 225, and 241. As one might expect, most of these are "Suggest" type rules. They indicate plausible moves for AM to make, promising new tasks to try, new concepts worth studying. Almost all the rest are "Fillin" type rules, providing empirical methods to find entries for a specified facet.

Another large set of heuristics is used to embody – or to modify – what can be called "focus of attention". When should AM keep on the same track, and when not? The first rules expressing varying nuances of this idea are numbers 1-5. The last such rules are numbers 209-216. Some of these rules are akin to goal-setting mechanisms (e.g., rule 141). In addition, many of the "Interest" type rules have some relation to keeping AM interested in recently-chosen concepts (or: in concepts related to them, e.g. by Analogy, by Genl/Spec, by Isa/Exs,...).

The remaining "Interest" rules are generally some re-echoing of the following notion: X is interesting if F(X) has an unexpected (interesting) value. For example, in rule 26, "F(X)" is just "Generalizations of X". In slightly more detail, the principle characteristics of interestingness are:

- symmetry (e.g., in an expanding analogy)
- coincidence (e.g., in a concept being re-discovered often)
- appropriateness (e.g., in choosing an operation H so that GoH will have nicer Domain/Range characteristics than G itself did)
- recency (see the previous paragraph on focus of attention)
- individuality (e.g., the first entity observed which satisfies some property)
- usefulness (e.g., there are many conjectures involving it)
- association (i.e., the given concept is related to an interesting one)

One group of heuristic rules embeds syntactic tricks for generalizing definitions (Lisp predicates), specializing them, instantiating them, symbolically evaluating them, inverting them, rudimentarily analyzing them, etc. For example, see rules 31 and 89. Some rules serve other syntactic functions, like ensuring that various limits aren't exceeded (e.g., rule 15), that the format for each facet is adhered to (e.g., rule 16), that the entries on each facet are used as they are meant to be (e.g., rules 9 and 59), etc. Many of the "Check" type heuristics fall into this category.

Finally, AM possesses a mass of miscellaneous rules which evade categorization. See, e.g., rules 185 and 236. These range from genuine math heuristics (rules which lead to discovery frequently) to simple data management hacks.

No detailed analysis has been performed on the set of heuristics AM possesses, as of the time of writing of this thesis.

4.6.2. Heuristics Grouped by How Specific They Are

Another dimension of distribution of heuristics, aside from the above *functional* one, is simply that of how high up in the Genl/Spec tree they are located. The table below summarizes how the rules were distributed in that tree:

LEVEL	e.g.	# Con's	# w/Heur	# Heurs	Avg	Avg w/Heur	# Fillin	# Sugg	# Check	# Int
0	Anything	1	1	10	10.0	10.0	0	5	0	5
1	Any-Concept	1	1	110	110.0	110.0	39	30	20	21
2	Active	2	2	24	12.0	12.0	7	10	4	3
3	Operation	6	3	31	5.2	10.3	11	3	3	14
≥4	Union	100	11	63	0.6	5.7	26	15	8	16

Here is a key to the column headings:

LEVEL: How far down the Genl/Spec tree of concepts we are looking.
e.g.: A sample concept at that level.
- # Con's: The total number of concepts at that level.
- # w/Heur: How many of them have *some* heuristics.
- # Heurs: The total number of heuristics attached to concepts at that level.

Avg: (# Heurs) / (# Concepts); i.e., the mean number of heuristics per concept, at that level.

Avg w/Heur: (# Heurs) / (# w. Heurs)
- # Fillin: Total number of "Fillin" type heuristics at that level.
- # Sugg: Total number of "Suggest" type heuristics at that level.
- # Check: Total number of "Check" type heuristics at that level.
- # Int: Total number of "Interestingness" type heuristics at that level.

The heuristic rules are seen *not* to be distributed uniformly, homogeneously among all the initial concepts. The extent of this skewing was not realized by the author until the above table was constructed. A surprising proportion of rules are attached to the very general concepts. The top 10% of the concepts contain 73% of all the heuristics. One notable exception is the "Interest" type heuristics: they seem more evenly distributed throughout the tree of initial concepts. This tends to suggest that future work on providing "meta-heuristics" should concentrate on how to automatically synthesize those Interest heuristics for newly-created concepts.

Chapter 5. AM's Concepts

This chapter contains material about AM's anatomy. After a brief overview, we'll look in detail at the way concepts are represented (Section 5.2). This includes a discussion of each kind of facet a concept may possess. Wedged in among the implementation details and formats are a horde of tiny ideas; they should be useful to anyone contemplating working on a system similar in design to AM.

The chapter closes by sketching all the knowledge AM starts with. The concepts will be diagrammed, and will also have a brief description, sufficient for the reader to follow later chapters without trouble. Instead of using up a large number of pages for an unreadable listing of all of the specific information initially supplied *re* each concept, such complete coverage is relegated to Appendix 2.1. [1]

The next chapter starts on page 114.[2]

5.1. Motivation and Overview

Each concept consists merely of a bundle of facets. The facets represent the different aspects of each concept, the kinds of questions one might want to ask about the concept:

> How valuable is this concept?
> What is its definition?
> If it's an operation, what is legally in its domain?
> What are some generalizations of this concept?
> How can you separate the interesting instances of this concept from the dull ones?
> etc.

Since each concept is a mathematical entity, the kinds of questions one might ask are fairly constant from concept to concept. This set of questions might change significantly for a new domain of concept.

One "natural" representation for a concept in LISP is therefore as a set of attribute/value

---

[1] That appendix lists each concept, giving a condensed listing of the facts initially given (by the author) to AM about each facet of that concept. This material is translated from LISP into English and standard math notation. The appendix is preceded by an alphabetical index of the concepts and the page number on which they are presented. That index is on page 173. Some unmodified "concepts" -- still in LISP -- are displayed in Appendix 2.3.

[2] Though devoid of theoretical significance, that sentence has also proved of high empirical value.

pairs. That is, each concept is maintained as an atom with a property list. The names of the properties (Worth, Definitions, Domain/Range, Generalizations, Interestingness, etc.) correspond to the questions above, and the value stored under property F of atom C is simply the value of the F-facet of the C-concept. This value can also be viewed as the answer which expert C would give, if asked question F. Or, it can be viewed as the contents of slot F of frame C.

5.1.1. A Glimpse of a Typical Concept

As an example, here is a stylized rendition of the SETS concept. This is a concept which is meant to correspond to the notion of a set of elements. The format P: $v_1, v_2, \ldots$ is used to indicate that the value of property P is the list $v_1, v_2, \ldots$ That is, the concept Sets has entries $v_1, v_2, \ldots$ for its facet P. For example, according to the box below, "Singleton" is one entry on the Specializations facet of Sets.

I shall not digress here to explain each of these entries – and what are apparently omissions. Such things will be done later in this chapter[3]. For now, just glance at it to get the flavor of what a concept is like.

---

[3] The individual facets will be discussed one at a time. This particular concept is shown at an intermediate state of being filled in. Although several facets are blank, many are filled in which were initially empty (e.g., Examples). The reader wishing to see what this concept was like at the time that AM started up should turn ahead to page 211 (inside Appendix 2).

```
Name(s): Set, Class, Collection
Definitions:
        Recursive: λ (S) [S={} or Set.Definition (Remove(Any-member(S),S))]
        Recursive quick: λ (S) [S={} or Set.Definition (CDR(S))]
        Quick: λ (S) [Match S with {...} ]
Specializations: Empty-set, Nonempty-set, Set-of-structures, Singleton
Generalizations: Unordered-Structure, No-multiple-elements-Structure
Examples:
        Typical: {{}}, {A}, {A,B}, {3}
        Barely: {}, {A, B, {C, { { { A, C, (3,3,3,9), <4,1,A,{B},A>}}}}}
        Not-quite: {A,A}, (), {B,A}
        Foible: <4,1,A,1>
Conjec's: All unordered-structures are sets.
Intu's:
        Geometric: Venn diagram. {See [Venn 89], or [Skemp 71].}
Analogs: bag, list, oset
Worth: 600
View:
        Predicate: λ (P) {x∈Domain(P) | P(x)}
        Structure: λ (S) Enclose-in-braces(Sort(Remove-multiple-elements(S)))
Suggest: If P is an interesting predicate over X, consider {x∈X | P(x)}.
In-domain-of: Union, Intersection, Set-difference, Set-equality, Subset, Member
In-range-of: Union, Intersection, Set-difference, Satisfying
```

To decipher the Definitions facet, there are a few things you must know. An expression of the form "(λ (x) E)" is called a Lambda expression after Church[4], and may be considered an executable procedure. it accepts one argument, binds the variable "x" to the value of that argument, and then evaluates "E" (which is probably some expression involving the variable x). For example, "(λ (x) (x+5))" is a function which adds 5 to any number; if given the argument 3, this lambda expression will return the value 8.

The second thing you must know is that facet F of concept C will occasionally be abbreviated as C.F. In those cases where F is "executable", the notation C.F will refer to applying the corresponding function. So the first entry in the Definitions facet is recursive because it contains an embedded call on the function Set.Definition. Notice that we are implying that the *name* of that lambda expression itself is "Set.Definition".

There are some bizarre implications of this: since there are three separate but equivalent definitions, AM may choose whichever one it wants when it recurs. AM can choose one via a random selection scheme, or always try to recur into the same definition as it was just in, or perhaps suit its choice to the form of the argument at the moment.

For example, one definition might be great for arguments of size 10 or less, but slow for bigger ones, and another definition might be mediocre for all size arguments; then AM

[4] Before and during Church, it's called a function. See [Church 41].

should use the mediocre definition over and over again, until the argument becomes small enough, and from then on recur only into the fast definition. Although AM embodies this "smart" scheme, the little comments necessary to see how it does so have be excised from the version shown above in the box. This will be explained later in this chapter, on page 90.

All concepts possess executable definitions, though not necessarily effective ones. They each have a LISP predicate, but that predicate is not guaranteed to terminate. Notice that the definitions for Sets are all definitions of finite sets.[5]

5.1.2. The main constraint: Fixed set of facets

One important constraint on the representation is that the set of facets be fixed for all the concepts. An additional constraint is that this set of facets not grow, that it be fixed once and for all. So there is one fixed, universal list of two dozen types of facets. Any facet of any concept *must* have one of those standard names. All concepts which have some examples must store them as entries on a facet called Examples; they can't call them Instances, or Cases, or G00037's. This constraint is known as the "**Boings** constraint"[6], and has three important consequences:

1. OUTLINE: First, it provides a nice, distributed, universal framework on which to display all that is known about a given concept. For example, when AM creates a new concept like "Square-root", the user can judge how well AM understands that concept by examining Square-root's property-list (the list of entries for each of its facets). Similarly, AM can instantly tell what facets are not yet filled in for any given concept, and this will in turn suggest new tasks to perform. In other words, this constraint helps define the "space" which AM must explore, and makes it obvious what parts of each concept have and have not yet been investigated.
2. STRUCTURE: The constraint specifies that there be a *set* of facets, not just one. This set was made large enough that all the efficiency advantages of a "structured" representation are preserved (unlike totally uniform representations, e.g. pure production systems with simple memories as data structures, or predicate calculus).
3. UNIFORMITY: The most important benefit of the **Boings** constraint arises when AM[7] wants to get a particular question answered – especially if the information pertains to related concepts. The advantage is that it'll have a very limited repertoire of questions it may ask, hence there will be no long searching, no misunderstandings. This is the same advantage that always arises when everyone uses a common language.

We shall illustrate the last two advantages by using the Sets concept pictured in the box a couple pages ago. How does AM handle a task of this form: "**Check examples of Sets**"? AM accesses the examples facet of the Sets concept, and obtains a bunch of items which are all

---

[5] The third definition, "{...}", may not look finite, but consider that ellipsis notation is not permitted within any specific set.

[6] See [Lenat 75b]. Historically, each concept module was called a "BEING".

[7] Actually, the requestor is not "AM" in toto, but rather simply a clause which is a part of a heuristic rule, or a bit of code embedded within an entry on an executable facet, such as Algorithms.

probably sets. If any isn't a set, AM would like to make it one, if that involves nothing difficult. AM locates all the generalizations of Sets[8], and comes up with the list <Sets, Unordered-Structures, No-multiple-elements-Structures, Structures, Objects, Any-concept, Anything>. Next, the "Check" facet of each of these is examined, and all its heuristics are collected. For example, the Check facet of the No-multiple-elements-Structures concept contains the following entry: "Eliminate multiple occurrences of each element" (of course this is present not as an English sentence but rather as a little LISP function). So even though Sets has no entries for its Check facet, several little functions will be gathered up by the rippling process. Each potential set would be subjected to all those checks, and might be modified or discarded as a result.

There is enough "structure" around to keep the heuristic rules relevant to this task isolated from very irrelevant rules, and there is enough "uniformity" to make finding those rules very easy.

The same rippling would be done to find predicates which tell whether a set is interesting or dull. For example, one entry on the Interestingness facet of the Structure concept says that a structure is interesting if all pairs of members satisfy the same rare predicate P(x,y) [for any such P]. So a set, all pairs of whose members satisfy "Equality," would be considered interesting. In fact, every Singleton is an interesting *Structure* for just that reason. A singleton might be an interesting *Anything* because it takes only a few characters to type it out (thereby satisfying a criterion on Anything.Interest).

To locate all the specializations of Sets, the rippling would go in the opposite direction. For example, one of the entries on the Specializations facet of Sets is Set-of-structures; one if *its* Specialization entries is Set-of-sets. So this latter concept will be caught in the net when rippling away from Sets in the Specializations direction.

If AM wants lots of examples of sets, it has only to ripple in the Specializations direction, gathering Examples of each concept it encounters. Examples of Sets-of-sets (like this one: {{A},{C,D}}}) will be caught in this way, as will examples of Sets-of-numbers (like this one: {1,4,5}), because two specializations of Sets are Sets-of-Sets and Sets-of-Numbers[9].

In addition to the three main reasons for keeping the set of facets the same for all the concepts (see previous page), we claimed there were also reasons for keeping that set fixed once and for all. Why not dynamically enlarge it? To add a new facet, its value has to be filled in for lots of concepts. How could AM develop the huge body of heuristics needed to guide such filling-in and checking activities? Also, the number of facets is small to begin with because people don't seem to use more than a few tens of such "properties" in classifying knowledge about a concept[10]. If the viability of AM seemed to depend on this ability, I would have worked on it. AM got along fine without being able to enlarge its set of facets, so no time was ever spent on that problem. I leave it as a challenging, ambitious "open research problem".

---

[8] by "rippling" upward from Sets, in the Gen direction

[9] We are assuming that AM has run for some time, and already discovered Numbers, and already defined Sets-of-Numbers.

[10] This data is gathered from introspection by myself and a few others, and should probably be tested by performing some psychological experiments.

5.1.3. BEINGs Representation of Knowledge

Before discussing each facet in detail, let's interject a brief historic digression, to explain the origins of this modular representation scheme.

The ideas arose in an automatic programming context, while working out a solution to the problem of constructing a computer system capable of synthesizing a simple concept-discrimination program (similar to [Winston 70]). The scenario envisioned was one of mutual cooperation among a group of a hundred or so experts, each a specialist in some minute detail of coding, concept formation, debugging, communicating, etc. Each expert was modelled by one module, one BEING. Each BEING had the same number of slots (parts, facets), and each slot was interpreted as a *question* which that BEING could answer. The community of experts carried on a round-table discussion of a programming task which was specified by a human user. Eventually, by cooperating and answering each other's questions, they hammered out the program he desired. See [Lenat 75b] for details.

The final system, called PUP6, did actually synthesize several large LISP programs, including many variants of the concept-learning program. This is described fully in [Lenat 75a]. Unfortunately, PUP6 had virtually no natural language ability and was therefore unusable by an untrained human. Its modal output was "*EH*?".

The search for a new problem domain where this communication difficulty wouldn't be so severe led to consideration of elementary mathematics.

The other main defect of PUP6 was its narrowness, the small range of 'target' programs which could be synthesized. PUP6 had been designed with just one target in mind, and almost all it could do was to hit that target. The second constraint on the new task domain was then one of having a non-specific target, a very broad or diffuse goal. This pointed to an automated researcher, rather than a problem-solver.

These two constraints then were (i) elementary math, because of communication ease, and (ii) self-guided exploration, because of the danger of too specific a goal. Together, they directed the author to an investigation which ultimately resulted in the AM project.

5.2. Facets

How *is* each concept represented? Without claiming that this is the "best" or preferred scheme, this section will treat in detail AM's representation of this knowledge.

We have seen that the representation of a concept can loosely be described as a collection of facet/value pairs, where the facets are drawn from a fixed set of about 25 total possible facets.

The facets break down into three categories:

1. Facets which relate this concept C to some other one(s): Generalizations, Specializations, Examples, Isa's, In-domain-of, In-range-of, Views, Intu's, Analogies, Conjec's
2. Facets which contain information intensive to this concept C: Definitions, Algorithms, Domain/Range, Worth, Interest

3. Sub-facets, containing heuristics, which can be tacked onto facets from either group above. These include: Suggest, Fillin, Check

Some facets come in several flavors (e.g., there are really four separate facets -- not just one -- which point to Examples: boundary, typical, just-barely-failing, foibles).

This section will cover each facet in turn. Let's begin by listing each of them. For a change of pace, we'll show a typical question that each one might answer about concept C:[11]

Name: What shall we call C when communicating with the user?
Generalizations: Which other concepts have less restrictive definitions than C?
Specializations: Which concepts satisfy C's definition plus some additional constraints?
Examples: What are some things that satisfy C's definition?
Isa's: Which concepts' definitions does C itself satisfy?[12]
In-domain-of: Which operations can be performed on C's?
In-range-of: Which operations result in values which are C's?
Views: How can we view some other kind of entity as if it were a C?
Intu's: What is an abstract, analogic representation for C?
Analogies: Are there similar (though formally unrelated) concepts?
Conjec's: What are some potential theorems involving C?
Definitions: How can we tell if x is an example of C?
Algorithms: How can we execute the operation C on a given argument?
Domain/Range: What kinds of arguments can operation C be executed on? What kinds of values will it return?
Worth: How valuable is C? (overall, aesthetic, utility, etc.)
Interestingness: What special features make a C especially interesting?

In addition, each facet F of concept C can possess a few little subfacets which contain heuristics for dealing with that facet of C's:

F.Fillin: How can entries on C.F be filled in? These heuristics get called on when the current task is "Fillin facet F of concept X", where X is a C.
F.Check: How can potential entries on C.F be checked and patched up?
F.Suggest: If AM gets bogged down, what are some new tasks (related to C.F) it might consider?

We'll now begin delving into the syntax and semantics of each facet, one by one. Future chapters will not depend on this material. The reader may wish to skip to Section 5.3 (page 105).

5.2.1. Generalizations/Specializations

---

[11] In this discussion, "C" represents the name of the concept whose facet is being discussed, and may be read "the given concept".

[12] Notice that C will therefore be an example of each member of Isa's(C).

Generalization makes possible conscious, controlled, and accurate accomodation of one's existing schemas, not only in response to the demands for assimilation of new situations as they are encountered, but ahead of these demands, seeking or creating new examples to fit the enlarged concept.

-- Skemp

We say concept A "*is a generalization of*" concept B iff every example of B is an example of A. Equivalently, this is true iff the definition of B can be phrased as "λ (x) [A.Defn(x) and P(x)]"; that is, for x to satisfy B's definition, it must satisfy A's definition plus some additional predicate P. The Generalizations facet of concept C will be abbreviated as C.Genl.

C.Genl does not contain *all* generalizations of C; rather, just the "immediate" ones. More formally, if A is a generalization of B, and B of C, then C.Genl will *not* contain a pointer to A. Instead, C will point to B[13].

Here are the recursive equations which permit a search process to quickly find all generalizations or specializations of a given concept X:

Generalizations(X) = Genl*(X) = {X} U Generalizations(X.Genl)
Specializations(X) = Spec*(X) = {X} U Specializations(X.Spec)

For the reader's convenience, here are the similar equations, presented elsewhere in the text, for finding all examples of – and Isa's of – X:

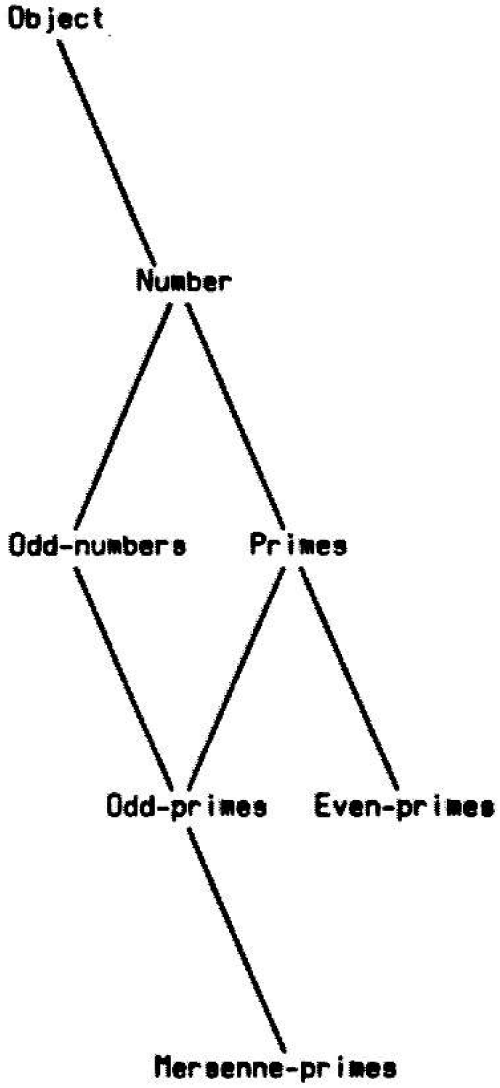
Examples(X) = Spec*(Exs(Spec*(X)))
Isa's(X) = Genl*(Isa(Genl*(X)))

The format of the Generalizations facet is quite simple: it is a list of concept names. The Generalizations facet for Odd-primes might be:

(Odd-numbers Primes)

[13] In general, C.Genl will contain an entry X1; X1.Genl will contain an entry X2;...; Xn.Genl will contain B as one entry; B.Genl will contain Y1;...; Yn.Genl will contain A.

Here is a small diagram representing generalization relationships. The only lines drawn represent the pointers found in the Genl facets of these concepts:



Each of those lines represents an arrow which slants upwards, indicating a Genl link. For example, we see that the Generalizations facet of Odd-primes contains pointers to both Odd-numbers and to Primes. There is no pointer from Odd-primes upward to Number, because there is an "intermediate" concept (namely, Primes). There is no pointer from Mersenne-primes to Object, since a chain of intermediate concepts links them.

The reason for these strange constraints is so that the total number of links can be minimized. There is no harm if a few redundant ones sneak in. In fact, frequently-used paths are granted the status of single links, as we shall soon see.

We've been talking about both Specializations and Generalizations as if they were very similar to each other. It's time to make that more explicit:

Specializations are the converse of Generalizations. The format is the same, and (hopefully) A is an entry on B's Specializations facet iff B is an entry on A's Generalizations facet.

The uses of these two facets are many:

1. AM can sometimes establish independently that A is both a generalization and a specialization of B; in that case, AM would like to recognize that fact easily, so it can conjecture that A and B specify equivalent concepts. Such coincidences are easily detected as *cycles* in the Genl (or Spec) graph. In these cases, AM may physically merge A and B (and all the other concepts in the cycle) into one concept.
2. Sometimes, AM wants to assemble a list of all specializations (or generalizations) of X, so that it can test whether some statement which is just barely true (or false) for X will hold for any of those specializations of X.
3. Sometimes, the list of generalizations is used to assemble a list of isa's; the list of specializations helps assemble a list of examples.[14]
4. A common and crucial use of the list of generalizations is to locate all the heuristic rules which are relevant to a given concept. Typically, the relevant rules are those tacked onto Isa's of that concept, and the list of Isa's is built up from the list of generalizations of that concept. This was also mentioned on page 56.
5. To incorporate new knowledge. If AM learns, conjectures, etc. that A is a specialization of B, then all the machinery (all the theorems, algorithms, etc.) for B become available for working with A.

Here is a little trick that deserves a couple paragraphs of its own. AM stores the answers to common questions (like "What are *all* the specializations of Operation") explicitly, by intentionally permitting redundant links to be maintained. If two requests arrive closely in time, to test whether A is a generalization of B, then the result is stored by adding "A" as an entry on the Generalizations facet of B, and adding "B" as a new entry on the Specializations facet of A. The slight extra space is more than recompensed in cpu time saved.

If the result were False (A turned out not to be a generalization of B) then the links would specify that finding explicitly, so that the next request would not generate a long search again. Such failures are recorded on two additional facets: Genl-not and Spec-not. Since most concept pairs A/B are related by Spec-not and by Genl-not, the only entries which get recorded here are the ones which were frequently called for by AM. If space ever gets tight, all such facets can be wiped clean with no permanent damage done.

These two "shadow" facets (Genl-not and Spec-not) are not useful or interesting in their own right. If AM ever wished to know all the concepts which are *not* generalizations of C, the fastest way would be to take the set-difference of all concepts and Generalizations(C). Since they are quite incomplete, Genl-not and Spec-not are used more like a cache memory: they save time whenever they are applicable, and don't really cost much when they aren't applicable. Because of their superfluity, these two facets will not be mentioned again. I only mentioned them above because they do greatly speed up AM's execution time, and because they may have some psychological analog.

---

[14] This process was called RIPPLING, and was described in Chapter 4. See also footnote 34 in that chapter.

5.2.2. Examples/Isa's

> *Usually, to show that a definition implies no contradiction, we proceed by example, we try to make an example of a thing satisfying the definition. We wish to define a notion A, and we say that, by definition, an A is anything for which certain postulates are true. If we can demonstrate directly that all these postulates are true of a certain object B, the definition will be justified; the object B will be an example of an A.*
>
> -- Poincare'

Following Poincare', we say "*concept A is an example of concept B*" iff A satisfies B's definition.[15] Equivalently, we say that "*A isa B*". It would be legal (in that situation) for "A" to be an entry on B.Exs (the Examples facet of concept B) and for "B" to be an entry on A.Isa (the Isa's facet of concept A). Some earlier mention of the Examples and Isa's facets can be seen in Chapter 4, page 57.

The Examples facet of C does not contain *all* examples of C; rather, just the "immediate" ones. The examples facet of Numbers will not contain "11" since it is contained in the examples facet of Odd-primes. A "rippling" procedure is used to acquire a list of all examples of a given concept. The basic equation is:

Examples(x) = Specializations(Exs(Specializations(x)))

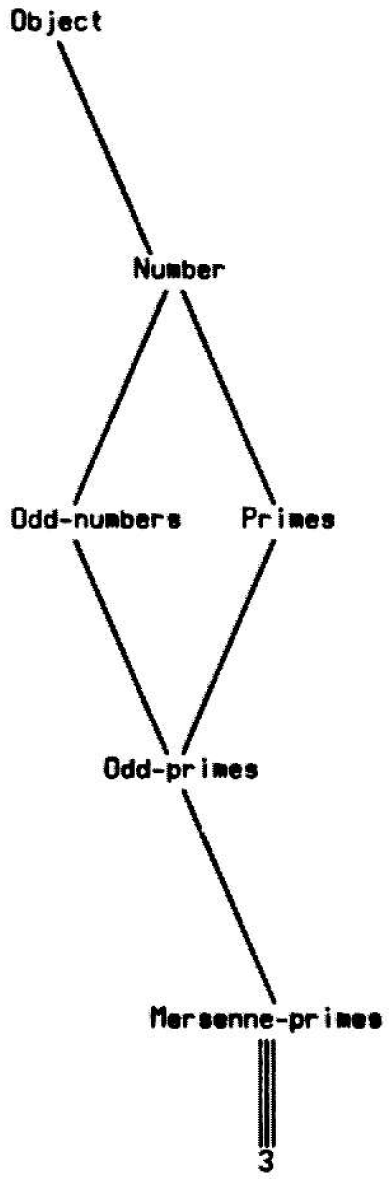
where Exs(x) is the contents of the examples facet of x. Examples(x) represents the final list of all known items which satisfy the definition of X. Examples(x) thus must include Exs(x). Specializations(x) might be more regularly written $Spec^0(x)$. That is, all members of x.Spec, all members of *their* Spec facet, etc. Note the similarity of this to the formula for Isa's(x), given on page 57. We could also write the above equation as follows:

Examples(x) = $Spec^*$(Exs($Spec^*$(x)))

As an illustration, we shall show how AM would recognize that "3" is an example of Object:

---

[15] What does this mean? B.Defn is a Lisp predicate, a Lambda expression. If it is fed A as its argument, and it returns True, we say that A is a B, or that A satisfies B's definition. If B.Defn returns NIL, we say that A is not a B, or that A fails B's definition. If B.Defn runs out of time before returning a T/NIL value, there is no definite statement of this form we can make. In that case, AM might check to see whether A satisfies the definition of some specialization of B, or whether A fails the definition of some generalization of B.



As the graph above shows, AM would ripple in the Spec direction 4 times, moving from Object all the way to Mersenne-primes; then descend once in the Exs direction, to reach "3"; then ripple 0 more times in the Spec direction. Thus "3" is seen to be an example of Object, according to the above formula. Similarly, we see that "3" is also an example of Number, of Primes, of Odd-number, of Odd-primes, and of course an example of Mersenne-primes.

As with Generalizations/Specializations, the reasons behind the incomplete pointer structure is simply to save space, and to minimize the difficulty of updating the graph structure

whenever new links are found. Suppose a new Mersenne prime[16] is computed. Wouldn't it be nice simply to add a single entry to the Exs facet of Mersenne-primes, rather than to have to update the Exs pointers from a dozen concepts?

There is no harm if a few redundant links sneak in. In fact, frequently-used paths are granted the status of single links. If two requests arrive closely in time, to test whether A isa B, then the result is stored as an entry on the Isa facet of A, and the Exs facet of B. If the result were False, then the links would specify that, so that the next request would not generate a long search. In fact, there is a separate facet called Exs-not, and one called Isa-not. These two shadowy facets are quite analogous to the unmentionable facets "Genl-not" and "Spec-not", discussed in the previous subsection.

"Isa's" is the converse of "Examples". The format is the same, and (if A and B are both concepts) A is an entry on B.Isa iff B is an entry on A.Exs. In other words, A is a member of Examples(B) iff B is a member of Isa's(A). Due to an ugly lack of standardization, non-concepts are allowed to exist. Thus, "3" is an example of Primes, but is not itself a concept. Examples of X sometimes are concepts, of course: "IntersectoIntersect" is an example of Compose-with-self. And Isa's(x) are always concepts. The highest level concept is called "Anything". Its definition is the atom T. That is, "λ(x) T". This high-level concept can claim everything as its examples.

The *uses* of the Exs/Isa's facets are similar to those for Genl/Spec (see previous subsection).

Their formats are quite a bit more complicated than the Genl/Spec facets' formats, when we finally get to the implementation level, however. There are really a cluster of different facets all related to Examples:

1. TYPICAL: This is a list of average examples. Care must be taken to include a wide spectrum of allowable kinds of examples. For "Sets", these would include sets of varying size, nesting, complexity, type of elements, etc.
2. BOUNDARY: Items which just barely pass the definition of this concept. This might include items which satisfy the base step of a recursive definition, or items which were intuitively believed to be *non*-examples of the concept. For "Sets", this might include the empty set.
3. BOUNDARY-NOT: Items which just barely fail the definition. This might include an item which had to be slightly modified during checking, like {A,B,A} becoming {A,B}.
4. FOIBLES: Total failures. Items which are completely against the grain of this concept. For "Sets", this might include the operation "Compose".
5. NOT: This is the "cache" trick used to store the answers to frequently-asked questions. If AM frequently wants to know whether X is an example of Y, and the answer is *No*, then much time can be saved by adding X as an entry to the Exs-not facet of Y.

An individual item on these facets may just be a concept name, or it may be more complicated. In the case of an operation, it is an item of the form $\langle a_1 a_2 \ldots \rightarrow v \rangle$; i.e., actual

---

[16] "Mersenne prime", without a hyphen, refers to a number satisfying certain properties [see glossary]. "Mersenne-primes", with a hyphen, refers to one specific AM concept, a data structure with facets. Each Mersenne prime is an example of the concept Mersenne-primes.

arguments and the value returned. In the case of objects, it is an object of that form. An Exs facet of the concept Sets might contain {a} as one entry.

Here is a more detailed illustration. Consider the Examples facet of Set-union. It might appear thus:

TYPICAL: {A}U{A,B}→{A,B};
{A,B}U{A,B}→{A,B};
{A,<3,4,3>,{A,B}}U{3,A}→{A,<3,4,3>,{A,B},3}.
BOUNDARY: {}UX→X [17]
BOUNDARY-NOT: {A,B}U{A,C}→{A,B,A,C};
{A,B,C,D}U{E,F,G,H,I,J}→{A,B,C,E,F,G,H,I,J}
FOIBLES: <2,A,2>
NOT: no entries

The format for Isa's are much simpler: there are only two kinds of links, and they're each merely a list of concept names. Here is the Isa facet of Set-union:

ISA: (Operation[18] Domain=Range-op)
ISA-NOT: (Structure Composition Predicate)

At some time, some rule asked whether Set-union isa Composition. As a result, the negative response was recorded by adding "Composition" to the Isa-not facet of Set-union, and adding "Set-union" to the Exs-not subfacet of the Examples facet of the concept Composition (indicating that Set-union was definitely not an example of Composition, yet there was no reason to consider it a foible).

5.2.3. In-Domain-of/In-Range-of

We shall say that A is in the domain of B (written "A In-dom-of B") iff

1. A and B are concepts
2. B isa Operation
3. A is equal to (or at least a specialization of) one of the domain components of the operation B. That is, B can be executed using any example of A as one of its arguments.[19]

For example, Odd-perfect-squares is In-dom-of Add, since Odd-perfect-squares is a specialization of Numbers, and Numbers is one component of the following entry which is

---

[17] Actually, AM is not quite smart enough to use the variable X as shown in the boundary examples. It would simply store a few instances of this general rule, plus have an entry of the form <Equivalent: Identity(X) and Set-union(X,{})> on the Exs facet of Conjectures. Notice that because of the asymmetric way Set-union was defined, only one lopsided boundary example was found. If another definition were supplied, the converse kind of boundary examples would be found.

[18] This entry is redundant.

[19] More formally, we can say that this occurs whenever some entry on the Domain/range facet of B has the form <D_1 D_2 ... D_i → R> with some D_j a member of Generalizations(A). Then A is a specialization of some domain component of some entry on B.Domain/range.

located on Add.Domain/range: <Numbers Numbers → Numbers>. Since Odd-perfect-squares is a specialization of Numbers, the operation 'Add' can be executed using any example of Odd-perfect-squares as its argument.

As another example, Odd-perfect-squares is also In-dom-of Set-insert, one of whose Domain/range entries is <Anything Sets → Sets>. This is because Odd-perfect-squares is a specialization of Anything. So Set-insert is executed on two arguments, and the first argument can be any example of Odd-perfect-squares (the second argument must be an example of Sets).[20]

Although it can be recomputed very easily, we may wish to record the fact that A In-dom-of B by adding the entry "B" to the In-dom-of facet of A. AM may even wish to add this new entry to the Domain/range facet of B (where A is a specialization of the jth domain component of B):
$< D_1\ D_2 \ldots D_{j-1}\ A\ D_{j+1} \ldots D_i \rightarrow R>$. The two examples given above would produce new domain/range entries of <Odd-perfect-squares Numbers → Numbers> for Add, and <Odd-perfect-squares Sets → Sets> for Set-insert.

The semantic content of "In-dom-of" is: what can be done to any example of a given concept C? Given an example of concept C, what operations can be run on that thing? Here are some illustrations:

"Odd-perfect-squares In-dom-of Set-insert" tells us that Set-insert can be run on any particular Odd-perfect-square we can grab hold of.

"Operation In-dom-of Compose" tells us that Compose can be run on any operation we want.

"Dom=Range-operation In-dom-of Compose" tells us that Compose can be run on any operation which has its range equal to one of its domain components.

"Primes In-dom-of Squaring" tells us that we can apply the operation Squaring to any particular prime number we wish.

Let us now turn from In-dom-of to the related facet In-ran-of.

We say that concept A is in the range of B iff B is an Activity[21] and A is a specialization of the range of B. More precisely, we can say that "A In-ran-of B" iff

1. A and B are concepts
2. B isa Operation (i.e., B is an example of the concept "Operation")
3. Some entry on the Domain/range facet of B has the form $<D_1\ D_2 \ldots D_i \rightarrow R>$ with R a generalization of A.

For example, Odd-perfect-squares is In-ran-of Squaring, since (1) both of those are concepts, (2) Squaring is an operation, (3) one of its Domain/range entries is <Numbers→Perf-

---

[20] Since Odd-perfect-squares is more closely related to Numbers than to the concept Anything (half as many Genl links away), AM expects that restricting Add to Odd-perfect-squares will probably yield a more promising new operation than restricting Set-insert to only insert odd perfect squares into sets.

[21] I.e., iff B isa Active, iff B∈Examples(Active), iff Active.Defn(B)=True. Actually, since the range of Predicates is merely {T,F}, we may as well assume that B is an operation, not a predicate. This is in fact assumed, in the text and in the actual AM system.

squares>, and Perf-squares is a generalization of Odd-perfect-squares[22].

Here is what the In-ran-of facet of Odd-perfect-squares might look like:

(Squaring Add TIMES Maximum Minimum Cubing)

Each of these operations will – at least sometimes – produce an odd perfect square as its result.

Semantically, the In-ran-of relation between A and B means that one might be able to produce examples of A by running operation B. Aha! This is a potential mechanism for finding examples of a concept A. All you need do is get hold of In-ran-of(A), and run each of those operations. Even more expeditious is to examine the Examples facets of each of those operations, for already-run examples whose values should be tested using A.Defn, to see if they are examples of A's. AM relies on this in times of high motivation; it is too "blind" a method to use heavily all the time.

This facet is also useful for generating situations to investigate. Suppose that the Domain/range facet of Doubling contains only one entry: < Numbers → Numbers >. Then syntactically, Odd-numbers is in the range of Doubling. Eventually a heuristic rule may have AM spend some time looking for an example of Doubling, where the result was an odd number. If none is quickly found, AM conjectures that it *never* will be found. Since one definition of Odd-number(x) is "Number(x) and Not(Even-number(x))", the only non-odd numbers are even numbers. So AM will increment the Domain/range facet of Doubling with the entry <Numbers→Even-numbers>, and remove the old entry. Thus Odd-numbers will no longer be In-dom-of Doubling. AM can of course chance upon this conjecture in a more positive way, by noticing that all known examples of Doubling have results which are examples of Even-numbers.[23].

A more productive result is suggested by examining the cases where Odd-perfect-squares are the result of cubing. The smallest such odd numbers are 1, 729, and 15625. In general, these numbers are all those of the form $(2n+1)^6$. How could AM notice such an awkward relationship?

The general question to ask, when A In-ran-of B, is "What is the set of domain items whose values (under the operation B) are A's?" In case the answer is "All" or "None", some special modifications can be made to the Domain/range facets and In-dom-of, In-ran-of facets of various concepts, and a new conjecture can be printed. In other cases, a new concept might get created, representing precisely the set of all arguments to B which yield values in A. If you will, this is the inverse image of A, under operation B. In the case of B a predicate, this might be the set of all arguments which satisfy the predicate.

---

22 Why? Because Generalizations(Odd-perfect-squares) is the set of concepts {Odd-numbers Perf-squares Numbers Objects Any-concept Anything}, hence contains Perf-squares. So Perf-squares is a generalization of Odd-perfect-squares.

23 This positive approach is in fact the way AM noticed this particular relationship.

6 Wrong. That was an exponent, not a footnote!

In the case of B=Cubing and A=Odd-perfect-squares, the heuristic mentioned above will have AM create a new concept: the inverse image of Odd-perfect-squares under the operation of Cubing. That is, find numbers whose cubes are Odd-perfect-squares. It is quickly noticed that such numbers are precisely the set of Odd-perfect-squares themselves! So The Domain/range facet of Cubing might get this new entry: <Odd-perfect-squares → Odd-perfect-squares>. But not all squares can be reached by cubing, only a few of them can. AM will notice this, and the new range would then be isolated and might be renamed by the user "Perfect-sixth-powers". Note that all this was brought on by examining the In-ran-of facet of Odd-perfect-squares. "Cubing" was just one of the seven entries there. There are six more stories to tell in this tiny nook of AM's activities.

How exactly does AM go about gathering the In-ran-of and In-dom-of lists? Given a concept C, AM can scan down the global tree of operations (the Exs and Spec links below the concept 'Active'). For if C is not In-dom-of F, it certainly won't be In-dom-of any specialization of F. Similarly, if it can't be produced by F, it won't be produced by any specialization of F. If you can't get x using Doubling you'll never get it by Quadrupling. So AM simply ripples around, as usual. The precise code for this algorithm is of little interest. There are not that many operations, and it is cheap to tell whether X is a specialization of a given concept, so even an exhaustive search wouldn't be prohibitive. Finally, recall that such a search is not done all the time. It will be done initially, perhaps, but after that the In-dom-of and In-ran-of networks will only need slight updating now and then.

5.2.4. Views

Often, two concepts A and B will be inequivalent, yet there will be a "natural" bijection between one and (a subset of) the other. For example, consider a finite set S of atoms, and consider the set of all its subsets, 2^S, also called the *power set* of S. Now S is a member of, but not a *subset* of, 2^S (e.g., if S={x,y,...}, then x is not a member of 2^S). On the other hand, we can identify or view S as a subset by the mapping v→{v}. Then S is associated with the following subset of 2^S: { {x}, {y},... }. Why would we want to do this? Well, it shows that S is identified with a *proper* subset of 2^S, and indicates that S has a lower cardinality (remember: all sets are finite).

As another example, most of us would agree that the set {x, {y}, z} can be associated with the following bag: (x, {y}, z). Each of them can be viewed as the other. Sometimes such a viewing is not perfectly natural, or isn't really a bijection: how could the bag (2, 2, 3) be viewed as a set? Is {2,3} better or worse than {2,{2},3}?

The View facet of a concept C describes how to view instances of another concept D as if they were C's. For example, this entry on the View facet of Sets explains how to view any given structure as if it were a Set:

Structure: λ (x) Enclose-in-braces(Sort(Remove-multiple-elements(x)))

If given the list <z,a,c,a>, this little program would remove multiple elements (leaving <z,a,c>), sort the structure (making it <a,c,z>), and replace the "<...>" by "{...}", leaving the final value as {a,c,z}. Note that this transformation is not 1-1; the list <a,c,z> would get transformed into this same set. On the other hand, it may be more useful than

transforming the original list into {z,{a,{c,{a}}}} which retains the ordering and multiple element information. Both of those transformations may be present as entries on the View facet of Sets.

As it turns out, the View facet of Sets actually contains only the following information:

Structure: λ (x) Enclose-in-braces(x)

Thus the Viewing will produce entities which are not quite sets. Eventually, AM will get around to executing a task of the form "Check Examples of Sets", and at that time the error will be corrected. One generalization of Sets is No-multiple-elements-Structures, and one of its entries under Examples.Check says to remove all multiple elements. Similarly, Unordered-structures is a generalization of Sets, and one of its Examples.Check subfacet entries says to sort the structure. If either of these alters the structure, the old structure is added to the Boundary-not subfacet (the 'Just-barely-miss' kind) of Examples facet of Sets.

The syntax of the View facet of a concept C is a list of entries; each entry specifies the name of a concept, X, and a little program P. If it is desired to view an instance of X as if it were a C, then program P is run on that X; the result is (hopefully) a C. The programs P are opaque to AM; they must have no side effects and be quick.

Here is an entry on the View facet of Singleton:

Anything: λ (x) Set-insert(x, PHI)

In other words, to view anything as a singleton set, just insert it into the empty set. Note that this is also one way to view anything as a set. As you've no doubt guessed, there is a general formula explaining this:

Views(X) = View(Specializations(X))

Thus, to find all the ways of viewing something as a C, AM ripples away from C in the Spec direction, gathering all the View facets along the way. All of their entries are valid entries for C.View as well.

In addition to these built-in ways of using the Views facets, some special uses are made in individual heuristic rules. Here is a heuristic rule which employs the Viewing facets of relevant concepts in order to find some examples of a given concept C:

IF the current task is to Fill-in Examples of C,
and C has some entries on its View facet,
and one of those entries <X,P> indicates a concept X which has some known Examples,
THEN run the associated program P on each member of Examples(X),
and add the following task to the agenda: "Check Examples of C", for the following reason: "Some very risky techniques were used to find examples of C", and that reason's rating is computed as: Average(Worth(X), ||the examples of C found in this manner||).

Say the task selected from the agenda was "Fill-in Examples of Sets". We saw that one entry on Sets.View was Structure: λ(x) Enclose-in-braces(x). Thus it is of the form <X,P>,

with X=Structure. The above heuristic rule will trigger if any examples of Structures are known. The rule will then use the View facet of Sets to find some examples of Sets. So AM will go off, gathering all the examples of structures. Since Lists is a Specialization of Structure, the computation of Examples(Structures) will eventually ripple downwards and ask for Examples of Lists. If the Examples facet of Lists contains the entry <z,a,c,a,a>, then this will be retrieved as one of the members of Examples(Structure). The heuristic rule takes each such member in turn, and feeds it to Set.View's little program P. In this case, the program replaces the list brackets with set braces, thus converting <z,a,c,a,a> to {z,a,c,a,a}.

In this manner, all the existing structures will be converted into sets, to provide examples of sets. After all such conversions take place, a great number of potential examples of Sets will exist. The final action of the right side of the above heuristic rule is to add the new task "Check examples of Sets" to the agenda. When this gets selected, all the "slightly wrong" examples will be fixed up. For example, {z,a,c,a,a} will be converted to {a,c,z}.

If any reliance is made on those unchecked examples, there is the danger of incorrectly rejecting a valid conjecture. This is not too serious, since the very first such reliance will boost the priority of the task "Check examples of Sets", and it would then probably be the very next task chosen.

5.2.5. Intuitions

> *The mathematician does not work like a machine; we cannot overemphasize the fundamental role played in his research by a special intuition (frequently wrong), which is not common-sense, but rather a divination of the regular behavior he expects of mathematical beings.*
>
> -- Bourbaki

This facet turned out to be a "dud", and was later excised from all the concepts. It will be described below anyway, for the benefit of future researchers. Feel free to skip directly to the next subsection.

The initial idea was to have a set of a few (3-10) large, global, opaque LISP functions. Each of these functions would be termed an "Intuition" and would have some suggestive name like "jigsaw-puzzle", "see-saw", "archery", etc. Each function would somehow model the particular activity implied by its name. There would be a multitude of parameters which could be specified by the "caller" as if they were the arguments of the function. The function would then work to fill in values for any unspecified parameters. That's all the function does. The caller would also have to specify which parameters were to be considered as the "results" of the function.

For the see-saw, the caller might provide the weight of the left-hand-side sitter, and the final position of the see-saw, and ask for the weight of the right-hand sitter. The function would then compute that weight (as any random number greater/less-than the left-hand weight,

depending on the desired tilt of the board). Or, the caller might specify the two weights and ask for the final position.

The See-saw function is an expert on this subject; it has efficient code for computing any values which can be computed, and for randomly instantiating any variables which may take on any value (e.g., the first names of the people doing the sitting). When an individual call is made on this function, the caller is not told how the final values of the variables were computed, only what those values end up as.

So the Intuitions were to be experimental laboratories for AM, wherein it could get some (simulated) real-world empirical data. If the seesaw were the Intuition for ">", and weight corresponded to Numbers, then several relationships might be visualized intuitively (like the anti-symmetry of ">"). This is a nice idea, but in practice the only relationships derived in this way were the ones that were thought up while trying to encode the Intuition functions. This shameful behavior led to the excision of the Intuitions facets completely from the system.

As another example, suppose AM is considering composing two relations R and S. If they have no common Intuition reference, then perhaps they're not meaningfully composable. If they do both tie into the same Intuition function, then perhaps that function can tell us something about the composition. This is a nice idea, but in practice very few prunings were accomplished this way, and no unanticipated combinations were fused.

Each Intuition entry is like a "way in" to one of the few global scenarios. It can be characterized as follows:

1. One of the salient features of these entries -- and of the scenarios -- is that AM is absolutely forbidden to look inside them, to try to analyze them. They are *opaque*. Most Intuition functions use numbers and arithmetic, and it would be pointless to say that AM discovered such concepts if it had access to those algorithms all along.
2. The second characteristic of an Intuition is that it be *fallible*. As with human intuition, there is no guarantee that what is suggested will be verified even empirically, let alone formally. Not only does this make the programming of Intuition functions easier, it was meant to provide a degree of "fairness" to them. AM wasn't cheating quite as much if the See-saw function was only antisymmetric 90% of the time.
3. Nevertheless, the intuitions are very *suggestive*. Many conjectures can be proposed only via them. Some analogies (see the next subsection) can also be suggested via common intuitions.

After they were coded and running, I decided that the Intuition functions were unfair; they contained some major discoveries "built-in" to them. They had the power to propose otherwise-obscure new concepts and potential relationships. They contributed nothing other than what was originally programmed into them; *they were not synergetic*. Due to this dubious character of the contributions by AM's few Intuition functions, they were removed from the system. All the examples and all the discoveries listed in this document were made without their assistance.

We shall now drop this de-implemented idea. I think there is some real opportunity for research here. For the benefit of any future researchers in this area, let me point to the excellent discussion of analogic representations in [Sloman 71].

5.2.6. Analogies

The whole idea of analogy is that 'Effects', viewed as a function of situation, is a continuous function.

-- Poincare'

As with Views and Intuitions, this facet is useful for shifting between one part of the universe and another. Views dealt with transformations between two specific concepts; Intuitions dealt with transformations between a bunch of concepts and a large standard scenario which was carefully hand-crafted in advance. In contrast, this facet deals with transforming between a list of concepts and another list of concepts.

Analogies operate on a much grander scale than Views. Rather than simply transforming a few isolated items, they initiate the creation of many new concepts. Unlike Intuitions, they are not limited in scope beforehand, nor are they opaque. They are dynamically proposed.

The concept of "prime numbers" is *analogous* to the notion of "simple groups". While not isomorphic, you might guess at a few relationships involving simple groups just by my telling you this fact: simple groups are to groups what primes are to numbers.[24]

Let's take 3 elementary examples, involving very fundamental concepts.

1. AM was told how to View a set as if it were a bag.

2. AM was told it could Intuit the relation "|" as the predetermined "See-saw" function.

3. AM, by itself, once Analogized that these two lists correspond:

<Bags	Same-length	Operations-on-and-into Bags>
<Bags-of-T's	Equality	Those operations restricted to Bags-of-T's>

The concept of a bag, all of whose elements are "T"'s, is the unary representation of *numbers* discovered by AM. When the above analogy (#3) is first proposed, there are many known Bag-operations[25], but there are as yet no numeric operations[26]. This triggers one of AM's heuristic rules, which spurs AM on to finding the analogues of specific Bag-

---

[24] If a group is not simple, it can be factored. Unfortunately, the factorization of a group into simple groups is not unique. Another analogizing contact: For each prime p, we can associate the cyclic group of order p, which is of course simple. AM never came up with the concept of simple groups; this is just an illustration for the sophisticated reader.

[25] i.e., all entries on In-dom-of(Bag) and In-ran-of(Bag); a few of these are: Bag-insert, Bag-union, Bag-intersection

[26] Examples of Operation whose domain/range contains "Number".

operations. That is, what special properties do the bag-operations have when their domains and/or ranges are restricted from Bags to Bags-of-T's (i.e, Numbers). In this way, in fact, AM discovers Addition (by restricting Bag-union to the Domain/range <Bags-of-T's Bags-of-T's → Bags-of-T's>), plus many other nice arithmetic functions.

Well, if it leads to the discovery of Addition, that analogy is certainly worth having. How would an analogy like that be proposed? As the reader might expect by now, the mechanism is simply some heuristic rule adding it as an entry to the Analogies facet of a certain concept. For example:

> **IF the current task has just created a canonical specialization C2 of concept C1, with respect to operations F1 and F2, [i.e., two members of C2 satisfy F1 iff they satisfy F2],**
> **THEN add the following entry to the Analogies facet of C2:**
> **<C1 F1 Operations-on-and-into(C1)>**
> **<C2 F2 Those operations restricted to C2's>**

After generalizing "Equality" into the operation "Same-length", AM seeks to find a canonical[27] representation for Bags. That is, AM seeks a canonizing function f, such that (for any two bags x,y)

> **Same-length(x,y) iff Equal(f(x), f(y)).**

Then the range of f would delineate the set of "canonical" Bags. AM finds such an f and such a set of canonical bags: the operation f involves replacing each element of a bag by "T", and the canonical bags are those whose elements are all T's. In this case, the above rule triggers, with C1=Bags, C2=Bags-of-T's, F1=Same-length, F2=Equality, and the analogy which is produced is the one shown as example #3 above.

The Analogy facets are not implemented in full generality in the existing LISP version of AM, and for that reason I shall refrain from delving deeper into their format. Since good research has already been done on reasoning by analogy[28], I did not view it as a central feature of my work. Very little space will be devoted to it in this document.

An important type of analogy which was untapped by AM was that between heuristics. If two situations were similar, then conceivably the heuristics useful in one situation might be useful (or have useful analogues) in the new situation. Perhaps this is a viable way of enlarging the known heuristics. Such "meta-level" activities were kept to a minimum throughout AM, and this proved to be a serious limitation.

Let me stress that the failure of the Intuitions facets to be nontrivial was due to the lack of spontaneity which they possessed. Analogies facets were useful and "fair" since their uses were not predetermined by the author.

---

[27] A natural, standard form. All bags differing in only "unimportant" ways should be transformed into the same canonical form. Two bags B1 and B2 which have the same length should get transformed into the same canonical bag.

[28] An excellent discussion of reasoning by analogy is found in [Polya 54]. Some early work on emulating this was reported in [Evans 68]; a more recent thesis on this topic is [Kling 71].

5.2.7. Conjec's

Basically, facet Conjec of concept C is a list of relationships which involve C. We shall discuss its semantics (uses of this facet) before its syntax.

Perhaps the most obvious use for this facet would be to hold conjectures which could not be phrased simply. Yet it turns out that luckily (I think), all the conjectures "fell out" naturally as trivial relationships, e.g. simply as arcs in the Genl/Spec/Exs/Isas pointer format. Specifically, the modal conjecture had the form "the range of F is not just C, but actually S".

For example, AM restricted TIMES to perfect squares, and noted that the result was not merely a number but a perfect square each time. The unique factorization theorem was noticed similarly (the range of Prime-factorings was always a singleton, not merely a set).

In all the cases encountered by AM, there was never any real need for a place to "park" an awkwardly-phrased conjecture, because *no awkward conjecture could ever possibly be noticed.* Why is this so? AM was constructed explicitly on the assumption that all (enough?) important theorems could be discovered in quite natural ways, as very simple (already-known) relationships on already-defined concepts. AM embodies several such assumptions about math research; they are collected and packaged for display in Section 7.2.6, on page 162.

What else might this facet be useful for, if not the storage of awkwardly-worded conjectures? It might be a good place to store *flimsy* conjectures: those which were strong enough to get considered, yet for which not much empirical confirmation had been done. This in fact was one important role of this facet.

For example, AM was initially told that there are two specializations of Unordered-structures, namely Bags and Sets. But AM was not given any examples of any structures at all. Early on, it chose the task "Fillin examples of Bags" from the agenda. After filling them in, a heuristic rule had AM consider whether or not this concept of Bags was really any more specialized than the concept of Unordered-structures. To test this empirically, AM tried to verify whether or not there were any examples of Unordered-structures that were *not* examples of Bags. Failure to find any led to proposing the conjecture "All Unordered-structures are really Bags". This could have been recorded quite easily: Bags was already known to be specialization of Unordered-structure, so all AM had to do was tag it as a generalization as well (add "Bags" to the Generalizations facet of the Unordered-structures concept). But a heuristic rule which knows about such equivalence conjectures first asked whether there were any specializations of Unordered-structures which had no known examples, and for which AM had not (recently, at least) tried to fill in examples. In fact, such an entry was "Sets". So the conjecture was stored on the Conjec facet of Unordered-structures, and a new job was added to the agenda: "Fill in examples of Sets". The reason was that such examples might disprove this flimsy conjecture. In fact, the job already existed on the agenda, so only the new reason was added, and its priority was boosted. When such examples were found, they did of course disprove that conjecture: each set was

an Unordered-structure and yet was not a Bag.[29]

This last example has suggested another use for this facet: holding heuristic rules which are relevant to filling in and checking conjectures. For example, the Conjec facet of Operations has some special heuristics which look for certain kinds of relationships involving any given operation (e.g., "Pick any example F(x)=y. See what interesting statements can be made about y. Then try to verify or disprove each one by looking at the values of all the other known calls on operation F"). The Conjec facet of Any-concept will contain knowledge which is much more general in scope (e.g., "See whether concept C is an example of some member of (C.Isa).Spec"). Compose.Conjec will contain more specific heuristics (e.g., "See if the composition A∘B is really no different from B").

Given any concept C, AM will ripple upwards, locating Isas(C), and collect the heuristics which are tacked onto their Conjec facets. These heuristic rules will then be evaluated (in order of increasing generality), and some conjectures will probably be proposed, checked, discarded, modified, etc. In fact, each Conjec facet of each concept can have two separate subfacets: Conjec.Fillin and Conjec.Check. The former contains heuristics for noticing conjectures, the second for verifying and patching them up.

There is yet another use for this facet, one of efficiency of storage. After discovering that all primes except 2 are Odd-primes, there is very little reason to keep around Odd-primes as a separate concept from Primes. Yet they are not quite equivalent. Primes.Conjec is a good place for AM to store the conjecture "Prime(x) implies that x=2 or Odd(x)", and to pull over to Primes any efficient definition/algorithm which Odd-primes might possess (patching it up to work for "2"), and then destroy the concept Odd-primes. Another way out is merely to destroy "Primes", and make 2 a distinguished number tacked onto the Just-barely-missed subfacet of Odd-primes.Exs (just like "1" is already).

Here is another example: AM discovers that Set-insert∘Set-insert is the same as just Set-insert. That is, if you insert x twice into a set S, it's no different than inserting it just once (because Sets don't allow multiple copies of the same element). Then there's no longer any reason for keeping Set-insert∘Set-insert hanging around as a separate concept. Instead, just add a small new entry to Set-insert.Conjec and forget that space-consuming composition forever.

There is another use of the Conjec facet: untangling paradoxes. It is with no sorrow that I mention that this facility was never needed by AM: no genuine contradictions ever were believed by AM. What would one look like? Suppose a chain of Spec links indicates that X is a specialization of Y, and yet AM finds some example x of X which does not satisfy Y.Definition). So X is – and is not – a specialization of Y. In such cases, the Conjecs facets of the concepts involved would indicate which of those Spec links were initially-supplied (hence unchallengable), which links were created based on formal verifications (barely challengable), and which links were established based only on empirical evidence (yes, these are the ones which would then fade into the sunset). If it has to, AM should be able to recall the justification for each new link it created. AM can deduce this by examining the Conjec facets of the concepts involved.

---

[29] Bags are not multisets, although those two notions are very closely related to each other. Each set is a multiset by definition; but each set is guaranteed by definition to <u>not</u> be a bag.

Periodically (at huge intervals) AM chose a task of the form "Check conjecs about C", at which time all the entries on C.Conjec would be re-examined in light of existing data. Some would be discarded (perhaps causing some Exs/Isa/Spec/Genl links to vanish with them). Some of the conjectures might be believed much more strongly now (causing some new links to be recorded). This turned out to be a surprisingly ineffective activity; very few new revelations were obtained this way. Ultimately, this kind of task was muzzled (AM was inhibited from doing this).

Theoretically, AM might possess rules which transformed a conjecture into a more efficient algorithm for an operation, or which used the knowledge contained therein to speed up an existing algorithm. Another sophisticated use of a conjec would be to set up a new representation scheme for a concept[30].

Finally, the Conjec's facet is used as a showcase, to highlight some nice discovery that AM wants to display. The user can look at the entries on each concept's Conjec facet (after a long run) and get a better feeling for AM's abilities. If there are several powerful conjectures listed for concept C, then it appears to the user that AM "understands" the concept much better than if C.Conjecs is empty.

Let's recapitulate the uses of this facet:

1. Store awkwardly-phrased conjectures: this wasn't really useful.
2. Store flimsy conjectures: apparent relationships worth remembering, yet not quite believed.
3. Hold heuristics which notice and check conjectures.
4. Obviate the need for many similar concepts: Collapse the entire essence of a related concept into one or two relationships involving this one.
5. Untangling paradoxes: a historic record, which wasn't really used.
6. Improve existing algorithms, definition testing procedures, representations.
7. Display AM's most impressive observed relationships in a form which is easily inspectable by the user.

The syntax of this facet is simply a list of conjectures, where each conjecture has the form of a relationship: (R a b c...d). R is the name of a known operation (in which case, abc... are its arguments and we claim that d is its value), or R is a predicate (and d is either True or False), or R is the name of a kind of link (Genl, Spec, Isa, or Exs), and the claim is that a and b are related by R. Here are three example of conjectures, illustrating the possible formats:

1. **(Compose Set-insert Set-insert Set-insert)**. This says that if you apply the known operation Compose, to the two arguments Set-insert and Set-insert, then the resultant composition is indistinguishable from Set-insert.
2. **(Same-size Insert(S,S) S False)**. That is, inserting a set into itself will always (for finite sets) give you a set of a different length.
3. **(Example-of Prime-factorings Function)**. This conjecture is the unique factorization

---

[30] e.g., after unique factorization is discovered, begin representing numbers as a bag of primes: n is represented as the prime factorization of n. This is exponentially better than unary notation: bags-of-1's. AM had a tiny ability for this kind of ongoing transformation, so crude it's better left undescribed.

theorem. The operation which takes a number n, and finds all prime factorizations of n, is claimed to be a function, not merely a relation. That is, each n has precisely one such prime factoring.

5.2.8. Definitions

A typical way to disambiguate a concept from all others is to provide a "definition" for it.[31] Almost every concept had some entries initially supplied on its "Definitions" facet. The format of this facet is a list of entries, each one describing a separate definition. A single entry will have the following parts:

1. Descriptors: Recursive/Linear/Iterative, Quick/Slow, Opaque/Transparent, Once-only/Early/Late, Destructive/Nondestructive.
2. Relators: Reducing to the definition of concept X, Same as Y except..., Specialized version of Z, Using the definition of W, etc.
3. Predicate: A small, executable piece of LISP code, to tell if any given item is an example of this concept.

The predicate or "code" part of the entry must be faithfully described by the Descriptors, must be related to other concepts just as the Relators claim. The predicate must be a LISP function which take argument(s) and return either T or NIL (for True/False), depending on whether or not the argument(s) can be regarded as examples of the concept.

The argument "{A B}" should satisfy the predicate of any valid definition entry of the Sets concept. This triple of arguments <{A B}, {A C}, {A B C}> should satisfy any definition of the Set-union concept, since the third is equal to the Set-union of the first two arguments.

Here is a typical entry from the Definitions facet of the Set-union concept:

```
Descriptors: Slow, Recursive, Transparent

Relators: Uses the algorithm for Set-insert, Uses the definition of Empty-set,
          Uses the definition of Set-equal, Uses the algorithm for Some-member,
          Uses the algorithm for Set-delete, Uses the definition of Set-union

Code: λ (A B C)
        IF  Empty-set.Defn(A)  THEN  Set-equal.Defn(B,C)  ELSE
            X ← Some-member.Alg(A)
            A ← Set-delete.Alg(X,A)
            B ← Set-insert.Alg(X,B)
            Set-union.Defn(A,B,C)
```

[31] As EPAM studies showed [Feigenbaum 63], one can never be sure that this definition will specify the concept uniquely for all time. In the distant future, some new concept may differ in ways thought to be ignorable at the present time.

Let me stress that this is just one entry, from one facet of one concept.

The notation "X ← Some-member.Alg(A)" means that any one algorithm for the concept Some-member should be accessed, and then it should be run on the argument A. The result, which will be an element of A, is to be assigned the name "X". The effect is to bind the variable X to some member of set A.

In the actual LISP implementation, the ELSE part of the conditional is really coded[32] as:

```
(Set-union.Defn (Set-delete.Alg (SETQ X (Some-member.Alg A))  A)
                (Set-insert.Alg  X  B)
                C
)
```

This particular definition is not very efficient, but it is described as Transparent. That means it is very well suited to analysis and modification by AM itself. Suppose some heuristic rule wants to generalize this definition. It can peer inside it, and, e.g., replace the base step call on Set-equal, by a call on a generalization of Set-equal (say "Same-length"[33]).

How could *different* definitions help here? Suppose there were a definition which first checked to see if the three arguments were Set-equal to each other, and if so then it instantly returned T as the value of the definition predicate; otherwise, it recurred into Set-union.Defn again. This might be a good algorithm to try at the very beginning, but if the Equality test fails, we don't want to keep recurring into this definition. This algorithm should thus have a descriptor labelling it ONCE-ONLY EARLY.

A typical kind of entry for the Definitions facet of an operation is to simply call on the *Algorithms* part of that same concept. Here is such an entry from the Definitions facet of the Set-union concept:

> **Descriptors: none**
>
> **Relators: Uses the definition of Set-equal, Uses the algorithm for Set-union**
>
> **Code: λ (A B C) Set-equal.Defn(C, Set-union.Alg(A,B))**

This definition is a trivial call on the "Algorithms" facet of Set-union. That is, one way to test whether C is the set-union of A and B, is simply to *run* set-union on A and B, and compare the result against C. The descriptors and relators of the particular algorithm which is chosen will then be added to the descriptors and relators which exist so far on this entry. Note that the box above (like the box on the previous page) is simply one entry on the Definitions facet of the Set-union concept.

---

[32] The expression "(f.Defn a1 a2..)" means "apply the predicate part of a definition of f, to arguments a1, a2,..". This definition is to be randomly selected from the entries on the Definitions facet of concept f.

[33] For disjoint sets, the new definition would specify the operation which we call "addition"

There are three purposes to having descriptors and relators hanging around:

1. For the benefit of the user. AM appears more intelligent because it can *describe* the kind of definition it is using – and why.
2. For the sake of efficiency. When all AM wants to do is to evaluate Set-union(A,B), it's best just to grab a *fast* definition. When trying to generalize Set-union, it's more appropriate to modify a very clean, transparent definition – even if it is a slow one.
3. For the benefit of the heuristic rules. Often, a left- or a right-hand-side will ask about a certain kind of definition. For example, **"If a transparent definition of X exists, then try to specialize X".**

Granted that Descriptors and Relators are useful, how do these "meta-level" modifiers get filled in, for newly-created[34] concepts? All such powers are embedded in the fine structure of the heuristic rules. This is true for the Algorithms facet as well, and will be illustrated in the very next subsection.

Let me pull back the curtain a little further, and expose the actual implementation of these ideas in AM. The secrets about to be revealed will not be acknowledged anywhere else in this document. They may, however, be of interest to future researchers. Each concept may have a cluster of Definition facets, just as it can have several kinds of Examples facets. These include three types: Necessary and sufficient definitions, necessary definitions, and sufficient definitions. These three types have the usual mathematical meanings. All that has been alluded to before (and after this subsection) is the necc&suff type of definition (x is an example of C *if and only if* x satisfies C.Def/necc&suff). Often, however, there will be a much quicker sufficient definition (x satisfies C.Def/suf, *only if* x is certainly a C). Similarly, entries on C.Def/nec are useful for quickly checking that x is *not* an example of C (to check this, it suffices to verify that x *fails* to satisfy a necessary definition of C).

So given the task of deciding whether or not x is an example of C, we have many alternatives:

1. If x is a concept, see if C is a member of x.ISA (if so, then x is an example of C).
2. Try to locate x within C.Exs. (depending upon the flavor of subfacet on which x is found, this may show that x is or is not an example of C).
3. If x is a concept, ripple to collect ISA's(x), and see if C is a member of ISA's(x).
4. If there is a fast sufficent definition of C, see if x satisfies it.
5. If there is a fast necessary definition of C, see if x fails it (if so, then x is *not* an example of C).
6. If there is a necessary and sufficient definition of C, see whether or not x satisfies that definition (this may show that x is or is not an example of C).
7. Try to locate x within C.Exs. (depending upon the flavor of subfacet on which x is found, this may show that x is or is not an example of C).
8. Recur: check to see if x is an example of any specialization of C.
9. Recur: check to see if x is *not* an example of some generalization of C (if so, then x is *not* an example of C),

In fact, there is a LISP function, IS-EXAMPLE, which performs those steps in that order.

---

[34] For initially-supplied definition entries, the author hand-coded these modifiers.

At each moment, there is a timer set, so even if there is a necessary and sufficient definition hanging around, it might run out of time before settling the issue one way or the other. Each time the function recurs, the timer is granted a smaller and smaller quantum, until finally it has too little to bother recurring anymore. There is a potential overlap of activity: to see if x is an example of C, the function might ask whether x is or is not an example of a particular generalization of C (step 9, above); to test *that*, AM might get to step 8, and again ask if x is an example of C. Even though the timer would eventually terminate this fiasco (and even though the true answer might be found despite this wasted effort) it is not overly smart of AM to fall into this loop. Therefore, a stack is maintained, of all concepts whose definitions the IS-EXAMPLE function tried to test on argument x. As the function recurs, it adds the current value of C to that stack; this value gets removed when the recursion pops back to this level, when that recursive call "returns" a value.

5.2.9. Algorithms

Earlier, we said that each concept can have any facets from the universal fixed set of 25 facets. This is not strictly true. Sometimes, a whole class of concepts will possess a certain type of facet which no others may meaningfully have. If C can have that facet, then so can any specialization of C. Typically, there will be some concept C such that the examples of C are precisely the set of concepts which can possess the new facet. That is, there will be a *domain of applicability* for the facet, just as we defined such domains of applicability for heuristics. For example, consider the "Domain/Range" facet. It is meaningful only to "operations", but really *is* an important feature of all operations. Its domain of applicability is Operation.

The kinds of facets – including all such limited "jargon" facets – is fixed once and for all. New kinds of facets cannot be conceived and added by AM itself. Nor does AM have any control over the domain of applicability of each facet.

If desired, one can view all this in a more general light. For each facet f, the only concepts which can have entries for facet f are examples of some particular concept J(f) – the "J" stands for "jargon". J(f) is the domain of applicability of facet f. If C is any concept which is not an example of J(f), then it can never meaningfully possess any entries for that facet f. For almost all facets f, J(f) is "Any-concept". Thus any concept can possess almost any facet. For example, J(Defn)="Any-concept", so any concept may have definitions.

There are a few more restricted facets. For example, J(Domain/range)="Operation". So only operations can have domain/range facets.[35] The concept "Sets", which is not an operation, can't have a domain/range facet.

Similarly, J(Algorithms)="Actives". This facet is the subject of this section. The Algorithms facet is present for all – but only for – Actives (predicates, relations, operations).

The representation is, as usual, a list of entries, each one describing a separate algorithm. A single entry will have the following parts:

---

[35] Actually, Predicates also have domain/range facets, even though the Range parts are all necessarily the same: {T,F}.

1. Descriptors: Recursive/Linear/Iterative, Quick/Slow, Opaque/Transparent, Once-only/Early/Late, Destructive/Nondestructive.
2. Relators: Reducing to the algorithm for concept X, Same as Y except..., Specialized version of Z's algorithm, Using the algorithm for W, etc.
3. Program: A small, executable piece of LISP code, for actually running C.

Note the similarity to the format for the Definitions facets of concepts. Instead of a LISP predicate, however, the Algorithms facets possess a LISP function (an executable piece of code whose value will in general be other than True/False). That "program" part of the entry must be faithfully described by the Descriptors, must be related to other concepts just as the Relators claim, must take arguments and return values as specified in the Domain/Range facet of C, and when run on any arguments, the resultant <args value> pair must satisfy the Definitions facet of C.

There is an extra level of sophistication which is available but rarely used in AM. The descriptors can themselves be small numeric-valued functions. For example, instead of just including the Descriptor "Quick", and instead of just giving a fixed number for the speed of the algorithm, there might be a little program there, which looked at the arguments fed to the algorithm, and then estimated how fast this algorithm would be. The main reason for not using this feature more heavily is that most of the algorithms are fairly fast, and fairly constant in performance. It would be silly to spend much time recomputing their efficiency each time they were called. If the algorithm is recursive, this conjures up even sillier pictures. The main reason in support of using this feature is of course "intelligence": in the long run, processing a little bit before deciding which algorithm to run *has* to be the winning solution. At the moment, it is not yet cost-effective.

Here is a typical entry from the Algorithms[36] facet of the Set-union concept:

```
Descriptors: Slow, Recursive, Transparent

Relators: Uses the algorithm for Set-insert, Uses the definition of Empty-set,
        Uses the algorithm for Some-member, Uses the algorithm for Set-insert,
        Uses the algorithm for Set-union

Code: λ (A B)
        IF  Empty-set.Defn(A) THEN B  ELSE
              X ← Some-member.Alg(A)
              A ← Set-delete.Alg(X,A)
              B ← Set-insert.Alg(X,B)
              Set-union.Alg(A,B)
```

[36] note that it is similar to -- but not identical to -- the entry shown on page 86, of a Definition of Set-union.

Note that the Descriptors don't say whether this algorithm is destructive[37] or not. That means that this same algorithm can be used either destructively or not, depending on what AM wants. More precisely, it's up to the algorithms which get called on by this one. If they are all chosen to be destructive, so will Set-union. If they all copy their arguments first then Set-union will *not* be destructive. For example, note how the algorithm calls on Set-insert(X,B). If this is destructive, then at the end B will have been physically modified to contain X; the original contents of B will be lost.

This particular algorithm is not very efficient, but it is described as Transparent. That means it is very well suited to analysis and modification by AM itself. Suppose some heuristic rule wants to specialize this algorithm. It can peer inside it, and, e.g., replace the variable X in (Set-insert X B) by the constant "T".[38]

Why should AM bother storing multiple algorithms for the same concept? Consider this example again, of Set-union. Suppose there were an algorithm which first checked to see if the two arguments were Equal to each other, and if so then it instantly returned one of them as the final value for Set-union; otherwise, it recurred into Set-union.Alg. This might be a good algorithm to try at the very beginning, but if the Equality test fails, we don't want to keep recurring into this definition. This algorithm should thus have a descriptor labelling it ONCE-ONLY EARLY.

Also, there is an iterative algorithm which checks to see if A equals B, and if so then it returns B. If not, the algorithm proceeds to check that A is shorter than B, and if not it switches them. Finally, it enters an iterative loop similar to the recursive one above: it repeatedly transfers an element from A to B, using Some-member, Set-delete and Set-insert. This iterative loop repeats until A becomes empty. While more efficient than the recursive one, this definition is less transparent.

An even more efficient algorithm is provided, but it is totally opaque:

> **Descriptors: Quick, Non-recursive, Non-destructive, Opaque**
>
> **Relators: none**
>
> **Code: λ (A B) (UNION A B)**

This algorithm calls on the LISP function "UNION" to perform the set-union. It is the "best" algorithm to choose unless space is critical, in which case a destructive algorithm must be chosen, or unless AM wishes to inspect it rather than run it, in which case a transparent one must be picked.

---

[37] A LISP algorithm is destructive if it physically, permanently modifies the list structures it is fed as arguments. Set-union(A,B) is destructive if -- after running -- A and B don't have the same values they started with. The advantages of destructive operations are increased speed, decreased space used up, fewer assignment statements. The danger of course is in accidentally destroying some information you didn't mean to.

[38] This is a fairly useless new operation, of course. It adds T to B unless A is empty, in which case this operation has no effect at all.

All the details about understanding the descriptors and relators are embedded in the fine structure of the heuristic rules. A left-hand-side may test whether a certain kind of algorithm exists for a given concept. A right-hand-side which fills in a new algorithm must also worry about filling in the appropriate descriptors and relators. As with newly created concepts, such information is trivial to fill in at the time of creation, but becomes much harder after the fact.

Here is a typical heuristic rule which results in a new entry being added to the Algorithms facet of the newly-created concept named Compose-Set-Intersect&Set-Intersect:

```
IF the task is to Fillin Algorithms for F,
     and F is an example of Composition
     and F has a definition of the form F=GoH,
     and F has no transparent, nonrecursive algorithm,
THEN add a new entry to the Algorithms facet of F,
     with Descriptors: Transparent, Non-recursive
     with Relators: Reducing to G.Alg and H.Alg, Using the Definition of <G.Domain>
     with Program: λ (||<G.Domain>||,||<H.Domain>||-1,X)
                                   (SETQ X (H.Alg ||<G.Domain>||))
                                   (AND
                                       (<G.Domain>.Defn X)
                                       (G.Alg X ||<H.Domain>||-1))
```

The intent of the little program which gets created is to apply the first operator, check that the result is in the domain of the second, and then apply the second operator. The expression ||<G.Domain>|| means find a domain/range entry for G, count how many domain components there are, and form a list that long from randomly-chosen variable names (u,v,w,x,y,z).

For the case mentioned above, F = Compose-Set-Intersect&Set-Intersect, G = Set-Intersect, and H = Set-Intersect. The domain of G is a pair of Sets, so ||<G.Domain>|| is a list of 2 variables, say (u v). Similarly, ||<H.Domain>||-1 is a list of 1 variable, say (w). Putting all this together, we see that the new definition entry created for Compose-Set-Intersect&Set-Intersect would look like this:

```
Descriptors: Non-Recursive, Transparent

Relators: Reducing to Set-Intersect.Alg, Using the definition of Sets

Code: λ (u,v,w,X)
          (SETQ X (Set-Intersect.Alg u v))
          (AND
            (Sets.Defn X)
            (Set-Intersect.Alg X w)
```

Let me make clear here one "kluge" of the AM program. At times, AM will be capable of producing only a slow algorithm for some new concept C. For example, $TIMES^{-1}(x)$ was

originally defined by AM as a blind, exhaustive search for bags of numbers whose product is x. As AM uses that algorithm more and more, AM records how slow it is. Eventually, a task is selected of the form "Fillin new algorithms for C", with the two reasons being that the existing algorithms are all too slow, and they are used frequently. At this point, AM should draw on a body of rules which take a declarative definition and transform it into an efficient algorithm, or which take an inefficient algorithm and speed it up. Doing a good job on just those rules would be a mammoth undertaking, and the author decided to omit them. Instead, the system will occasionally beg the user for a better (albeit opaque) algorithm for some particular operation. In general, the only requests were for inverse operations, and even then only a few of them. The reader who wishes to know more about rules for creating and improving LISP algorithms is directed to [Darlington and Burstall 73]. A more general discussion of the principles involved can be found in [Simon 72].

5.2.10. Domain/Range

Another facet possessed only by active concepts is Domain/Range. The syntax of this facet is quite simple. It is a list of entries, each of the form $\langle D_1\ D_2 \ldots \rightarrow R \rangle$, where there can be any number of D_i's preceding the arrow, and R and all the D_i's are the names of concepts. Semantically, this entry means that the active concept may be run on a list of arguments where the first one is an example of D_1, the second an example of D_2, etc., and in that case will return a value guaranteed to be an example of R. In other words, the concept may be considered a relation on the cross-product $D_1 x D_2 x \ldots x R$. We shall say that the *domain* of the concept is $D_1 x D_2 x \ldots$, and that its *range* is R. Each D_i is called a *component* of the domain.

For example, here is what the Domain/Range facet of TIMES might look like:

```
(
  < Numbers Numbers → Numbers >
  < Odd-numbers Odd-numbers → Odd-numbers >
  < Even-Numbers Even-Numbers → Even-numbers >
  < Odd-numbers Even-Numbers → Even-Numbers >
  < Perf-Squares Perf-Squares → Perf-Squares >
  < Bags-of-Numbers → Numbers >
                                        )
```

Here is what the Domain/Range facet of Set-Union might look like:

```
(
  < Sets Sets → Sets >
  < Nonempty-sets Sets  → Non-empty-sets >
  < Sets-of-Sets → Sets >
                                  )
```

The Domain/Range part is useful for pruning away absurd compositions, and for syntactically suggesting compositions and "coalescings". Let's see what this means.

Suppose some rule sometime tried to compose TIMES∘Set-union. A rule tacked onto Compose says to ensure that the range of Set-union at least intersects (and preferably is *equal* to) some component of the domain of TIMES. But there are no entities which are both sets and numbers[39]; ergo this fails almost instantaneously.

This is too bad, since there was probably a good reason (e.g., intuition) for trying this composition. If the activation energy (priority of the current task) is high enough, AM will continue trying to force it through. The failure arose because Sets could not be viewed as if they were Numbers. A relevant rule says:

> **IF you want to view X's as if they were Y's,**
> **THEN seek an interesting operation F from X to Y, to do the viewing.**

So AM had to locate any and all operations whose domain/range had an entry of the form <Sets→Numbers>. The only such operation known to AM at the time was F=Length. So the composition produced was TIMES[X, Length(Set-union(Y,Z))].

Notice that if the composition Set-union∘Set-union is proposed, there will be no conflict, since the range of Set-union obviously intersects one component of the domain of Set-union. How can AM determine the domain/range of this composition? A rule tacked onto Compose indicates that if F=G∘H, and a domain/range entry for G is <A...X...B → C>, and an entry for H is <D...E → Y>, and Y intersects X, then an entry for F's domain/range is <A...D...E...B → C>. That is, the domain of H is substituted for the single component of the domain of G which can be shown to intersect the range of H. Purely syntactically, AM can thus compute some domain/range entries for the composition Set-union∘Set-union.

< Sets Sets → Sets> and < Sets Sets → Sets> combine to yield < Sets Sets Sets → Sets >;
<Non-empty-sets Sets → Non-empty-sets> and <Sets Sets → Sets> combine to yield <Non-empty-sets Sets Sets → Non-empty-sets>;

and so on. Similarly, one can compute an entry for the domain/range facet of the previous composition of three operations TIMES∘Length∘Set-union:

< Sets Sets → Sets>, < Sets → Numbers>, and < Numbers Numbers → Numbers > combine to yield < Numbers Sets Sets → Numbers >

So when computing TIMES(X, Length(Set-union(Y,Z))), both Y and Z can be sets, and X a number, and the result will be a number.

The claim was also made that Domain/Range facets help propose plausible coalescings. By "*coalescing*" an operation, we mean defining a new one, which differs from the original one in that a couple of the arguments must now coincide. For example, coalescing TIMES(x,y) results in the new operation F(x) defined as TIMES(x,x). Syntactically, we can coalesce a pair of domain components of the domain/range facet of an operation if those two domain components are equal, or if one of them is a specialization of the other, or even if they

---

[39] Why? The number n, to AM, is represented in unary, as a bag of n T's. None of these are sets. The composition "TIMES∘BAG-UNION" would have made sense to AM, but would have been defined only for bags-of-T's. Then TIMES∘BAG-UNION(x,y,z) would be just x(y•z).

merely intersect. In the case of one related to the other by specialization, the more specialized concept will replace both of them, In case of merely intersecting, an extra test will have to be inserted into the definition of the new coalesced operation.

Given this domain/range entry for Set-insert: < Anything Sets → Sets >, we see that it is ripe for coalescing. Since Sets is a specialization of Anything, the new operation F(x), which is defined as Set-insert(x,x), will have a domain/range entry of the form < Sets → Sets >. That is, the specialized concept Sets will replace both of the old domain elements (Anything and Sets). F(x) takes a set x and inserts it into itself. Thus F({a,b})={a,b,{a,b}}. In fact, this new operation F is very exciting because it always seems to give a new, larger set than the one you feed in as the argument.

We have seen how the Domain/range facets can prune away meaningless coalescings, as well as meaningless compositions. Any proposed composition or coalescing will at least be syntactically meaningful. If all compositions are proposed only for at least one good semantic reason, then those passing the domain/range test, and hence those which ultimately get created, will all be valuable new concepts. Since almost all coalescings are semantically interesting, *any* of them which have a valid Domain/Range entry will get created and probably will be interesting.

This facet is occasionally used to suggest conjectures to investigate. For example, a heuristic rule says that if the domain/range entries have the form <D D D... → genl(D) >, then it's worthwhile seeing whether the value of this operation doesn't really always lie inside D itself. This is used right after the Bags↔Numbers analogy is found, in the following way. One of the Bag-operations known already is Bag-union. The analogy causes AM to consider a new operation, with the same algorithm as Bag-union, but restricted to Bags-of-T's (numbers in unary representation). The Domain/range facet of this new, restricted mutation of Bag-union contains only this entry: <Bags-of-T's Bags-of-T's → Bags>. Since Bags is a generalization of Bags-of-T's, the heuristic mentioned above triggers, and AM sees whether or not the union of two Bags-of-T's is always a bag containing only T's. It appears to be so, even in extreme cases, so the old Domain/range entry is replaced by this new one: <Bags-of-T's Bags-of-T's → Bags-of-T's>. When the user asks AM to call these bags-of-T's "numbers", this entry becomes <Numbers Numbers → Numbers>. In modern terms, then, the conjecture suggested was that the sum of two numbers is always a number.

To sum up this last ability in fancy language, we might say that one mechanism for proposing conjectures is the prejudicial belief in the unlikelihood of asymmetry. In this case, it is asymmetry in the parts of a Domain/range entry that draws attention. Such conjecturing can be done by any action part of any heuristic rule; the Conjec facet entries don't have a monopoly on initiating this type of activity.

5.2.11. Worth

How can we represent the worth of each concept? Here are some possible suggestions:

1. The most intelligent (but most difficult) solution is "purely symbolically". That is, an individualized description of the good and bad points of the concept; when it is useful, when misleading, etc.
2. A simpler solution would be to "standardize" the above symbolic description once

and for all, fixing a universal list of questions. So each concept would have to answer the questions on this list (How good are you at motivating new concepts?, How costly is your definition to execute?,...). The answers might each be symbolic; e.g., arbitrary English phrases.

3. To simplify this scheme even more, we can assume that the answers to each question will be numeric-valued functions (i.e, LISP code which can be evaluated to yield a number between 0 and 1000). The vector of numbers produced by Evaluating all these functions will then be easy to manipulate (e.g. using dot-product, vector-product, vector-addition, etc.), and the functions themselves may be inspected for semantic content. Nevertheless, much content is lost in passing from symbolic phrases to small LISP functions.
4. A slight simplification of the above would be to just store the vector of numbers answering the fixed set of questions; i.e., don't bother storing a bunch of programs which compute them dynamically.
5. Even simpler would be to try to assign a single "worthwhileness" number to each concept, in lieu of the vector of numbers. Simple arithmetic operations could manipulate Worth values then. In some cases, this linear ordering seems reasonable ("primes" really are better than "palindromes".) Yet in many cases we find concepts which are too different to be so easily compared (e.g., "numbers" and "angles".)
6. The least intelligent solution is none at all: each concept is considered equally worthwhile as any other concept. This threatens to be combinatorial dynamite.

As we progress along the intelligent→→→trivial dimension, we find that the schemes get easier and easier to code, the Worth values get easier and easier to deal with, but the amount of reliable knowledge packed into them decreases.

Initially, scheme #3 above was chosen for AM: a vector of numeric-valued procedural answers to a fixed set of questions. Here are those questions, the components of the Worth vectors for each concept:

1. Overall aesthetic worth.
2. Overall utility. Combination of usefulness, ubiquity.
3. Age. How many cycles since this concept was created?
4. Life-span. Can this concept be forgotten yet?
5. Cost. How much cpu time has been spent on this concept, since its creation?

Notice that in general no constant number can answer one of these questions once and for all (consider, e.g., Life-span). Each 'answer' had to be a numeric-valued LISP function.

A few questions which crop up often are not present on this list, since they can be answered trivially using standard LISP functions (e.g., "How much space does concept C use up?" can be found by calling the function "COUNT" on the property-list of the LISP atom "C").

Another kind of question, which was anticipated and did in fact come up frequently, is of the form "How good are the entries on facet F of this concept?", for various values of F. Since there are a couple dozen kinds of facets, this would mean adding a couple dozen more questions to the list. The line must be drawn somewhere. If too much of AM's time is drained by evaluating where it is already, it can never progress.

The heuristic rules are responsible for initially setting up the various entries on the Worth facets of new concepts, and for periodically altering those entries for *all* concepts, and for delving into those entries when required.

Recent experiments have shown (see Experiment 1, page 127) there was little change in behavior when each vector of functions was replaced by a single numeric function (actually, the sum of the values of the components of the "old" vector of functions). There wasn't even too much change when this was replaced by a single number. There *was* a noticeable degradation (but no collapse) when all the concepts' numbers were set equal to each other initially.

For the purposes of this document, then (except for this page and the discussion of Experiment 1), we may as well assume that each concept has a single number (between 0 and 1000) attached as its overall "Worth" rating. This number is set[40] and referenced and updated by heuristic rules. Experiment 1 can be considered as showing that a more sophisticated Worth scheme is not necessary for the particular kinds of behaviors that AM exhibits.

5.2.12. Interest

Now that we know how how to judge the overall worth of the concept "Composition", let's turn to the question of how interesting some specific composition is. Unfortunately, the Worth facet really has nothing to say about that problem. The Worth of the concept "Compose" has little effect on how interesting a particular composition is: "CountoDivisors-of" is very interesting, and "InsertoMember"[41] is less so. The Worth facets of *those* concepts will say something about their overall value. And yet there is some knowledge, some "features" which would make any composition which possessed them more interesting than a composition which lacked them:

Are the domain and range of the composition equal to each other?
Are interesting properties of each component of the composition preserved?
Are undesirable properties lost (i.e., *not* true about the composition)?
Is the new composition equivalent to some already-known operation?

These hints about "features to look for" belong tacked onto the Composition concept, since they modify all compositions. Where and how can this be done?

For this purpose each concept – including "Composition" – can have entries on its "*Interest*" facet. It contains a bunch of features which (if true) would make any particular example of the current concept interesting.

The format for the Interest facet is as follows:

< Conflict-matrix
<Feature$_1$, Value$_1$, Reason$_1$, Used$_1$>
<Feature$_2$, Value$_2$, Reason$_2$, Used$_2$>

---

[40] The author initially sets this value for the 115 initial concepts. Heuristic rules set it for each concept created by AM.

[41] INSERTtoMEMBER(x,y,z)= if x∈y, then insert 'T' into z, else insert 'NIL' into z.

...
<Feature$_K$, Value$_K$, Reason$_K$, Used$_K$>
>

This is the format of the facet itself, not of each entry. The conflict-matrix is special and will be discussed below. Each Feature/Value/Reason/Used quadruple will be termed an "entry" on the Interest facet.

Each "Feature," is a LISP predicate, indicating whether or not some interesting property is satisfied. The corresponding "Value$_i$" is a numeric function for computing just how valuable this feature is. The "Reason," is a token (usually an English phrase) which is tacked along and moved around, and can be inspected by the user. The "Used$_i$" subpart is a list of all the concepts whose definitions are known to incorporate[42] this feature; all examples of such concepts will then automatically satisfy this Feature$_i$.

For example, here is one entry from the Interest facet of Compose:

FEATURE: Domain(Arg1)=Range(Arg2)
VALUE: .4 * .4xWorth(Domain(Arg1)) + .2xPriority(current task)
REASON: "The composition of Arg1 and Arg2 will map from a set back into that same set"
USED: Compose-with-self-Domain=Range-operation, Interesting-compose-4

Just as with Isa's and Generalizations, we can make a general statement about Interest features:

Any feature tacked onto the Interest facet of any member of ISA's(C), also applies to C.

That is, X.Interest is relevant to C iff C is an example of X. For example, any feature which makes an operation interesting, also makes a composition interesting.

So we'd like to define the function Interests(C) as the union of the Interest features found tacked onto any member of ISA's(C).[43] But some of these might have already been conjoined to a definition, to form the concept C (or a generalization of C). So all C's will trivially (by definition) satisfy such features. The USED subparts can be employed to find such features. In fact, the final value of Interests(C) is the one computed above, using ISA's(C), but after eliminating all the features whose USED subparts pointed to any member of ISA's(C).

This covers the purpose of each subpart of each entry on a typical Interest facet. Now we're ready to motivate the presence of the Conflict-matrices.

Often, AM will specialize a concept by conjoining onto its definition some features which would make any example of the concept interesting. So *any* example of this new specialized

---

[42] Not SATISFY the feature. Thus the general concept Domain=Range-op incorporates the feature "range(x)is one component of domain(x)" as just one of the conjuncts in its definition. On the other hand, Set-union satisfies the feature, since its range, Sets, really is one component of its domain.

[43] Recall that the formula for this is ISA's(C) = Generalizations(Isa(Generalizations(C))).

concept is thus guaranteed to be an *interesting* example of the old concept. Sometimes, however, a pair of features are exclusive: both of them can never be satisfied simultaneously. For example, a composition can also be interesting if "arg2" is an operation from Range(arg1) into a set which is much more interesting than either Domain(arg1) or Range(arg1). Clearly, this feature and the one shown above can't both be true ("x=y" and "x much more interesting than y" can't occur simultaneously). If AM didn't have some systematic way to realize this, however, it might create a new concept, called Interesting-composition, defined as any composition satisfying both of those features. But then this concept will be vacuous: *no* operation can possibly satisfy that over-constrained definition; this new concept will have no examples; it is the null concept; it is trivially forgettable. Merely to think of it is a blot on AM's claim to rationality.

The "Conflict-matrix" is specified to prevent many such trivial combinations from eating up a lot of AM's time (and, as usual, it helps to make AM appear smarter). If there are K features present for the Interest facet of the concept, then its conflict-matrix will be a K×K matrix. In row i, column j of this matrix is a letter, indicating the relationship between features i and j:

E Exclusive of each other: they both can't be true at the same time.
→ Implies: If feature i holds, then feature j must hold.
← Implied by: If feature j holds, then so does feature i.
= Equal. Feature i holds precisely when feature j holds.
U Unrelated. As far as known, there is no connection between them.

These little relations are utilized by some of the heuristic rules. Here is one such rule. Its purpose is to create a new, specialized form of concept C, if many examples of C were previously found very quickly.

IF Current-task is (Fillin Specializations of C)
and ||C.Examples||>30
and Time-spent-on-C-so-far < 3 cpu seconds,
and Interests(C) is not null,
THEN create a new concept named Interesting-C,
Defined as the conjunction of C.Defn and the highest-valued member of Interests(C) which is U (unrelated) to any feature USED in the definition of C.
and add the following task to the agenda: Fillin examples of Interesting-C, with value computed as the Value subpart of the chosen feature, for this reason: "Any example of Interesting-C is automatically an interesting example of C".
and add "Interesting-C" to the USED subpart of the entry where that feature was originally plucked from.

Of course, the LISP form of the above rule is really more detailed about what to do, but the general flavor of the interaction with the Interest facet should come across. As before, the value desired is not C.Interest, but rather the post-rippling value Interests(C). C.Int contains a few features pertaining just to C's, but Interests(C) contains many additional features which are not limited in scope to merely judging C's, but pertain to a more general class of concepts. The quantity 'Time-spent-on-C-so-far' is one component of the Worth facet of C; it might just as well have been accessed from some "Past-history" record of AM's activities. The numbers in the rule – and every little bit of that rule – were specified ad hoc by the author. This is true for each rule initially present in AM. As Section 6.2 will discuss, the precise numbers don't drastically affect the system's performance.

5.2.13. Suggest

This section describes a space-saving "trick", and a "fix-up" to undo some potentially serious side-effects of that trick. Readers not interested in this level of detail may skip to the next subsection.

AM maintains a long list of tasks (the agenda), ordered by a global priority rating scheme. Besides this, AM maintains two threshholds: Do-threshhold and a lower one, Be-threshhold.

When a new task is proposed, if its global priority is below Be-threshhold, then it won't even be entered on the agenda. This value is set so low that any task having even one mediocre reason will make it onto the agenda.

After a task is finished executing, the top-rated one from the agenda is selected to work on next. If its priority rating is below Do-threshhold, however, it is put back on the agenda, and AM complains that no task on the agenda is very interesting at the moment. AM then spends a minute or so looking around for new tasks, re-evaluating the priorities of the tasks on the agenda already, etc.

One way to find new tasks (and new reasons for already-existing tasks) is to evaluate the "*Suggest*" facets of all the concepts in the system. More precisely, each Suggest facet contains some heuristics, encoded into LISP functions. Each function accepts a number N as an argument (representing some minimum value tolerable for a new task), and the function returns as its value a list of new tasks. These are then merged into the agenda, if desired.

Semantically, each function is one heuristic rule for suggesting a new task which might be very plausible, promising, and *a propos* at the current time. For example, here is one entry from the Suggest facet of Any-concept:

> **IF there are no examples for concept C filled in so far,**
> **THEN consider the task "Fillin examples of C", for the following reason: "No examples of C filled in so far", whose value is half of Worth(C). If that value is below arg1, then forget it; otherwise, try to add to to the agenda.**

The argument "arg1" is that low numeric value, N, supplied to the Suggest facet.

This entry alone will produce a multitude of potential tasks; for concepts whose Worth numbers are high, or for which a task is already on the agenda to fill in their examples, these suggested tasks will be remembered; most of the other ones will typically be forgotten.

One use of this facet is thus to "beef up" the agenda whenever AM is discontented with all the tasks thereon. At such a time, AM may call on all the Suggest facets in the system, and a large volume of new tasks will be added to the agenda. Many of them will exist there already, but for different reasons, so many old tasks' priority values will rise. After this period of suggesting is over, the agenda's highest-ranking task will hopefully have a higher value than any did before. Also at this time, the Be-threshhold and Do-threshhold numbers are reduced. So there are two reasons why the top task may now be rated higher than Do-threshhold. If it isn't, then the threshholds are lowered again, and again all the Sugg facets are triggered (this time with a lower N value).

Both threshholds are raised slightly every time AM succeeds in picking and executing a task. So they follow a pattern of slow increase, followed by a sudden decrement, followed by another slow increase, etc. This was intended to mimic a human's increasing expectations as he makes progress.[44] It also mimics the way a human strains his mind when an obstacle to that progress appears; if the straining doesn't produce a brilliant new insight, he grudgingly is willing to reduce his expectations, and perhaps resume some "old path" abandoned earlier.

Another use of this facet is to re-suggest tasks that might have been dropped from (or never made it onto) the agenda, because they weren't valued above Be-threshhold. How might this work? Suppose that, at an earlier time, a task was proposed but never made it onto the agenda because Be-threshhold was quite high. Now, suppose Be-threshhold is much lower (due to a succession of failures). If a Sugg facet re-proposes that same task, it will be accepted, will "stick" onto the agenda (albeit near the bottom). The Suggest facets can reproduce most of the common tasks, and try to stick them on the agenda (though usually for a mediocre to poor reason). It will still usually require another reason for such a task to rise to the very top of the agenda, and be selected and executed.

So the use of the two threshholds is really an unaesthetic space-saving device, and the role of the Suggest facets is merely to correct the errors introduced in this way. There may be no convincing intuitive reason for having these facets at all in a "just" world.

5.2.14. Fillin/Check

To doubt everything doesn't suffice; one must know why he doubts.

-- Poincare'

There is one more level of structure to AM's representation of a concept than the simple "properties on a property-list" image. Each concept consists of a bunch of facets; each facet follows the format layed down for it (and described in the preceding several subsections). Yet each facet of each concept can have two additional "subfacets" (little slots that are hung onto any desired slot) named *Fillin* and *Check*.

The "Fillin" field of facet F of concept C is abbreviated C.F.Fillin. The format of that subfield is a list of heuristic rules, encoded into LISP functions. Semantically, each rule in C.F.Fillin should be relevant to filling in entries for facet F of any concept which is a C. This substructure is an implementation answer to the question of where to place certain heuristic rules.

As an illustration, let me describe a typical rule which is found on Compose.Examples.Fillin. According to the last paragraph, this must be useful for filling in examples of any operation

[44] This was based on personal introspection, and should be tested experimentally.

which is a composition. The rule says that if the composition A∘B is formed from two very time-consuming operations A and B, then it's worth trying to find some examples of A∘B by symbolic means; in this case, scan the examples of A and of B, for some pair of examples x→y (example of B) and y→z (example of A). Then posit that x→z is an example of A∘B. This rule applies precisely to the task of filling in examples of Examples(Composition). Thus, it is relevant to the task "Fill in examples of InsertoInsert". It is irrelevant if you change the action (e.g., "*Check* examples of InsertoInsert"), or if you change the facet to be dealt with (e.g., "Fill in *algorithms* for InsertoInsert"), or if you change the class of concept (e.g., "Fill in examples of *Set-union*)[45].

As another illustration, let me describe a typical rule which is found on Compose.Conjec.Fillin. It says that one potential conjecture about a given composition A∘B is that it is unchanged from A (or from B). This happens often enough that it's worth examining each time a new composition is made. This rule applies precisely to the task of filling in conjectures about particular compositions.

The subfacet Any-Concept.Examples.Fillin is quite large; it contains all the known methods for filling in examples of C (when all we know is that C is a concept). Here are a few of those techniques[46]:

1. Instantiate C.Defn
2. Search the examples facets of all the concepts on Generalizations(C) for examples of C
3. Run some of the concepts named in In-ran-of(C) [i.e., operations whose range is C] and collect the resultant values.

Any-Concept.Examples.Check is large for similar reasons. A typical entry there says to examine each verified example of C: if it is also an example of a specialization of C, then it must be removed from C.Examples and inserted[47] into the Examples facet of that specialized concept.

Here is one typical entry from Operation.Domain/Range.Check:

IF a domain/range entry has the form (D D D... → R),
and all the D's are equal, and R is a generalization of D,
THEN it's worth seeing whether (D D D... → D) is consistent with all known examples of the operation.
If there are no known examples, add a task to the agenda requesting they be filled in.
If there are examples, and (D D D... → D) is consistent, add it to the Domain/range facet of this operation.
If there are some contradicting examples, create a new concept which is defined as this operation restricted to (D D D... → D).

---

[45] Note that it does make sense if you replace the concept "Insert o Insert" by any other example of a Composition (e.g., "Fill in examples of Set-Union o Set-intersection").

[46] The interested reader will find them all listed in Appendix 3, beginning on page 233

[47] Conditionally. Since each concept is of finite worth, it is allotted a finite amount of space. A random number is generated to decide whether or not to actually insert this example into the Examples facet of the specialization of C. The more that specialized concept is "exceeding its quota", the narrower the range that the random number must fall into to have that new item inserted. The probability is never precisely 1 or 0

Note that this "Checking" rule doesn't just passively check the designated facet; it actively "fixes up" faulty entries, adds new tasks, creates new concepts etc. All the check rules are very aggressive in this way. For example, one entry on No-multiple-elements-structure.Examples.Check will actually remove any multiple occurrences of an element from a structure.

As you might expect, the set Checks(C.F) of all relevant rules for checking facet F of concept C is obtained as (ISA's(C)).F.Check. That is, look for the Check subfacet of the F facet of all the concepts on ISA's(C)). Similarly, Fillins(C.F) is the union of the Fillin subfacets of the F facets of all the concepts on ISA's(C).

When AM chooses a task like "Fillin examples of Primes", its first action is to compute Fillins(Primes.Exs). It does this by asking for ISA's(Primes); that is, a list of all concepts of which Primes is an example. This list is: <Objects Any-concept Anything>. So the relevant heuristics are gathered from Objects.Exs.Fillin, etc. This list of heuristics is then executed, in order (last executed are the heuristics attached to Anything.Exs.Fillin).

It should now be clear what is meant when a concept's facets are listed in the following format:

Name(s)	Frob, Frobnation
•	
•	
Algorithms	A1 A2
Examples	E1 E2 E3 E4 E5 E6
Fillin	Rule1 Rule2
Check	Rule3 Rule4 Rule5
Domain/range	DR1 DR2 DR3
Check	Rule6
Conjecs	C1 C2 C3 C4 C5 C6
Fillin	Rule7 Rule8
Check	Rule9 Rule10
•	
•	

E.g., the entry Rule9 is a heuristic rule which may help to check entries on the Conjecs facet of any Frob[48]. This notation will not be used actually in this document, partly for the benefit of those readers who skip this subsection, partly for consistency between concepts diagrammed before and after this subsection. Rather, all the Fillin heuristics for a concept will be gathered together into what appears to be just one coherent facet. Theoretically, of course, one could organize them that way, with an extra precondition on each Fillin heuristic to indicate which facet it is useful for filling in.

[48] 'Frob' is a nonsense word, a variable identifier which stands for any concept.

5.2.15. Other Facets which were Considered

Most facets (like "Definitions") were anticipated from the very beginning planning of AM, and proved just as useful as expected. Others (like "Intuitions") were also expected to be very important, yet were a serious disappointment. Still others (like "Suggest") were unplanned and grumblingly acknowledged as necessary for the particular LISP program that bears the name AM. Finally, we turn to a few facets which were initially planned, and yet which were adjudged useless around the time that AM was coded. They were therefore never really a part of the LISP program AM, although they figured in its proposal. Let me list them, and explain why each one was dropped.

1. UN-INTERESTINGNESS. This was to be similar to the Interest part. It would contain entries of the form feature/value/reason, where the feature would be a *bad* (dull, trivializing, undesirable, uninteresting) property that an entity (a concept or a task) might possess. If it did, then the value component would return a negative number as its contribution to the worth/priority of that entity. This sounded plausible, but turned out to be useless in practice: (i) There were very few features one could point to which explicitly indicated when something was boring; (ii) Often, a conjunction of many such features would make the entity seem unusual, hence interesting; (iii) Most entities were viewed as very mediocre unless/until specific reasons to the contrary, and in those cases the presence a few boring properties would be outshadowed by the few non-boring ones. In a sea of mediocrity, there is little need to separate the boring from the very boring.

2. JUSTIFICATION. For conjectures which were not yet believed with certainty, this part would contain all the known evidence supporting hem. This would hopefully be convincing, if the user (or a concept) ever wanted to know. In cases of contradictions arising somehow, this facet was to keep hold of the threads that could be untangled to resolve those paradoxes. As described earlier, this duty could naturally be assumed by the Conjecs facet of each concept. The other intended role for this facet was to hold sketches of the proofs of theorems. Unfortunately, the intended concepts for Proof and Absolute truth were never implemented, and thus most of the heuristic rules which would have interacted with this facet are absent from AM. It simply was never needed.

3. RECOGNITION Originally, it was assumed that the location of relevant concepts and their heuristics would be much more like a free-for-all (pandemonium) than an orderly rippling process. As with the original use of BEINGs[49], the expectation was that each concept would have to "shout out" its relevance whenever the activities triggered some recognition predicate inside that concept. Such predicates were to be stored in this facet. But it quickly became apparent that the triggering predicates which were the left-hand-sides of the heuristic rules were quick enough to obviate the need for pre-processing them too heavily. Also, the only rules relevant to a given activity on concept C always seemed to be attainable by rippling in a certain direction away from C. This varied with the activity, and a relatively small table could be written, to specify which direction to ripple in (for any given desired activity). We see that for "Fill-in examples of...", the direction to ripple in is "Generalizations", to locate relevant heuristic rules. For "Judge interest of..." the direction is also generalizations. For "Access specializations of", the

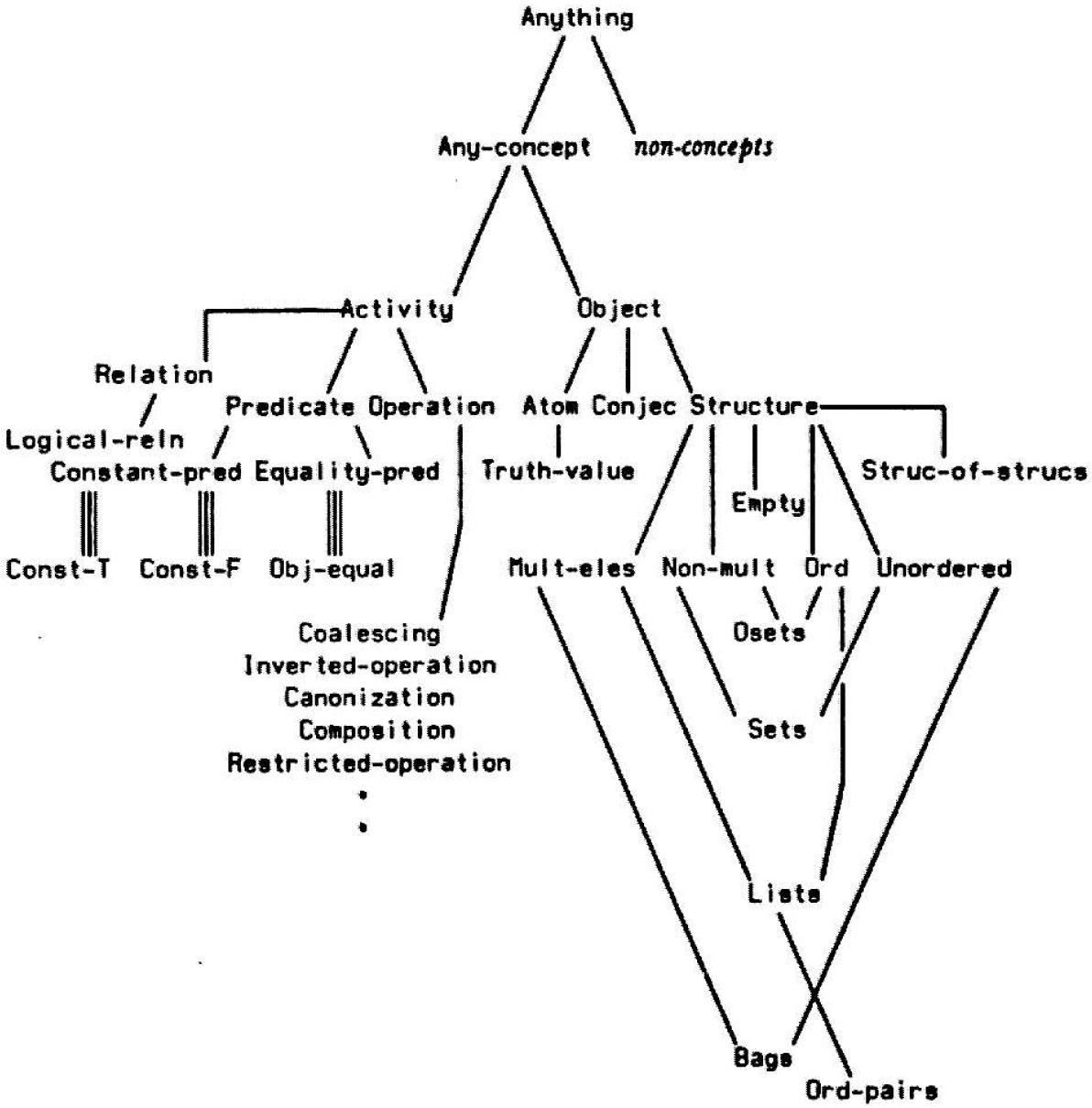
[49] Interacting knowledge modules, each module simulating a different expert at a round-table meeting. See [Lenat 75b].

direction is Specializations, etc. The only important point here is that the Recognition facet was no longer needed.

5.3. AM's Starting Concepts

The first subsection presents a diagram of the top-level (general) concepts AM started with, with the lines indicating the Generalizations/Specializations kinds of relationships (single line links) and a few Examples/Isa's links (triple vertical lines). Several specific concepts have been omitted from that picture. All the concepts initially fed to AM are then listed alphabetically and described in Section 5.3.2. A full facet-by-facet description of each concept is provided in Appendix 2. Finally, Section 5.3.3 discusses the choice of starting concepts.

5.3.1. Diagram of Initial Concepts



The diagram above represents the "topmost" concepts which AM had initially, shown connected via Specialization links (\) and Examples links (||). The only concepts not diagrammed are *examples* of the concept Operation. There are 17 such operations.

Also, we should note that many entities exist in the system which are not themselves concepts. For example, the number "3", though it be an *example* of many concepts, is not itself a concept. All entities which *are* concepts are present on the list called CONCEPTS, and they all have property lists (with facet names as the properties). In hindsight, this somewhat arbitrary scheme is regrettable. A more aesthetic designer might have come up with a more uniform system of representation than AM's.

5.3.2. Summary of Initial Concepts

Since the precise set of concepts is not central to the design of AM, or the quality of behaviors of AM, they are not worth detailing here. On the other hand, a cursory familiarity with their names and definitions should aid the reader in building up an understanding of what AM has done. For that reason, the concepts will now be briefly described, in alphabetical order. This is the same order as concepts are listed on page 173. A fuller description of the concepts is provided in Appendix 2. The ordering within that appendix is different; concepts are grouped together if they are semantically related, by starting at the top of the diagram and meandering downward.

ACTIVITY represents something that can be "performed". All Actives – and *only* Actives – have Domain/range facets and Algorithms facets.

ALL-BUT-FIRST-ELEMENT is an operation which takes an ordered structure and removes the first element from it. It is similar in spirit to the Lisp function "CDR".

ALL-BUT-LAST-ELEMENT takes an ordered structure and removes its last element.

ANY-CONCEPT is useful because it holds all the very general tactics for filling in and checking each facet. The definition of Any-concept is "λ (x) x∈CONCEPTS". 'CONCEPTS' is AM's global list of entities known to be concepts. Initially, this list contains the hundred or so concepts which AM starts with (e.g., all those diagrammed on the preceding page).

ANYTHING is defined as "λ (x) T"; i.e., a predicate which will *always* return true. Notice that the singleton {a} is an example of Anything, but (since it's not on the list **CONCEPTS**) it is not an example of Any-concept.

ATOM contains data about all primitive, indivisible objects (identifiers, constants, variables).

BAG is a type of structure. It is unordered, and multiple occurrences of the same element are permitted. They are isomorphic to the concept known as 'multiset', except that we stipulate that sets are *not* bags.

BAG-DELETE is an operation which takes two arguments, x and B. Although x can be anything, B must be a bag. The procedure is to remove one occurrence of x from B.

BAG-DIFF is an operation which takes two bags B,C. It repeatedly picks a member of C, and removes it (one occurrence of it) from both B and C. This continues until C is empty.

BAG-INSERT is an operation which adds (another occurrence of) x into bag B.

BAG-INTERSECT takes two bags B,C, and creates a new bag D. An item occurs in D the *minimum* number of times it occurs in either B or C.

BAG-UNION takes bag C and dumps all its elements into bag B.

CANONIZE is both an example of and a specialization of 'Operation'. It accepts two predicates P1 and P2 as arguments, both defined over some domain A×A, where P1 is a

generalization of P2. Canonize then tries to produce a "standard representation" for elements of A, in the following way. It creates an operation f from A into A, satisfying: P1(x,y) iff P2(f(x),f(y)). Then any item of the form f(x) is called a canonical member of A. The set of such canonical-A's is worth naming, and it is worth investigating the restrictions of various operations' domains and ranges to this set of canonical-A's[50]. "Canonize" contains lots of information relevant to creating such functions f (given P1 and P2). Thus Canonize is an example of the concept Operation. Canonize also contains information relevant to dealing with any and all such f's. So Canonize is a specialization of Operation.

COALESCE admits the same duality[51]. This very useful operation takes as its argument any operation F(a,b,c,d...), locates two domain components which intersect (preferably, which are equal; say the second and third), and then creates a new operation G defined as G(a,b,d...)≡F(a,b,b,d...). That is, F is called upon with a pair of arguments equal to each other. If F were Times, then G would be Squaring. If F were Set-insert, then G would be the operation of inserting a set S into itself.

COMPOSITION involves taking two operations A and B, and applying them in sequence: A∘B(x)≡A(B(x)). This concept deals with (i) the activity of creating new compositions, given a pair of operations; (ii) all the operations which were created in this fashion. That is why this concept is both a specialization of and an example of Operation.

CONJECTURES are a kind of object. This concept knows about – and can store – conjectures. When proof techniques are inserted into AM, this tiny twig of the tree of concepts will grow to giant proportions.

CONSTANT-PREDICATE is a predicate which can afford to have a very liberal domain: it always ignores its arguments and just returns the same logical value all the time.

DELETE is an operation which contains all the information common to all flavors of removing an element from a structure (regardless of the type of structure which is being attenuated). When called upon to actually perform a deletion, this concept determines the type of structure and then calls the appropriate specialized delete concept (e.g., Bag-delete).

DIFFERENCE is another general operation, which accepts two structures, determines their type (e.g., Bags), and then calls the appropriate specialized version of difference (e.g., Bag-diff).

EMPTY-STRUCTURE contains data relevant to structures with no members.

FIRST-ELEMENT is an operation which takes an ordered structure and returns the first element. It is like the Lisp function 'CAR'.

IDENTITY is just what it claims to be. It takes one argument and returns it immediately. The main purpose of knowing about this boring transformation is just in case some new concept turns out unexpectedly to be equivalent to it.

---

[50] i.e., take an operation which used to have "A" as one of its domain components or as its range, and try to create a new operation with essentially the same definition but whose domain/range says "Canonical-A" instead of "A".

[51] Both a specialization of Operation and an example of Operation.

INSERT takes an item x and a structure S, determines S's type, and calls the appropriate flavor of specialized Insertion concept. The general INSERT concept contains any information common to all of those insertion concepts.

INTERSECT is an operation which computes the intersection of any two structures. It, too, has a separate specialization for Bags, Sets, Osets, and Lists.

INVERT-AN-OPERATION is a very active concept. It can invert any given operation. If $F:X \rightarrow Y$ is an operation, then its inverse will be abbreviated F^{-1}, and $F^{-1}(y)$ is defined as all the x's in X for which $F(x)=y$. The domain and range of F^{-1} are thus the range and domain of F.

INVERTED-OP contains information specific to operations which were created as the inverses of more primitive ones.

LAST-ELEMENT takes an ordered structure and returns its final member.

LIST is a type of structure. It is ordered, and multiple occurrences of the same element are permitted. Lists are also called vectors, tuples, and obags (for "ordered bags").

LIST-DELETE is an operation which takes two arguments, x and B. Although x can be anything, B must be a list. The procedure is to remove the first occurrence of x from B.

LIST-DIFF is an operation which takes two lists B,C. It repeatedly picks a member of C, and removes it (the first remaining occurrence of it) from both B and C. This continues until there are no more members in C.

LIST-INSERT is an operation which adds (another occurrence of) x onto the front of list B. It is like the Lisp function 'CONS'.

LIST-INTERSECT takes two lists B,C, and creates a new list D. An item occurs in D the *minimum* number of times it occurs in either B or C. D is arranged in order as (a sublist of) list B.

LIST-UNION takes list C glues it onto the end of list B. It's like 'APPEND' in Lisp.

LOGICAL-RELATION contains knowledge about Boolean combinations: disjunction, conjunction, implication, etc.

MULTIPLE-ELEMENTS-STRUCTURES are a specialization of Structure. They permit the same atom to occur more than once as a member. (e.g., Bags and Lists)

NO-MULTIPLE-ELEMENTS-STRUCTURES are a specialization of Structure. They permit the same atom to occur only once as a member. (e.g., Sets and Osets)

NONEMPTY-STRUCTURES are a specialization of Structure also. They contain data about all structures which have some members.

OBJECT is a general, static concept. Objects are like the subjects and direct objects in

sentences, while the Actives are like the verbs[52].

OBJECT-EQUALITY is a predicate. It takes a pair of objects, and returns True if (i) they are identical, or (ii) they are structures, and each corresponding pair of members satisfies Object-Equality. Often we'll call this 'Equal', and denote it as '='.

OPERATIONS are Actives which take arguments and return a value. While a predicate examines its arguments and returns either True or False, an operation examines *its* arguments and returns any number of values, of varying types. Assuming that the arguments lay in the domain of the operation (as specified by some entry on its Domain/range facet), then every value returned must lie within its range (as specified by that same Domain/range entry).

ORDERED-PAIR is a kind of List. It has just two 'slots', however: a front and a rear element.

ORDERED-STRUCTURE is a specialized type of Structure. It includes all structures for which the order of insertion of two members can make a difference in whether the structures are equal or not. Ordered-structures are those for which it makes sense to talk about a front and a rear, a first element and a last element.

OSET is a type of structure. It is ordered, and multiple occurrences of the same element are not permitted. The short-term-memory of Newell's PSG [Newell 73] is an Oset, as is a cafeteria line. Not much use was found for this concept by AM.

OSET-DELETE removes x from oset B (if x was in B).

OSET-DIFF is an operation which takes two osets B,C. It removes each member of C from B.

OSET-INSERT is an operation which adds x to the front of oset B. If x was in B previously, it is simply moved to the front of B.

OSET-INTERSECT takes two osets B,C, and removes from B any items which are *not* in C as well. B thus 'induces' the ordering on the resultant oset.

OSET-UNION takes oset C, removes any elements in B already, then glues what's left of C onto the rear of B.

PARALLEL-JOIN is an operation which takes a kind of structure and an operation H. It creates a new operation F, whose domain is that type of structure. For any such structure S, F(S) is computed by appending together H(x) for each member x of S.

PARALLEL-JOIN2 is a similar operation. It creates an operation F with two structural arguments. F(S,L) is computed by appending the values of H(x,L), as x runs through the elements of S.[53]

---

[52] As in English, a particular Activity can sometimes itself be the subject.

[53] Here, the args to PARALLEL-JOIN2 are two types of structures SS and LL, and an operation H whose range is also a structural type DD. Then a new operation is created, with domain SSxLL and range DD.

PARALLEL-REPLACE is an operation used to synthesize new substitution operations. It takes a structural type and an operation H as its arguments, and creates a new operation F. F(S) is computed by simply replacing each member x of S by the value of F(x). The operation produced is very much like the Lisp function MAPCAR.

PARALLEL-REPLACE2 is a slightly more general operation. It creates F, where F(S,L) is computed by replacing each $x \in S$ by F(x,L).

PREDICATES are actives which examine their arguments and then return T or NIL (True or False). It is only due to the capriciousness of AM's initial design that predicates are kept distinct from operations. Of course, each example of an operation can be viewed as if it were a predicate; if $F:A \rightarrow B$ is any operation from A to B, then we can consider F a relation on AxB, that is a subset of AxB, and from there pass to viewing F as a (characteristic) predicate $F:AxB \rightarrow \{T,F\}$. Similarly, any predicate on Ax...xBxC may be considered an operation (a multi-valued, not-always-defined function) from Ax...xB into C. There are no unary predicates. If there were one, say $P:A \rightarrow \{T,F\}$, then that predicate would essentially be a new way to view a certain subset of A; the predicate would then be transformed into $\{a \in A \mid P(a)\}$, made into a new concept, tagged as a specialization of A, and its definition would be "$\lambda(a)$ [A.Defn(a) $\wedge$ P(a)]".

PROJECTION1 is a simple operation. It is defined as λ (x y) x. Notice that Identity is just a specialized restriction of Proj1. Proj1(Me,You)=Me.

PROJECTION2 is a similar operation. It is defined as λ (x y) y.

RELATION is any Active which has been encapsulated into a set of ordered pairs. 'Relation' bridges the gap between active and static concepts.

REPEAT is an operation for generating new operations by repeating old ones. Given as its argument a structural type SS and an existing operation H (with domain and range of the form $SSxSS \rightarrow SS$), Repeat(SS,H) synthesizes a brand new operation F. The domain/range of F is just that of H. F(S) is computed by repeating $TEMP \leftarrow H(x,TEMP)$ for each element x of S. TEMP is initialized as some member (preferably the first element) of S.

REPEAT2 is similar, but requires that H take three arguments, and it creates F, where F(S,L) is gotten by repeatedly doing $TEMP \leftarrow H(x,TEMP,L)$.

RESTRICT is an operation which turns out new operations. Given an argument operation (or predicate) F, the synthesized concept would have the same definition as F, but would have its domain and/or range curtailed.

REVERSE-ORDERED-PAIR transforms the ordered pair <x,y> into <y,x>.

SET is a type of structure. It is unordered, and multiple occurrences of the same element are not permitted.

SET-DELETE is an operation which takes two arguments, x and B. Although x can be anything, B must be a set. The procedure is to remove x from B (if x was in B), then return the resultant value of B.

SET-DIFF is an operation which takes two sets B,C. It removes each member of C from B.

SET-INSERT is an operation which adds x to set B.

SET-INTERSECT removes from set B any items which are *not* in set C, too.

SET-UNION dumps into B all the members of C which weren't in there already.

STRUCTURE, the antithesis of ATOM, is inherently divisible. A structure is something that has members, that can be broken into pieces. There are two questions one can ask about any kind of structure: Is it ordered or not? Can there be multiple occurrences of the same element in it or not? There are four sets of answers to these two questions, and each of the four specifies a well-known kind of structure (Sets, Lists, Osets, Bags).

STRUCTURE-OF-STRUCTURES is a specialization of Structure, representing those structures all of whose *members* are themselves structures.

TRUTH-VALUE is a specialized kind of atomic object. Its only examples are True and False. This concept is the range set for all predicates.

UNION is a general kind of joining operation. It takes two structures and combines them. Four separate variants of this concept are given to AM initially (e.g., Set-union).

UNORDERED-STRUCTURE is a specialized type of Structure. It includes all structures for which the order of insertion of two members never makes any difference in whether the structures are equal or not. Unordered-structures cannot be said to have a front or a rear, a first element or a last element.

5.3.3. Rationale behind Choice of Concepts

A necessary part of realizing AM was to choose a particular set of starting concepts. But how should such a choice be made?

My first impulse was to gather a *complete* set of concepts. That is, a basis which would be sufficient to derive all mathematics. The longer I studied this, the larger the estimated size of this basis grew. It immediately became clear that this would never fit in 256k. [54] One philosophical problem here is that future mathematics may be inspired by some real-world phenomena which haven't even been observed yet. Aliens visiting Earth might have a different mathematics from ours, since their collective life experiences could be quite different from we Terrans.

Scrapping the idea of a sufficient basis, what about a necessary one? That is, a basis which would be *minimal* in the following sense: if you ever removed a concept from that basis, it could never be re-discovered. In isolated cases, one can tell when a basis is *not* minimal: if it contains both addition and multiplication, then it is too rich, since the latter can be

[54] This is the size of the core memory of the computer I had at my disposal.

derived from the former.[55] And yet, the same problem about "absoluteness" cropped up: how can anyone claim that the discovery of X can *never* be made from a given starting point? Until recently, mathematicians didn't realize how natural it was to derive numbers and arithmetic from set theory (a task which AM does, by the way)[56]. So 50 years ago the concepts of set theory and number theory would both have been undisputedly placed into a "minimal" basis. There are thus no absolute conceptual primitives; each culture (perhaps even each individual) possesses its own basis.

Since I couldn't give AM a minimal basis, nor a complete one, I decided AM might as well have a *nice* one. Although it can never be minimal, it should nevertheless be made very small (order of magnitude: 100 concepts). Although it can never be complete, it should suffice for re-discovering much of already-known mathematics. Finally, it should be *rational*, by which I mean that there should be a simple rule for deciding which concepts do and don't belong in that basis.

The concepts AM starts with are meant to be those possessed by young children (age 4, say). This explains some omissions of concepts which would otherwise be considered fundamental: (i) Proof and techniques for proof/disproof; (ii) Abstract properties of relations, like associativity, single-valued, onto; (iii) Cardinality, arithmetic; (iv) Infinity, continuity, limits. The interested reader should see [Piaget 55] or [Copeland 70].

Because my programming time and the PDP-10's memory space were both quite small, only a small percentage of these 'pre-numerical' concepts could be included. Some unjustified omissions are: (i) visual operations, like rotation, coloration; (ii) Games, rules, procedures, strategies, tactics; (iii) Geometric notions, e.g., outside and between.

AM is not supposed to be a model of a child, however. It was never my intention (and it would be much too hard for me) to try to emulate a human child's whimsical imagination and emotive drives. And AM is not ripe for "teaching", as are children.[57] Also, though it possesses a child's ignorance of most concepts, AM is given a large body of sophisticated "adult" heuristics. So perhaps a more faithful image is that of Ramanujan, a brilliant modern mathematician who received a very poor education, and was forced to re-derive much of known number theory all by himself. Incidentally, Ramanujan never did master the concept of formal proof.

There is no formal justification for the particular set of starting concepts. They are all reasonably primitive (sets, composition), and lie several levels "below" the ones which AM managed to ultimately derive (prime factorization, square-root). It might be valuable to attempt a similar automated math discoverer, which began with a very different set of concepts (e.g., start it out as an expert in lattice theory, possessing all known concepts thereof). The converse kind of experiments are to vary the initial base of concepts, and observe the effects on AM's behavior. A few experiments of that form are described in Section 6.2.

---

[55] by AM, and by any mathematician. As Don Cohen points out, if the researcher lacked the proper discovery methods, then he might never derive Times from Plus.

[56] The "new math" is trying to get young children to do this as well, unfortunately, no one showed the elementary-school teachers the underlying harmony, and the results have been saddening.

[57] Learning psychologists might label AM as neo-behavioristic and cognitivistic. See [LeFrancois].

Chapter 6. Results

This chapter opens by summarizing what AM "did". Section 1 gives a fairly high-level description of the major paths which were explored, the concepts discovered along the way, the relationships which were noticed, and occasionally the ones which "should" have been but but weren't.

The next section (6.2) continues this exposition by presenting the results of experiments which were done with (and *on*) AM.

Chapter 7 will draw upon these results – and others given in the appendices – to form conclusions about AM. Several meta-level questions will be tackled there (e.g., "What are AM's limitations?").

6.1. What AM Did

> *Now we have seen that mathematical work is not simply mechanical, that it could not be done by a machine, however perfect. It is not merely a question of applying rules, of making the most combinations possible according to certain fixed laws. The combinations so obtained would be exceedingly numerous, useless, and cumbersome. The true work of the inventor consists in choosing among these combinations so as to eliminate the useless ones or rather to avoid the trouble of making them, and the rules which must guide this choice are extremely fine and delicate. It is almost impossible to state them precisely; they are felt rather than formulated. Under these conditions, how imagine a sieve capable of applying them mechanically?*
>
> -- Poincare'

AM is both a mathematician of sorts, and a big computer program.

By granting AM more anthropomorphic qualities than it deserves, we can describe its progress through elementary mathematics. It rediscovered many well-known concepts, a couple interesting but not-generally-known ones, and several concepts which were hitherto unknown and should have stayed that way. Section 1.3, on page 10, recaps what AM did, much as a historian might critically evaluate Euler's work. A more detailed prose description of everything AM did is found in Appendix 5.1, beginning on page 287.

Instead of repeating any of this descriptive prose here, Section 6.1.1 will provide a very brief listing of what AM did in a single good run, task by task. A much more detailed version of this same list is found in Appendix 5.2, beginning on page 294. The task numbers there correspond to the numbering below[1]. These task-by-task listings are not complete listings of every task AM ever attempted in any of its many runs, but rather a trace of a single, better-than-average run of the program.[2] The reader may wish to consult the brief alphabetized glossary of concept names in the last chapter (page 107), or the more detailed appendix of concept descriptions (following page 173).

Following this linear trace of AM's behavior is a more appropriate representation of what it did: namely, a two-dimensional graph of that same behavior as seen in "concept-space". This forms Section 6.1.2, and is found on page 123.

By under-estimating AM's sophistication, one can demand answers to the typical questions to ask about a computer program: how big is it, how much cpu time does it use, what language it's coded in, etc. These are found in Section 6.1.3.

6.1.1. Linear Task-by-task Summary of a Good Run

1. Fill in examples of Compose. Failed, but suggested next task:
2. Fill in examples of Set-union. Also failed, but suggested:
3. Fill in examples of Sets. Many found (e.g., by instantiating Set.Defn) and then more derived from those examples (e.g., by running Union.Alg).
4. Fill in specializations of Sets (because it was very easy to find examples of Sets). Creation of new concepts. One, INT-Sets, is related to "Singletons". Another, "BI-Sets", is all nests of braces (no atomic elements).
5. Fill in examples of INT-Sets. This indirectly led to a rise in the worth of Equal.
6. Check all examples of INT-Sets. All were confirmed. AM defines the set of Nonempty INT-Sets; this is renamed "Singletons" by the user.
7. Check all examples of Sets. To check a couple conjectures, AM will soon look for Bags and Osets.
8. Fill in examples of Bags.
9. Fill in specializations of Bags. Created INT-Bags (contain just one kind of element), and BI-Bags (nests of parentheses).
10. Fill in examples of Osets.
11. Check examples of Osets.
12. Fill in examples of Lists.
13. Check examples of Lists.
14. Fill in examples of All-but-first.
15. Fill in examples of All-but-last.
16. Fill in specializations of All-but-last. Failed.

---

[1] They do *not* precisely match the task numbers accompanying the example given in Chapter 2.

[2] In fact, it is perhaps the best overall run. It occurred in two stages (due to space problems; unimportant). In this particular run, AM misses the few "very best" discoveries it ever made, since the runs they occurred in went in somewhat different directions. It also omits some of the more boring tasks: see, e.g., the description of task number 69.

17. Fill in examples of List-union.
18. Fill in examples of Proj1.
19. Check examples of All-but-first.
20. Check examples of All-but-last.
21. Fill in examples of Proj2.
22. Fill in examples of Empty-structures. 4 found.
23. Fill in generalizations of Empty-structures. Failed.
24. Check examples of List-union.
25. Check examples of Bags. Defined Singleton-bags.
26. Fill in examples of Bag-union.
27. Check examples of Proj2.
28. Fill in examples of Set-union.
29. Check examples of Set-union. Define λ (x,y) x∪y=x, later called Superset.
30. Fill in examples of Bag-insert.
31. Check examples of Bag-insert. Range is really Nonempty bags. Isolate the results of insertion restricted to Singletons: call them Doubleton-bags.
32. Fill in examples of Bag-intersect.
33. Fill in examples of Set-insert.
34. Check examples of Set-insert. Range is always Nonempty sets. Define λ (x,S) Set-insert(x,S)=S; i.e., set membership. Define Doubleton sets.
35. Fill in examples of Bag-delete.
36. Fill in examples of Bag-difference.
37. Check examples of Bag-intersect. Define λ (x,y) x∩y={}; i.e. disjoint bags.
38. Fill in examples of Set-intersect.
39. Check examples of Set-intersect. Define λ (x,y) x∩y=x; i.e., subset. Also define disjoint sets: λ (x,y) x∩y={}.
40. Fill in examples of List-intersect.
41. Fill in examples of Equal. Very difficult to find examples; this led to:
42. Fill in generalizations of Equal. Define "Same-size", "Equal-CARs", and some losers.
43. Fill in examples of Same-size.
44. Apply an Algorithm for Canonize to the args Same-size and Equal. AM eventually synthesizes the canonizing function "Size". AM defines the set of canonical structures: bags of T's; this later gets renamed as "Numbers".
45. Restrict the domain/range of Bag-union. A new operation is defined, Number-union, with domain/range entry <Number Number → Bag>.
46. Fill in examples of Number-union. Many found.
47. Check the domain/range of Number-union. Range is 'Number'. This operation is renamed "Add2".
48. Restrict the domain/range of Bag-intersect to Numbers. Renamed "Minimum".
49. Restrict the domain/range of Bag-delete to Numbers. Renamed "SUB1".
50. Restrict the domain/range of Bag-insert to Numbers. AM calls the new operation "Number-insert". Its domain/range entry is <Anything Number → Bag>.
51. Check the domain/range of Number-insert. This doesn't lead anywhere.
52. Restrict the domain/range of Bag-difference to Numbers. This becomes "Subtract".
53. Fill in examples of Subtract. This leads to defining the relation LEQ (≤).[3]
54. Fill in examples of LEQ. Many found.

---

[3] If a larger number is "subtracted" from a smaller, the result is zero. AM explicitly defines the set of ordered pairs of numbers having zero "difference". <x,y> is in that set iff x is less than or equal to y.

55. Check examples of LEQ.
56. Apply algorithm of Coalesce to LEQ. LEQ(x,x) is Constant-True.
57. Fill in examples of Parallel-join2. Included is Parallel-join2(Bags,Bags,Proj2), which is renamed "TIMES", and Parallel-join2(Structures,Structures,Proj1), a generalized Union operation renamed "G-Union", and a bunch of losers.
58. – 69. Fill in and check examples of the operations just created.
70. Fill in examples of Coalesce. Created: Self-Compose, Self-Insert, Self-Delete, Self-Add, Self-Times, Self-Union, etc. Also: Coa-repeat2, Coa-join2, etc.
71. Fill in examples of Self-Delete. Many found.
72. Check examples of Self-Delete. Self-Delete is just Identity-op.
73. Fill in examples of Self-Member. No positive examples found.
74. Check examples of Self-Member. Self-member is just Constant-False.
75. Fill in examples of Self-Add. Many found. User renames this "Doubling".
76. Check examples of Coalesce. Confirmed.
77. Check examples of Add2. Confirmed.
78. Fill in examples of Self-Times. Many found. Renamed "Squaring" by the user.
79. Fill in examples of Self-Compose. Defined Squaring∘Squaring. Created Add∘Add (two versions: Add21 which is λ (x,y,z) (x+y)+z, and Add22 which is x+(y+z)). Similarly, two versions of Times∘Times and of Compose∘Compose.
80. Fill in examples of Add21. (x+y)+z. Many are found.
81. Fill in examples of Add22. x+(y+z). Again many are found.
82. Check examples of Squaring. Confirmed.
83. Check examples of Add22. Add21 and Add22 appear equivalent. But first:
84. Check examples of Add21. Add21 and Add22 still appear equivalent. Merge them. So the proper argument for a generalized "Add" operation is a Bag.
85. Apply algorithm for Invert to argument 'Add'. Define Inv-add(x) as the set of all bags of numbers (>0) whose sum is x. Also denoted $Add^{-1}(x)$.
86. Fill in examples of TIMES21. (xy)z. Many are found.
87. Fill in examples of TIMES22. x(yz). Again many are found.
88. Check examples of TIMES22. TIMES21 and TIMES22 may be equivalent.
89. Check examples of TIMES21. TIMES21 and TIMES22 still appear equivalent. Merge them. So the proper argument for a generalized "TIMES" operation is a Bag. Set up an analogy between TIMES and ADD, because of this fact.
90. Apply algorithm for Invert to argument 'TIMES'. Define Inv-TIMES(x) as the set of all bags of numbers (>1) whose product is x. Analogic to Inv-Add.
91. Fill in examples of Parallel-replace2. Included are Parallel-replace2(Bags,Bags,Proj2) (called MR2-BBP2), and many losers.
92. – 107. Fill in and check examples of the operations just created.
108. Fill in examples of Compose. So easy that AM creates Int-Compose.
109. Fill in examples of Int-Compose. The two chosen operations G,H must be such that ran(H)⊂dom(G), and ran(G)⊂dom(H); both G and H must be interesting. Create G-Union∘MR2-BBP2,[4] Insert∘Delete, Times∘Squaring, etc.
110. – 127. Fill in and check examples of the compositions just created. Notice that G-Union∘MR2-BBP2 is just TIMES.
128. Fill in examples of Coa-repeat2. Among them: Coa-repeat2(Bags-of-Numbers, Add2) [multiplication again!], Coa-repeat2(Bags-of-Numbers, Times)

---

[4] an alternate derivation of the operation of multiplication

[exponentiation], Coa-repeat2(Structures, Proj1) [CAR], Coa-repeat2(Structures, Proj2) [Last-element-of], etc.

129. Check the examples of Coa-repeat2. All confirmed.
130. Apply algorithms for Invert to 'Doubling'. The result is called "Halving" by the user. AM then defines "Evens".
131. Fill in examples of Self-Insert.
132. Check examples of Self-Insert. Nothing special found.
133. Fill in examples of Coa-repeat2-Add2.
134. Check examples of Coa-repeat2-Add2. It's the same as TIMES.
135. Apply algorithm for Invert to argument 'Squaring'. Define "Square-root".
136. Fill in examples of Square-root. Some found, but very inefficiently.
137. Fill in new algorithms for Square-root. Had to ask user for a good one.
138. Check examples of Square-root. Define the set of numbers "Perfect-squares".
139. Fill in examples of Coa-repeat2-Times. This is exponentiation.
140. Check examples of Coa-repeat2-Times. Nothing special noticed, unfortunately.
141. Fill in examples of Inv-TIMES. Many found, but inefficiently.
142. Fill in new algorithms for Inv-TIMES. Obtained opaquely from the user.
143. Check examples of Inv-TIMES. This task suggests the next one:
144. Compose G-Union with Inv-TIMES. Good domain/range. Renamed "Divisors".
145. Fill in examples of Divisors. Many found, but not very efficiently.
146. Fill in new algorithms for Divisors. Obtained from the user.
147. Fill in examples of Perfect-squares. Many found.
148. Fill in specializations of TIMES. Times1(x)=1•x, Times2(x)=2x, Times-sq is TIMES with its domain restricted to bags of perfect squares, Times-ev takes only even arguments, Times-to-evens requires that the result be even, Times-to-sq, ...
149. Check examples of Divisors. Define 0-Div, 1-Div, 2-Div, and 3-Div, the sets of numbers whose Divisors value is the empty set, a singleton, a doubleton, and a tripleton, respectively.
150. Fill in examples of 1-Div. Only one example found: "1". Lower 1-Div.Worth.
151. Fill in examples of 0-Div. None found. Lower the worth of this concept.
152. Fill in examples of 2-Div. A nice number are found. Raise 2-Div.Worth.
153. Check examples of 2-Div. All confirmed, but no pattern noticed.
154. Fill in examples of 3-Div. A nice number found.
155. Check examples of 3-Div. All confirmed. All are perfect squares.
156. Restrict Square-root to numbers which are in 3-Div. Call this Root3.
157. Fill in examples of Root3. Many found.
158. Check examples of Root3. All confirmed. All are in 2-Div. Raise their worths.
159. Restrict Squaring to 2-divs. Call the result Square2.
160. Fill in examples of Square2. Many found.
161. Check the range of Square2. Always 3-Divs. Conjecture: x has 2 divisors iff x^2 has 3 divisors.
162. Restrict Squaring to 3-Divs. Call the result Square3.
163. Restrict Square-rooting to 2-Divs. Call the result Root2.
164. Fill in examples of Square3. Many found.
165. Compose Divisors-of and Square3. Call the result Div-Sq3.
166. Fill in examples of Div-Sq3. Many found.
167. Check examples of Div-Sq3. All such examples are Same-size.
168. – 175. More confirmations and explorations of the above conjecture. Gradually, all its ramifications lead to dead-ends (as far as AM is concerned).
176. Fill in examples of Root2. None found. Conjecture that there are none.

177. Check examples of Inv-TIMES. Inv-TIMES always contains a singleton bag, and always contains a bag of primes.
178. Restrict the range of Inv-TIMES to bags of primes. Call this Prime-Times.
179. Restrict the range of Inv-TIMES to singletons. Called Single-Times.
180. Fill in examples of Prime-times. Many found.
181. Check examples of Prime-times. Always a singleton set. User renames this conjecture "The unique factorization theorem".
182. Fill in examples of Single-TIMES. Many found.
183. Check examples of Single-TIMES. Always a singleton set. Single-TIMES is actually the same as Bag-insert!
184. Fill in examples of Self-set-union. Many found.
185. Check examples of Self-set-union. This operation is same as Identity.
186. Fill in examples of Self-bag-union. Many found.
187. Check examples of Self-bag-union. Confirmed. Nothing interesting noticed.
188. Fill in examples of Inv-ADD.
189. Check examples of Inv-ADD. Hordes of boring conjectures, so:
190. Restrict the domain of Inv-ADD to primes (Inv-Add-primes), to evens (Inv-Add-evens), to squares, etc.
191. Fill in examples of Inv-add-primes. Many found.
192. Check examples of Inv-add-primes. Confirmed, but nothing special noticed.
193. Fill in examples of Inv-add-evens. Many found.
194. Check examples of Inv-add-evens. Always contains a bag of primes.
195. Restrict the range of Inv-Add-evens to bags of primes. Called Prime-ADD.
196. Restrict the range of Inv-ADD to singletons. Call that new operation Single-ADD.
197. Fill in examples of Prime-ADD. Many found.
198. Check examples of Prime-ADD. Always a nonempty set (of bags of primes). User renames this conjecture "Goldbach's conjecture".
199. Fill in examples of Single-ADD. Many found.
200. Check examples of Single-ADD. Always a singleton set. This operation is the same as Bag-insert and Single-TIMES.
201. Restrict the range of Prime-ADD to singletons, by analogy to Prime-TIMES.[5] Call the new operation Prime-ADD-SING.
202. Fill in examples of Prime-ADD-SING. Many found.
203. Check examples of Prime-ADD-SING. Nothing special noticed.
204. Fill in examples of Times-sq.[6] Many examples found.
205. Check domain/range of Times-sq. Is the range actually Perfect-squares? Yes!
206. Fill in examples of Times1. Recall that Times1(x)=TIMES(1,x).
207. Check examples of Times1. Apparently just a restriction of Identity.
208. Check examples of Times-sq. Confirmed.
209. Fill in examples of Times0.
210. Fill in examples of Times2.
211. Check examples of Times2. Apparently the same as Doubling. That is, $x*x=2*x$. Very important. By analogy, define Ad2(x) as $x*2$.
212. Fill in examples of Ad2.
213. Check examples of Ad2. Nothing interesting noticed.

---

[5] In this case, AM is asking which numbers are uniquely representable as the sum of two primes.

[6] Recall that this is just TIMES restricted to operate on perfect squares.

214. Fill in specializations of Add. Among those created are: Add0 (x+0), Add1, Add3, ADD-sq (addition restricted to perfect squares), Add-ev (sum of even numbers), Add-pr (sum of primes), etc.
215. Check examples of Times0. The value always seems to be 0.
216. Fill in examples of Times-ev.[7] Many examples found.
217. Check examples of Times-ev. Apparently all the results are Evens.
218. Fill in examples of Times-to-ev.[8] Many found.
219. Fill in examples of Times-to-sq. Only a few found.
220. Check examples of Times-to-sq. All arguments always seem to be squares. Conjec: Times-to-sq is really the same as Times-sq. Merge the two. This is a false conjecture, but did AM no harm.
221. Check examples of Times-to-ev. The domain always contains an even number.
222. Fill in examples of Self-Union.
223. Check examples of Self-Union.
224. Fill in examples of SubSet.
225. Check example of SubSet.
226. Fill in examples of SuperSet.
227. Check examples of SuperSet. Conjec: Subset(x,y) iff Superset(y,x). Important.
228. Fill in examples of Compose∘Compose-1. AM creates some explosive combination (e.g., (Compose∘Compose)∘(Compose∘Compose)∘(Compose∘Compose)), some poor ones (e.g., Square∘Count∘ADD^{-1}), and even a few – very few – winners (e.g., SUB1∘Count∘Self-Insert).
229. Check examples of Compose∘Compose-1.
230. Fill in examples of Compose∘Compose-2.[9] AM recreates many of the previous tasks' operations.
231. Check examples of Compose∘Compose-2. Nothing noticed yet[10].
232. – 252. Fill in and check examples of the losing compositions just created.
253. Fill in examples of Add-sq (i.e., sum of squares).
254. Check domain/range entries of Add-sq. The range is not always perfect squares. Define Add-sq-sq(x,y), which is True iff x and y are perfect squares and their sum is a perfect square as well.
255. Fill in examples of Add-pr; i.e., addition of primes.
256. Check Domain/range entries of Add-pr. AM defines the set of pairs of primes whose sum is also a prime. This is a bizarre derivation of prime pairs.

---

[7] Recall that Times-ev is just like TIMES restricted to operating on even numbers.

[8] That is, consider bags of numbers which multiply to give an even number.

[9] Recall that the difference between this operation and the last one is merely in the order of the composing: Fo(GoH) versus (FoG)oH.

[10] Later on, AM will use these new operations to discover the associativity of Compose.

6.1.2. Two-Dimensional Behavior Graph

On the next two pages is a graph of the same "best run" which AM executed. The nodes are concepts, and the links are actions which AM performed. Labels on the links indicate when each action was taken, so the reader may observe how AM jumped around. It should also easy to perceive from the graph which paths of development were abandoned, which concepts ignored, and which ones concentrated upon. These are precisely the features of AM's behavior which are awkward to infer from a simple linear trace (as in the previous section).

In more detail, here is how to read the graph: Each node is a concept. To save space, these names are often highly abbreviated. For example, "x0" is used in place of "TIMES-0".

Each concept name is surrounded by from zero to four numbers:

```
318     288
FROBNATION
310     291
```

The upper right number indicates the task number (see last section) during which examples of this concept were filled in. The lower right number tells when they were checked. The upper left number indicates when the Domain/range facet of that concept was modified. Finally, the lower left number is the task number during which some new Algorithms for that concept were obtained. A number in parentheses indicates that the task with that number was a total failure.

Because of the limited space, it was decided that if a concept were ever renamed by the user, then only that newer, mnemonic name would be given in the diagram. Thus there is an arrow from "Coalesce" to "*Square*", an operation originally called "Self-Times" by AM.

Sometimes, a concept will have under it a note of the form =GROK. This simply means that AM eventually discovered that the concept was equivalent to the already-known concept "Grok", and probably forgot about this one (merged it into the one it already knew about). The "trail" of discovery may pick up again at that pre-existing concept. A node written as =GROK means that the concept was really the same as "Grok", but AM never investigated it enough to notice this.

Each node may have an arrow leading into it, and any number of arrows emanating from it. The arrows indicate the creation of new concepts. Thus an arrow leading to concept "Frobnate" indicates how that concept was created. An arrow directed away from Frobnate points to a concept created as, e.g., a specialization or an example of Frobnate. No arrowheads are in practice necessary: all arrows are directed *downwards*.

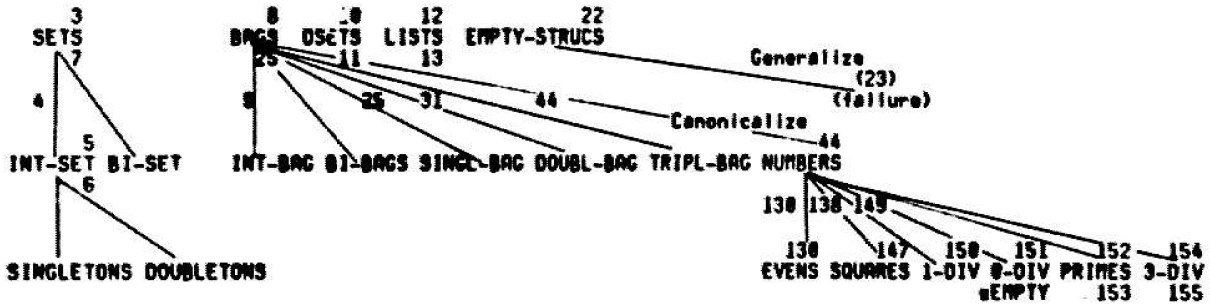
The arrows may be labelled, indicating precisely what they represent (e.g., composition, restriction) and what the task number was when they occurred. For space reasons, the following convention has proven necessary: if an arrow emanating from C is un-numbered, it is assumed to have occurred at the same time as the arrow to its immediate left which also points from C; if all the arrows emanating from C have no number, than all their times of

occurrence are assumed to be the *lower right*[11] number of C. Finally, if C has no lower right number, the arrow is assumed to have the value of the upper right number of C.

An unlabelled arrow is assumed to be an act of Specialization or the creation of an Example.[12] Labels, when they do occur, are given in capitals and small letters; concept names (nodes) are by contrast in all capitals.

All the numbers correspond to those given to the tasks in the task-by-task traces presented in the last section (p. 115) and in Appendix 5 (p. 294).

The first part of this graph (presented below) contains static structural (and ultimately numerical) concepts which were studied by AM:

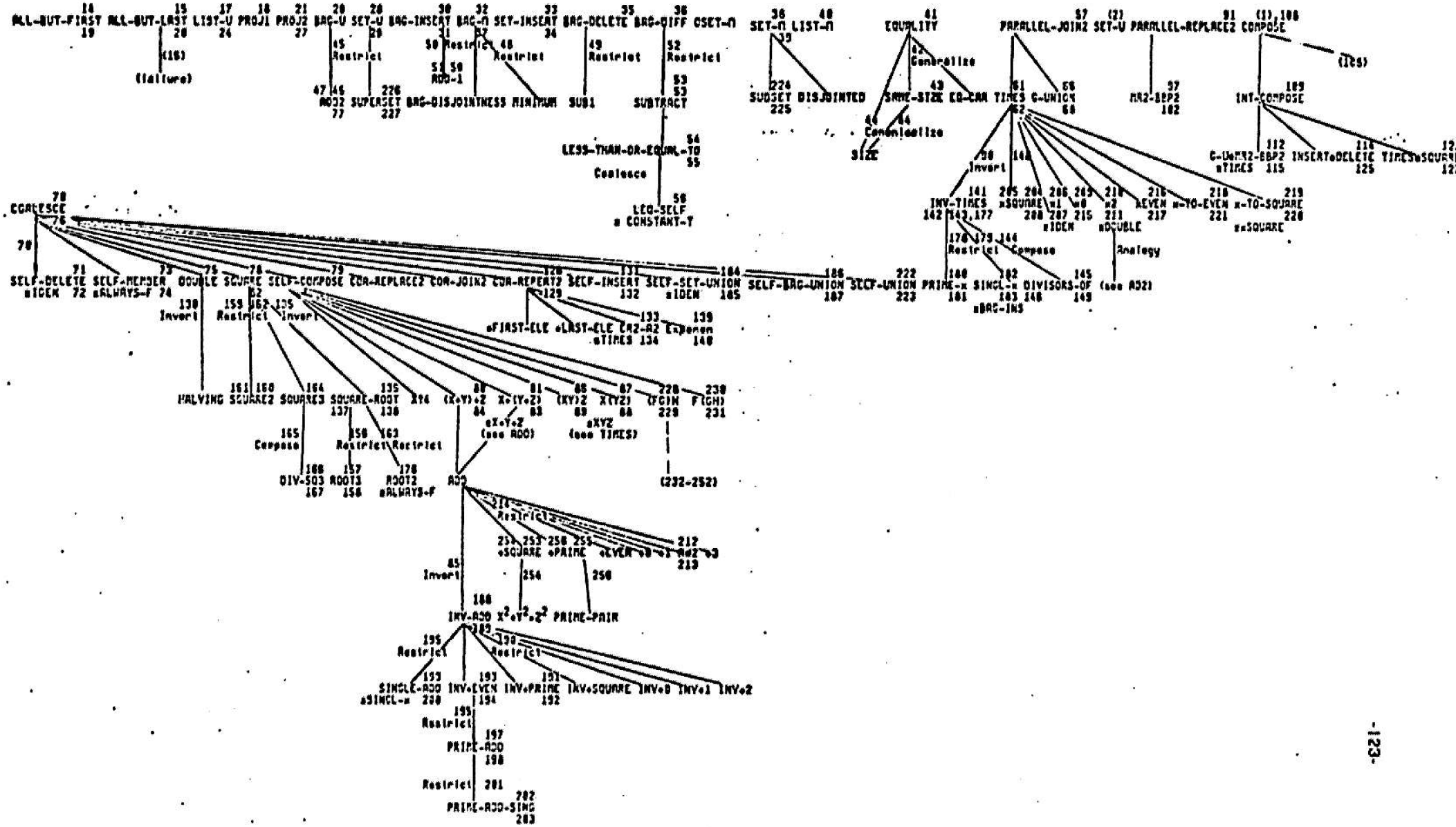


The rest of the graph (presented on the next page) deals with activities which were investigated:

[11] This is often true because many concepts are created while checking examples of some known concept.

[12] It should be clear in each context which is happening. If not, refer to the short trace in the preceding section, and look up the appropriate task number.

ACTIVED


ALL-BUT-FIRST
ALL-BUT-LAST
LIST-U
PROJ1
PROJ2
BAG-U
SET-U
BAG-INSERT
BAG-M
SET-INSERT
BAG-DELETE
BAG-DIFF
OSET-M
SET-M
LIST-M
EQUALITY
PARALLEL-JOIN2
SET-U
PARALLEL-REPLACE2
COMPOSE
(1),108
(16)
(failure)
Restrict
Restrict
Restrict
Restrict
Restrict
ADD2
SUPERSET
BAG-DISJOINTNESS
MINIMUM
SUB1
SUBTRACT
SUBSET
DISJOINTED
Generalize
SAME-SIZE
EQ-CAR
TIMES
G-UNION
MR2-BBP2
INT-COMPOSE
(105)
Canonicalize
SIZE
LESS-THAN-OR-EQUAL-TO
Coalesce
LEQ-SELF
x CONSTANT-T
Invert
INV-TIMES
xSQUARE
xIDEN
xDOUBLE
xEVEN
x-TO-EVEN
x-TO-SQUARE
xxSQUARE
Restrict
Compose
Analogy
PRIME-x
SINGL-x
DIVISORS-OF
(see ADD2)
xBAG-INS
COALESCE
SELF-DELETE
xIDEN
SELF-MEMBER
xALWAYS-F
DOUBLE
SQUARE
SELF-COMPOSE
COA-REPLACE2
COA-JOIN2
COA-REPEAT2
SELF-INSERT
SELF-SET-UNION
xIDEN
SELF-BAG-UNION
SELF-UNION
xFIRST-ELE
xLAST-ELE
CR2-A2
Expansion
xTIMES
Invert
Restrict
Invert
HALVING
SQUARE2
SQUARE3
SQUARE-ROOT
X+4
(X+Y)+Z
X+(Y+Z)
(XY)Z
X(YZ)
(FGH)
F(GH)
xX+Y+Z
(see ADD)
xXYZ
(see TIMES)
Compose
Restrict
Restrict
DIV-SQ3
ROOT3
ROOT2
xALWAYS-F
ADD
(232-252)
Restrict
xSQUARE
xPRIME
xEVEN
Invert
INV-ADD
X²+Y²+Z²
PRIME-PAIR
Restrict
Restrict
SINGLE-ADD
xSINGL-x
INV+EVEN
INV+PRIME
INV+SQUARE
INV+0
INV+1
INV+2
Restrict
PRIME-ADD
Restrict
PRIME-ADD-SING

6.1.3. AM as a Computer Program

When viewed as a large LISP program, there is very little of interest about AM. There are the usual battery of customized functions (e.g., a conditional PRINT function), the storage hacks (special emergency garbage collection routines, which know which facets are expendible), the time hacks (omnisciently arrange clauses in a conjunction so that the one most likely to fail will come first), and the bugs (if the user renames a concept while it's the current one being worked on, there is a 5% chance of AM entering an infinite loop).

Below are listed a few parameters of the system, although I doubt that they hold any theoretical significance. The reader may be curious about how big AM, how long it takes to execute, etc.

Machine: SUMEX, PDP-10, KI-10 uniprocessor, 256k core memory.

Language: Interlisp, January '75 release, which occupies 140k of the total 256k, but which provides a surplus "shadow space" of 256k additional words available for holding compiled code.

AM support code: 200 compiled (not block-compiled) utility routines, control routines, etc. They occupy roughly 100k, but all are pushed into the shadow space.

AM itself: 115 concepts, each occupying about .7k (about two typed pages, when Pretty-printed with indentation). Facet/entries stored as property/value on the property list of atoms whose names are concepts' names.[13] Each concept has about 8 facets filled in.

Heuristics are tacked onto the facets of the concepts. The more general the concept, the more heuristic rules it has attached to it.[14] "Any-concept" has 121 rules; "Active concept" has 24; "Coalesce" has 7; "Set-Insertion" has none. There are 250 heuristic rules in all, divided into 4 flavors (Fillin, Check, Suggest, Interestingness). Although the mean number of rules is therefore only about 2.2 (i.e., less than 1 of each flavor) per concept, the standard deviation of this is a whopping 127.4. The average number of heuristics (of a given flavor) encountered rippling upward from a randomly-chosen concept C (along the network of generalization links) is about 35, even though the mean path length is only about 4.[15]

The total number of jobs executed in a typical run (from scratch) is about 200. The run ends because of space problems, but AM's performance begins to degrade near the end anyway.

"Final" state of AM: 300 concepts, each occupying about 1k. Many are swapped out onto

---

[13] Snazzy feature: Executable entries on facets (e.g., an entry on UnionAlg) are stored uncompiled until the first time they are actually called on, at which time they are compiled and then executed

[14] This was not done consciously, and may or may not hold some theoretical significance.

[15] If the heuristics were homogeneously distributed among the concepts, the number of heuristics (of a given type) along a typical path of length 4 would only be about 2, not 35. If all the heuristics were tacked onto Anything and Any-concept, the number encountered in any path would be 75, not 35.

disk. Number of winning concepts discovered: 25 (estimated). Number of acceptable concepts defined: 100 (est.).[16] Number of losing concepts unfortunately worked on: 60 (est.). The original 115 concepts have grown to an average size of 2k. Each concept has about 11 facets filled in.

About 30 seconds of cpu time were allocated to each task, on the average, but the task typically used only about 18 seconds before quitting. Total CPU time for a run is about 1 hour. Total cpu time consumed by this research project was about 500 cpu hours.

Real time: about 1 minute per task, 2 hours per run. The idea for AM was formulated in the Fall of 1974, and AM was coded in the summer of 1975. Total time consumed by this project to date has been about 2500 man-hours: 700 for planning, 500 for coding, 600 for modifying and debugging and experimenting, and 700 for writing this thesis.

6.2. Experiments with AM

Now we've described the activities AM carried out during its best run. AM was working by itself, and each time executed the top task on the agenda. It received no help from the user, and all its concepts' Intuitions facets had been removed.

One valuable aspect of AM is that it is amenable to many kind of interesting experiments. Although AM is too *ad hoc* for numerical results to have much significance, the qualitative results perhaps do have some valid things to say about research in elementary mathematics, about automating research, and at least about the efficacy of various parts of AM's design.

This section will explain what it means to perform an experiment on AM, what kinds of experiments are imaginable, which of those are feasible, and finally will describe the many experiments which were performed on AM.

By modifying AM in various ways, its behavior can be altered, and the *quality* of its behavior will change as well. As a drastic example, one experiment involved forcing AM to select the next task to work on *randomly* from the agenda, not the top task each time. Needless to say, the performance was very different from usual.

By careful planning, each experiment can tell us something new about AM: how valuable a certain piece of it is, how robust a certain scheme really is, etc. The results of these experiments would then have something to contribute to a discussion of the "real intelligence" of AM (e.g., what features were superfluous), and contribute to the design of the "next" AM-like system. Generalizing from those results, one might suggest some hypotheses about the larger task of automated math research.

Let's cover the different *kinds* of experiments one could perform on AM:

(i) Remove individual concept modules, and/or individual heuristic rules. Then examine

---

[16] For a list of most of the 'winners' and 'acceptables', see the final section in Appendix 2, page 224.

how AM's performance is degraded. AM should operate even if most of its heuristic rules and most of its concept modules were excised. If the remaining fragment of AM is too small, however, it may not be able to find anything interesting to do. In fact, this situation was actually encountered experimentally, when the first few partially complete concepts were inserted. If only a little bit of AM is removed, the remainder will in fact keep operating without this "uninteresting collapse". The converse situation should also hold: although still functional with any concept module unplugged, AM's performance *should* be noticeably degraded. That is, while not indispensable, each concept should nontrivially help the others. The same holds for each individual heuristic rule. When a piece of AM is removed, which concepts does AM then "miss" discovering? Is the removed concept/heuristic later discovered anyway by those which are left in AM? This should indicate the importance of each kind of concept and rule which AM starts with.

(ii) Vary the relative weights given to features by the criteria which judge aesthetics, interestingness, worth, utility, etc. See how important each factor is in directing AM along successful routes. In other words, vary the little numbers in the formulae (both the global priority-assigning formula and the local reason-rating ones inside heuristic rules). One important result will be some idea of the robustness or "toughness" of the numeric weighting factors. If the system easily collapses, it was too finely tuned to begin with.

(iii) Add several new concept modules (including new heuristics relevant to them) and see if AM can work in some unanticipated field of mathematics (like graph theory or calculus or plane geometry). Do earlier achievements – concepts and conjectures AM synthesized already – have any impact in the new domain? Are some specialized heuristics from the first domain totally wrong here? Do all the old general heuristics still hold here? Are they sufficient, or are some "general" heuristics needed here which weren't needed before? Does AM "slow down" as more and more concepts get introduced?

(iv) Try to have AM develop nonmathematical theories (like elementary physics, or program verification). This might require limiting AM's freedom to "ignore a given body of data and move on to something more interesting". The exploration of very non-formalizable fields (e.g., politics) might require much more than a small augmentation of AM's base of concepts. For some such domains, the "Intuitions" scheme, which had to be abandoned for math, might prove valid and valuable.

(v) Add several new concepts dealing with proof, and of course add all the associated heuristic rules. Such rules would advise AM on the fine points of using various techniques of proof/disproof: when to use them, what to try next based on why the last attempt failed, etc. See if the *kinds* of discoveries AM makes are increased.

Just prior to the writing of this document, several experiments (of types i, ii, and iii above[17]) were set up and performed on AM. We're now ready to examine each of them in detail. The following points are covered for each experiment:

1. How was it thought of?
2. What will be gained by it? What would be the implications of the various possible outcomes?

---

[17] experiments of type (iv) weren't tried and are left as "open problems", as invitations for future research efforts. Experiment (v) will probably be carried out this year (1976).

3. How was the experiment set up? What preparations/modifications had to be made? How much time (man-hours) did it take?
4. What happened? How did AM's behavior change? Was this expected? Analysis.
5. What was learned from this experiment? Can we conclude anything which suggests new experiments (e.g., use a better machine, a new domain) or which bears on a more general problem that AM faced (e.g., a new way to teach math, a new idea about doing math research)?

6.2.1. Must the Worth numbers be finely tuned?

Each of the 115 initial concepts has supplied to it (by the author) a number between 0 and 1000, stored as its Worth facet, which is supposed to represent the overall value of the concept. "Compose" has a higher initial Worth than "Structure-delete", which is higher than "Equality"[18].

Frequently, the priority of a task involving C depends on the overall Worth of C. How sensitive is AM's behavior to the initial settings of the Worth facets? How finely tuned must these initial Worth values be?

This experiment was thought of because of the 'brittleness' of many other AI systems, the amount of fine tuning needed to elicit coherent behavior. For example, see the discussion of limitations of PUP6, in [Lenat 75b]. The author believed that AM was very resilient in this regard, and that a demonstration of that fact would increase credibility in the power of the ideas which AM embodies.

To test this, a simple experiment was performed. Just before starting AM, the mean value of all concepts' Worth values was computed. It turned out to be roughly 200. Then each concept had its Worth reset to the value 200.[19] This was done "by hand", by the author, in a matter of seconds. AM was then started and run as if there were nothing amiss, and its behavior was watched carefully.

What happened? By and large, the same major discoveries were made – and missed – as usual, in the same order as usual. But whereas AM proceeded fairly smoothly before, with little superfluous activity, it now wandered quite blindly for long periods of time, especially at the very beginning. Once AM "hooked into" a line of productive development, it followed it just as always, with no noticeable additional wanderings. As one of these lines of developments died out, AM would wander around again, until the next one was begun.

It took roughly three times as long for each major discovery to occur as normal. This "delay" got shorter and shorter as AM developed further. In each case, the tasks preceding the discovery and following it were pretty much the same as normal; only the tasks "between" two periods of development were different – and much more numerous. The precise numbers involved would probably be more misleading than helpful, so they will not

---

[18] As AM progresses, it notices something interesting about Equality every now and then, and pushes its Worth value upwards.

[19] The initial spread of values was from 100 to 600.

be given[20].

The reader may be interested to learn that the Worth values of many of the concepts – and most of the new concepts – ended up very close to the same values that they achieved in the original run. Overrated concepts were investigated and proved boring; underrated concepts had to wait longer for their chances, but then quickly proved interesting and had their Worth facets boosted.

The conclusion I draw from this change in behavior is that the Worth facets are useful for making blind decisions – where AM must choose based only on the overall worths of the various concepts in its repertoire. Whenever a specific reason existed, it was far more influential than the "erroneous" Worth values. The close, blind, random decisions occur between long bursts of specific-reason-driven periods of creative work.[21]

The general answer, then, is *No*, the initial settings of the Worth values are not crucial. Guessing reasonable initial values for them is merely a time-saving device. This suggests an interesting research problem: what impact does the quality of initial starting values have on humans? Give several bright undergraduate math majors the same set of objects and operators to play with, but tell some of them (i) nothing, and some of them (ii) a certain few pieces of the system are very promising, (iii) emphasize a different subset of the objects and operators. How does "misinformation" impede the humans? How about no information? Have them give verbal protocols about where they are focussing their attention, and why.

Albeit at a nontrivial cost, the Worth facets did manage to correct themselves by the end of a long[22] run. What would happen if the Worth facets of those 115 concepts were not only initialized to 200, but were held fixed at 200 for the duration of the run?

In this case, the delay still subsided with time. That is, AM still got more and more "back to normal" as it progressed onward. The reason is because AM's later work dealt with concepts like Primes, Square-root, etc., which were so far removed from the initial base of concepts that the initial concepts' Worths were of little consequence.

Even more drastically, we could force all the Worth facets of all concepts – even newly-created ones – to be kept at the value 200 forever. In this case, AM's behavior doesn't completely disintegrate, but that delay factor actually increases with time: apparently, AM begins to suffer from the exponential growth of "things to do" as its repertoire of concepts grows linearly. Its purposiveness, its directionality depends on "focus of attention" more and more, and if that feature is removed, AM loses much of its rationality. A factor of 5 delay doesn't sound that bad "efficiency-wise", but the actual apparent behavior of AM is as staccato bursts of development, followed by wild leaps to unrelated concepts. AM no longer can "permanently" record its interest in a certain concept.

So we conclude that the Worth facets are (i) not finely tuned, yet (ii) provide important

---

[20] Any reader who wishes to perform this experiment can simply say [MAPC CONCEPTS '(LAMBDA (c) (SETB c WORTH 200)] to Interlisp, just before typing (START) to begin AM

[21] Incidentally, GPS behaved just this same way See, e.g., [Newell&Simon 72]

[22] A couple cpu hours, about a thousand tasks total selected from the agenda

global information about the relative values of concepts. If the Worth facets are completely disabled, the rationality of AM's behavior hangs on the slender thread of "focus of attention".

6.2.2. How finely tuned is the Agenda?

The top few candidates on the agenda always appear to be reasonable (to me). If I work with the system, guiding it, I can cause it to make a few discoveries it wouldn't otherwise make, and I can cause it to make its typical ones much faster (about a factor of 2). Thus the *very* top task is not always the best.

If AM randomly selects one of the top 20 or so tasks on the agenda each time, what will happen to its behavior? Will it disintegrate, slow down by a factor of 10, slow down slightly,...?

This experiment required only a few seconds to set up, but demanded a familiarity with the LISP functions which make up AM's control structure. At a certain point, AM asks for Best-task(Agenda). Typically, the LISP function Best-task is defined as CAR – i.e., pick the first member from the list of tasks. What I did was to redefine Best-task as a function which randomly selected n from the set {1,2,...,20}, and then returned the nth member of the job-list.

If you watch the top job on the agenda, it will take about 10 cycles until AM chooses it. And yet there are many good, interesting, worthwhile jobs sprinkled among the top 20 on the agenda, so AM's performance is cut by merely a factor of 3, as far as cpu time per given major discovery. Part of this better-than-20 behavior is due to the fact that the 18th best task had a much lower priority rating than the top few, hence was allocated much less cpu time for its quantum than the top task would have received. Whether it succeeded or failed, it used up very little time. Since AM was frequently working on a low-value task, it was unwilling to spend much time or space on it. So the mean time allotted per task fell to about 15 seconds (from the typical 30 secs). Thus, the "losers" were dealt with quickly, so the detriment to cpu-time performance was softened.

Yet AM is much less rational in its sequencing of tasks. A topic will be dropped right in the middle, for a dozen cycles, then picked up again. Often a "good" task will be chosen, having reasons all of which were true 10 cycles ago – and which are clearly superior to those of the last 10 tasks. This is what is so annoying to human onlookers.

To carry this investigation further, another experiment was carried out. AM was forced to alternate between choosing the top task on the agenda, and a randomly-chosen one. Although its rate of discovery was cut by less than half, its behavior was almost as distasteful to the user as in the last (always-random) experiment.

Conclusion: Picking (on the average) the 10th-best candidate impedes progress by a factor less than 10 (about a factor of 3), but it dramatically degrades the "sensibleness" of AM's behavior, the continuity of its actions. Humans place a big value on absolute sensibleness, and believe that doing something silly 50% of the time is *much* worse than half as productive as always doing the next most logical task.

Corollary: Having 20 multi-processors simultaneously execute the top 20 jobs will increase the rate of "big" discoveries, but not by a full factor of 20.

Another experiment in this same vein was done, one which was designed to be far more crippling to AM. Be-threshhold was held at 0 always, so *any* task which ever got proposed was kept forever on the agenda, no matter how low its priority. The Best-task function was modified so it randomly selected any member of the list of jobs. As a final insult, the Worth facets of all the concepts were initialized to 200 before starting AM.

Result: Many "explosive" tasks were chosen, and the number of new concepts increased rapidly. As expected, most of these were real "losers". There seemed no rationality to AM's sequence of actions, and it was quite boring to watch it floundering so. The typical length of the agenda was about 500, and AM's performance was "slowed" by at least a couple orders of magnitude. A more subjective measure of its "intelligence" would say that it totally collapsed under this random scheme.

Conclusion: Having an unlimited number of processors simultaneously execute all the jobs on the agenda would increase the rate at which AM made big discoveries, at an ever accelerating pace (since the length of the agenda would grow exponentially).

Having a uniprocessor *simulate* such parallel processing would be a losing idea, however. The truly "intelligent" behavior AM exhibits is its plausible sequencing of tasks.

6.2.3. How valuable is tacking reasons onto each task?

Let's dig inside the agenda scheme now. One idea I've repeatedly emphasized is the attaching of reasons to the tasks on the agenda, and using those reasons and their ratings to compute the overall priority value assigned to each task. An experiment was done to ascertain the amount of intelligence that was emanating from that idea.

The global formula assigning a priority value to each job was modified. We let it still be a function of the reasons for the job, but we "trivialized" it: the priority of a job was computed as simply the number of reasons it has (normalized by multiplying by 100, and cut-off if over 1000).

This raised the new question of what to do if several jobs all have the same priority. In that case, I had AM execute them in stack-order (most recent first)[23].

Result: I secretly expected that this wouldn't make too much difference on AM's apparent level of directionality, but such was definitely not the case. While AM opened by doing tasks which were far more interesting and daring than usual (e.g., filling in various Coalescings right away), it soon became obvious that AM was being swayed by hitherto trivial coding decisions. Whole classes of tasks – like Checking Examples of C – were never chosen, because they only had one or two reasons supporting them. Previously, one

[23] Why? Because (i) it sounds right intuitively to me, (ii) this is akin to human focus of attention, and mainly because (iii) this is what AM did anyway, with no extra modification

or two good reasons were sufficient. Now, tasks with several poor reasons were rising to the top and being worked on. Even the LIFO (stack) policy for resolving ties didn't keep AM's attention focussed.

Conclusion: Unless a conscious effort is made to ensure that each reason really will carry roughly an equal amount of semantic impact (charge, weight), it is not acceptable merely to choose tasks on the basis of how many reasons they possess. Even in those constricted equal-weight cases, the similarities between reasons supporting a task should be taken into account.

Another experiment, not yet performed, will pin down the value of this rule-attaching idea even more precisely. A threshhold value – say 400 – will be fixed. Any reason whose rating is above that threshhold will be called a *good* reason, and every other reason will be a minor reason. Then tasks will be ordered by the number of good reasons they possess, and ties will be broken by the number of minor reasons. Still another experiment would be to randomly pick any task with at least one good reason.

6.2.4. What if certain concepts are eliminated/added?

Feeling in a perverse mood one day, I eliminated the concept "Equality" from AM, to see what it would then do. Equality was a key concept, because AM discovered Numbers via the technique of generalizing the relation "Equality" (exact equality of 2 given structures, at all internal levels). What would happen if we eliminate this path? Will AM rederive Equality? Will it get to Cardinality via another route? Will it do some set-theoretic things?

Result: Rather disappointing. AM never did re-derive Equality, nor Cardinality. It spent its time thrashing about with various flavors of data-structures (unordered vs. ordered, multiple-elements allowed or not, etc.), deriving large quantities of boring results about them. Very many composings and coalescings were done, but no exciting new operations were produced.

It is expected that eliminating other, less central concepts than Equality will do less damage to AM's progress. The reader is invited to try such experiments himself.

To eliminate a concept, like equality, one need merely type *KILB(OBJ-EQUALITY*[24]*)* at the beginning of the session, before typing *(START)*.

An even kinder type of experiment would be to *add* a few concepts. One such experiment was done: the addition of Cartesian-product. This operation, named C-PROD, accepts two sets as arguments and returns a third set as its value: the Cartesian product of the first two.

Result: The only significant change in AM's behavior was that TIMES was discovered first as the restriction of C-PROD to Canonical-Bags. When it soon was rediscovered in a few other guises, its Worth was even higher than usual. AM spent even more time exploring concepts concerned with it, and deviated much less for quite a long time.

---

[24] To find out the precise PNAME of each concept, just type *CONCEPTS*.

Synthesis of the above experiments: It appears that AM may really be more specialized than expected; AM may only be able to forge ahead along one or two main lines of development – at least if we demand it make very interesting, well-known discoveries quite frequently. Removing certain key concepts can be disastrous. On the other hand, adding some carefully-chosen new ones can greatly enhance AM's directionality (hence its apparent intelligence).

Conclusion: In its current state, AM is thus seen to be *minimally competent*: if any knowledge is removed, it appears much less intelligent; if any is added, it appears slightly smarter.

Suggestion for future research: A hypothesis, which should be tested experimentally, is that the importance of the presence of each individual concept decreases as the number of – and *depth* of – the synthesized concepts increase. That is, any excision would eventually "heal over", given enough time. The failure of AM to verify this may be due to the relatively small amount of development in toto (an hour of cpu time, a couple hundred new concepts, a few levels deeper than the starting ones).

6.2.5. What if certain heuristics are tampered with?

The class of experiments described by this section's heading should prove entertaining, but it will probably be difficult to learn from their results.

Why is this? Some of the heuristics were added to correct a specific problem; removing them would simply re-initiate that problem. Others were never actually used by AM, so their deletion would have no effect. If AM enlarged the range of what it worked on, their absence might then be felt.

What good would these experiments be, then? We might learn something about the "redundancy of reasoning chains". We'd stop AM just before it made a big discovery, remove the heuristic rule it was about to use, and see if it ever makes that big discovery anyway, later on. If not, perhaps the discarded rule was very important, or there are alternate rules which exist but haven't been inserted in AM. If the same discovery is made by an alternate route, does that indicate an unexpected duplication of heuristic knowledge? If heuristic H2 is used now, instead of H1, does that suggest a new meta-rule: "if you want to apply one of H1/H2 but can't, see if the other rule *can* be applied."? Is that last sentence really a Meta-meta-rule?

Before this discussion enters an infinite loop, I'd better extract myself – and the reader – by commenting that there may be an idea in all this, perhaps of use to whoever writes Meta-AM. It was decided not to carry out a systematic series of experiments of this type until AM is much further developed in abilities.

6.2.6. Can AM work in a new domain: Plane Geometry?

A true strategy should be domain-independent.

-- Adams

As McDermott points out [McDermott 76], just labelling a bunch of heuristics 'Operation heuristics' doesn't suddenly make them relevant to any operation; all it does is give that impression to a human who looks at the code (or a description of it). Since the author hoped that the labelling really was fair, an experiment was done to test this. Such an experiment would be a key determiner of how general AM is.

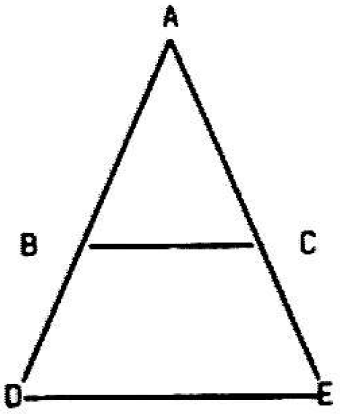
How might one demonstrate that the "Operation" heuristics really could be useful or dealing with any operation, not just the ones already in AM's initial base of concepts?

One way would be to pick a new domain, and see how many old heuristics contribute to -- and how many new heuristics have to be added to elicit -- some sophisticated behavior in that domain. Of course, some new primitive concepts would have to be introduced (defined) to AM.

Only one experiment of this type was attempted. The author added a new base of concepts to the ones already in AM. Included were: Point, Line, Angle, Triangle, Equality of points/lines/angles/triangles. These simple plane geometry notions were sufficiently removed from set-theoretic ones that those pre-existing specific concepts would be totally irrelevant; on the other hand, the general concepts -- the ones with the heuristics attached -- would still be just as relevant: Any-concept, Operation, Predicate, Structure, etc.

For each new geometric concept, the only facet filled in was its Definition. For the new predicates and operators, their Domain/range entries were also supplied. No new heuristics were added to AM.

Results: fairly good behavior. AM was able to find examples of all the concepts defined, and to use the character of the results of those examples searches to determine intelligent courses of action. AM derived congruence and similarity of triangles, and several other well-known simple concepts. An unusual result was the repeated derivation of the idea of "timberline". This is a predicate on two triangles: Timberline(T1,T2) iff T1 and T2 have a common angle, and the side opposite that angle in the two triangles are parallel:



Timberline(ABC,ADE)

Since AM kept rederiving this in new ways, it seems surprising that there is no very common name for the concept. It could be that AM is using techniques which humans don't – at least, for geometry.

The only new bit of knowledge that came out of this experiment was a "use" for Goldbach's conjecture: any angle (0-180 degrees) can be built up (to within 1 degree) as the sum of two angles of prime degrees (<180). This result is admittedly esoteric at best, but is nonetheless worth reporting.

The total effort expended on this experiment was: a few months of subconscious processing, ten hours of designing the base of concepts to insert, ten hours inserting and debugging them. The whole task took about two days of real time.

The conclusion to be drawn is that heuristics really can be generally useful; their attachment to general-sounding concepts is not an illusion.[25] The implication of this is that AM can be grown incrementally, domain by domain. Adding expertise in a new domain requires only the introduction of concepts local to that domain; all the very general concepts – and their heuristics – already exist and can be used with no change.

The author feels that this result can be generalized: AM can be expanded in scope, even to non-mathematical fields of endeavor. In each field, however, the rankings of the various heuristics[26] may shift slightly. As the domain gets further away from mathematics, various heuristics are important which were ignorable before (e.g., those dealing with ethics), and some pure math research-oriented heuristics become less applicable ("giving up and moving on to another topic" is not an acceptable response to the 15-puzzle, nor to a hostage situation).

Well, it sounds as if we've shifted our orientation from 'Results' to a subjective evaluation of those results. Let's start a new chapter to legitimize this type of commentary.

---

[25] Or: it's a very good illusion! But note: if this phenomenon is repeatable and useful, then (like Newtonian mechanics) it won't pragmatically matter whether it's only an illusion.

[26] the numeric values that should be returned by the local ratings formulae which are attached to the heuristic rules.

Chapter 7. Evaluating AM

All mathematicians are wrong at times.

-- Maxwell

This chapter contains discussions "meta" to AM itself.

First comes an essay about judging the performance of a system like AM. This is a very hard task, since AM has no "goal". Even using current mathematical standards, should AM be judged on what it produced, or the quality of the path which led to those results, or the difference between what it started with and what it finally derived?

Section 7.2 then deals with the capabilities and limitations of AM:

- What concepts can be elicited from AM now? With a little tuning/tiny additions?
- What are some notable omissions in AM's behavior? Can the user elicit these?
- What could probably be done within a couple months of modifications?
- Aside from a total change of domain, what kinds of activities does AM lack (e.g., proof capabilities)? Are any discoveries (e.g., analytic function theory) clearly beyond its design limitations?

Finally, all the conclusions will be gathered together, and a short summary of this project's 'contribution to knowledge' will be tolerated.

7.1. Judging Performance

One may view AM's activity as a progression from an initial core of knowledge to a more sophisticated "final"[1] body of concepts and their facets. Then each of the following is a reasonable way to measure success, to "judge" AM:

1. By AM's ultimate achievements. Examine the list of concepts and methods AM

[1] As has been stressed before, AM has no fixed goal, no "final" state. For practical purposes, however, the totality of explorations by AM is about the same as the "best run so far"; either of these can be thought of as defining what is meant by the "final" state of knowledge.

developed. Did AM ever discover anything interesting yet unknown to the user?[2] Anything new to Mankind?

2. By the character of the difference between the initial and final states. Progressing from set theory to number theory is much more impressive than progressing from two-dimensional geometry to three-dimensional geometry.
3. By the quality of the route AM took to accomplish these advances: How clever, how circuitous, how many of the detours were quickly identified as such and abandoned?
4. By the character of the User–System interactions: How important is the user's guidance? How closely must he guide AM? What happens if he doesn't say anything ever? When he does want to say something, is there an easy way to express that to AM, and does AM respond well to it? Given a reasonable kick in the right direction, can AM develop the mini-theories which the user intended, or at least something equally interesting?
5. By its intuitive heuristic powers: Does AM believe in "reasonable" conjectures? How accurately does AM estimate the difficulty of tasks it is considering? Does AM tie together (e.g., as analogous) concepts which are formally unrelated yet which benefit from such a tie?
6. By the results of the experiments described in Section 6.2 (beginning on page 125). How "tuned" is the worth numbering scheme? The task priority rating scheme? How fragile is the set of initial concepts and heuristic rules? How domain-specific are those heuristics really? The set of facets?
7. By the very fact that the kinds of experiments outlined in Section 6.2 can easily be "set up" and performed on AM. Regardless of the experiments' outcomes, the features of AM which allow them to be carried out at all are worthy of note.
8. By the implications of this project. What can AM suggest about educating young mathematicians (and scientists in general)? What can AM say about doing math? about empirical research in general?
9. By the number of new avenues for research and experimentation it opens up. What new projects can we propose?
10. By comparisons to other, similar systems.

For each of these 10 measuring criteria, a subsection will now be provided, to illustrate (i) the biggest achievement and (ii) the biggest failure of AM along each dimension, and (iii) to try to objectively characterize AM's performance according to that measure. Other measures of judging performance exist[3], of course, but haven't been applied to AM.

7.1.1. AM's Ultimate Discoveries

---

[2] The "user" is a human who works with AM interactively, giving it hints, commands, questions, etc. Notice that by "new" we mean new to the user, not new to Mankind. This might occur if the user were a child, and AM discovered some elementary facts of arithmetic. This is not really so provincial: mathematicians take "new" to mean new to Mankind, not new in the Universe. I feel philosophy slipping in, so this footnote is terminated.

[3] For example, Colby sent transcripts of a session with PARRY to various psychiatrists, and had them evaluate each interaction along several dimensions. The same kind of survey could be done for AM. A quite separate measure of AM would be to wait and see how many future articles in the field refer to this work (and in what light!).

Two of the ideas which AM proposed were totally new and unexpected:[4]

1. Consider numbers with an abnormally high number of divisors. If d(n) represents the number of divisors of n,[5] then AM defines the set of "maximally-divisible numbers" to be {n∈N | (∀m<n) d(m)<d(n)}. By factoring each such number into primes, AM noticed a regularity in them. The author then developed a "mini-theory" about these numbers. It later turned out that Ramanujan had already proposed that very same definition (in 1915), and had found that same regularity. His results only partially overlap those of AM and the author, however, and his methods are radically different.
2. AM found a cute geometric application of Goldbach's conjecture. Given a set of all angles of prime degree, from 0 to 180°,[6] then *any* angle between 0 and 180 degrees can be approximated to within 1° by adding a pair of angles from this prime set. In fact, it is hard to find smaller sets than this one which approximate any angle to that accuracy.

By and large, the other concepts which AM developed were either already-known, or real losers. For example, AM composed Set-insert with the predicate Equality. The result was an operation Insert∘Equal(x,y,z), which first tested whether x was Equal to y or not. The value of this was either True or False[7]. Next, this T/F value was inserted into z. For example, Insert∘Equal({1,2},{3,4},{5,6}) = {False,5,6}. The first two arguments are not equal, so the atom 'False' was inserted into the third. Although hitherto "unknown", this operation would clearly be better off left in that state.

Another kind of loser occurred whenever AM entered upon some "regular" behavior. For example, if it decided that Compose was interesting, it might try to create some examples of compositions. It could do this by picking two operations and composing them. What better operations to pick than Compose and Compose! Thus Compose∘Compose would be born. By composing that with itself, an even more monstrous operation is spawned: Compose∘Compose∘Compose∘Compose. Since AM actually uses the word "Compose" instead of that little infix circle, the PNAME of the data structure it creates is horrendous. Its use is almost nonexistent: it must take 5 operations as arguments, and it returns a new operation which is the composition of those five. An analogous danger which exists is for AM to be content conjecturing a stream of very similar relationships (e.g., the multiplication table). In all such cases, AM must have meta-rules which pull it up out of such whirlpools, to perceive a higher generalization of its previous sequence of related activities.

In summary, then, we may say that AM produced a few winning ideas new to the author, a couple of which were new to Mankind. Several additional "new" concepts were created

---

[4] Note that these are "ultimate discoveries" only in the sense of what has been done at the time of writing this thesis. For one of AM's ideas to be "new", it should be previously unknown to both the author and the user. Why? If the author knew about it, then the heuristics he provided AM with might unconsciously encode a path to that knowledge. If the user knew about that idea, his guidance might unconsciously help AM to derive it. An even more stringent interpretation would be that the idea be hitherto unknown to the collective written record of Mathematics.

[5] e.g., d(12) = Size({1,2,3,4,6,12}) = 6.

[6] Included are 0° and 1°, as well as the "typical" primes 2°, 3°, 5°, 7°, 11°,.., 179°.

[7] Actually, in LISP, it was easier to have such results always be either T or NIL.

which both AM and the user agreed were better forgotten. The "level" of AM's fruits could be classified as an undergraduate math major, although this is deceptive since AM lacks the *breadth* of abilities any human being possesses.

7.1.2. The Magnitude of AM's Progress

> *Even with men of genius, with whom the birth rate of hypotheses is very high, it only just manages to exceed the death rate.*
>
> -- W. H. George[8]

We can ask the following kind of question: how many "levels" did AM progress along? This is a fuzzy notion, but basically we shall say that a new level is reached when a valuable new bunch of connected concepts are defined in terms of concepts at a lower level.

For example, AM started out knowing about Sets and Set-operations. When it progressed to numbers and arithmetic, that was one big step up to a new level. When it zeroed in on primes, unique-factorization, and divisibility, it had moved up another level.

When fed simple geometry concepts, AM moved up one level when it defined some generalizations of the equality of geometric figures (parallel lines, congruent and similar triangles, angles equal in measure) and their invariants (rotations, translations, reflections).

The above few examples are unfortunately exhaustive: that just about sums up the major advances AM made. Its progress was halted not so much by cpu time and space, as by a paucity of meta-knowledge: heuristic rules for filling in new heuristic rules. Thus AM's successes are finite, and its failures infinite, along this dimension.

A more charitable view might compare AM to a human who was forced to start from set theory, with AM's sparse abilities. In that sense, perhaps, AM would rate quite well. The "unfair" advantage it had was the presence of many heuristics which themselves were gleaned from mathematicians: i.e., they are like compiled hindsight. A major purpose of mathematics education in the university is to instil these heuristics into the minds of the students.

AM is thus characterized as possessing heuristics which are powerful enough to take it a few "levels" away from the kind of knowledge it began with, but *only* a few levels. The limiting factors are (i) the heuristic rules AM begins with, and more specifically (ii) the expertise in recognizing and compiling new heuristics, and more generally (iii) a lack of real-world situations to draw upon for analogies, intuitions, and applications.

---

[8] Quoted from [Beveridge 50].

7.1.3. The Quality of AM's Route

Thinking is not measured by what is produced, but rather is a property of the way something is done.

-- Hamming

No matter what its achievements were, or the magnitude of its advancement from initial knowledge, AM could[9] still be judged "unintelligent" if, e.g., it were exploring vast numbers of absurd avenues for each worthwhile one it found. The quality of the route AM followed is thus quite significant.

AM performed better in this respect than expected. It is not obvious[10] how well a human would have fared under similar circumstances. Of the two hundred new concepts it defined, about 130 were acceptable – in the sense that one can defend AM's reasoning in at least exploring them; in the sense that a human mathematician might have considered them. Of these "winners", about two dozen were significant – that is, useful, catalytic, well-known by human mathematicians, etc. Unfortunately, the sixty or seventy concepts which were losers were *real* losers. In this respect, AM fell far below the standards a mathematician would set for acceptable behavior: *all* his failures should have at least seemed promising at the beginning. Half of AM's adventures were poorly grounded, and (perhaps due to a lack of intuition) AM bothered with concepts which were "obviously" trivial: the set of even primes, the set of numbers with only one divisor, etc. The human mathematician would momentarily consider many poor courses of action, whereas AM on the other hand managed to avoid truly lunatic activities without even momentary consideration of them, But a human would only spend a significant amount of time on very promising tasks, and AM wasted a huge amount of time on tasks which a human would have quickly recognized as dead-ends.

Once again we must observe that the quality of the route is a function of the quality of the heuristics. If there are many clever little rules, then the steps AM takes will often seem clever and sophisticated. If the rules superimpose nicely, joining together to collectively buttress some specific activity, then their effectiveness may surprise – and surpass – their creator.

Such moments of great insight (i.e., where AM's reasoning surpassed mine) did occur, although rarely. Both of AM's "big discoveries" started by its examining concepts I felt weren't really interesting. For example, I didn't like AM spending so much time worrying about numbers with many divisors; I "knew" that the converse concept of primes was

---

[9] not necessarily WOULD be so judged. Humans may very well consider an incredible number of silly ideas before the right pair of "hooked atoms" collide into a sensible thought, which is then considered in full consciousness. If, like humans, AM was capable of doing this processing in a sufficiently brief period of real time, it would not reflect ill on its evaluation. Of course, this may simply be the DEFINITION of "sufficiently brief".

[10] Or whether that even makes sense to consider. Comparisons with mathematicians would be desirable, but are beyond the scope of this investigation.

infinitely more valuable. And yet AM saw no reason to give up on maximally-divisible numbers; it had several good reasons for continuing that inquiry (they were the converse to primes which had already proved interesting, their frequency within the integers was neither very high nor very low nor very regular, their definition was simple, they were extremals of the interesting operation "Divisors-of", etc., etc.) Similarly, I "knew" that Goldbach's conjecture was useless, so I was unhappy that AM was bothering to try to apply it in the domain of geometry. In both cases, AM's reasons for its actions were unassailable, and in fact it did discover some interesting new ideas both times.

Sometimes AM's behavior was displeasing, even though it wasn't "erring". Occasionally it was simultaneously developing two mini-theories (say primes and maximally-divisibles). Then it might pick a task or two dealing with one of these topics, then a task or two dealing with the other topic, etc. The task picked at each moment would be the one with the highest priority value. As a theory is developed, the interestingness of its associated tasks go up and down; there may be doldrums for a bit, just before falling into the track that will lead to the discovery of a valuable relationship. During these temporary lags, the interest value of tasks related to the other theory's concepts will appear to have a higher priority value: i.e., better reasons supporting it. So AM would then skip over to one of *those* concepts, develop it until *its* doldrums, then return to the first one, etc. Most humans found this behavior unpalatable[11] because AM had no compunction about skipping from one topic to another. Humans have to retune their minds to do this skipping, and therefore treat it much more seriously. For that reason, AM was given an extra mobile reason to use for certain tasks on its agenda: "focus of attention". Any task with the same kind of topic as the ones just executed are given this extra reason, and it raises their priority values a little. This was enough sometimes to keep AM working on a certain mini-theory when it otherwise would have skipped somewhere else.

The above "defect" is a cute little kind of behavior AM exhibited which was non-human but not clearly "wrong". There were *genuine* bad moments also, of course. For example, AM became very excited[12] when the conjunction of "empty-set" and other concepts kept being equal to empty-set. AM kept repeating conjunctions of this form, rather than stepping back and generalizing this data into a (phenomenological) conjecture. Similar blind looping behavior occurred when AM kept composing Compose with itself, over and over. In general, one could say that "regular" behavior of any kind signals a probable fiasco. A heuristic rule to this effect halted most of these disgraceful antics. This rule had to be careful, since it was almost the antithesis of the "focus of attention" idea mentioned in the preceding paragraph. Together, those two rules seem to say that you should continue on with the kind of thing you were just doing, but not for *too* long a time.

The moments of insight were 2 in number; the moments of stupid misdirection were about twenty times as many.

AM has very few heuristics for deciding that something was *uninteresting*, that work on it

---

[11] Although it might be the "best" from a dynamic management point of view, it probably would be wrong in the long run. Major advances really do have lulls in their development.

[12] Please excuse this anthropomorphism. Technically, we may say that the priority value of the best job on the agenda is the "level of excitement" of AM. 700 or higher is called "excitement", on a scale of 0-1000.

should halt for a long time. Rather, AM simply won't have anything positive to say about that concept, and other concepts are explored instead, essentially by default. Each concept has a worth component which corresponded to its right to life (its right to occupy storage in core). This number slowly declines with time, and is raised whenever something interesting happens with that concept. If it ever falls below a certain threshhold, and if space is exhausted[13], then the concept is forgotten: its list cells are garbage collected, and all references to it are erased, save those which will keep it from being re-created. This again is not purposeful forgetting, but rather by default; not because X is seen as a dead-end, but simply because other concepts seem so much more interesting for a long time.

Thus AM did not develop the sixty "losers" very much: they ended up with an average of only 1.5 tasks relevant to them ever having been chosen. The "winners" averaged about twice as many tasks which helped fill them out more. Also, the worth ratings of the losers were far below those of the winners. So AM really did judge the value of its new concepts quite well.

The final aspect of this important dimension of evaluation is the quality of the reasons AM used to support each task it chose to work on. Again, the English phrases corresponded quite nicely to the "real" reasons a human might give to justify why something was worth trying, and the ordering of the tasks on the agenda was rarely far off from the one that I would have picked myself. This was perhaps AM's greatest success: the rationality of its actions.

7.1.4. The Character of the User-System Interactions

AM is not a "user-oriented" system. There were many nice human-interaction features in the original grandiose proposal for AM which never got off the drawing board. At the heart of these features were two assumptions:

1. The user must understand AM, and AM must likewise have a good model of the particular human using AM. The only time either should initiate a message is when his model of the other is not what he wants that model to be. In that case, the message should be specifically designed to fix that discrepancy.[14]
2. Each kind of message which is to pass between AM and its user should have its own appropriate language. Thus there should be a terse comment language, whereby the user can note how he feels about what AM is doing, a questioning language for either party to get/give reasons to the other, a picture language for communicating certain relationships, etc.

Neither of these ideas ever made it into the LISP code that is now AM, although they are certainly not prohibited in any way by AM's design. It would be a separate project, at or above the level of a master's thesis, for someone to build a nice user interface for AM[15].

---

[13] No concepts were forgotten in this way until near the end of AM's runs, when AM would usually collapse from several causes including lack of space.

[14] This idea was motivated by a lecture given in 1975 by Terry Winograd

[15] I am not actually calling for this to be done, merely indicating the magnitude of the effort involved. A VERY nice user interface might be much harder, at the level of a dissertation.

As one might expect, the reason for this atrophy is simply because very little guidance from the user was needed by AM. In fact, all the discoveries, cpu time quotes, etc. mentioned in this document are taken from totally unguided runs by AM. If the user guides as well as he can, then about a factor of 2 or 3 speedup is possible. Of course, this assumes that the user is dragging AM directly along a line of development he knows will be successful. The user's "reasons" at each step are based essentially on hindsight. Thus this is not at all "fair". If AM ever becomes more user-oriented, it would be nice to let children (say 6-12 years old) experiment with it, to observe them working with it in domains unfamiliar to either of them.[16]

The user can "kick" AM in one direction or another, e.g., by interrupting and telling AM that Sets are more interesting than Numbers[17]. Even in that particular case, AM fails to develop any higher-level set concepts (diagonalization, infinite sets, etc.) and simply wallows around in finite set theory (de Morgan's laws, associativity of Union, etc.). When geometric concepts are input, and AM is kicked in *that* direction, much nicer results are obtained. See the report on the Geometry experiment, page 133.

There is one important result to observe: the very best examples of AM in action were brought to full fruition only by a human developer. That is, AM thought of a couple great concepts, but couldn't develop them well on its own. A human (the author) then took them and worked on them by hand, and interesting results were achieved. These results could be told to AM, who could then go off and look for new concepts to investigate. This interaction is of course at a much lower frequency than the kind of rapidfire question/answering talked about above. Yet it seems that such synergy may be the ultimate mode of AM-like systems.

7.1.5. AM's Intuitive Powers

> *Intuitive conviction surpasses logic as the brilliance of the sun surpasses the pale light of the moon.*
>
> -- Kline

Let me hasten to mention that the word "intuitive" in this subsection's title is not related to the (currently non-existent) "Intuitions" facets of the concepts. What is meant is the totality of plausible reasoning which AM engages in: empirical induction, generalization, specialization, maintaining reasons for jobs on the agenda list, creation of analogies between bunches of concepts, etc.

---

[16] Starred (*) exercise for the reader: carry out such a project on a statistically significant sample of children, wait thirty years, and observe the incidence of mathematicians and scientists in general, compared to the national averages. Within whatever occupation they've chosen, rate their creativity and productivity.

[17] To actually do this, the user will type control-I to interrupt AM. He then types I, meaning "alter the interest of", followed by the word "Sets". AM then asks whether this is to be raised or lowered. He types back R, and AM asks how much, on a 1-10 scale. He replies 9, say, and then repeats this process for the concept "Numbers".

AM only considers conjectures which have been explicitly suggested: either by empirical evidence, by analogy, or (de-implemented now:) by Intuition facets. Once a conjecture has been formulated, it is tested in all ways possible: new experimental evidence is sought (especially extreme cases), it is examined formally[18] to see if it follows from already-known conjectures, etc.

Because of this grounding in plausibility, the only conjectures the user ever sees (the ones AM is testing) are quite believable. If they turn out to be false, both he and AM are surprised. For example, both AM and the user were disappointed when nothing came out of the concept of Uniquely-prime-addable numbers (positive integers which can be represented as the sum of two primes in precisely one way). Several conjectures were proposed via analogy with unique prime factorization, but none of them held experimentally. Each of them seemed worth investigating, to both the user and the system.[19]

AM's estimates of the value of each task it attempts were often far off from what hindsight proved their true values to be. Yet this was not so different from the situation a real researcher faces, and it made little difference on the discoveries and failures of the system. AM occasionally mismanaged its resources due to errors in these estimates. To correct for such erroneous prejudgments, heuristic rules were permitted to dynamically alter the time/space quanta for the current task. If some interesting new result turned up, then some extra resources would be allotted. If certain heuristics failed, they could reduce the time limits, so not as much total cpu time would be wasted on this loser.

An example of a nice conjecture is the unique factorization one. A nice analogy was the one between angles and numbers (leading to the application of Goldbach's conjecture). Another nice analogy was between numbers and bags (and hence between bag-operations and what we commonly call arithmetic operations).

Some poor analogies were considered, like the one between bags and singleton-bags. The ramifications of this analogy were painfully trivial[20].

7.1.6. Experiments on AM

The experiments described in Section 6.2 (page 125 ff) provide some results relevant to the overall value of the AM system. The reader should consult that section for details; neither the experiments nor their results will be repeated here. A few conclusions will be summarized, to show that AM fared well in this dimension of evaluation.

The worth-numbering scheme for the concepts is fairly robust: even when all the concepts's

---

[18] Currently, this is done in trivial ways. An open problem, which is under attack now, is to add more powerful formal reasoning abilities to AM.

[19] It is still not known whether there is anything interesting about that concept or not.

[20] The bag-operations, applied to singletons, did not produce singletons as their result: (x)U(y) is (x,y) which is not a singleton. Whether they did or not depended only on the equality or inequality of the two arguments. There were many tiny conjectures proposed which merely re-echoed this general conclusion.

worths are initialized at the same value, the performance of AM doesn't collapse, although it is noticeably degraded.

Certain mutilations of the priority-value scheme for tasks on the agenda will cripple AM, but it can resist most of the small changes tried in various experiments.

Sometimes, removing just a single concepts (e.g., Equality) was enough to block AM from discovering some valuable concepts it otherwise got (in this case, Numbers). This makes AM's behavior sound very fragile, like a slender chain of advancement. But on the other hand, many concepts (e.g., TIMES, Timberline, Primes[21]) were discovered in several independent ways. If AM's behavior is a chain, it is multiply-stranded[22]. More experiments of this sort should be done to test this general conclusion about AM.

The heuristics are specific to their stated domain of applicability. Thus when working in geometry, the Operation heuristics were just as useful as they were when AM worked in elementary set theory or number theory. The set of facets seemed adequate for those domains, too. The Intuition facet, which was rejected as a valid source of information about sets and numbers, might have been more acceptable in geometry (e.g., something similar to Gelernter's model of a geometric situation).

All in all, then, we conclude that AM was fairly tough, and about as general as its heuristics claimed it was. AM is not invincible, infallible, or universal. Its strength lies in careful use of heuristics. If there aren't enough domain-specific heuristics around, the system will simply not perform well in that domain. If the heuristic-using control structure of AM is tampered with[23], there is some chance of losing vital guiding information which the heuristics would otherwise supply.

7.1.7. How to Perform Experiments on AM

The very fact that the kinds of experiments mentioned in the last section (and described in detail in Section 6.2) can be "set up" and performed on AM, reflects a nice quality of the AM program.

Most of those experiments took only a matter of minutes to set up, only a few tiny modifications to AM. For example, the one where all the Worth ratings were initialized to the same value was done by evaluating the single LISP expression:

```
(MAPC CONCEPTS '(λ (c) (PUT c 'Worth 200)))
```

---

[21] Primes was discovered independently as follows: all numbers (>0) were seen to be representable as the sum of smaller numbers; Add was known to be analogous to TIMES; But not all numbers (>1) appeared to be representable as the product of two smaller ones; Rule number 81 triggered (see Appendix 3, page 243), and AM defined the set of exceptions: the set of numbers which could not be expressed as the product of two smaller ones; i.e., the primes

[22] except for a few weak spots, like Numbers. If they don't get discovered, AM loses.

[23] e.g., treat all reasons as equivalent, so you just COUNT the number of reasons a task has, to determine its priority on the agenda

Similarly, here is how AM was modified to treat all tasks as if they had equal value: the function Pick-task has a statement of the form

(SETQ Current-task (First-member-of Agenda))

All that was necessary was to replace the call on the function "First-member-of"[24] by the function "Random-member-of".

Even the most sophisticated experiment, the introduction of a new bunch of concepts – those dealing with geometric notions like Between, Angle, Line – took only a day of conscious work to set up.

Of course *running* the experiment involves the expenditure of hours of cpu time, so only a limited number were actually performed.[25]

There are certain experiments one can't easily perform on AM: removing all its heuristics, for example. Most heuristic search programs would then wallow around, displaying just how big their search space really was. But AM would just sit there, since it'd have nothing plausible to do.

Many other experiments, while cute and easy to set up, are quite costly in terms of cpu time. For example, the class of experiments of the form: "remove heuristics x, y, and z, and observe the resultant affect on AM's behavior". This observation would entail running AM for an hour or two of cpu time! Considering the number of subsets of heuristics, not all these questions are going to get answered in our universe's lifetime. Considering the small probable payoff from any one such experiment, very few should actually be attempted.

One nice experiment would be to monitor the contribution each heuristic is making. That is, record each time it is used and record the final outcome of its activation (which may be several cycles later). Unfortunately, AM's heuristics are not all coded as separate Lisp entities, which one could then "trace". Rather, they are often interwoven with each other into large program pieces. So this experiment can't be easily set up and run on AM.

Most of the experiments one could think of can be quickly set up – but only by someone familiar with the LISP code of AM. It would be quite hard to modify AM so that the untrained user could easily perform these experiments. Essentially, that would demand that AM have a deep understanding of its own structure. This is of course desirable, fascinating, challenging, but wasn't part of the design of AM.[26]

---

[24] In LISP, this function is actually abbreviated "CAR".

[25] Those described in the last chapter. The series of experiments began at the same time that this document was being written, and was intended originally only as a diversion from the tedium of writing. The interesting character of their results convinced me they should be included, even though they are few in number and quite incomplete.

[26] A suggestion for future research projects in this general area: such systems should be designed in a way which facilitates a poorly-trained user not only using the system but experimenting on it.

7.1.8. Future Implications of this Project

One harsh measure of AM would be to demand what possible applications it will have. This really means (i) the uses for the AM system, (ii) the uses for the ideas of how to create such systems, (iii) conclusions about math and science one can draw from experiments with AM.

Here are some of these implications, both real and potential:

1. New tools for computer scientists who want to create large knowledge-based systems to emulate some creative human activity.

   1a. The modular representation of knowledge that AM uses might prove to be effective in any knowledge-based system. Division of a global problem into a multitude of small chunks, each of them of the form of setting up one quite local "expert" on some concept, is a nice way to make a hard task more managable. Conceivably, each needed expert could be filled in by a human who really is an expert on that topic. Then the global abilities of the system would be able to rely on quite sophisticated local criteria. Fixing a set of facets once and for all permits effective inter-module communication.

   1b. Some ideas may carry over unchanged into many fields of human creativity, wherever local guiding rules exist. These include: (a) ideas about heuristics having domains of applicability, (b) the policy of tacking them onto the most general knowledge source (concept, module) they are relevant to, (c) the rippling scheme to locate relevant knowledge, etc.,

2. A body of heuristics which can be built upon by others.

   2a. Most of the particular heuristic judgmental criteria for interestingness, utility, etc., might be valid in developing theorizers in other sciences. Recall that each rule has its domain of applicability; many of the heuristics in AM are quite general.

   2b. Just within the small domain in which AM already works, this base of heuristics might be enlarged through contact with various mathematicians. If they are willing to introspect and add some of their "rules" to AM's existing base, it might gradually grow more and more powerful.

   2c. Carrying this last point to the limit of possibility, one might imagine the program possessing more heuristics than any single human. Of course, AM as it stands now is missing so much of the 'human element', the life experiences that a mathematician draws upon continually for inspiration, that merely amassing more heuristics won't automatically push it to the level of a super-human intelligence. Another far-out scenario is that of the great mathematicians of each generation pouring their individual heuristics into an AM-like system. After a few generations have come and gone, running that program could be a valuable way to bring about 'interactions' between people who were not contemporaries.

3. New and better strategies for math educators. [optional]

   3a. Since the key to AM's success seems to be its heuristics, and not the particular concepts it knows, the whole orientation of mathematics education should perhaps be modified, to provide experiences for the student which will build up these rules in his mind. Learning a new theorem is worth much less than learning a new heuristic which

lets you discover new theorems.[27] I am far from the first to urge such a revision (see, e.g., [Koestler 67], p.265, or see [Papert 72]).

3b. If the repertoire of intuition (simulated real-world scenarios) were sufficient for AM to develop elementary concepts of math, then educators should ensure that children (4-6 years old) are thoroughly exposed to those scenarios. Such activities would include seesaws, slides, piling marbles into pans of a balance scale, comparing the heights of towers built out of cubical blocks, solving a jigsaw puzzle, etc. Unfortunately, AM failed to show the value of these few scenarios. This was a potential application which was not confirmed.

3c. One use for AM itself would be as a "fun" teaching tool. If a very nice user interface is constructed, AM could serve as a model for, say, college freshmen with no math research experience. They could watch AM, see the kinds of things it does, play with it, and perhaps get a real flavor for (and get turned on by) doing math research. A vast number of brilliant minds are too turned off by high-school drilling and college calculus to stick around long enough to find out how exciting – and different – research math is compared to textbook math.

4. Further experiments on AM might tell us something about how the theory formation task changes as a theory grows in sophistication. For example, can the same methods which lead AM from premathematical concepts to arithmetic also lead AM from number systems up to abstract algebra? Or are a new set of heuristic rules or extra concepts required? My guess is that a few of each are lacking currently, but *only* a few. There is a great deal of disagreement about this subject among mathematicians. By tracing along the development of mathematics, one might categorize discoveries by how easy they would be for an AM-like system to find. Sometimes, a discovery required the invention of a brand new heuristic rule, which would clearly be beyond AM as currently designed. Sometimes, discovery is based on the lucky random combination of existing concepts, for no good *a priori* reason. It would be instructive to find out how often this is necessarily the case: how often *can't* a mathematical discovery be motivated and "explained" using heuristic rules of the kind AM possesses?

5. An unanticipated result was the creation of new-to-Mankind math (both directly and by defining new, interesting concepts to investigate by hand). The amount of new bits of mathematics developed to date is minuscule.

5a. As described in (2c) above, AM might absorb heuristics from several individuals and thereby integrate their particular insights. This might eventually result in new mathematics being discovered.

5b. An even more exciting prospect, which never materialized, was that AM would find a new redivision of existing concepts, an alternate formulation of some established theory, much like Hamiltonian mechanics is an alternate unification of the data which led to Newtonian mechanics. The only rudimentary behavior along these lines was when AM occasionally derived a familiar concept in an abnormal way (e.g., TIMES was derived in four ways; Prime pairs were noticed by restricting Addition to primes).

---

[27] Usually. One kind of exception is the following: the ability to take a powerful theorem, and extract from it a new, powerful heuristic. AM cannot do this, but it may turn out that this mechanism is quite crucial for humans' obtaining new heuristics. This is another open research problem.

7.1.9. Open Problems: Suggestions for Future Research

While AM can and should stand as a complete research project, part of its value will stem from whatever future studies are sparked by it. Of course the "evaluation" of AM along this dimension must wait for years, but even at the present time several such open problems come to mind:

- Devise Meta-heuristics, rules capable of operating on and synthesizing new heuristic rules. AM has shown the solution of this problem to be both nontrivial and indispensable. AM's progress ground to a halt because fresh, powerful heuristics were never produced. The next point suggests that the same need for new rules exists in mathematics as a whole:

- Examine the history of mathematics, and gradually build up a list of the heuristic rules used. Does the following thesis have any validity: "*The development of mathematics is essentially the development of new heuristics.*" That is, can we 'factor out' all the discoveries reachable by the set of heuristics available (known) to the mathematicians at some time in history, and then explain each new big discovery as requiring the synthesis of a brand new heuristic? For example, Bolyai and Lobachevsky did this a century ago when they decided that counter-intuitive systems might still be consistent and interesting. Non-Euclidean geometry resulted, and no mathematician today would think twice about using the heuristic they developed. Einstein invented a new heuristic more recently, when he dared to consider that counter-intuitive systems might actually have physical reality.[28] What was once a bold new method is now a standard tool in theoretical physics.

- In a far less dramatic vein, a hard open problem is that of building up a body of rules for symbolically instantiating a definition (a LISP predicate). These rules may be structured hierarchically, so that rules specific to operating on 'operations whose domain and range are equal' may be gathered. Is this set finite and managable; i.e., does some sort of "closure" occur after a few hundred (thousand?) such rules are assembled?

- More generally, we can ask for the expansion of all the heuristic rules, of all categories. This may be done by eliciting them from famous mathematicians, or automatically by the application of very sophisticated meta-heuristics. Some categories of rules include: how to generalize/specialize definitions, how to find examples of a given concept, how to optimize LISP algorithms.

- Experiments can be done on AM. A few have been performed already, many more are proposed in Section 6.2, and no doubt some additional ones have already occurred to the reader.

- Extend the analysis already begun (see p. 59) of the set of heuristics AM possesses. One reason for such an analysis would be to achieve a better understanding of the

---

[28] As Courant says, "When Einstein tried to reduce the notion of 'simultaneous events occurring at different places' to observable phenomena, when he unmasked as a metaphysical prejudice the belief that this concept must have a scientific meaning in itself, he had found the key to his theory of relativity."

contribution of the heuristics. In some sense, the heuristics and the choice of starting concepts "encode" the discoveries which AM makes, and the way it makes them. A better understanding of that encoding may lead to new ideas for AM and for future AM-like systems.

- Rewrite AM. In Chapter I, on page 9, it was pointed out that there are two common species of heuristic search programs. One type has a legal move generator, and heuristics to constrain it. The second type, including AM, has only a set of heuristics, and they act as plausible move generators. Since AM seemed to create new concepts, propose new conjectures, and formulate new tasks in a very few distinct ways, it might very well be feasible to find a purely syntactic "legal move generator" for AM, and to convert each existing heuristic into a form of constraint. In that case, one could, e.g., remove all the heuristics and still see a meaningful (if explosive) activity proceed. There might be a few surprises down that path.

- A more tractible project, a subset of the former one, would be to recode just the conjecture-finding heuristics as constraints on a new, purely syntactic "legal conjecture generator". A simple Generate-and-Test paradigm would be used to synthesize and examine large numbers of conjectures. Again, removing all the heuristics would be a worthwhile experiment.

- At the reaches of feasability, one can imagine trying to extend AM into more and more fields, into less-formalizable domains. International politics has already been suggested as a very hard future applications area.

- Abstracting that last point, try to build up a set of criteria which make a domain ripe for automating (e.g., it possesses a strong theory, it is knowledge-rich (many heuristics exist), the performance of the professionals/experts is much better than that of the typical practitioners, the new discoveries in that field all fall into a small variety of syntactic formats,...?). Initially, this study might help humans build better and more appropriate scientific discovery programs. Someday, it might even permit the creation of an automatic-theory-formation-program-writer.

- The interaction between AM and the user is minimal and painful. Is there a more effective language for communication? Should several languages exist, depending on the type of message to be sent (pictures, control characters, a subset of natural language, induction from examples, etc.)? Can AM's output be raised in sophistication by introducing an internal model of the user and his state of knowledge at each moment?

- Human protocol studies may be appropriate, to test out the model of mathematical research which AM puts forward. Are the sequences of actions similar? Are the mistakes analogous? Do the pauses which the humans emit *quantitatively* correspond to AM's periods of gathering and running 'Suggest' heuristics?

- Can the idea of Intuition functions be developed into a useful mechanism? If not, how else might real-world experiences be made available to an automated researcher to draw upon (for analogies, to base new theories upon)? Could one

interface physical effectors and receptors and quite literally allow the program to 'play around in the real world' for his analogies?

- Most of the 'future implications' discussed in the last section suggest future activities (e.g., new educational experiments and techniques).

- Most of the 'limiting assumptions' discussed in a later section (page 157) can be tackled with today's techniques (plus a great deal of effort). Thus each of them counts as an open problem for research.

- Perform an information-theoretic analysis on AM. What is the value of each heuristic? the new information content of each new conjecture?

- If you're interested in natural language, the very hard problem exists of giving AM (or a similar system) the ability to really do inferential processing on the *reasons* attached to tasks on the agenda. Instead of just being able to test for equality of two reasons, it would be much more intelligent to be able to infer the kind of relationship between any two reasons; if they overlap semantically, we'd like to be able to compute precisely how that should that effect the overall rating for the task; etc.

- Modify the control structure of AM, as follows. Allow mini-goals to exist, and supply new rules for setting them up (plausible goal generators) and altering those goals, plus some new rules and algorithms for satisfying them. The modification I have in mind would result in new tasks being proposed because of certain current goals, and existing tasks would be reordered so as to raise the chance of satisfying some important goal. Finally, the human watching AM would be able to observe the rationality (hopefully) of the goals which were set. The simple "Focus of Attention" mechanism already in AM is a tiny step in this goal-oriented direction. Note that this proposal itself demonstrates that AM is not inherently opposed to a goal-directed control structure. Rather, AM simply possesses only a partial set of mechanisms for complete reasoning about its domain.

7.1.10. Comparison to Other Systems

One popular way to judge a system is to compare it to other, similar systems, and/or to others' proposed criteria for such systems. There is no other project (known to the author) having the same objective: automated math research.[29] Many somewhat related efforts have been reported in the literature and will be mentioned here.

Several projects have been undertaken which overlap small pieces of the AM system and in addition concentrate deeply upon some area *not* present in AM. For example, the CLET

[29] In [Atkin & Birch 1971], e.g., we find no mention of the computer except as a number cruncher.

system [Badre 73] worked on learning the decimal addition algorithm[30] but the "*mathematics discovery*" aspects of that system were neither emphasized nor worth emphasizing; it was an interesting natural language communication study. The same comment applies to several related studies by IMSSS[31].

Boyer and Moore's theorem-prover [Boyer&Moore 75] embodies some of the spirit of AM (e.g., generalizing the definition of a LISP function), but its motivations are quite different, its knowledge base is minimal, and its methods purely formal.[32] The same comments apply to the SAM program [Guard 69], in which a resolution theorem-prover is set to work on unsolved problems in lattice theory.

Among the attempts to incorporate heuristic knowledge into a theorem prover, we should also mention [Wang 60], [Pitrat 70], [Bledsoe 71], and [Brotz 74]. How did AM differ from these "heuristic theorem-provers"? The goal-driven control structure of these systems is a real but only minor difference from AM's control structure (e.g., AM's "focus of attention" is a rudimentary step in that direction; see p. 150). The fact that their overall activity is typically labelled as deductive is also not a fundamental distinction (since constructing a proof is usually in practice quite *in*ductive). Even the character of the inference processes are analogous: The provers typically contain a couple binary inference rules, like Modus Ponens, which are relatively risky to apply but can yield big results; AM's few "binary" operators have the same characteristics: Compose, Canonize, Logically-combine (disjoin and conjoin). The main distinction is that the theorem provers each incorporate only a handful of heuristics. The reason for this, in turn, is the paucity of good heuristics which exist for the very general task environment in which they operate: domain-independent (asemantic) predicate calculus theorem proving. The need for additional guidance was recognized by these researchers. For example, see [Wang 60], p. 3 and p. 17. Or as Bledsoe says[33]:

> There is a real difference between *doing* some mathematics and *being* a mathematician. The difference is principally one of judgment: in the selection of a problem (theorem to be proved); in determining its relevance;... It is precisely in these areas that machine provers have been so lacking. This kind of judgment has to be supplied by the user... Thus a crucial part of the resolution proof is the *selection* of the reference theorems by the *human* user; the human, by this one action, usually employs more skill than that used by the computer in the proof.

Many researchers have constructed programs which pioneered some of the techniques AM uses[34]. [Gelernter 63] reports the use of prototypical examples as analogic models to guide search in geometry, and [Bundy 73] employs models of "sticks" to help his program work with natural numbers. The single heuristic of analogy was studied in [Evans 68] and

---

30 Given the addition table up to 10 + 10, plus an English text description of what it means to carry, how and when to carry, etc., actually write a program capable of adding two 3-digit numbers

31 See [Smith 74a], for example

32 This is not meant as criticism; considering the goals of those researchers, and the age of that system, their work is quite significant.

33 [Bledsoe 71], p. 73

34 In many cases, those techniques were used for the first time, hence were thought of as "tricks"

[Kling 71].[35]

Theory formation systems in *any* field have been few. Meta-Dendral [Buchanan 74] represents perhaps the best of these. Its task is to unify a body of mass spectral data (examples of "proper" identifications of spectra) into a small body of rules for making identifications. Thus even this system is given a fixed task, a fixed set of data to find regularities within. AM, however, must find its own data, and take the responsibility for managing its own time, for not looking too long at worthless data.[36] There has been much written about scientific theory formation (e.g., [Hempel 52]), but very little of it is specific enough to be of immediate use to AI researchers. A couple pointers to excellent discussions of this sort are: [Fogel 66], [Simon 73], and [Buchanan 75]. Also worth noting is a discussion near the end of [Amarel 69], in which "formation" and "modelling" problems are treated:

> The problem of model finding is related to the following general question raised by Schutzenberger [in discussion at the Conference on Intelligence and Intelligent Systems, Athens, Ga., 1967]: *'What do we want to do with intelligent systems that relates to the work of mathematicians?'*. So far all we have done in this general area is to emulate some of the reasonably simple activities of mathematicians, which is finding consequences from given assumptions, reasoning, proving theorems. A certain amount of work of this type was already done in the propositional and predicate calculi, as well as in some other mathematical systems. But this is only one aspect of the work that goes on in mathematics.
>
> Another very important aspect is the one of finding general properties of structures, finding analogies, similarities, isomorphisms, and so on. This is the type of activity that is extremely important for our understanding of model-finding mechanisms. Work in this area is more difficult than theorem-proving. The problem here is that of theorem finding.

AM is one of the first attempts to construct a "theorem-finding" program. As Amarel noted, it may be possible to learn from such programs how to tackle the general task of automating scientific research.

Besides "math systems", and "creative thinking systems", and "theory formation systems", we should at least discuss others' thoughts on the issue of algorithmically doing math research. Some individuals feel it is not so far-fetched to imagine automating mathematical research (e.g., Paul Cohen). Others (e.g., Polya) would probably disagree. The presence of a high-speed, general-purpose symbol manipulator in our midst now makes investigation of that question possible.

There has been very little published thought about discovery in mathematics from an algorithmic point of view; even clear thinkers like Polya and Poincare' treat mathematical ability as a sacred, almost mystic quality, tied to the unconscious. The writings of philosophers and psychologists invariably attempt to examine human performance and belief, which are far more managable than creativity *in vitro*. Belief formulae in inductive

---

[35] Brotz's program, [Brotz 74], uses this to propose useful lemmata.

[36] In case that wasn't clear: Meta-Dendral has a fixed set of templates for rules which it wishes to find, and a fixed vocabulary of mass spectral concepts which can be plugged into those templates. AM also has only a few stock formats for conjectures, but it selectively enlarges its vocabulary of math concepts.

logic[37] invariably fall back upon how well they fit human measurements. The abilities of a computer and a brain are too distinct to consider blindly working for results (let alone algorithms!) one possesses which match those of the other.

7.2. Capabilities and Limitations of AM

The first two subsections contain a general discussion of what AM can and can't do. Later subsections deal with powers and limitations inherent in using an agenda scheme, in fixing the domain of AM, and in picking one specific model of math research to build AM upon. The AM program exists only because a great many simplifying assumptions were tolerated; these are discussed in Section 7.2.4 (p. 157). Finally, some speculation is made about the ultimate powers and weaknesses of any systems which are designed very much like AM.

7.2.1. Current Abilities

What fields has AM worked in so far? AM is now able to explore a small bit of the theory of sets, data types, numbers, and plane geometry. It by no means has been fed – nor has it rediscovered – a large fraction of what is known in any of those fields. It might be more accurate to be humble and restate those domains as: elementary finite set theory, trivial observations about four kinds of data types, arithmetic and elementary divisibility theory, and simple relationships between lines, angles, and triangles. So a sophisticated concept in each domain – which was discovered by AM – might be:

- de Morgan's laws
- the fact that Delete∘Insert[38] never alters Bags or Lists
- unique factorization
- similar triangles

Can AM work in a new field, like politics? AM can work in a new elementary, formalized domain, if it is fed a supplemental base of conceptual primitives for that domain. To work in plane geometry, it sufficed to give AM about twenty new primitive concepts, each with a few parts filled in. Another domain which AM could work in would be elementary mechanics. The more informal the desired field, the less of AM that is relevant. Perhaps an AM-like system could be built for a constrained, precise political task.[39] Disclaimer: Even for a very small domain, the amount of common-sense knowledge such a system would need is staggering. It is unfortunate to provide such a trivial answer to such an important question, but there is no easy way to answer it more fully until years of additional research are performed.

Can AM discover X? Why didn't it do Y? It is difficult to predict whether AM will (without

---

[37] For example, see [Hintikka 62], [Pietarinin 72]. The latter also contains a good summary of Carnap's $\lambda,\alpha\iota$ formalization.

[38] Take an item x, insert it into (the front of) structure B, then delete one (the first) occurrence of x from B.

[39] For example, such a politico-oriented AM-like system might conceive the notion of a group of political entities which view themselves as quite disparate, but which are viewed from the outside as a single unit: e.g., 'the Arabs', 'the American Indians'. Conjectures about this concept might include its reputation as a poor combatant (and why). Many of the same facets AM uses would carry over to represent concepts in that new domain.

modifications) ever make a specific given discovery. Although its capabilities are small, its limitations are hazy. What makes the matter even worse is that, given a concept C which AM missed discovering, there is probably a reasonable heuristic rule which is missing from AM, which would enable that discovery. One danger of this "debugging" is that a rule will be added which only leads to that one desired discovery, and isn't good for anything else. In that case, the new heuristic rule would simply be an *encoding* of a specific bit of mathematics which AM would then appear to discover using general methods. This must be avoided at all costs, *even at the cost of intentionally giving up a certain discovery.* If the needed rule is general – it has many applications and leads to many interesting results – then it really was an oversight not to include it in AM. Although I believe that there are not too many such omissions still within the small realm AM explores, there is no objective way to demonstrate that, except by further long tests with AM.

In what ways are new concepts created? Although the answer to this is accurately given in Section 4.3, page 42 (namely, this is mainly the jurisdiction of the right sides of heuristic rules), and although I dislike the simple-minded way it makes AM sound, the list below does characterize the major ways in which new concepts get born:

Fill in examples of a concept (e.g., by instantiating or running its definition)
Create a generalization of a given concept (e.g., by weakening its definition)
Create a specialization of a given concept (e.g., by restricting its domain/range)
Compose two operations f,g, thereby creating a new one h. [Define h(x)=f(g(x))]
Coalesce an operation f into a new one g. [Define g(x)=f(x,x)]
Permute the order of the arguments of an operation. [Define g(x,y)=f(y,x)]
Invert an operation [g(x)=y iff f(y)=x] (e.g., from Squaring, create Square-rooting)
Canonize one predicate P1 with respect to a more general one P2 [create a new concept f, an operation, such that: P2(x,y) iff P1(f(x),f(y))]
Create a new operation g, which is the repeated application of an existing operation f.
The usual logical combinations of existing concepts x,y: x∧y, x∨y, ¬x, etc.

Below is a similar list, giving the primary ways in which AM formulates new conjectures:

Notice that concept C1 is really an example of concept C2
Notice that concept C1 is really a specialization (or: generalization) of C2
Notice that C1 is equal to C2; or: *almost always* equal
Notice that C1 and C2 are related by some known concept
Check and update the domain/range of an existing operation
If two concepts are analogous, extend the analogy to their conjectures as well

In summary, we can say that AM has achieved its original purpose: to be guided successfully by a large set of local heuristic rules, in the discovery of new mathematical theories. Besides creating new concepts and noticing conjectures, AM has the key "ability" of appearing to decide rationally what to work on at each moment. This is a result of the agenda of tasks – containing associated reasons. Of course all of these abilities stem from the quality and the quantity of local heuristic rules: little plausible move generators and evaluators.

7.2.2. Current Limitations

Below are several shortcomings of AM, which hurt its behavior but are not believed to be inherent limitations of its design. They are presented in order of decreasing severity.

Perhaps the most serious limitation on AM's current behavior arose from the lack of constraints on left sides of heuristic rules. It turned out that this excessive freedom made it difficult for AM to inspect and analyze and synthesize its own heuristics; such a need was not foreseen at the time AM was designed. It was thought that the power to manipulate heuristic rules was an ability which the author must have, but which the system wouldn't require. As it turned out, AM did successfully develop new concepts several levels deeper than the ones it started with. But as the new concepts got further and further away from those initial ones, they had fewer and fewer specific heuristics filled in (since they had to be filled in by AM itself). Gradually, AM found itself relying on heuristics which were very general compared to the concepts it was dealing with (e.g., forced to use heuristics about Objects when dealing with Numbers). Heuristics for dealing with heuristics do exist, and their number could be increased. This is not an easy job: finding a new meta-heuristic is a tough process. Heuristics are rarely more than compiled hindsight; hence it's difficult to create new ones "before the fact".

AM has no notion of proof, proof techniques, formal validity, heuristics for finding counterexamples, etc. Thus it never really establishes any conjecture formally. This could probably be remedied by adding about 25 new concepts (and their 100 new associated heuristics) dealing with such topics. The needed concepts have been outlined on paper, but not yet coded. It would probably require a few hundred hours to code and debug them.

The user interface is quite primitive, and this again could be dramatically improved with just a couple hundred hours' work. AM's explanation system is almost nonexistent: the user must ask a question quickly, or AM will have already destroyed the information needed to construct an answer. A clean record of recent system history and a nice scheme for tracking down reasons for modifying old rules and adding new ones dynamically does not exist at the level which is found, e.g., in MYCIN [Davis 76]. There is no trivial way to have the system print out its heuristics in a format which is intelligible to the untrained user.

An important type of analogy which was untapped by AM was that between heuristics. If two situations were similar, conceivably the heuristics useful in one situation might be useful (or have useful analogues) in the new situation (see [Koppelman 75]). Perhaps this is a viable way of enlarging the known heuristics. Such "meta-level" activities were kept to a minimum throughout AM, and this proved to be a serious limitation. My intuition tells me that the "right" ten meta-rules could correct this particular deficiency.

The idea of "Intuitions" facets was a flop. Intuitions were meant to model reality, at least little pieces of it, so that AM could perform (simulate) physical experiments, and observe the results. The major problem here was that *so* little of the world was modelled that the only relationships derivable were those foreseen by the author. This lack of generality was unacceptable, and the intuitions were completely excised. The original idea might lead somewhere if it were developed fully. As with all limitations of AM, I leave this as an open suggestion for future research.

Several limitations arose from the constraints of the agenda scheme, from the choice of finite set theory as the domain to work in, and from the particular model of math research that was postulated. These will be discussed in the next few subsections.

7.2.3. Limitations of the Agenda scheme

The following quibbles with the agenda scheme get less and less important. When you get bored, skip to the next subsection.

Currently, it is difficult to include heuristics which interact with one another in any significant way. The whole fibre of the Agenda scheme assumes perfect independence of heuristics. The global formula used to rate tasks on the agenda assumes perfect superposition of reasons: there are no "cross-terms". Is this assumption always valid? Unfortunately *no*, not even for the limited domain AM has explored. Sometimes, two reasons are very similar: "Examples of Sets would permit finding examples of Union" and "Examples of Sets would permit finding examples of Intersection". In that case, their two ratings shouldn't cause such a big increase in the overall priority value of the task "Fillin examples of Sets".

Sometimes, a heuristic rule will want to *dissuade* the system from some activity. Thus a *negative* numeric contribution to a task's priority value is desired. This is not figured into the current scheme. With a slight modification, the global formula could preserve the sign (signum) of each reason's rating.

Tasks on the agenda list are ordered by their numeric priority value. Each reason's numeric value is kept, too. When new reasons are added, these values are used to recompute a new priority for the task. Each reason's rating was computed by a little formula found inside some heuristic rule. Those formulae are not kept hanging around. One big improvement in apparent intelligence could be attained by tacking on those little formulae to the reasons. When a new reason is added, the old reasons' rating formulae would be evaluated again. They might indeed give new numbers. For example, suppose one reason was "Few examples of X are known". But by now, other tasks have meanwhile inadvertantly filled in several examples of X. Then that little reason's formula would come up with a much lower value than it did originally. In fact, the value might be so low that the reason was dropped altogether. If the formulae were kept, it might be good practice to evaluate them for the top two or three tasks on the agenda, to see if they might change their ordering. Also, the top task's priority would then be more accurate, and recall that its value is used to determine the cpu time and list cell space quanta that the task is allowed to use up. At the moment, AM is not set up to store the little functions, and if modified to do so, it uses up a lot more space than it can afford. Also, the top few jobs are almost never semantically coupled (except by "focus of attention"), so the precise order in which they are executed rarely matters.

Perhaps what is needed is not a single priority value for each task, but a vector of numbers. At each cycle, AM would construct a vector of its current "interests" and needs, and each task's vector would be dot-multiplied against this global vector of AM's desires. The highest scorer would then be chosen. For example, one dimension of the rating could be "safety", and one could be "best possible payoff", one could be "average expected payoff", etc. Sometimes, AM would have to break out of a stagnant situation, and it would be willing to try riskier tasks than usual. This was not implemented because of the great increase in cpu time it would cause. It is, however, probably a better design than the current one. Even more intelligent schemes can be envisioned – involving more and more symbolic data being stored with each task. Ultimately, this would be just the English reasons themselves; by that

time, the task-orderer would have grown into an incredibly complex AI program itself (a natural language program plus an interrelator plus...).

The agenda list should really be an agenda *tree*[40], since the ordering of tasks is really just partial, not total. If this is clear, then skip the rest of this paragraph. There are some "legitimate" orderings of tasks on the agenda; if task X is supported by a subset of the reasons which support Y, then typically the priority of X will be less than or equal to the priority of Y. Two tasks of the form "Fillin examples of A", "Fill in examples of B" can be ordered simply because A is currently much more interesting than B. But often, two tasks will have no ironclad ordering between them: compare "Fillin examples of Sets" and "Check generalizations of Union". Thus the ordering is only partial, and it is the artifice of the global evaluation function which embeds this into a linear ordering. If multiprocessors are used, it might be advantageous to keep the original partial ordering around.

7.2.4. Limiting Assumptions

AM only "got off the ground" because a number of sweeping assumptions were made, pertaining to what could be ignored, how a complex process could be adequately simulated, etc. Now that AM *is* running, however, those same simplifications crop up as limitations to the system's behavior. Each of the following points is a 'convenient falsehood'. Although the reader has already been told about some of these, it's worth listing them all together here:

- The only communication necessary from AM to the user is keeping the user informed of what AM is doing. No natural language ability is required by AM; simple template instantiation is sufficient.
- The only communication from the user to AM is an occasional interrupt, when the user wishes to provide some guidance or to pose a query. Both of these can be stereotyped and passed easily through a very narrow channel.[41]
- Each heuristic has a well-defined domain of applicability, which can be specified just by giving the name of a single concept.
- If concept C1 is more specialized than C2, then C1's heuristics will be more powerful and should be executed before C2's (whenever both concepts' heuristics are relevant).
- If h1 and h2 are two heuristics attached to concept C, then it is not necessary to spend any time ordering them.
- Heuristics superimpose perfectly; they never interact strongly with each other.

---

[40] maybe an agenda Heap.

[41] E.g., a set of escape characters, so 1W means '*Why did you do that?*', 1U means '*Uninteresting! Go on to something else*', etc.

- The reasons supporting a task can be mere tokens; it suffices to be able to inspect them for equality. They need not follow a constrained syntax. The value of a reason is adequately characterized by a unidimensional numeric rating.

- The reasons supporting a task superimpose perfectly; they never interact with each other.

- Supporting reasons – and their ratings – never change with time, with one exception: the ephemeron 'Focus of attention'.

- It doesn't matter in what order the supporting reasons for a task were added.

- There is no need for negative or inhibitory reasons, which would decrease the priority value of a task.

- At any moment, the top few tasks on the agenda are not coupled strongly; it is not necessary to expend extra processing time to carefully order them.

- The tasks on the agenda are completely independent of each other, in the sense of one task 'enabling' or 'waking-up' another.

- Mathematics research has a clean, simple model (see Section 7.2.6, page 162), which indicates that it is a search process governed by a large collection of heuristic rules.

- Elementary mathematics is such that valuable new concepts will be discovered fairly regularly.

- The worth of each new concept can be estimated easily, after just a brief investigation.

- Contradictions will arise very rarely, and it is not disastrous to ignore them when they do occur. The same indifference applies to the danger of believing in false conjectures.

- When doing theory formation in elementary mathematics, proof and formal reasoning are dispensable.

- Even as more knowledge is obtained, the set of facets need never change.

- For any piece of knowledge sought or obtained, there is precisely one facet of one existing[42] concept where that knowledge ought to be stored, and it is easy to determine that proper location.

- Even as more concepts are defined, the body of heuristics need not grow much.

---

[42] The only allowable exception is that a new piece of information might require the creation of a brand new concept, and then require storage somewhere on that concept.

- Any common-sense knowledge required by AM is automatically present within the heuristic rules. So, e.g., no special spatial visualization abilities are needed.

It is worth repeating here that the above assumptions are all clearly *false*. Yet none of them was too damaging to AM's behavior, and their combined presence made the creation of AM feasible.

7.2.5. Choice of Domain

> *The genesis of mathematical creation is a problem which should intensely interest the psychologist. It is the activity in which the human mind seems to take least from the outside world, in which it acts or seems to act only of itself and on itself, so that in studying the procedure of mathematical thought we may hope to reach what is most essential in man's mind.*
>
> -- Poincare'

Here are some questions this subsection will address:

- What are the inherent limitations -- and advantages -- in fixing a domain for AM to work in?
- What characteristics are favorable to automating research in any given domain?
- What are the specific reasons for and against elementary finite set theory as the chosen starting domain?

Research in various domains of science and math proceeds slightly differently. For example, psychology is interested in explaining people, not creating new kinds of people. Math is not interested in individual entities so much as new kinds of entities. There are ethical restrictions on physicians which prevent certain experiments from being done. Political experiments rarely permit backtracking, etc. Each has its own peculiarities.

If we want a system to work in many domains, we have to sacrifice some power.[43]. Within a given field of knowledge (like math), the finer the category we limit ourselves to, the more specific are the heuristics which become available. So it was reasonable to make this first attempt limited to one narrow domain.

This brings up the choice of domain. What should it be? As the DENDRAL project illustrated so clearly[44], choice of subject domain is quite important when studying how researchers discover and develop their theories. Mathematics was chosen as the domain of this investigation, because

1. In doing math research, one needn't cope with the uncertainties and fallability of

---

[43] This is assuming a system of a given fixed size. If this restriction isn't present, then a reasonable "general-purpose" system could be built as several systems linked by one giant switch.

[44] see [Feigenbaum et. al. 71]. In that case, the choice of subject was enabled by [Lederberg 64].

testing equipment; that is, there are no uncertainties in the data (compared to, e.g., molecular structure inference from mass spectrograms).

2. Reliance on experts' introspections is one of the most powerful techniques for codifying the judgmental criteria necessary to do effective work in a field; I personally have had enough training in elementary mathematics so that I didn't have to rely completely on external sources for guidance in formulating such heuristic rules. Also, several excellent sources were available [Polya, Skemp, Hadamard, Kershner, etc.].
3. The more formal a science is, the easier it is to automate. For a machine to carry out research in psychology would require more knowledge about human information processing than now is known, because psychology deals with entities as complex as you and I. Also, in a formal science, the *languages* to communicate information can be simple even though the *messages* themselves be sophisticated.
4. Since mathematics can deal with any conceivable constructs, a researcher there is not limited to explaining observed data. Related to this is the freedom to investigate – or to give up on – whatever the researcher wants to. There is no single discovery which is the "goal", no given problem to solve, no right or wrong behavior.
5. Unlike "simpler" fields, such as propositional logic, there is an abundance of heuristic rules available for the picking.

The limitations of math as a domain are closely intertwined with its advantages. Having no ties to real-world data can be viewed as a limitation, as can having no clear goal. There is always the danger that AM will give up on each theory as soon as the first tough obstacle crops up.

Since math has been worked on for millenia by some of the greatest minds from many different cultures, it is unlikely that a small effort like AM would make any new inroads, have any startling insights. In that respect, Dendral's space was much less explored. Of course math – even at the elementary level that AM explored it – still has undiscovered gems (e.g., the recent unearthing of Conway's numbers [Knuth 74]).

One point of agreement between Weizenbaum and Lederberg[45] is that AI can succeed in automating an activity only when a "strong theory" of that activity exists. AM is built on a detailed model of how humans do math research. In the next subsection, we'll discuss the model of math research that AM assumes.

Before that, consider for a moment how few other fields of human endeavor have a good model, and also enjoy all the advantages listed above: other domains of math, classical physics,... not many others.

7.2.6. Limitations of the Model of Math Research

[45] See the quote at the front of the next subsection. It is from [Lederberg 76], a review of [Weizenbaum 76]. This review also exists as file WEIZENLED[pub,jmc]@SAIL.

Weizenbaum does point to projects in mathematics and chemistry where computers have shown their potential for assisting human scientists in solving problems. He correctly points out that these successes are based on the existence of "strong theories" about their subject matter.

-- Lederberg

AM, like anything else in this world, is constrained by a mass of assumptions. Most of these are "compiled" or interwoven into the very fabric of AM, hence can't be tested by experiments on AM. Some of these were just discussed a few pages ago, in Section 7.2.4.

Another body of assumptions exists. AM is built around a particular model of how mathematicians actually go about doing their research. This model was derived from introspection, but can be supported by quotes from Polya, Kershner, Hadamard, Saaty, Skemp, and many others. No attempt will be made to justify any of these premises. On the next page is a simplified summary of that information processing model for math theory formation:

MODEL OF MATH RESEARCH

1. The order in which a math textbook presents a theory is almost the exact opposite of the order in which it was actually discovered and developed. In a text, new definitions are stated with little or no motivation, and they turn out to be just the ones needed to state the next big theorem, whose proof then magically appears. In contrast, a mathematician doing research will examine some already-known concepts, perhaps trying to find some regularity in experimental data involving them. The patterns he notices are the conjectures he must investigate further, and these relationships directly motivate him to make new definitions.
2. Each step the researcher takes while developing a new theory involves choosing from a large set of "legal" alternatives – that is, searching. The key to keeping this from becoming a blind, explosive search is the proper use of evaluation criteria. Each mathematician uses his own personal heuristics to choose the "best" alternative available at each moment.
3. Non-formal criteria (aesthetic interestingness, inductive inference from empirical evidence, analogy, and utility) are much more important than formal deductive methods in developing mathematically worthwhile theories, and in avoiding barren diversions.
4. Progress in *any* field of mathematics demands much non-formal heuristic expertise in *many* different "nearby" mathematical fields. So a broad, universal core of knowledge must be mastered before any single theory can meaningfully be developed.
5. It is sufficient (and pragmatically necessary) to have and use a large set of informal heuristic rules. These rules direct the researcher's next activities, depending on the current situation he is in. These rules can be assumed to superimpose ideally: the combined effect of several rules is just the sum of the individual effects.
6. The necessary heuristic rules are virtually the same in all branches of mathematics, and at all levels of sophistication. Each specialized field will have some of its own heuristics; those are normally much more powerful than the general-purpose heuristics.
7. For true understanding, the researcher should grasp[46] each concept in several ways: declaratively, abstractly, operationally, knowing when it is relevant, and as a bunch of examples.
8. Common metaphysical assumptions about nature and science: Nature is fair, uniform, and regular. Coincidences have meaning. Statistical considerations are valid when looking at mathematical data. Simplicity and symmetry and synergy are the rule, not the exception.

[46] Have access to, relate to, store, be able to manipulate, be able to answer questions about

7.2.7. Ultimate powers and weaknesses

Consider now *any* system which is consistent with the preceding model of math research, and whose orientation is to discover and develop new (to the system) mathematical theories. This includes AM itself, but might also include a bright high-school senior who has been taught a large body of heuristic rules.

What can such systems ultimately achieve? What are their ultimate limits? Answers to ultimate questions are hard to come by experimentally, so this discussion will be quite philosophical, speculative, and short. The model of math research hinges around the use of heuristic rules for guidance at all levels of behavior. It is questionable whether or not all known mathematics could evolve smoothly in this way. As a first order fixup, we've mentioned the need to provide good meta-heuristics, to keep enlarging the set of heuristics. If this is not enough (if meta-meta-...-heuristics are needed), then the model is a poor one and has some inherent limitations.[47] If some discoveries can only be made non-rationally (by random chance, by Gestalt, etc.) then any such system would be incapable of finding those concepts.

Turning aside from math, what about systems whose design -- as a computer program -- is similar to AM?[48] Building such systems will be "fun", and perhaps will result in new discoveries in other fields. Eventually, scientists (at least in a few very hard domains) may relegate more and more of their "hack" research duties to AM-like systems. The ultimate limitations will be those arising from incorrect (e.g., partial) models of the activities the system must perform. The systems themselves may help improve these models: experiments that are performed on the systems are actually tests of the underlying model; the results might cause revisions to be made in the model, then in the system, and the whole cycle would begin again.

7.3. Final Conclusions

Before quitting, let's summarize what's worth remembering about this thesis.

- It is a demonstration that a few hundred general heuristic rules suffice to guide an automated math researcher as it explores and expands a large but incomplete knowledge base of math concepts. AM serves as a living existence proof that creative research can be effectively modelled as heuristic search.

---

[47] If Ptolemy had had access to a digital computer, all his data could have been made to fit (to any desired accuracy), just by computing epi-cycles, epi-epi-cycles,.. to the needed number of epi's. We in AI must constantly be on guard against that error.

[48] Having an agenda of tasks with reasons and reason-ratings combining to form a global priority for each task, having units/modules/frames/Beings/Actors/concepts which have parts/slots/facets, etc. Heuristic rules are tacked onto relevant concepts, and are executed to produce new concepts, new tasks, new facet entries.

- The thesis also introduces a control structure based upon an agenda of small research tasks, each with a list of supporting reasons attached.

- The main limitation of AM was its inability to synthesize powerful new heuristics for the new concepts it defined.

- The main successes were the few novel ideas it came up with, the ease with which a new task domain was fed to the system, and – most importantly – the overall rational sequences of behavior AM exhibited.

- The greatest long-range importance of AM may well lie in the body of heuristics assembled (Appendix 3), either as the seed for a huge base of experts' heuristics, or as a new orientation for mathematics education.

Appendix 1. Glossary of Technical Terms

The "jargon" of a field facilitates communication among practitioners of that field, but it too often excludes novices. I have tried to soften the impact of each "buzz-word" when it was first used, but the reader may need to frequently refresh his memory about the meanings of certain terms.

This glossary is divided into two sections. The first contains primarily Mathematics terms, strangely biassed because it just covers what is referenced in this thesis. The second glossary, of Computer Science and Artificial Intelligence terms, suffers from the same tunnel vision. They may suffice for reading this document, but they are certainly *not* meant to be used for more general purposes.

Appendix 1.1. Glossary of Math Terms

Abduction: In logic, a syllogism of the form "from A, conclude that B is probably true". If your mental frame for an automobile contains a hundred necessary features, and you see something satisfying only 90 of them, you can *abductively* conclude it is probably an automobile.

Cardinality: the concept of "number". Two sets are of the same cardinality iff they have the same number of elements.

Composition of two relations R and S: This is a new relation denoted R∘S, and defined as R∘S(x) = R(S(x)). So R∘S maps elements of the domain of S into elements of the range of R. Notice that if R and S are both functions, then so is R∘S. The intuitive picture of this process is to operate on x with the relation S, and *then* apply R to the results.

Function: an operation f which associates, to each element x of some set D, an element f(x) of some set R. D and R are the domain and range of f. Notice that a function may be considered a special kind of relation. For a *relation* f (on DxR) to be called a *function*, f must satisfy two important constraints: (i) it must be always-defined on its domain; that is, for all domain elements x∈D, f(x) must exist. (ii) f must be single-valued; that is, f(x) must be a singleton.

Iff: if and only if; implies and is implied by; is equivalent to; <==>.

Integers: positive and negative whole numbers; i.e. ...,-2, -1, 0, 1, 2,...

Map: used as a verb, this word indicates the action of applying a function or a relation; e.g., we say that *squaring* maps 7 into 49. Used as a noun, it is a synonym for function.

Mathematical concept: this is taken to mean all the constructions, definitions, conjectures, operations, structures, etc. that a mathematician deals with. Some examples: Set-intersection, Sets, The unique factorization theorem, every entry listed in this glossary.

Mathematical intuition: this is the mental imagery which can be brought to bear. Typically, we transform the situation to an abstract, simplified one, manipulate it there, and re-translate the results into the original notation. For example, our intuition about "ordering" may involve the image of marks on a yardstick. We can then answer questions involving ordering rapidly, using this representation. Three features of the intuitive image should be noted: (i) it is typically fast and simple, (ii) it is opaque, one cannot introspect too easily on "why it works", and (iii) it is fallible, occasionally leading to wrong results.

Mathematical research: The fundamental idea here is that mathematics is an *empirical* science, just as much as chemistry or physics. In doing research, the ultimate goal is the creation of new, interesting theories, but the techniques used include looking for patterns in empirical data, inducing new conjectures, modelling some aspects of the real world, etc. Although the final product looks like a smooth, formal development, magically flowing from postulates to lemmas to theorems, the actual research process involved untold blind alleys, rough guesses, and hard work. (Analogy: The process of painting is rarely itself artistic.)

Mathematical theory: to qualify as a theory, we must have (i) a basis of undefined primitive terms, (ii) definitions involving these, (iii) axioms involving all the primitives and defined terms (iv) conjectures and theorems relating these terms. To be at all worthwhile, however, the theory must also meet the fuzzy requirements that (v) there is some correspondence between the primitives and some "real-world" concepts, between the axioms and some "real" relationships, and (vi) some of the theorems are unexpected, hard to prove, elegant, interesting, etc.

Mersenne prime: a prime number which happens to be of the form 2^P-1, where p is prime.

Natural numbers: non-negative integers; i.e., 0, 1, 2, 3,...

No.: an abbreviation for "Number".

Number: in the typical loose fashion of computer scientists, I intend this to mean a non-negative integer: i.e., a *natural* number.

Ordering: the concept of "before" and "after". This distinguishes a list from a bag (multiset). The formal axioms for ordering simply state the obvious properties of the intuitive image of a list.

Prime numbers: natural numbers which have no divisors other than 1 and themself; e.g., 17, but *not* 15 (=3x5). Primes are interesting because of the myriad times they crop up in diverse theorems – from the Chinese Remainder Theorem (solving systems of linear congruence equations), to the Law of Quadratic Reciprocity, to Fermat's Theorem (for all integers n, for all primes p, n^P is congruent to n (mod p)). The "secret" of their value lies in the fact that all integers can be factored *uniquely* into a set of prime divisors. This "Unique Factorization Theorem" lets us reduce questions about integers to questions about primes.

Prime pairs: two prime numbers whose difference is two; e.g., 17 and 19.

Relation: an operation which associates, for each element of some set D, a set of elements E = $\{e_1, e_2,...\}$ of some set R. D and R are the domain and range of the relation. For example,

the relation "≤" associates to 5 the set of numbers {5, 6, 7, 8,...} – i.e., all integers which 5 is less than or equal to. The domain and range of this relation are the integers.

Set-theoretic: having to do (in the context of this thesis) with elementary finite set theory, and the primitive notions of mathematics (e.g., union, insert, predicate, conjecture).

Unity: a fancy way of referring to the natural number "1".

|: The relation "divides-evenly-into". Thus we say 2|6.

¬: The operation of negation. "¬X" is read as "not X".

∨: Disjunction. "A∨B" is read as "A or B".

∧: Conjunction. "A∧B" is read as "A and B".

●: Exclusive or. "A●B" is read as "A or B, but not both".

→: Implication. "A→B" is read as "If A then B".

↔: Logical equivalence. "A↔B" is read as "A if and only if B".

∀: Universal quantification. "∀X" is read as "For all X".

∃: Existential quantification. "∃X" is read as "For some X".

Appendix 1.2. Glossary of AI Terms

ACTORs: A modular form of representation, useful for distributing of the task of *control* among several components in a computer program. Each ACTOR is a black box, with no parts or slots, but which does have some assertions (a "contract") which he must honor. It merely responds to a fixed set of messages, by sending out certain messages of his own. These are delivered via a bureaucracy. See [Hewitt 76].

AI: an abbreviation for Artificial Intelligence.

Bag: A bag is a kind of list structure, a bunch of elements which are unordered, but one in which multiple copies of the same element are permitted. One may visualize a paper bag filled with cardboard letters. Technically, we shall say that a set is *not* considered to be a bag. A bag is denoted by enclosure within parentheses, just as sets are within braces. So the bag containing X and four Y's might be written (X Y Y Y Y), and would be considered indistinguishable from the bag (Y Y Y X Y).

BEINGs: A modular form of representation of knowledge, conceived as a collection of cooperating experts. Each expert is modelled by one module, which consists of a list of Question/Answering-program pairs. The set of questions is fixed for all the Beings in the system. When any Being has a question, he broadcasts it to the entire system, and some Being who recognizes it will take over control and try to answer it by running *his* appropriate Answering-program. In the process of running this, some new questions may arise. Notice that Beings distribute responsibility for control and for static knowledge. See [Lenat 75b].

Bug: a flaw in a computer program. As Corey Sacerdoti put it, a bug refers to something which is broken but not badly.

Concept: within the context of this document, the word "concept" typically refers to a precise frame-like data structure, a BEING. Semantically, each concept is meant to correspond to one abstract entity that we would intuitively call a concept: an object, an operator, a conjecture, etc. See "facet".

Cooperating Knowledge Sources: Very often, in tackling a problem, one receives some hints and some constraints from very different sources, phrased in very different languages, often addressing different representations of the problem. For example, in trying understand a human speaker, our memory of the previous discussion and knowledge of the speaker may narrow down the possible *meanings* of what he is saying. Our ears, of course, register the precise acoustic wave-forms he is uttering. Our English vocabulary forces us to interpret imperfect signals as real words. Our eyes see his gestures and his lip movements, and give us more information. All these different sources of information must be used, and yet they all are talking in different "languages" to us. The most trivial solution is to keep all the sources independent, and keep working until one of them can solve the problem all by itself. A much better solution is to transform all their babblings into one canonical representation, one single language. This way, all the knowledge sources can cooperate.

Coupled: two functional subsystems are causally connected; one influences the other. See the entry for "Linear".

CPU time: Central-Processing-Unit runtime (cpu time) is the number of execution cycles of the computer that the AM program has used up. This is conveniently measured in seconds, minutes, and hours, where one cpu minute is the amount of processing done in one minute of real time, when AM has 100% of the machine, and is runninng without any input or output.

CS: an abbreviation for Computer Science.

Execution: a program is actually used by *running* it on a particular set of input data. This process is known as program *execution*.

Facet: Within the context of this document, the word "facet" denotes a slot of the kind of data-structure known as "concepts" (qv). Thus "*a facet of the Compose concept*" really just means a slot of a particular frame, a part of certain BEING, one single attribute/value pair taken from the property list of the Lisp atom named Compose. Semantically, each facet holds information pertaining to a single aspect of the concept it is a part of; hence the suggestive name: "facet".

FRAMEs: A modular representation of knowledge. Each module is a list of Feature/Value pairs. The *value* represents a default assumption which can be relied on until/unless new information comes in about that feature. Each frame has whatever *features* (called "slots") seem appropriate. Whenever a situation S is encountered, the frame(s) for S are activated. As new information rolls in, it replaces the default information in various slots. Notice the emphasis on distributing static knowledge (*data*), not necessarily control, in such a system. See [Piaget 55] or [Minsky 75].

Function: a small, executable part of a program. When fed the proper kind of argument(s), a function will "run" and ultimately produce some sort of value. Unlike pure mathematical functions (see the previous glossary), a Lisp function can have side effects (qv).

Garbage collection: As a Lisp program executes, various list structures (pointer networks) are created. When the last pointer to a structure is removed, that structure has essentially been irretrievably forgotten. If the operating system knew which storage cells were thus "free", it could re-cycle them, reuse them. The process of finding and liberating such discarded lists is called garbage collection. This is performed automatically by the Lisp language, whenever space is almost all filled up.

Hack: A quick job that produces what is needed, but not well. Introducing a heuristic which was only used once, in a predetermined way (e.g., to fix a particular bug), would be a real hack.

Hand-crafting: the human programmer carefully designs his system in such a way that the pieces just manage to mesh. For instance: he provides just the perfect set of axioms so that his theorem-prover can solve a certain problem, or he modifies the program's strategies so that they efficiently manipulate the axiom set in just the right way.

Heterarchy: A kind of control structure for a computer program which is distinct from hierearchy. Heterarchical structuring views the whole program as a collection of equal partners, an unstructured set of functions. "Control" is viewed as a spotlight, which can be

flicked from one function to another. The functions can affect who does or doesn't get control next, but there is no guarantee who will get control, or that control will revert back to some function which once had it. Aside from the lure of its democratic flavor, it is clearly a natural way to represent cooperating knowledge modules.

Hierarchy: This term refers to a kind of control structure for a computer program. The typical hierarchical structure is one in which a function calls a subroutine, which processes and then returns a value to that function. A program is viewed as a tree structure, with lines indicating "calling".

Interact: a dynamic mode of communication between a human and a computer program. The human reacts to what the program is printing out on his terminal, and the program in turn reacts to what the user types in. This may take the form of questioning and answering, or interrupting and commenting.

Interestingness: Note that this is not a valid English word. In the context of AM, it refers to a numeric value, computed by little Lisp programs stored in the "Interest" facets of various concepts. Despite the danger of imbuing such a humble scheme with all the mystique of what is and isn't interesting, it is felt that a sufficient component of that evaluation has been captured to warrant the name. Pragmatically, it is of much more use to the user to see "Interestingness of Compose has just risen" than to see a message like "G00034 incremented".

Kludge (or Kluge): This is a program feature which is an unfair shortcut around a specific problem. One "kludgy" way of improving the algorithm of a given concept is to ask the user for a better algorithm.

Linear: a system whose components, inputs, and outputs *superimpose* -- i.e., don't couple.

Lisp: a LISt-Processing programming language. Primitive operations exist for manipulating nested list structures. Since Lisp functions are also merely lists, it is easy to create and modify entities which are then executed (qv).

Modular Representations of Knowledge in AI Systems: Knowledge is partitioned into packets (called modules, frames, units, productions, Beings, experts, Actors) along lines of: different applicabilities, expertise, purpose, importance, generality, etc. Each packet is structurally similar to all the rest. Advantages: By having the knowledge discretized, pieces can be added and/or removed with no trouble. The knowledge of the system is easily inspected and analyzed. The structural similarity yields several advantages: a simple control system suffices to "run" all the knowledge, the modules can intercommunicate easily, new modules can be inserted without knowing precisely "who else" is already in the system. In general, the less similarly-structured the modules are, the simpler the inter-communication media must be. Modular representation is a natural way to implement cooperating knowledge sources.

Number: in the typical loose fashion of computer scientists, I intend this to mean a non-negative integer: i.e., a *natural* number.

Open research problem: a limitation of the AM system.

Recur: Often, part of a definition will refer back to that very same definition. This may lead to an infinite circular loop, or it may terminate. The following definition of "is larger than" is recursive, because the last line recurs:

```
set R is larger than set S
    if R={} but S≠{}, or
    if neither is empty and
        Remove-element(R) is larger than Remove-element(S).
```

Recurse: a transitive verb which means "to swear again." It must be distinguished from "recur", above.

Side effects: while a function is executing, it may cause changes in the state of its environment which persist even after the function has returned a value. This is like hysteresis effects. For example, a function may create or destroy some list structure, define a new function, reset some variable, etc. Such activities are called side effects of the function.

Space: The memory of a computer is quite finite. Though it may be supplemented by slow auxilliary devices (tapes, discs, etc.), the actual number of storage cells in the computer's fast "core" memory is a limiting factor in program behavior. Storage space, or just "space", refers to these internal memory cells. When space is exhausted, the only remedy is to perform a garbage collection (qv).

System: this can mean a computer program, and occasionally is just an another way of referring to AM. In general, a system is any collection of entities related to form a meaningful whole.

Terminal: a communications device for passing information between a computer system and a human. This could be a teletype, a TV screen and keyboard, etc. The terminal is usually portable and remotely located from the computer.

User: the human being who sits at a computer terminal and watches AM run (occasionally, perhaps, interacting with AM).

Appendix 2. AM's Concepts

The first part of this huge appendix (Appendix 2.1.2 to 2.1.75) lists the set of knowledge AM started with: its initial concepts. It is not very readable, nor is it central to any of the *ideas* on which AM is based. The reader is therefore warned to proceed at his own risk through this material.

Section 2 of this appendix contains a brief description of those concepts which were only partially implemented in AM (e.g., "Destructive-op"). It was decided not to give each of them a full "box" of their own.

The third part of this appendix lists a couple concepts as they were actually coded into Lisp. The reader is shown which entry – or heuristic rule – each bit of Lisp code corresponds to.

Finally, starting on page 224, a list is provided of some of the concepts which AM created. This is intended not as an exhaustive catalog, but merely to show the breadth of what was done by AM, the smart guesses and the lunacies. This list could have been pieced together by studying Appendix 5, wherein some examples of AM in action are given. There the reader may dynamically observe what kinds of concepts – and infer what kinds of entries for *their* facets – AM was able to derive from its initial base.

Appendix 2.1. Initial Concepts

Each concept will be listed, followed by a description of the entries in each of its facets[1]. For each such "slot", a condensation is provided (in English, LISP, and math notation) of all the knowledge initially supplied to AM about that facet of that concept.

If there is any unmentioned facet for a concept, then it started out blank. Many of the facets of the original concepts were left blank intentionally, knowing that AM would be able to fill *them* in as well. After all, if you can fill in examples of any new concept, you ought to be able to fill in examples of Sets!

The concepts are grouped semantically, much like the tree shown on page 105, like the order in which heuristics are listed in Appendix 3. This section of the appendix is prefaced by an index which is arranged alphabetically, since the primary use of it will probably be as an encyclopedia. When the reader encounters a poorly-named or poorly-explained concept somewhere in the text, he may wish to glance first at Chapter 5, page 107, where very brief definitions of the concepts are also given alphabetically. If that "dictionary" is insufficent, he can turn to the appropriate page in this appendix, and see the same concept presented in much more detail.

---

[1] Each of these entries was supplied by hand, by the author.

Appendix 2.1.1 Index to Initial Concepts

CONCEPT	PAGE	CONCEPT	PAGE
Active	175	Multiple-elements-structure	210
All-but-the-first-element	201	No-multiple-elements-structure	211
All-but-the-last-element	202	Nonempty-structure	211
Any-concept	174	Object	207
Anything	174	Object-equality	176
Atom-obj	208	Operation	177
Bag-Delete	184	Ord-Structure	210
Bag-Diff	194	Ordered-pairs	213
Bag-insert	182	Oset-Delete	185
Bag-Intersect	189	Oset-Diff	193
Bag-Union	191	Oset-insert	181
Bags	212	Oset-Intersect	187
Canonize	196	Oset-Union	190
Coalesce	195	Osets	214
Compose	178	Parallel-join	199
Conjecture	207	Parallel-join2	199
Constant-False	177	Parallel-replace	197
Constant-predicate	176	Parallel-replace2	197
Constant-True	176	Predicate	175
Delete	183	Projection1	203
Difference	192	Projection2	203
Empty-structure	211	Relation	206
First-element	201	Repeat	198
Identity	204	Repeat2	198
Insert	179	Restrict	204
Intersect	186	Reverse-ord-pair	200
Invert-an-operation	205	Set-Delete	183
Inverted-op	205	Set-Diff	193
Last-element	200	Set-insert	180
List-Delete	184	Set-Intersect	188
List-Diff	192	Set-Union	191
List-insert	182	Sets	212
List-Intersect	186	Structure	209
List-Union	190	Structure-of-Structures	209
Lists	213	Truth-value	208
Logical-combination	206	Union	189
Member	202	Unord-Structure	210

Appendix 2.1.2 Anything

Name(s): Anything, Entity, Thing, Item
Definitions:
Non-Recursive, Trivial, Quick: λ () T
Specializations: Any-concept, Non-concepts
Generalizations: none
Examples: Anything, Any-concept
Isa's: Any-concept
Worth: 100
Interest: 5 heuristics (see Appendix 3.1, page 229).[2]
Sugg: 5 heuristics
In-domain-of: Delete, Insert[3], Member, Proj1, Proj2, Identity, Constant-pred.
In-range-of: First-ele, Last-ele, Member, Proj1, Proj2, Identity.

Appendix 2.1.3 Any-concept

Name(s): Any-concept, Any-Being, Anybody
Definitions:
Non-Recursive, Opaque, Quick: λ (x) FMEMB(x,Concepts)
Non-Recursive, Opaque, Quick: λ (x) GETP(x,Name)
Specializations: Active, Object
Generalizations: Anything
Examples: Anything, Any-concept, Active, Object
Isa's: Anything, Any-concept
Worth: 100
View: to view any X as if it were a Y, find an op. whose domain contains[4] X, and whose range is contained in Y, and apply that op. to the given X.
Fillin: 39 heuristics (see Appendix 3.2, beginning on page 230).[5]
Check: 20 heuristics
Interest: 21 heuristics
Sugg: 30 heuristics

---

[2] In general, this appendix will omit heuristics. They will instead be presented in one big collection, as the next appendix. For each concept, we will however mention how many heuristics of each variety are present. The interested reader may turn immediately to Appendix 3 if he desires, to see those heuristic rules.

[3] All four specializations of each of Delete (e.g., Bag-delete) and Insert (e.g., List-insert) are also listed here.

[4] That is, the domain of the operation is D1xD2xD3..., and X is a subset of some Di, a specialization of Di.

[5] As usual, the heuristics are listed in Appendix 3, not here. But the reader is forewarned that this concept has so many heuristics that they are grouped by facet in the next appendix, occupying Appendices 3.2.1 through 3.2.8, pages 230 to 251.

Appendix 2.1.4 Active

Name(s): Active, activity, action
Definitions:
Sufficient, Non-Recursive, Quick: λ (x) GETP(x,Algorithms)
Sufficient, Non-Recursive, Quick: λ (x) GETP(x,Dom/range)
Specializations: Predicate, Relation, Operation
Generalizations: Any-concept
Examples: none.[6]
Isa's: Any-concept
In-domain-of: Constructive, Destructive, Coalesce, Compose, Restrict
In-range-of: Compose, Coalesce, Restrict.
Worth: 100
Fillin: 7 heuristics.
Check: 4 heuristics
Interest: 3 heuristics
Sugg: 10 heuristics

Appendix 2.1.5 Predicate

Name(s): Predicate, sometimes: logical operation, Boolean function.
Definitions:
Nonrecursive quick opaque: λ (P) Range(P) is Truth-value; i.e., {T,F}.
Generalizations: Active
Examples: Equality, Constructive, Destructive, Empty, Nonempty, Constant-pred,
the Defn entries of each concept.[7]
In-domain-of: Canonize
Worth: 100
Fillin: 2 heuristics.
Sugg: 1 heuristic.
Interest: 1 heuristic.

---

[6] Recall that each active will be an example of an operation, predicate, etc, hence need not be pointed to explicitly here.

[7] Thus the predicate 'Empty', while it exists in AM, is superfluous, since the definition facet of 'Empty-struc' contains that very predicate.

Appendix 2.1.6 Object-equality

Name(s): Equality, Object equality, Obj-equal, Equal, Same.
Definitions:
Nonrecursive opaque: λ (x,y) EQUAL(x,y)
Sufficient, very quick, opaque: λ (x,y) EQ(x,y)).
Recursive slow: λ (x,y) x and y are both identical atoms,
or x and y are both empty structures,
or x and y are both nonempty structures and
Equality.Defn(CAR(x),CAR(y)) and
Equality.Defn(CDR(x),CDR(y)).
Nonrecursive transform slow: λ (x y) Identity.Defn(x,y).
Quick: λ (x,y) y=Equality.Algs(x).
Domain/range: <Object Object → {T,F}>
<Structure Structure → {T,F}>
Algorithms:
Nonrecursive quick: λ (x) x.
Conjec: 'Identity, restricted to Objects, is the same as Obj-Equality.'
Isa's: Predicate
Worth: 200
What: the Equality of two list structures; closely related to Identity op.

Appendix 2.1.7 Constant-predicate

Name(s): Constant-predicate, Const pred, Logical constant function.
Definitions: none.
Domain/range: <Anything... Anything → {T,F}>
Isa's: Predicate
Specializations: Constant-True, Constant-False
Conjec: (∀x,∀y) Constant-pred.Defn(x)=Constant-pred.Defn(y).
Worth: 100
What: a predicate which always returns the same logical value.

Appendix 2.1.8 Constant-True

Name(s): Constant-True, Constant T, Always-T, sometimes: Always.
Definitions:
Nonrecursive, very quick: λ (...) T.
Domain/range: <Anything... Anything → {T,F}>
<Anything... Anything → {T}>
Generalizations: Constant-Predicate
Worth: 100
What: a predicate which always returns True.

Appendix 2.1.9 Constant-False

Name(s): Constant-False, Constant F, Always-F, sometimes: Never.
Definitions:
Nonrecursive, very quick: λ (...) F[8]
Domain/range: <Anything... Anything → {T,F}>
<Anything... Anything → {F}>
Generalizations: Constant-Predicate
Worth: 100
What: a predicate which always returns False.

Appendix 2.1.10 Operation

Name(s): Operation, sometimes: function, mapping.
Definitions: none.[9]
Specializations: Inverted-op, Composition, Canonization,
Coalesced-op, Constructive-ω,[10]
Generalizations: Active
Examples: Insert, Delete, Union, Intersect, Difference, Compose, Canonize,
Coalesce, Identity, Proj1, Proj2, First-ele, Last-ele, All-but-first-ele,
All-but-last-ele, Restrict, Reverse-ord-pair, Member, Invert, Repeat(2),
Parallel-join(2), Parallel-replace(2).
In-domain-of: Invert, Parallel-join(2), Parallel-replace(2), Repeat(2).
In-range-of: Canonize, Invert, Parallel-join(2), Parallel-replace(2), Repeat(2)
Worth: 100
Fillin: 7 heuristics.
Check: 3 heuristics
Interest: 11 heuristics
Sugg: 2 heuristics

---

[8] Actually, the value returned is 'NIL', not False or F.

[9] Recall that all this means is that computationally, any entity x is considered to be an Operation iff it is in Operation.Exs, or if it is an example of some Specialization of this concept.

[10] The concepts of Constructive and Destructive operations are not encoded as concepts yet. The distinction between specialization of Operation and Example of operation is quite blurry. E.g., why not consider th class of Insertion operations a whole specialization of Operation, instead of just an example? The decision as to what status each operation would have was quite arbitrary, I'm afraid.

Appendix 2.1.11 Compose

Name(s): Compose, Composition, sometimes: afterwards;
Definitions:
Declarative slow: λ (A,B,C) ∀x, C(x)=A(B(x)).
Sufficient Nonrecursive Quick: λ (A,B,C) C has the Name 'A∘B'.
Sufficient, Slow: Are-equivalent(C,Compose.Algs(A,B)).
Sufficient, Quick: C=Compose.Algs(A,B).
Domain/range: <Active Active → Active>
<Operation Active → Operation>[11]
<Predicate Active → Predicate>
<Relation Relation → Relation>
Algorithms: [12]
Distributed: use the heuristics attached to Compose to guide the filling in of various facets of the new composition.
Generalizations: Operation
Isa's: Operation
Worth: 300
Fillin: 9 heuristics.
Check: 2 heuristic.
Suggest: 2 heuristics.
Interest: 11 heuristics.

[11] Note that while this entry would imply that Operation.In-ran-of and Operation.In-dom-of could both contain 'Compose' as an entry, only the most general concept (i.e., 'Active') has 'Compose' in its In-dom-of and In-ran-of facets.

[12] An algorithm for COMPOSE is a procedure for taking a pair of operations, i.e. a pair of concepts G and H, and creating a new active concept F which is defined to be their composition, whose Algorithms facet contains 'λ (x) G(H(x))', or, more precisely, '(APPLYB G ALGS (APPLYB H ALGS x))'.

Appendix 2.1.12 Insert

Name(s): Insert, Insertion, sometimes: Add, Merge;
Definitions:
Quasi-recursive cases: λ (x,A,B) [determine the type of structure that A and B are, say S, then use S-insert.Defn(x,A,B)].
Necessary, Nonrecursive, Quick: λ (x,A,B) Member.Defn(x,B).
Necessary Declarative: λ (x,A,B) z∈B iff z∈A or z=x.
Necesary, Declarative: λ (x,A,B) [(∀a∈A)(a∈B), and (∀b≠x ∈B)(b∈A), and x∈B]
Sufficient, Quick: B=Insert.Algs(x,A).
Domain/range: ⟨Anything, Structures → Structures⟩
Algorithms:
Quasi-recursive cases: λ (x,A) [determine the type of structure A is, say S, then use S-insert.Algs(x,A)].
Isa's: Operation
Specializations: Bag-insert, Set-insert, List-insert, Oset-insert.
Worth: 100
Check: 1 heuristic.

Appendix 2.1.13 Set-insert

Name(s): Set-insert, Set insertion, sometimes: insert, Tag.
Definitions:
Declarative Slow: λ (x,A,B) [(∀a∈A)(a∈B), and (∀b∉x ∈B)(b∈A), and x∈B]
Recursive Slow: λ (x,A,B) (A={} and B={x}, or else:
[AND: z←Member.Alg(A); Member.Defn(z,B);
Set-insert.Defn(x,Set-delete.Alg(z,A),Set-delete.Alg(z,B))])
Recursive: λ (x,A,B) (A={} and B={x}, or else:
[AND: z←CAR(A); Member.Defn(z,B);
Set-insert.Defn(x,CDR(A),Set-delete.Alg(z,B))])
Declarative: λ (x,A,B) (∀z) z∈B iff z∈A ● z=x.
Quick: B=Set-insert.Alg(x,A).
Domain/range: <Anything, Sets → Sets>
Algorithms:[13]
Non-recursive quick: λ (x,A) (if Member Defn(x,A) then A, else MERGE(x,A)))
Non-recursive quick: λ (x,A) (MERGE(x,A) and Elim-adjacent-mult-elements(A))
Recursive: λ (x,A) (if A={} then {x}, else if A={x} then A, else
[z←CAR(A); if z=x then A, else CONS(z,Set-insert.Alg(x,CDR(A)))]).
Generalizations: Insert
Worth: 100
What:[14] If x isn't already in A, then add it and re-sort the set A.

[13] Actually, this operation, like all the other structural operations, are much more sophisticated than this simple presentation implies. In this case, if A is not supplied, AM chooses a random example of a Set and inserts x into that set. If x is missing, then AM finds a random example of Anything and inserts it into A.

[14] The 'What' facet doesn't really exist, but is occasionally present in this Appendix for the aid of the reader. A fuller English description of any concept can be obtained by looking in the alphabetical summary of concepts, in Chapter 5, beginning on page 107.

Appendix 2.1.14 Oset-insert

Name(s): Oset-insert, Oset insertion, sometimes: Insert;
Definitions:
Declarative Slow: λ (x,A,B) [(∀a∈A)(a∈B), and (∀b∉x ∈B)(b∈A), and x=CAR(B)]
Recursive Slow: λ (x,A,B) (A=[] and B=[x], or else:
[AND: z←Member.Alg(A); Member.Defn(z,B);
Oset-insert.Defn(x,Oset-delete.Alg(z,A),Oset-delete.Alg(z,B))])
Non-recursive, Quick: λ (x,A,B) (B=CONS(x,Oset-delete.Algs(x,A)).
Quick: λ (x,A,B) (B=Oset-insert.Algs(x,A)).
Necessary Quick: λ (x,A,B) (x=CAR(B)).
Necessary, Declarative: λ (x,A,B) (∀z) z∈B iff z∈A ● z=x.
Domain/range: <Anything, Osets → Osets>
Algorithms:
Non-recursive quick: λ (x,A) (CONS(x,[if Member.Defn(x,A) then DREMOVE(x,A)[15], else A]))
Non-recursive quick: λ (x,A) (CONS(x,A) and DREMOVE(x,CDR(A)))
Non-recursive quick: λ (x,A) (CONS(x,DREMOVE(x,A))
Recursive: λ (x,A) (if A=[] then [x], else if A=[x...] then A, else
CONS(x,Oset-delete.Algs(x,A))).
Generalizations: Insert
Worth: 100
What: Eliminate x from A and add x as the first element of the oset A.

[15] The INTERLISP function DREMOVE(x,A) destructively removes all occurrences of x from the list structure A.

Appendix 2.1.15 List-insert

Name(s): List-insert, List insertion, sometimes: Insert, sometimes: CONS;
Definitions:
Nonrecursive Quick: λ (x,A,B) (B=CONS(x,A)).
Nonrecursive: λ (x,A,B) (A=CDR(B) and x=CAR(B)).[16]
Quick: λ (x,A,B) (B=List-insert.Algs(x,A)).
Necessary Quick: λ (x,A,B) (x=CAR(B)).
Necessary Quick: λ (x,A,B) (A=CDR(B)).
Domain/range: <Anything, Lists → Lists>
Algorithms:
Non-recursive quick: λ (x,A) CONS(x,A).
Recursive slow: λ (x,A) (if A=◇ then <x>, else
NCONC1[17](List-insert.Algs(x,All-but-last.Algs(A)),CAR(A)).
Generalizations: Insert
Worth: 100
What: Add the element x onto the front of the List A.

Appendix 2.1.16 Bag-insert

Name(s): Bag-insert, Bag insertion, sometimes: Insert;
Definitions:
Nonrecursive Quick: λ (x,A,B) (B=SORT(CONS(x,A))).
Quick: λ (x,A,B) (B=Bag-insert.Algs(x,A)).
Domain/range: <Anything, Bags → Bags>
Algorithms:
Non-recursive quick: λ (x,A) MERGE(x,A).
Non-recursive: λ (x,A) SORT(CONS(x,A)).
Recursive slow: λ (x,A) (if A=() then (x), else
if CAR(A)<[18]x then CONS(CAR(A),Bag-insert.Algs(x,CDR(A))), else CONS(x,A)).
Generalizations: Insert
Worth: 100
What: Merge the element x into the Bag A.

---

[16] Here's how this would really appear in LISP: (LAMBDA (x A B) (AND [APPLYB OBJ-EQUAL ALGS A (APPLYB ALL-BUT-FIRST-ELE ALGS B)] [APPLYB OBJ-EQUAL ALGS x (APPLYB FIRST-ELE ALGS B)])).

[17] This LISP function means 'λ(S,x) add the element x to the end of list S'. CDR means All-but-the-first-element, CAR means The-first-element.

[18] Here, 'less than' means 'precedes alphanumerically', using ALPHORDER.

Appendix 2.1.17 Delete

Name(s): Delete, Deletion, Remove, sometimes, Subtract;
Definitions:
Quasi-recursive cases: λ (x,A,B) [determine the type of structure A and B are, say S, then use S-Delete.Defn(x,A,B)].
Slow[19]: λ (x,A,B) List-delete.Defn(x,A,B)
Sufficient, Nonrecursive, Quick: λ (x,A,B) NOT(Member.Defn(x,B)).
Sufficient, Quick: B=Delete.Algs(x,A).
Domain/range: <Anything, Structures → Structures>
Algorithms:
Quasi-recursive cases: λ (x,A) [determine the type of structure A is, say S, then use S-Delete.Algs(x,A)].
Slow: λ (x,A) List-delete.Algs(x,A).
Isa's: Operation
Specializations: Set-delete, List-delete, Oset-delete, Bag-delete.
Worth: 100
What: Remove (one occurrence of) x from (the front of) structure A.

Appendix 2.1.18 Set-Delete

Name(s): Set-Delete, Set Deletion, sometimes: Delete;
Definitions:
Declarative Slow: λ (x,A,B) (∀a∈A)(a∈B xor a=x) ∧ (∀b∈B)(b∈A) ∧ ¬x∈B
Recursive Slow: λ (x,A,B) (A={} and B={}, or else A={x} and B={}, or else:
[AND: z←Member.Alg(A) until z≠x; Member.Defn(z,B);
Set-Delete.Defn(x,Set-delete.Alg(z,A),Set-delete.Alg(z,B))])
Quick: B=Set-Delete.Algs(x,A).
Domain/range: <Anything, Sets → Sets>
Algorithms:
Non-recursive quick: λ (x,A) DREMOVE(x,A)
Non-recursive quick: λ (x,A) (if NOT(Member.Defn(x,A)) then A, else DREMOVE(x,A)))
Recursive: λ (x,A) (if A={} then {}, else if A={x} then {}, else
[z←CAR(A); if z=x then CDR(A), else CONS(z,Set-Delete.Alg(x,CDR(A)))]).
Generalizations: Delete
Worth: 100
What: remove the element x from the set S, if it's there initially.

---

[19] The List-delete definitions and algorithms are relatively slow, since x might occur anywhere in A, and it might occur more than once. Special tricks are available to speed up the other kinds of deletions. For Set-delete and Oset-delete, we can use DREMOVE, since deleting all occurrences of x is fine -- there can only be at most one occurrence. For Bag-delete, we can walk down the bag and quit when any element is seen to be alphabetically-greater-than x. These speed-ups are the reason for maintaining four separate kind of deletion operations.

Appendix 2.1.19 Bag-Delete

Name(s): Bag-Delete, Bag Deletion, sometimes: Delete;
Definitions:
Recursive Slow: λ (x,A,B) (A=() and B=(), or else (A=(x...) and B=CDR(A),
or else Bag-delete.Defn(x,CDR(A),CDR(B).
Quick: B=Bag-Delete.Algs(x,A).
Domain/range: <Anything, Bags → Bags>
Algorithms:
Non-recursive quick opaque[20]: λ (x,A) [z←(MEMBER(x,A);
RPLACA(z,CADR(z)); RPLACD(z,CDDR(Z))]
Recursive: λ (x,A) (if A=() then (), else if CAR(A)=x then CDR(A), else
CONS(CAR(A),Bag-Delete.Alg(x,CDR(A)))).
Generalizations: Delete
Worth: 100
What: remove one copy of x from the Bag A, if x was int there initially.

Appendix 2.1.20 List-Delete

Name(s): List-Delete, List Deletion, sometimes: Delete;
Definitions:
Recursive Slow: λ (x,A,B) (A=◇ and B=◇, or else CAR(A)=x and CDR(A)=B,
or else List-delete.Defn(x,CDR(A),CDR(B).
Quick: B=List-Delete.Algs(x,A).
Domain/range: <Anything, Lists → Lists>
Algorithms:
Non-recursive quick opaque: λ (x,A) FRPLACD(z←(MEMBER(x,A),CDDR(Z))
Recursive: λ (x,A) (if A=◇ then ◇, else if CAR(A)=x then CDR(A), else
CONS(CAR(A),List-Delete.Alg(x,CDR(A)))).
Generalizations: Delete
Worth: 100
What: remove the first copy of x from the List A, if x is in A.

---

[20] This algorithm is labelled Opaque because it contains very tight 'sneaky' code, implementing a highly non-standard linked data structure deletion algorithm. The call on the interlisp function MEMBER binds z to the tail of A, beginning with the first occurrence of x.

Appendix 2.1.21 Oset-Delete

Name(s): Oset-Delete, Oset Deletion, sometimes: Delete;
Definitions:
Recursive Slow: λ (x,A,B) (A=[] and B=[], or else CAR(A)=x and CDR(A)=B, or else Oset-delete.Defn(x,CDR(A),CDR(B).
Recursive Slow: λ (x,A,B) (A=[] and B=[], or else A=[x] and B=[], or else: [AND: z←Member.Alg(A) until z≠x; Member.Defn(z,B); Set-Delete.Defn(x,Set-delete.Alg(z,A),Set-delete.Alg(z,B))])
Necessary Quick: λ(x,A,B) (CAR(A)=CAR(B) xor CAR(A)=x).
Quick: B=Oset-Delete.Algs(x,A).
Domain/range: [Anything, Osets → Osets]
Algorithms:
Non-recursive quick opaque: λ (x,A) DREMOVE(x,A).
Non-recursive quick opaque: λ (x,A) FRPLACD(z←(MEMBER(x,A),CDDR(z))
Recursive: λ (x,A) (if A=[] then [], else if CAR(A)=x then CDR(A), else CONS(CAR(A),Oset-Delete.Alg(x,CDR(A)))).
Non-recursive quick: λ (x,A) (if NOT(Member.Defn(x,A)) then A, else DREMOVE(x,A)))
Non-recursive quick: λ (x,A) DREMOVE(x,A)
Recursive: λ (x,A) (if A=[] then [], else if A=[x] then [], else [z←CAR(A); if z=x then CDR(A), else if z>x then A, else Oset-Delete.Alg(x,CDR(A)))]).
Generalizations: Delete
Worth: 100
What: remove the element x from the Oset A, if it's present there initially.

Appendix 2.1.22 Intersect

Name(s): Intersect, Intersection, sometimes: Product;
Definitions:
Quasi-recursive cases: λ (A,B,C) [determine the type of structure A and B are, say S, then use S-Intersect.Defn(A,B,C)].
Slow: λ (A,B,C) List-intersect.Defn(A,B,C)
Necessary, Nonrecursive: λ (A,B,C) Member.Defn(x,C) iff Member.Defn(x,A) and Member.Defn(x,B).
Quick: C=Intersect.Algs(A,B).
Domain/range: <Structures Structures → Structures>
Algorithms:
Quasi-recursive cases: λ (A,B) [determine the type of structure A and B are, say S,[21] then use S-Intersect.Algs(A,B)].
Slow: λ (A,B) List-Intersect.Algs(A,B).
Isa's: Operation
Specializations: Set-intersect, Bag-intersect, List-intersect, Oset-intersect.
Worth: 100

Appendix 2.1.23 List-Intersect

Name(s): List-Intersect, List-Intersection, sometimes: Intersect.
Definitions:
Recursive slow: λ (A,B,C) if A=◇ then C=◇, else if Member.Defn(CAR(A),B) then [CAR(A)=CAR(C) and List-intersect.Defn(CDR(A),List-delete.Alg(CAR(A),B),CDR(C))], else List-intersect.Defn(CDR(A),B,C).
Quick: C=List-Intersect.Algs(A,B).
Domain/range: <Lists Lists → Lists>
Algorithms:
Non-recursive: λ (A,B) [for each x in A (in order), do the following: if Member.Defn(x,B) then List-delete.Alg(x,B), else List-delete.Alg(x,A). Finally, return the value of 'A' as the result.
Recursive: λ (A,B) if A=◇ then ◇, else if Member.Defn(CAR(A),B) then CONS(CAR(A),List-intersect.Alg(CDR(A),List-delete.Alg(CAR(A),B))), else List-intersect.Alg(CDR(A),B).
Generalizations: Intersect
Worth: 100
What: Move along list A. Remove it (once) from B if it's there, else from A. Return A.

[21] S might be 'Sets', or S can be 'Lists', etc

Appendix 2.1.24 Oset-Intersect

Name(s): Oset-Intersect, Oset-Intersection, sometimes: Intersect.
Definitions:
Recursive[22]: λ (A,B,C) if A=[] then C=[], else
if CAR(A)∈[23]B then [CAR(A)=CAR(C) and
Oset-intersect.Defn(CDR(A),Oset-delete.Alg(CAR(A),B),CDR(C))], else
Oset-intersect.Defn(CDR(A),B,C).
Quick: C=Oset-Intersect.Algs(A,B).
Once Early Quick Opaque: λ (A,B,C) if B is shorter than A,
then Oset-intersect.Defn(B,A,C).
Domain/range: <Osets Osets → Osets>
Algorithms:
Once Early Quick Opaque: λ (A,B) if B is shorter than A,
then Oset-intersect.Alg(B,A).
Non-recursive: λ (A,B) [for each x in A (in order), do the following:
if x¬∈B then DREMOVE(x,A). Finally, return the value of A.
Non-recursive: λ (A,B) [for each x in A (in order), do the following:
if x∈B then Oset-delete.Alg(x,B), else Oset-delete.Alg(x,A).
Finally, return the value of 'A' as the result.
Recursive: λ (A,B) if A=[] then [], else if CAR(A)∈B
then CONS(CAR(A),Oset-intersect.Alg(CDR(A),Oset-delete.Alg(CAR(A),B))),
else Oset-intersect.Alg(CDR(A),B).
Generalizations: Intersect
Worth: 100
What: Move along Oset A, eliminating elements not found in Oset A.

---

[22] The difference between this definition and the similar one for List-intersect is that here we can use the very fast DREMOVE algorithm stored in Oset-Delete.Alg, whereas for lists it was necessary to use a slow List-delete algorithm.

[23] To save space, we may henceforth write 'x∈B' to mean 'Member.Defn(x,B)'.

Appendix 2.1.25 Set-Intersect

Name(s): Set-Intersect, Set-Intersection, sometimes: Intersect.
Definitions:
Once Early Quick Opaque: λ (A,B,C) if B is shorter than A,
then Set-intersect.Defn(B,A,C).
Recursive: λ (A,B,C) if A={} then C={}, else
z←Some-memb.Alg(A);
If Member.Defn(z,B)
then [Member.Defn(z,C) and Set-intersect.Defn(Set-delete.Alg(z,A),
Set-delete.Alg(z,B),
Set-delete.Alg(z,C))]
else Set-intersect.Defn(Set-delete.Alg(z,A),B,C).
Nonrecursive Declarative: For all x, x∈C iff x∈A and x∈B.
Quick: C=Set-intersect.Algs(A,B).
Domain/range: <Sets Sets → Sets>
Algorithms:
Once Early Quick Opaque: λ (A,B) if B is shorter than A,
then Set-intersect.Alg(B,A).
Non-recursive: λ (A,B) [for each x in A, do the following:
if x¬∈B then DREMOVE(x,A). Finally, return the value of A.
Recursive: λ (A,B) if A={} then {}, else if CAR(A)∈B
then CONS(CAR(A),Set-intersect.Alg(CDR(A),Set-delete.Alg(CAR(A),B))),
else Set-intersect.Alg(CDR(A),B).
Generalizations: Intersect
Worth: 100
What: Eliminate any elements of Set A which are absent from Set B.

Appendix 2.1.26 Bag-Intersect

Name(s): Bag-Intersect, Bag-Intersection, sometimes: Intersect.
Definitions:
Once Early Quick Opaque: λ (A,B,C) if B is shorter than A,
then Bag-intersect.Defn(B,A,C).
Recursive: λ (A,B,C) if A=() then C=(), else
z←CAR(A); If Member.Defn(z,B) then [Member.Defn(z,C) and
Bag-intersect.Defn(CDR(A),Bag-delete.Alg(z,B),Bag-delete.Alg(z,C))]
else Bag-intersect.Defn(CDR(A),B,C).
Quick: C=Bag-Intersect.Algs(A,B).
Domain/range: <Bags Bags → Bags>
Algorithms:
Once Early Quick Opaque: λ (A,B) if B is shorter than A,
then Bag-intersect.Alg(B,A).
Non-recursive: λ (A,B) [for each x in A, do the following:
if x∈B then B←Bag.delete.Alg(x,B), else A←Bag-delete.Alg(x,A).
Finally, return the value of A.
Generalizations: Intersect
Worth: 100
What: the intersection of bags A and B should contain all common elements,
with each element occurring the *minimum* number of times it occurs in A or B.

Appendix 2.1.27 Union

Name(s): Union, Join, Unite, sometimes: Combine, Append, Sum.
Definitions:
Quasi-recursive cases: λ (A,B,C) [determine the type of structure A and
B are, say S, then use S-Union.Defn(A,B,C)].
Necessary, Nonrecursive: λ (A,B,C) For all x, x∈C iff x∈A or x∈B
Quick: C=Union.Algs(A,B).
Domain/range: <Structures Structures → Structures>
Algorithms:
Quasi-recursive cases: λ (A,B) [determine the type of structure A and B are,
say S,[24] then use S-Union.Algs(A,B)].
Quasi-recursive cases: λ (A,B) [determine the type of structure A and B are,
say S, then do S-insert.Alg(CAR(A),Union(CDR(A),B))].
Isa's: Operation
Specializations: Set-Union, Bag-Union, List-Union, Oset-Union.
Worth: 100

[24] S might be 'Sets', or S can be 'Lists', etc.

Appendix 2.1.28 List-Union

Name(s): List-Union, Append, Nconc, List-join, sometimes: Union.
Definitions:
Recursive slow: λ (A,B,C) if A=◇ then C=B, else
CAR(A)=CAR(C) and List-union.Defn(CDR(A),B,CDR(C)).
Quick: C=List-Union.Algs(A,B).
Domain/range: ⟨Lists Lists → Lists⟩
Algorithms:
Nonrecursive, Quick, Non-destructive, Opaque: λ (A,B) (APPEND A B).
Nonrecursive, Quick, Destructive, Opaque: λ (A,B) (NCONC A B).
Recursive: λ (A,B) if A=◇ then B, else
CONS(CAR(A),List-Union.Alg(CDR(A),B)).
Generalizations: Union
Worth: 100
What: Append list B to the end of list A.

Appendix 2.1.29 Oset-Union

Name(s): Oset-Union, Oset-join, sometimes: Union, Append.
Definitions:
Recursive slow: λ (A,B,C) if A=[] then C=B,
else CAR(A)=CAR(C) and
Oset-union.Defn[CDR(A),
Oset-delete.Alg(CAR(A),B),
Oset-delete.Alg(CAR(A),C)].
Quick: C=Oset-Union.Algs(A,B).
Domain/range: ⟨Osets Osets → Osets⟩
Algorithms:
Nonrecursive, Quick, Non-destructive, Opaque: λ (A,B) (APPEND A B).
Nonrecursive, Quick, Destructive, Opaque: λ (A,B) (NCONC A B).
Recursive: λ (A,B) if A=[] then B, else
CONS(CAR(A),Oset-Union.Alg(CDR(A),Oset-delete.Alg(CAR(A),B))).
Generalizations: Union
Worth: 100
What: Append onto Oset A any new members of Oset B.

Appendix 2.1.30 Set-Union

Name(s): Set-Union, Set-join, sometimes: Union, Append.
Definitions:
Nonrecursive Declarative: λ (A,B,C) ∀x, x∈C iff x∈A or x∈B.
Recursive slow: λ (A,B,C) if A={} then C=B, else CAR(A)∈C and
Set-union.Defn(CDR(A),
Set-delete.Alg(CAR(A),B),Set-delete.Alg(CAR(A),C)).
Quick: C=Set-Union.Algs(A,B).
Domain/range: <Sets Sets → Sets>
Algorithms:
Nonrecursive, Quick, Destructive, Opaque: λ (A,B) (UNION A B).
Nonrecursive, Quick, Non-destructive, Opaque: λ (A,B)
(Self-intersect (APPEND A B)).
Recursive: λ (A,B) if A={} then B, else
Set-insert.Alg(CAR(A),Set-Union.Alg(CDR(A),Set-delete.Alg(CAR(A),B))).
Recursive: λ (A,B) if A={} then B, else
MERGE(CAR(A),Set-Union.Alg(CDR(A),DREMOVE(CAR(A),B))).
Generalizations: Union
Worth: 100
What: Merge into Set A any new members of Set B.

Appendix 2.1.31 Bag-Union

Name(s): Bag-Union, Bag-join, sometimes: Union, Append.
Definitions:
Recursive slow: λ (A,B,C) if A=() then C=B, else CAR(A)∈C and
Bag-union.Defn(
Bag-delete.Alg(CAR(A),A),[25]
Bag-delete.Alg(CAR(A),B),
Bag-delete.Alg(CAR(A),C)).
Quick: C=Bag-Union.Algs(A,B).
Domain/range: <Bags Bags → Bags>
Algorithms:
Recursive: λ (A,B) if A=() then B, else
Bag-insert.Alg(CAR(A),Bag-Union.Alg(CDR(A),Bag-delete.Alg(CAR(A),B))).
Generalizations: Union
Worth: 100
What: Bag-union(A,B) contains any x belonging to either bag, with multiplicity of x equal to the *maximum* of the multiplicity of the element x in A and in B.

[25] Yes, this is really the same as CDR(A), and in the other concepts in this appendix the shorter form is the one used. Here, we decided to show the nice, symmetric form that AM actually contains.

Appendix 2.1.32 Difference

Name(s): Difference, Structure-difference, sometimes: Minus, Subtract, Complement.
Definitions:
Quasi-recursive cases: λ (A,B,C) [determine the type of structure A and B are, say S, then use S-Diff.Defn(A,B,C)].
Necessary, Nonrecursive: λ (A,B,C) For all x, x∈C iff x∈A and ¬x∈B
Quick: C=Difference.Algs(A,B).
Domain/range: <Structures Structures → Structures>
Algorithms:
Quasi-recursive cases: λ (A,B) [determine the type of structure A and B are, say S, then use S-Diff.Algs(A,B)].
Quasi-recursive cases: λ (A,B) [determine the type of structure A and B are, say S, then do S-delete.Alg(CAR(B),Difference(A,CDR(B)))].
Isa's: Operation
Specializations: Set-Diff, Bag-Diff, List-Diff, Oset-Diff.
Worth: 100

Appendix 2.1.33 List-Diff

Name(s): List-Difference, List-diff.
Definitions:
Recursive slow: λ (A,B,C) if A=◇ then C=◇, else
If CAR(A)∈B then List-Diff.Defn(CDR(A),List-delete.Alg(CAR(A),B),C),
else CAR(A)=CAR(C) and List-Diff.Defn(CDR(A),B,CDR(C)).
Quick: C=List-Diff.Algs(A,B).
Domain/range: <Lists Lists → Lists>
Algorithms:
Nonrecursive: λ (A,B) for x in A (in order), if x is in B,
then use List-delete to remove an x from A and B.
Recursive: λ (A,B) if A=◇ then ◇, else
If CAR(A)∈B then List-Diff.Alg(CDR(A),List-delete.Alg(CAR(A),B)),
else CONS(CAR(A),List-Diff.Alg(CDR(A),B)).
Generalizations: Difference
Worth: 100
What: Move x along A. If x is also in B, remove it from A and from B.

Appendix 2.1.34 Oset-Diff

Name(s): Oset-Difference, Oset-diff.
Definitions:
Recursive slow: λ (A,B,C) if A=[] then C=[], else
If CAR(A)∈B then Oset-Diff.Defn(CDR(A),Oset-delete.Alg(CAR(A),B),C),
else CAR(A)=CAR(C) and Oset-Diff.Defn(CDR(A),B,CDR(C)).
Quick: C=Oset-Diff.Algs(A,B).
Domain/range: <Osets Osets → Osets>
Algorithms:
Nonrecursive: λ (A,B) for x in A, if x is in B, then remove x from A and B.
Recursive: λ (A,B) if A=[] then [], else
If CAR(A)∈B then Oset-Diff.Alg(CDR(A),Oset-delete.Alg(CAR(A),B)),
else CONS(CAR(A),Oset-Diff.Alg(CDR(A),B)).
Recursive: λ (A,B) if A=[] then [], else
If CAR(A)∈B then Oset-Diff.Alg(CDR(A),B),
else CONS(CAR(A),Oset-Diff.Alg(CDR(A),B)).
Generalizations: Difference
Worth: 100
What: Moving along A, when an element also in B is encountered,
use Oset-delete to remove it from A and from B.

Appendix 2.1.35 Set-Diff

Name(s): Set-Difference, Set-diff.
Definitions:
Recursive slow: λ (A,B,C) if A={} then C={}, else
If CAR(A)∈B then Set-Diff.Defn(CDR(A),Set-delete.Alg(CAR(A),B),C),
else CAR(A)=CAR(C) and Set-Diff.Defn(CDR(A),B,CDR(C)).
Quick: C=Set-Diff.Algs(A,B).
Declarative Nonrecursive: λ (A,B,C) ∀x, x∈C iff x∈A and ¬x∈B.
Domain/range: <Sets Sets → Sets>
Algorithms:
Nonrecursive: λ (A,B) for x in A, if x is in B, then remove x from A and B.
Recursive: λ (A,B) if A={} then {}, else
If CAR(A)∈B then Set-Diff.Alg(CDR(A),Set-delete.Alg(CAR(A),B)),
else CONS(CAR(A),Set-Diff.Alg(CDR(A),B)).
Recursive: λ (A,B) if A={} then {}, else
If CAR(A)∈B then Set-Diff.Alg(CDR(A),B),
else CONS(CAR(A),Set-Diff.Alg(CDR(A),B)).
Generalizations: Difference
Worth: 100
What: Members of set A which are not in Set B.

Appendix 2.1.36 Bag-Diff

Name(s): Bag-Difference, Bag-diff.
Definitions:
Recursive slow: λ (A,B,C) if A=() then C=(), else
If CAR(A)∈B then Bag-Diff.Defn(CDR(A),Bag-delete.Alg(CAR(A),B),C),
else CAR(A)=CAR(C) and Bag-Diff.Defn(CDR(A),B,CDR(C)).
Quick: C=Bag-Diff.Algs(A,B).
Domain/range: <Bags Bags → Bags>
Algorithms:
Nonrecursive: λ (A,B) for x in A, if x is in B, then remove an x from A and B.
Recursive: λ (A,B) if A=() then (), else
If CAR(A)∈B then Bag-Diff.Alg(CDR(A),Bag-delete.Alg(CAR(A),B)),
else CONS(CAR(A),Bag-Diff.Alg(CDR(A),B)).
Recursive: λ (A,B) if B=() then A, else
If CAR(B)∈A then Bag-diff.Alg(Bag-delete.Alg(CAR(B),A),CDR(B)),
else Bag-diff.Alg(A,CDR(B)).
Generalizations: Difference
Worth: 100
What: Move x along Bag B, removing one copy of each x from Bag A.

Appendix 2.1.37 Coalesce

Name(s): Coalesce, Self-apply, Condense, Collapse, Argument coincidence.
Definitions:
Declarative slow: λ (F,G) The domain of G has been collapsed, compared to F's, by the removal of one domain component D, and an algorithm for G is just a call on F, with two arguements the same. The only constraint on this situation is that the domain component from which duplicate argument is drawn is itself a specialization of D.[26]
Necessary, quick: λ (F,G) The length of each Domain/range entry for F is one larger than the length of each entry on G.Dom/range.
Necessary, quick: λ (F,G) The range of both F and G are equal.
Sufficient, slow: λ (F,G) Are-equivalent(G,Coalesce.Algs(F)).
Sufficient, quick: λ (F,G) G=Coalesce.Algs(F).
Domain/range: <Active → Active>
<Operation → Operation>
<Predicate → Predicate>
Algorithms:
Distributed: use the heuristics attached to Coalesce to guide the filling in of various facets of the new Coalesced concept.
Generalizations: Operation
Isa's: Operation
Worth: 300
Fillin: 4 heuristics.
Check: 1 heuristic.
Suggest: 2 heuristics.

---

26 Some examples of this: (i) Coalesce.Defn(TIMES,Square), because TIMES.Domain/range contains <Number Number → Number> and Square.Domain/range contains <Number → Number>, and a definition of Square is 'Times(x,x)', and clearly Number is a specialization of Number (a vacuous specialization). So Square is a coalesced form of TIMES. (ii) Coalesce.Defn(Insert,Self-insert), where the latter concept is defined as Insert(S,S). The domain of Insert is Anything x Structure; the domain of the new operation is just Structure. This passes Coalesce.Defn because Structure is a specialization of Anything: if we can insert ANYTHING into a structure, then certainly it is permissable to insert a STRUCTURE into a structure. (iii) Coalesce.Defn(Equality,Constant-T) because Equality is reflexive (x=x always).

Appendix 2.1.38 Canonize

Name(s): Canonize, Canonicalize, Standardize, sometimes: normalize.
Definitions:
Slow: λ (P1,P2,F) P1 and P2 are predicates over AxA,
and F is an operation from A to A,
and (∀x,y∈A) P1(x,y) iff P2(F(x),F(y)).[27]
Sufficient, slow: Are-equivalent(F,Canonize.Algs(P1,P2)).
Sufficient, quick: F=Canonize.Algs(P1,P2).
Domain/range: <Predicate Predicate → Operation>
Algorithms:
Distributed: use the heuristics attached to Canonize to guide the filling in of various facets of the new canonization.
Generalizations: Operation
Isa's: Operation
Worth: 200
Fillin: 6 heuristics.
Suggest: 5 heuristics.

---

[27] Some examples of this: (i) P1=Same-length, P2=Equality, F=Length, A=Lists. (ii) P1=Reversed-at-top-level, P2=Reversed-at-all-levels, F=Reverse-each-element, A=Lists. (iii) P1=Reversed-at-top-level, P2=Reversed-at-all-levels, F=Hash-each-element, A=Lists. (iv) P1=Congruent-triangles, P2=Identically-equal, F=Translate-and-rotate-to-standard-position, A=Triangles. The typical use for the concept is: given P2, find P1 and F. Or: given P1 and P2, find F.

Appendix 2.1.39 Parallel-replace2

Name(s): Parallel-replace2, Map-replace2, Parallel-substitute.
Definitions:
Quick: G=Parallel-Replace2.Algs(S1,S2,F).
Domain/range: <Type-of-structure Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,S2,F,G) G is an operation whose domain is S1xS2 and whose range is Range(F). For any structures s1∈S1, s2∈S2, G(s1,s2) is compute by replacing each element x of s1 by the value of F(x,s2). Notice this means that F must be an operation with a domain/range entry of the form <D S2 → R>, where R is unconstrained, but D is either 'Anything' or -- if S1 is of the form 'Structure-of-E's' -- E.
Non-recursive quick: λ (S1,S2,F) if F(x,y) doesn't depend on y, then just do Parallel-replace.Algs(S1,F).
Specializations: Parallel-replace
Isa's: Operation
Worth: 100
What: create a new operation, which takes 2 structures S1 and S2, and replaces each member x of S1 by F(x,S2).

Appendix 2.1.40 Parallel-replace

Name(s): Parallel-replace, Map-replace, Parallel-substitute, MAPCAR.
Definitions:
Quick: λ (S1,F,G) G=Parallel-Replace.Algs(S1,F).
Domain/range: <Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,F,G) G is an operation whose domain is S1 and whose range is Range(F). For any structure s1∈S1, G(s1) is computed by replacing each element x of s1 by the value of F(x). Notice this means that F must be an operation with a domain/range entry of the form <D → R>, where R is unconstrained, but D is either 'Anything' or -- if S1 is of the form 'Structure-of-E's' -- E.
Generalizations: Parallel-replace2
Worth: 100
Sugg: 2 heuristics.[28]
What: create a new operation, which takes a structures S1, and replaces each member x of S1 by F(x).

[28] These actually deal with substitution operations, the RESULTS of applying Parallel-replace and Parallel-replace2.

Appendix 2.1.41 Repeat2

Name(s): Repeat2, Map-repeat2, Iterate2, Map2, MAP2CONC.
Definitions:
Quick: λ (S1,S2,F,G) G=Repeat2.Algs(S1,S2,F).
Domain/range: <Type-of-structure Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,S2,F G=Repeat2(S1,S2,F) is an operation whose domain is S1xS2 and whose range is Range(F).
For any structures s1∈S1, s2∈S2,
G(s1,s2) is computed by the following algorithm:
y←CAR(s1); s1←CDR(s1);
while s1 do: y←F(y,s2,CAR(s1)); s1←CDR(s1);
Finally, return y.
Notice this means that F must be an operation whose domain/range has the form <s1 S2 s1 → s1>.
Non-recursive quick: λ (S1,S2,F) if F(x,y,z) doesn't depend on z, then just do Repeat.Algs(S1,F).
Specializations: Repeat
Isa's: Operation
Worth: 100
What: create a new operation, which takes 2 structures S1 and S2, and repeats F(x,y,s2) along the members x,y of S1.

Appendix 2.1.42 Repeat

Name(s): Repeat, Map, Iterate, Sequence.
Definitions:
Quick: λ (S1,F,G) G=Repeat.Algs(S1,F).
Domain/range: <Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,F) Repeat(S1,F)=G is an operation whose domain is S1 and whose range is Range(F). For any structure s1∈S1,
G(s1) is computed by the following algorithm:
y←CAR(s1); s1←CDR(s1);
while s1 do: y←F(y,CAR(s1)); s1←CDR(s1);
Finally, return y.
Notice this means that F must be an operation whose domain/range has the form <s1 s1 → s1>.
Generalizations: Repeat2
Worth: 100
What: create a new operation which repeats F all the way along an S1.

Appendix 2.1.43 Parallel-join2

Name(s): Parallel-join2, Map-join2, Parallel-union2, MAP2CONC.
Definitions:
Quick: λ (S1,S2,F,G) G=Parallel-join2.Algs(S1,S2,F).
Domain/range: <Type-of-structure Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,S2,F,G) G is an operation whose domain is S1xS2 and whose range is Range(F). For any structures s1∈S1, s2∈S2, G(s1,s2) is compute by appending together the values of F(x,s2), for each element x in s1. So F has to be an operation with a domain/range entry of the form <D S2 → R>, where R is a type of structure, but D is either 'Anything' or -- if S1 is of the form 'Structure-of-E's' -- E.
Non-recursive quick: λ (S1,S2,F) if F(x,y) doesn't depend on y, then just do Parallel-join.Algs(S1,F).
Specializations: Parallel-join
Isa's: Operation
Worth: 100
What: create a new operation, which takes 2 structures S1 and S2, and joins together F(x,s2) for each member x of S1.

Appendix 2.1.44 Parallel-join

Name(s): Parallel-join, Map-join, Parallel-union, MAPAPPEND, MAPCONC.
Definitions:
Quick: λ (S1,F,G) G=Parallel-join.Algs(S1,F).
Domain/range: <Type-of-structure Operation → Operation>
Algorithms:
Nonrecursive: λ (S1,F,G) G is an operation whose domain is S1 and whose range is Range(F). For any structure s1∈S1, G(s1) is computed by appending together the values of F(x), for each x∈s1. Notice this means that F must be an operation with a domain/range entry of the form <D → R>, where R is a type of structure, and D is either 'Anything' or -- if S1 is of the form 'Structure-of-E's' -- E.
Generalizations: Parallel-join2
Worth: 100
What: create a new operation, which takes a structure S1, and joins together F of each member of S1.

Appendix 2.1.45 Reverse-ord-pair

Name(s): Reverse-ord-pair, Reverse ordered pair, Switch CAR and CADR.
Definitions:
Nonrecursive quick: λ (P,Q) First.Alg(P)=Final.Alg(Q),
and Final.Alg(P)=First.Alg(Q).
Quick: λ (P,Q) Q=Reverse-ord-pair.Algs(P).
Domain/range: <Ordered-pair → Ordered-pair>
Algorithms:
Nonrecursive: λ (P) Q←P; First.Alg(Q,Final.Alg(P))[29]; Final.Alg(Q,First.Alg(P)); Q.
Nonrecursive quick opaque, nondestructive: λ (P) LIST(CADR(P),CAR(P).
Nonrecursive quick opaque, destructive: λ (P) z←Last-ele(P);
FRPLACA(CDR(P),CAR(P)); FRPLACA(P,z); P.
Nonrecursive quick opaque, nondestructive: λ (P) REVERSE(P).
Nonrecursive quick opaque, destructive: λ (P) DREVERSE(P).
Isa's: Operation
Worth: 100
What: turn the ordered pair <x,y> into the ordered pair <y,x>.

Appendix 2.1.46 Last-element

Name(s): Last-element, Final member.
Definitions:
Recursive: λ (S,x) z←First-element.Alg(S), and S←Delete.Alg(z,S),
and if Empty-struc.Defn(S) then x=z, else Last-element.Defn(S,x).
Quick: λ (S,x) x=Last-element.Algs(S).
Domain/range: <Ordered-structure → Anything>
Algorithms:
Recursive: λ (S) z←First-element.Alg(S), and S←Delete.Alg(z,S),
and if Empty-struc.Defn(S) then z, else Last-element.Alg(S).
Nonrecursive quick opaque: λ (S) CAR(LAST(S)).
Isa's: Operation
Worth: 100
What: find the final member of the ordered structure S.[30]

---

[29] The expression First.Alg(A,x) will result in a RPLACA: the first element of A will be removed, and in its place x will appear. Thus First.Alg(<a b c d>, z) will return as its value the new list <z b c d>.

[30] Actually, this concept is much more sophisticated. If Last-element.Algs is called with TWO arguments, S and v, then the intention is taken to be to REPLACE the last element of S by the element v. Thus that last element is deleted, and v is added at the end of S. This is done by: FRPLACA(LAST(S),v). To review: Last-element.Alg(A,x) resets the final member of A to x, while Last-element.Defn(A,x) merely tests whether the last member of A is x.

Appendix 2.1.47 First-element

Name(s): First-element, Initial member, Head, Front element, CAR.
Definitions:
Recursive: λ (S,x) z←Last-element.Alg(S), and S←Delete.Alg(z,S),
and if Empty-struc.Defn(S) then x=z, else First-element.Defn(S,x).
Quick: λ (S,x) x=First-element.Algs(S).
Domain/range: <Ordered-structure → Anything>
Algorithms:
Recursive: λ (S) z←Last-element.Alg(S), and S←Delete.Alg(z,S),
and if Empty-struc.Defn(S) then z, else First-element.Alg(S).
Nonrecursive, very quick, opaque: λ (S) CAR(S).
Isa's: Operation
Worth: 100
What: find the initial member of the ordered structure S.[31]

Appendix 2.1.48 All-but-the-first-element

Name(s): Rear, All but the first element, All-but-first, CDR, Tail, sometimes: back.
Definitions:
Nonrecursive: λ (S,R) List-delete.Defn(CAR(S),S,R).
Nonrecursive: λ (S,R) List-insert.Defn(CAR(S),R,S).
Nonrecursive: λ (S,R) CDR(S)=R.
Quick: λ (S,R) R=Rear.Algs(S).
Domain/range: <Ordered-structure → Ordered-structure>
Algorithms:
Nonrecursive, very quick, opaque: λ (S) CDR(S)
Nonrecursive: λ (S) z←First-ele.Alg(S); List-delete.Algs(z,S).
Isa's: Operation
Worth: 100
What: remove the initial member of the ordered structure S.

[31] Actually, this operation's algorithm, if fed two arguments S and v, will replace the first element of S by v, using FRPLACA(S,v). So this single concept contains both CAR and FRPLACA knowledge. This is not shown explicitly in the entries for First-element.Algs.

Appendix 2.1.49 All-but-the-last-element

Name(s): All-but-the-last-element, All-but-last, sometimes: front.
Definitions:
Quick: λ (S,R) R=All-but-last.Algs(S).
Domain/range: <Ordered-structure → Ordered-structure>
Algorithms:
Nonrecursive, very quick, opaque: λ (S) FRPLACD(LAST(S),NIL).
Isa's: Operation
Worth: 100
What: remove the final element from the ordered structure S.

Appendix 2.1.50 Member

Name(s): Some-element, Random member, Any element of, Member, In, Some-member.
Definitions:
Recursive: λ (x,S) Nonempty-struc.Defn(S) and
if First-ele.Defn(S,x) then True,
else Member.Defn(x,All-but-first-ele.Alg(S)).
Nonrecursive quick opaque: λ (x,S) MEMBER(x,S)).
Sufficient, very quick, opaque: λ (x,S) FMEMB(x,S)).
Quick: λ (S,x) x=Member.Algs(S).
Domain/range: <Structure → Anything>
Algorithms:
Nonrecursive opaque: λ (S) CAR(RAND-PERMUTE(S)).
Nonrecursive quick opaque: λ (S) CAR(S)).
Recursive slow: if S is empty then fail, otherwise if S=(x) then x,
else if RAND(0,1)=1 then First-ele.Alg(S),
else Member.Alg(All-but-last.Alg(S)).
Isa's: Operation
Worth: 100
What: find a random member of the structure S.

Appendix 2.1.51 Projection1

Name(s): Projection1, First-argument, Proj1.
Definitions:
    Nonrecursive quick: λ (x,y,...,z) z=x
    Quick: λ (x,y,...q,z) z=Some-element.Algs(x,y,...q).
Domain/range: <←D Anything...Anything → 'D'> [32]
Algorithms:
    Nonrecursive quick: λ (x,y,...,q) x.
Isa's: Operation
Specializations: Identity.
Worth: 100
What: accept a bunch of arguments and return the first one.

Appendix 2.1.52 Projection2

Name(s): Projection2, Second-argument, Proj2.
Definitions:
    Nonrecursive quick: λ (x,y,...,z) z=y
    Quick: λ (x,y,...q,z) z=Some-element.Algs(x,y,...q).
Domain/range: <Anything ←D Anything...Anything → 'D'>
Algorithms:
    Nonrecursive quick: λ (x,y,...,q) y.
Isa's: Operation
Specializations: Identity.
Worth: 200
What: accept a bunch of arguments and return the second one.

[32] This means that 'D' can be anything, so long as it's the same in both places in the domain/range template. Thus this includes <Sets Anything Anything → Sets>.

Appendix 2.1.53 Identity

Name(s): Identity, identity-operation, no-op, Self, no change.
Definitions:
Nonrecursive: λ (x,y) Equality.Defn(x,y)
Nonrecursive transform: λ (x,y) Proj1.Defn(x,x,y)
Nonrecursive transform: λ (x,y) Proj2.Defn(x,x,y)
Sufficient, very quick, opaque: λ (x,y) EQ(x,y)).
Quick: λ (x,y) y=Identity.Algs(x).
Domain/range: <Anything → Anything>
<Object → Object>
<Structures → Structures>
<Active → Active>
Algorithms:
Nonrecursive quick: λ (x) x.
Nonrecursive transform: λ (x) Projection1.Algs(x,x).
Nonrecursive transform: λ (x) Projection2.Algs(x,x).
Conjec: 'Identity, restricted to Objects, is the same as Obj-Equality.'
Generalizations: Projection1, Projection2.
Worth: 100
What: the identity operation, closely related to Equality.

Appendix 2.1.54 Restrict

Name(s): Restrict, Constrain the domain/range of an active.
Definitions:
Nonrecursive: λ (F,G) The domain/range of G are more restrictive[33]
than that of F, and G.Defn is just a call on F.Defn.
Sufficient, Quick: λ (F,G) G=Restrict.Algs(F).
Domain/range: <Active → Active>
<Operation → Operation>
<Predicate → Predicate>
Algorithms:
Distributed: use the heuristics attached to Restrict to guide the filling
in of various facets of the new Restricted concept.
Plus: an explicit little program for making the substitution
in the Domain/range facet, which is the essence of this concept.
Isa's: Operation
Worth: 200
Fillin: 3 heuristics.

[33] That is, one (or more) component of the G.Domain/range entry is a proper specialization of the corresponding F.Dom/ran entry, and all the other components match up equally.

Appendix 2.1.55 Invert-an-operation

Name(s): Invert, Find the inverse of an operation.
Definitions:
Declarative slow: λ (F,G) The domain of G is the range of F, plus all the domain components of F except one, D; the range of G is then D. The value of G.Defn(x1,...,r,...,d) must be the same as the value the value of F.Defn(x1,...,d,...,r), for any x1,...,d, and r.
Necessary, quick: λ (F,G) The length of each Domain/range entry for F is the same as the length of each entry on G.Dom/range.
Necessary, quick: λ (F,G) Taken as SETS, a domain/range entry from F and one from G are actually Equal.
Sufficient quick: λ (F,G) G has the Name 'F-inverse'.
Quick: λ (F,G) G=Invert.Algs(F).
Domain/range: <Operation → Operation>
<Operation → Inverted-op>
Algorithms:
Distributed: use the heuristics attached to Invert to guide the filling in of various facets of the new Inverted concept.
Isa's: Operation
Worth: 300
Fillin: 1 heuristic.
Suggest: 1 heuristic.

Appendix 2.1.56 Inverted-op

Name(s): Inverted operation,Inverse, sometimes: converse.
Definitions:
Declarative slow: λ (F) For some known operation G, Invert.Defn(G,F).
Necessary, quick: λ (F) The range of F is one single known concept.
Sufficient quick: λ (F) F has the Name 'G-inverse' for some G.
Generalizations: Operation
In-domain-of: Invert.[34]
In-range-of: Invert
Worth: 200

[34] This just means that such operations are themselves easily invertable.

Appendix 2.1.57 Relation

Name(s): Relation, relationship.
Definitions: none.
Generalizations: Active
Specializations: Logical-combination
Worth: 100
View: To view an operation F as a relation, consider it as the set of all ordered pairs, a subset of Dom(F)xRan(F), containing <x,y> iff F.Defn(x,y).
NOTE: This concept exists in only rudimentary form in AM at the moment.

Appendix 2.1.58 Logical-combination

Name(s): Logical Combination, Boolean relation.
Definitions: none.
Generalizations: Relation
Examples: Conjoin, Disjoin, Imply, Negate[35]
Worth: 200
Check: 1 heuristic
Interest: 3 heuristics
Sugg: 2 heuristics
NOTE: This concept exists in only rudimentary form in AM at the moment.

[35] These aren't coded seperately as concepts in AM, yet.

Appendix 2.1.59 Object

Name(s): Object, static concept, Passive
Definitions: none.[36]
Specializations: Structure, Atom-obj, Conjecture[37]
Generalizations: Any-concept
Examples: none.
Isa's: Any-concept
In-domain-of: Object-equality
Worth: 100
(No heuristics)[38]

Appendix 2.1.60 Conjecture

Name(s): Conjecture, Conjec, Hypothesis, Guess, Observation, Thesis, Belief.
Definitions:
Nonrecursive, Quick: λ (x) Match x with <CONJEC: ...>
Generalizations: Object
In-domain-of: Prove[39], Disprove, Test
In-range-of: none[40].
Worth: 200.

---

[36] Recall that all this means is that computationally, any entity x is considered to be an Object iff it is an example of some Specialization of this concept. Thus the list (3 A NIL) is an object, because it is a List, and List is one Specialization of Structure, and Structure is a Specialization of Object.

[37] This should be 'Statement', and that concept should have Conjecture as a specialization, along with Theorem, Falsehood, etc. This was never fully implemented in the AM code, however.

[38] The paucity of heuristics here attests to the little that structures, statements, and atoms have in common. They are merely non-actives. There is much that does not apply to any of them (see the Active and Operation concepts), but very few rules of thumb applicable to all 3 of them.

[39] At the moment, none of these three concepts is in AM.

[40] Conjectures are produced by heuristic rules, not mechanically by running some Active concept.

Appendix 2.1.61 Atom-obj

Name(s): Atom, Atomic object, sometimes: element.
Definitions:
Nonrecursive, Quick, Opaque: λ (x) ATOM(x)
Specializations: Truth-value, Variable[41], Identifier.
Generalizations: Object
In-domain-of: UNPACK; NthCHAR
In-range-of: MKATTOM, PACK;
View: To view any structure S as an atom, apply PACK to it.
Worth: 100. [42]

Appendix 2.1.62 Truth-value

Name(s): Truth value, Logical constant, T/F, {T,F}.
Definitions: none.[43]
Examples: True (T,Y,Yes), False (NIL,F,N,No).
Generalizations: Atom-obj
In-domain-of: Negation
In-range-of: all predicates; the Defn facet of each concept.
View: to view anything x as a truth value, do: λ (x) NOT(Equality.Defn(x,NIL)).[44]
Worth: 100.

---

[41] Many of the nouns in this box are not implemented as concepts in AM; e.g., Variable, Identifier, UNPACK, MKATOM,..

[42] The absence of any heuristics here just emphasizes the fact that literal constants, identifiers, variables, T, etc. have very little in common that ALL objects don't share.

[43] Since no definition is provided, AM never generalized or specialized this concept, looked for new examples of it, etc.

[44] Thus, as in Lisp itself, an entity is associated with False iff it is null, and with True iff it is anything else in the world.

Appendix 2.1.63 Structure

Name(s): Structure, Data-structure, sometimes:[45] List-structure.
Definitions:
Necessary, Non-Recursive, Quick, Opaque: λ (x) LISTP(x)
Specializations: Ord-struc, Unord-struc, Empty-struc, Non-empty-struc,
Multiple-elements-struc, No-multiple-elements-struc, Struc-of-strucs.
Generalizations: Object
In-domain-of: Insert, Delete, Member, Empty, Nonempty, Difference, Union, Intersect,
Parallel-replace(2), Parallel-join(2), Repeat(2).
In-range-of: Insert, Delete, Difference, Union, Intersect.
View: To view any entity x as a structure, insert x into an empty structure.
Worth: 200
Fillin: 2 heuristics.
Interest: 2 heuristics

Appendix 2.1.64 Structure-of-Structures

Name(s): Structure-of-structures, struc-of-strucs.
Definitions:
Recursive: λ (S) Empty-struc.Defn(S) or
[Structure.Defn(S) and z←Member.Alg(S) and Structure.Defn(z) and
Structure-of-Structures.Defn(Delete.Algs(z,S))].
Declarative PC: λ (S) Structure.Defn(S) and (∀x∈S) Structure.Defn(x).
Specializations: none.[46]
Generalizations: Structure
Worth: 300

[45] That is, the user might erroneously type 'List-structure' when he really means any kind of structure.

[46] AM specialized this by replacing each of the two calls on 'Structure.Defn' inside Struc-of-struc.Defn by a call on the definition of a single type of structure, thereby creating, e.g., Bag-of-Sets, List-of-Odds, Bag-of-Primes, etc. These specialized concepts were then kept around so, e.g., the sample traces in Chapter 6 and in Appendix 5 sometimes refer to them. Also, this concept and its specializations can be discovered independently by AM, using heuristic rule number 232 (see Appendix 3) to form a new interesting type of structure.

Appendix 2.1.65 Ord-Structure

Name(s): Ord-struc, Ordered Structure, sometimes: List-structure.
Definitions: none
Specializations: Osets, Lists
Generalizations: Structure
In-domain-of: First-ele, Last-ele, All-but-first-ele, All-but-last-ele.
In-range-of: All-but-first-ele, All-but-last-ele.
View: To view any unord-struc as an ord-struc, do nothing to it, or permute it.
Worth: 200
Fillin: 2 heuristics.
Check: 2 heuristics.
Interest: 1 heuristic

Appendix 2.1.66 Unord-Structure

Name(s): Unord-struc, Unordered Structure, sometimes: Collection
Definitions: none
Specializations: Sets, Bags.
Generalizations: Structure
View: To view any ordered-struc as an unord-struc, SORT it.
Worth: 200
Check: 1 heuristic.

Appendix 2.1.67 Multiple-elements-structure

Name(s): Multiple-elements-structure, Mult-ele-struc, sometimes: Lists.
Definitions: none
Specializations: Lists, Bags
Generalizations: Structure
In-domain-of: none.[47]
View: To view any nonmult-struc as a mult-struc, do nothing to it,
or: copy some elements inside it a random number of times.
Worth: 200
Fillin: 1 heuristic.

[47] There are many special functions which can only make sense for multiple-ele structures, e.g., Remove-1-occurrence(x,S), versus Remove-all-occurrences(x,S). Such operations have not yet been coded and added to AM.

Appendix 2.1.68 No-multiple-elements-structure

Name(s): No-Multiple-elements-structure, Nonmult-struc, sometimes: Sets.
Definitions: none
Specializations: Sets, Ordered-sets
Generalizations: Structure
View: To view any mult-struc as a nonmult-struc, eliminate multiple elements.
Worth: 200

Appendix 2.1.69 Empty-structure

Name(s): Empty-structure, Empty struc, sometimes: phi, NIL.
Definitions:
Nonrecursive quick opaque: λ (x) NULL(x)
Nonrecursive: λ (x) Structure.Defn(x) and NOT(Member.Alg(x)).
Generalizations: Structure
View: To view any structure as an empty-structure, repeatedly apply Delete.
Worth: 100

Appendix 2.1.70 Nonempty-structure

Name(s): Nonempty-structure, Nonempty struc, sometimes: structure
Definitions:
Nonrecursive quick opaque: λ (x) LISTP(x)
Nonrecursive: λ (x) NOT(NOT(Member.Alg(x))).
Generalizations: Structure
In-range-of: Insert
View: To view any structure as an Nonempty-structure, insert it into itself.
Worth: 100

Appendix 2.1.71 Sets

Name(s): Set, Class, Collection
Definitions:[48]
Recursive: λ (S) (S={} or Set.Definition (Set-Delete.Alg(Member.Alg(S),S)))
Recursive quick: λ (S) (S={} or Set.Definition (CDR(S)))
Quick: λ (S) (Match S with {...})
Intuitions: none at present.[49]
Specializations: Set-of-structures[50]
Generalizations: Unordered-Structure, No-multiple-elements-Structure
In-domain-of: Set-union, Set-intersect, Set-difference, Set-insert, Set-delete
In-range-of: Set-union, Set-intersect, Set-difference, Set-insert, Set-delete
View: To view any structure as a Set, do: λ (x) Enclose-in-braces(x)
To view any predicate as a Set, do: λ (P) S←{}.
Forall x in Examples(Domain(P)): If P(x) then Set-insert.Alg(x,S).
Worth: 400
Sugg: 1 heuristic.
Interest: 1 heuristic.

Appendix 2.1.72 Bags

Name(s): Bag, sometimes: Multiset, sometimes: Collection.
Definitions:
Recursive: λ (S) (S=( ) or Bag.Definition(Bag-delete.Alg(Member.Alg(S),S)))
Recursive quick: λ (S) (S=( ) or Bag.Definition (CDR(S)))
Quick: λ (S) (Match S with (...))
Specializations: Bag-of-structures[50]
Generalizations: Unordered-Structure, Multiple-elements-Structure
Worth: 400
In-domain-of: Bag-union, Bag-intersect, Bag-difference, Bag-insert, Bag-delete
In-range-of: Bag-union, Bag-intersect, Bag-difference, Bag-insert, Bag-delete
View: To view any structure as a Bag, do: λ (x) Enclose-in-parens(x)

---

[48] A surprising idea, which fell out naturally while designing the entries for the definition facets of Sets, Bags, etc., is that the differences between these structures is not in their definition so much as in the particular operators which work on them. Thus all 4 kinds of structures appear to have syntactically similar concepts, even including their definitions. The reader must examine, e.g., the definition of Bag-insert and Set-insert to discover the real differences between the Set and Bag structures which AM knows about.

[49] Several nice intuitions were originally provided, then scrapped when ALL intuitions were excised from AM.

[50] This concept was synthesized by AM, but was then left 'permanently' in place.

Appendix 2.1.73 Lists

Name(s): List, List-structure, Vector, Tuple, n-tuple, Sequence, Ordered-bag
Definitions:
Recursive: λ (S) (S=< > or List.Definition(List-Delete.Alg(Member.Alg(S),S)))
Recursive quick: λ (S) (S=< > or List.Definition (CDR(S)))
Quick: λ (S) (Match S with <...>)
Generalizations: Ordered-Structure, Multiple-elements-Structure
Specializations: Ordered-pairs
Worth: 400
In-domain-of: List-union, List-intersect, List-difference, List-insert, List-delete.[51]
In-range-of: List-union, List-intersect, List-difference, List-insert, List-delete
View: To view any structure as a List, do: λ (x) Enclose-in-angle-brackets(x)

Appendix 2.1.74 Ordered-pairs

Name(s): Ord-pair, Opair, Ordered pair, 2-tuple, sometimes: i/o pair, pair.
Definitions:
Declarative: λ (S) There exist x and y such that S=<x,y>.
Nonrecursive opaque: List.Definition (S) and CDR(S) and Null(CDDR(S)).
Nonrecursive slow: λ (S) List.Definition(S), and S≠◇, and z←Member.Alg(S),
and S←List-delete.Alg(z,S), and S≠◇, and y←Member.Alg(S),
and List-delete.Defn(y,S,◇).
Nonrecursive quick: λ (S) (Match S with <←x,←y>)
Generalizations: Lists
Worth: 200
In-domain-of: Reverse-ord-pair
In-range-of: Reverse-ord-pair
View: To view any entity x as an ordered pair, consider the pair <x,x>.
View: To view an example of an active concept F as an ord-pair, construct the
pair whose first element is a list of the arguments to F
[or: THE argument to F, if there is only one], and whose
second element is the value of F on those arg(s).
View: To view an (ordered) structure S as an Opair, consider the pair whose
first element is some member of (the first member of) S, and
whose second element is all the remaining members of S.
View: Transform the ordered structure (a b...c) into the Opair (a b) or (a c).

[51] There are many special functions which can only make sense for lists, e.g., this one: 'Between(x,S)' which returns a list of all elements lying after the first occurrence of x in S, but before the second occurrence. Such operations have not yet been coded and added to AM.

Appendix 2.1.75 Osets

Name(s): Oset, Oset-structure, Ordered-set, sometimes: Set.
Definitions:
Recursive: λ (S) (S=[ ] or Oset.Definition(Oset-Delete.Alg(Member.Alg(S),S)))
Recursive quick: λ (S) (S=[ ] or Oset.Definition (CDR(S)))
Quick: λ (S) (Match S with [...])
Generalizations: Ordered-Structure, No-multiple-elements-Structure
Worth: 400
In-domain-of: Oset-union, Oset-intersect, Oset-difference, Oset-insert, Oset-delete
In-range-of: Oset-union, Oset-intersect, Oset-difference, Oset-insert, Oset-delete
View: To view any structure as a Oset, do: λ (x) Enclose-in-square-brackets(x)

Appendix 2.2. Concepts never fully implemented

The following concepts were designed "on paper" before AM was coded, but were never put into AM – at least not fully. Future work on AM may include their coding, insertion into AM, and debugging. An asterisk (*) means that a crude, rudimentary version of the concept was coded and placed in AM, but had little impact on its behavior.

Statement: would include conjectures, theorems, axioms, hypotheses, conclusions, relationships.

Prove, Disprove, Proof, Counterexample, Theorem, Techniques for proving existence, Techniques for establishing universal conjectures,...: altogether about two dozen concepts were designed.

Mathematical Induction, including double induction.

Mathematical theory, system, basis, foundation, axiom, isomorphism,...

Cause and effect: their relation to theory formation.

Variable, Assignment, Binding, Quantification, Scope,...: a dozen concepts along these lines.

Constant, Identifier, PNAME/P2NAME,...: AM never really needed any non-opaque information about these, although future expansion of the system should probably include the coding and insertion of these concepts.

Inverse-coalesce: Given an active concept F(x), replace some occurrences of x in F.Defn by "y", thereby making a new operation which is a function of x and y.

Negate, Conjoin, Disjoin, Imply,...: These logical operators and relationships had too little semantic information to make it necessary to encode each one into a concept.

(*) **Constructive, Destructive:** these two predicates would judge any operation.

(∅) Non-concept: All entities which are not concepts. There was nothing to say about them, as a whole.

Appendix 2.3. Concepts and Heuristics as coded in LISP

The reader may wish to inspect the actual LISP encoding of concepts and their facets – including heuristic rules. For that reason, a few pages are excerpted from the AM program and shown below.

The facets of a concept are stored as properties on its property list. Each facet has a rigid format that it must adhere to; that format varies from facet to facet.

Two concepts have been selected: *Compose*, which is larger than the typical concept, and *Oset-structure*, which is a smaller and simpler concept.

Appendix 2.3.1. The 'Compose' Concept

Here is the property list of the atom "COMPOSE", when AM starts up. The reader should look for (and find!) parallels between the complete entries below and the abbreviated summaries on page 178. For that reason, after each entry, the corresponding summary line is repeated (in a box).

ENGN[52] (COMPOSE Compose Composition (Afterwards))

Appearance on page 178:

> **Name(s): Compose, Composition, sometimes: afterwards;**

DEFN (TYPE NEC&SUFF PC DECLARATIVE SLOW (FOREACH X IN (DOMAIN BA2)
RETURN (APPLYB[53] BA1 ALGS (APPLYB BA2 ALGS X]
DEFN-SUFF [[TYPE SUFFICIENT NONRECURSIVE QUICK
(AND (ISA BA1 'ACTIVE)
(ISA BA2 'ACTIVE)
(ISA BA3 'ACTIVE)
(ARE-EQUIV BA3 (ALREADY-COMPOSED[54] BA1 BA2]
[TYPE SUFFICIENT QUASIRECURSIVE SLOW (ARE-EQUIV BA3
(APPLYB 'COMPOSE 'ALGS BA1 BA2][55]
[TYPE SUFFICIENT QUASIRECURSIVE QUICK (EQUAL BA3
(APPLYB 'COMPOSE 'ALGS BA1 BA2]]

Appearance on page 178:

---

[52] This is short for "English name", and is the facet called "Name(s)" everywhere else in this thesis.

[53] The function "APPLYB" indicates that a concept's facet is to be accessed and then executed. (APPLYB C F x y_) means: access an entry on facet F of concept C, and then run it on the arguments x,y,_

[54] This LISP function checks to see whether the two operations have been composed before.

[55] The arguments to ComposeDefn (and to ComposeAlgs as well) are called BA1, BA2,_ Thus we would write each definition of Compose as "λ (BA1 BA2 BA3) _"

Definitions:
Declarative slow: λ (A,B,C) ∀x, C(x)=A(B(x)).
Sufficient Nonrecursive Quick: λ (A,B,C) C has the Name 'A∘B'.
Sufficient, Slow: Are-equivalent(C,Compose.Algs(A,B)).
Sufficient, Quick: C=Compose.Algs(A,B).

D-R ((OPERATION ACTIVE OPERATION)
(RELATION RELATION RELATION)
(PREDICATE ACTIVE PREDICATE)
(ACTIVE ACTIVE ACTIVE))
D-R-FILLIN1 (PROGN (ARGS-ASA COMPOSE F1 F2) (CADAR (CON-MERGE-ARGS[56] F1 F2)))
EXS-D-R-FILLIN1 [PROGN (ARGS-ASA COMPOSE F1 F2)
[SETQ RAN1 (LAST (ANY1OF (GETB F1 'D-R] (* RAN1 is the range of F1)
[SETQ DOM1 (ALL-BUT-LAST (ANY1OF (GETB F1 'D-R]
[SETQ RAN2 (LAST (ANY1OF (GETB F2 'D-R] (* RAN2 is the range of F2)
[SETQ DOM2 (ALL-BUT-LAST (ANY1OF (GETB F2 'D-R]
[SETQ X (MAXIMAL RAN2 DOM1 'FRAC-OVERLAP]
(NCONC1 (LSUBST DOM2 for X in DOM1) RAN1]

Appearance on page 178:

Domain/range: <Active Active → Active>
<Operation Active → Operation>
<Predicate Active → Predicate>
<Relation Relation → Relation>
Fillin: 2 *(out of a total of 9)* heuristics.
In Appendix 3, these are heuristics numbers 175 and 176.

ALGS ((TYPE QUASIRECURSIVE INDIRECT CASES [PROGN
(COND
((NULL BA1)
(APPLYB 'COMPOSE
'ALGS
(RAND-MEMB (EXS[57] ACTIVE))
BA2 BA3 BA4))[58]

---

[56] This is a LISP function, opaque to AM, which analyzes the Domain/range facets of the two operations F1 and F2, and sees how (if at all) the range of F1 can be made to overlap the domain of F2. Note that F2 is applied AFTER F1. The LISP code for this function is presented on page 221.

[57] The function "EXS" ripples outward from its argument, collecting examples as it goes.

[58] Note what this clause says: if Compose.Algs is ever called with its first argument missing, randomly select an Active to use as that constituent of the composition.

```
&[59]
((ALREADY-COMPOSED BA1 BA2)  (* Note: this sets GTEMP12)  GTEMP12)
((AND BA1 BA2 (IS-CON[60] BA1)
      (IS-CON BA2)
      (ISA BA1 'ACTIVE)
      (ISA BA2 'ACTIVE)
      (SETQ GTEMP11 (CON-MERGE-ARGS BA1 BA2 GTEMP12)))
 (* GTEMP12 is now the name of the new composition)
 (CREATEB[61] GTEMP12)
 [SETQ GUP1 (COND ((ISAG CS-B 'COMPOSE) CS-B)  (T 'COMPOSE]
 (* GUP1 is now the KIND of concept which GTEMP12 is to be an example of.
    This will usually be "COMPOSE" or some variant of it. )
 [INCRB[62] GTEMP12 'DEFN
  (LIST 'TYPE 'APPLICATION 'OF GUP1
   (APPEND (LIST 'APPLYB (Q[63] COMPOSE) (Q ALGS) (KWOTE BA1) (KWOTE BA2))
           (FIRSTN (LENGTH (CAAR GTEMP11))  BA-LIST]
(* Another way to fill in an entry for GTEMP12.Defn)
(COND
 ([SETQ GTEMP308 (CAR (SOME (EXS COMPOSE)
                            (FUNCTION (LAMBDA (C)
                               (MEMBER (LASTELE (GETB GTEMP12 'DEFN))
                                       (GETB (LASTELE C)  'DEFN]
  (FORGET-CONCEPT GTEMP12)
  (CPRIN1S 8 GTEMP12 turned out to be equivalent to GTEMP308 DCR)[64]
  GTEMP308)
 (T (INCRB GUP1 'EXS (NCONC1 (GEARGS GUP1)  GTEMP12))
    [SOME (RIPPLE GUP1 'GENL)
          (FUNCTION (LAMBDA (G)
             (SOME (GETB G 'D-R)
                   (FUNCTION (LAMBDA (D)
                      (AND (ISA BA1 (CAR D))
                           (ISA BA2 (CADR D))
                           (INCRB GTEMP12 'UP[65] (CADDR D))
                           (INCRB (CADDR D) 'EXS GTEMP12]
(* This last INCRB says that if an operation f maps onto range C,
 and we apply f and get a new Being, then that Being ISA C)[66]
    (INCRB GTEMP12 'IN-RAN-OF GUP1)
    (INCRB BA2 'IN-DOM-OF GUP1)
```

---

[59] Similar to last case: takes care of missing second argument. The ampersand, "&", indicates an omission from this listing.

[60] An abbreviation for (APPLYB 'ANY-CONCEPT 'DEFN BA1); i.e., test whether BA1 is a bona fide concept or not.

[61] CREATEB is a function which sets up a new blank data structure for a new concept.

[62] The function call (INCRB C F X) means: add entry X to the F facet of concept C.

[63] The LISP function "Q" is like a double quote: after one evaluation (Q X) returns 'X; after one more evaluation, 'X returns X; after a final evaluation, we get the VALUE of X.

[64] A conditional print statement. If the verbosity level is high enough (>8), this message is typed out to the user. Note the intermixing of variables (e.g., "GTEMP308") and undefined atoms (e.g., "equivalent"). CPRIN1S examines each argument, and if it is undefined, it quotes it.

[65] The ISA's facet is called "UP" in the LISP program.

[66] This is a streamlined, specialized version of the more general heuristic rule number 154; see page 259.

```
(INCRB BA1 'IN-DOM-OF GUP1)
(* Now see if the composition GTEMP12 shares any ISA's entries with
     either constituent operation: BA1 or BA2)[67]
[MAPC [INTERSECTION (SET-DIFF [UNION (GETB BA1 'UP) (GETB BA2 'UP]
                              (GETB GTEMP12 'UP]
    (FUNCTION (LAMBDA (Z)
       (COND
         ((DEFN Z GTEMP12)
           (INCRB Z 'EXS GTEMP12)
           (INCRB GTEMP12 'UP Z]
(COND
  [(GETB GTEMP12 'UP)
    (SETB GTEMP12 'GUP (COPY (GETB GTEMP12 'UP]
  (T (INCRB GTEMP12 'UP 'OPERATION)
     (INCRB 'OPERATION 'EXS GTEMP12)))
& (* A similar search now for GENL/SPEC of the composition)
(SETB GTEMP12 'D-R (CAR GTEMP11))
(INCRB GTEMP12 'ALGS
    (LIST 'TYPE 'NONRECURSIVE 'APPLICATION 'OF GUP1 (CADR GTEMP11)))
& (* Code for synthesizing a Defn entry for GTEMP12)
(SETB GTEMP12 'WORTH
    (MAP2CAR (GETB BA1 'WORTH) (GETB BA2 'WORTH) 'TIMES1000))
(GS-CHECK[68] GTEMP12]]))]
```

Appearance on page 178:

Algorithms:
Distributed: use the heuristics attached to Compose to guide the filling in of various facets of the new composition.
(The heuristics referred to are shown in Appendix 3.6, on page 263.)
Fillin: 5 ***(out of a total of 9)*** **heuristics.**
Check: 1 heuristic ***(out of a total of 2)***

UP (OPERATION)

Appearance on page 178:

Isa's: Operation

[67] This next MAPC is thus the LISP encoding of heuristic rule number 177; see page 263.

[68] This is a general-purpose function for testing that there is no hidden cycle in the Generalization network, that no two concepts are both generalizations and specializations of each other, unless they are tagged as being equivalent to each other.

WORTH (300)

Appearance on page 178:

Worth: 300

INT[69]
```
[(IMATRIX (1 2 3) (4 5))
(COND [(INTERSECTION (MAPAPPEND (GETB BA2 'D-R) 'LAST)
                     (MAPAPPEND (GETB BA1 'D-R) 'ALL-BUT-LAST))
       300
       (IDIFF 400 (ITIMES 100 (IPLUS (LENGTH (GETB BA1 'D-R))
                                     (LENGTH (GETB BA2 'D-R]
      (REASON (* In some interpretation, Range-of-op2 is 1 component of Domain-of-op1)))
(COND [[MEMB [CAR (LAST (CAR (GETB BA2 'D-R]
             (ALL-BUT-LAST (CAR (GETB BA1 'D-R]
       400
       (IDIFF 1000 (ITIMES 100 (LENGTH (CAR (GETB BA1 'D-R]
      (REASON (* In canonical interpretation, Range-of-op2 is a component of Domain of op1)))
(COND [(INTERSECTION (GETB CS-B TIES)
                     (UNION (GETB BA1 TIES)(GETB BA2 TIES)))
       100
       (ITIMES 100 [LENGTH (INTERSECTION (GETB CS-B TIES)
                                         (UNION (GETB BA1 TIES)(GETB BA2 TIES])
      (REASON (* This composition preserves some good properties of its constituents))])
(COND [(SET-DIFFERENCE (GETB CS-B TIES)
                       (UNION (GETB BA1 TIES)(GETB BA2 TIES)))
       100
       (ITIMES 100 [LENGTH (SET-DIFFERENCE (GETB CS-B TIES)
                                           (UNION (GETB BA1 TIES)(GETB BA2 TIES])
      (REASON (* This composition has some new props, not true of either constituent))])
(COND [(OR (GREATERP (GETB BA1 'WORTH) 500))
           (GREATERP (GETB BA2 'WORTH) 500)))
       300
       (IQUOTIENT (ITIMES (GETB BA1 'WORTH)(GETB BA2 'WORTH))
                  1000)
      (REASON (* Op1 and/or Op2 are very interesting themselves))])
(COND [[IS-ONE-OF [CAR (LAST (CAR (GETB BA2 'D-R]
                  (ALL-BUT-LAST (CAR (GETB BA1 'D-R]
       350
       (IDIFF [ITIMES 100 (IDIFF
                  [LENGTH (CAR (GETB BA1 'D-R]
                  (LENGTH (RIPPLE [IS-ONE-OF
                                   [SETQ TMP4 (CAR (LAST (GETB BA2 'D-R]
                                   (ALL-BUT-LAST (CAR (GETB BA1 'D-R]
                           'GENL]
              (ITIMES 50 (LENGTH (RIPPLE TMP4 'GENL]
      (REASON (* In canonical interpretation, Range-of-op2 is a specialization of a component
                 of Domain-of-op1)))
```

[69] Note that although the Fillin and Suggest heuristics are blended into the relevant facets (e.g., into the Algorithms for COMPOSE), the INTERESTINGNESS type heuristics are kept separate, in this facet.

```
(COND [[MEMB [CAR (LAST (CAR (GETB BA1 'D-R]
             (ALL-BUT-LAST (CAR (GETB BA2 'D-R]
       450
       (IPLUS 300 (COND ([MEMB [CAR (LAST (CAR (GETB BA1 'D-R]
                                   (ALL-BUT-LAST (CAR (GETB BA1 'D-R]
                          10)
                         (T 250))
              (COND ([MEMB [CAR (LAST (CAR (GETB BA2 'D-R]
                                 (ALL-BUT-LAST (CAR (GETB BA2 'D-R]
                        11)
                       (T 250))
              (ITIMES 70 (LENGTH (RIPPLE [CAR (LAST (CAR (GETB BA1 'D-R] 'GENL]
     (REASON (* In canonical interpretation,
                Range-of-op1 is one component of Domain-of-op2))
&
(COND [[ISA [CAR (LAST (CAR (GETB BA1 'D-R]
            (ALL-BUT-LAST (CAR (GETB BA2 'D-R]
       250
       (IPLUS 50 (COND ([ISA [CAR (LAST (CAR (GETB BA1 'D-R]
                                  (ALL-BUT-LAST (CAR (GETB BA1 'D-R]
                         10)
                        (T 100))
              (COND ([ISA [CAR (LAST (CAR (GETB BA2 'D-R]
                                 (ALL-BUT-LAST (CAR (GETB BA2 'D-R]
                        11)
                       (T 100))
              (ITIMES 50 (LENGTH (RIPPLE [CAR (LAST (CAR (GETB BA1 'D-R] 'GENL]
     (REASON (* Range-of-op1 is a specialization of a component of Domain-of-op2]
```

Appearance on page 178:

> **Interest: 11 heuristics.**
>
> **The heuristic rules encoded above are shown in English on page 265.**

Here is the code for CON-MERGE-ARGS, the function which decides how to overlap the domain/range facets of its two arguments, F1 and F2:

```
(CON-MERGE-ARGS
  [LAMBDA (F1 F2 F12 PGM1 SCHK SAPL DOM1 DOM2 RAN1 RAN2 TIL DOM3)
    [SETQ RAN1 (LAST (CAR (GETB F1 'D-R]
    (SETQ DOM1 (LDIFF (CAR (GETB F1 'D-R))
                      RAN1))
    [SETQ RAN2 (LAST (CAR (GETB F2 'D-R]
    (SETQ DOM2 (LDIFF (CAR (GETB F2 'D-R))
                      RAN2))
    [SETQ DOM3 (AND (CDR DOM1)
                    (LIST (CADR (MIN2 (APPEND RAN2 RAN2 RAN2
                                      RAN2) DOM1 'FRAC-OVERLAP]
    (* As DOMi and RANi are located, Switching of Args may be required, inside PGM1)
    (AND (MEMB (CAR DOM3) DOM2) (SETQ DOM3 NIL))
```

```
(SETQ GTEMP20 (LENGTH DOM2))
[SETQ SAPL (NCONC (LIST 'APPLYB (KWOTE F1) (Q ALGS))
                  (MAPCAR (SUB-ONCE 'X
                                    [SETQ GTEMP19 (COND
                                       ((IS-ONE-OF (CAR RAN2) DOM1))
                                       [(SETQ SCHK (ONE-ISAG DOM1 (CAR RAN2]
                                       ((SETQ SCHK (AND (SETQ TIL (EXS (CAR RAN2))
                                                        (CAR (SOME DOM1 (FUNCTION (LAMBDA (D)
                                                                                  (INTERSECTION
                                                                                    TIL
                                                                                    (EXS D]
                                    DOM1)
                                  (FUNCTION (LAMBDA (Z)
                                     (COND
                                      ((EQ Z 'X)
                                       'X)
                                      (T (SETQ GTEMP20 (ADD1 GTEMP20))
                                       (CAR (FNTH BA-LIST GTEMP20]
    (* SCHK is a flag which means that f2 maps us into an element of RAN2 which is not guaranteed
    a priori to be an element of DOM1, hence a check for this applicability of f1 will then have to be made)
(COND
 ((FMEMB 'X SAPL)
  (SETQ DOM3 (REM-ONCE GTEMP19 DOM1))
  (SETQ GTEMP7 (APPEND DOM3 DOM2))
  [COND
   [(NEQ (LENGTH GTEMP7)
         (LENGTH (SELF-INT GTEMP7)))
    (CPRINIS 9 CRLF CRLF AM can later coalesce the D-R of F12 DCR)
    [ADD-CANDS (LIST (LIST (LIST 'APPLYB (Q COALESCE) (Q ALGS) (KWOTE F12))
                           (IPLUS 100 (IQUO (DOTPROD (FIRSTN 2 (GETB F1 'WORTH))
                                                     (GETB F2 'WORTH)) 2000))
                           (LIST (SPLIST There is an overlap in the new combined
                                         domain of the operation F12]
    (SWHY 9 (There is an obvious overlap in (@ GTEMP7),the new combined domain of (@ F12]
```

The next piece of this function is the heuristic rule numbered 186 in Appendix 3.

```
   ([SOME GTEMP7 (FUNCTION (LAMBDA (X)
                           (IS-ONE-OF X (CDR (FMEMB X GTEMP7]
    (CPRINIS 10 CRLF CRLF AM may later coalesce the D-R of F12 DCR)
    [ADD-CANDS (LIST (LIST (LIST 'APPLYB (Q COALESCE) (Q ALGS) (KWOTE F12))
                           (IQUO (DOTPROD (FIRSTN 2 (GETB F1 'WORTH))
                                          (GETB F2 'WORTH))  2500))
                           (LIST (SPLIST There may be an overlap
                                         in the new combined domain of the operation F12]
    (SWHY 10 (There is a subtle overlap in (@ GTEMP7),the new combined domain of (@ F12]
  [SETQ PGM1 (LIST 'PROG
                   (LIST 'X)
                   [LIST 'SETQ 'X
                         (NCONC (LIST 'APPLYB (KWOTE F2) (Q ALGS))
                                (FIRSTN (LENGTH DOM2) (LIST 'BA1 'BA2 'BA3]
                   (LIST 'RETURN
                         (COND
                          (SCHK (LIST 'AND
                                      (LIST 'APPLY* (Q DEFN) (KWOTE SCHK) 'X)
                                      SAPL))
                          (T (LIST 'AND 'X SAPL]
  (LIST (LIST (APPEND DOM2 DOM3 RAN1)) PGM1))
 (T (* Composing is not possible)  NIL])
```

Appendix 2.3.2. The 'Osets' Concept

Here is the actual property list of the data-structure corresponding to the Osets concept:

```
ENGN (OSET Oset Oset-structure OSET-STRUC, Ordered-set (Set))
DEFN (TYPE NEC&SUFF RECURSIVE TRANSPARENT [COND
          ((EQUAL BA1 (OSET )) T)
          (T (APPLYB 'OSET 'DEFN (APPLYB 'OSET-DELETE 'ALGS
                                    (APPLYB 'SOME-MEMB 'ALGS BA1)
                                    BA1])
     (TYPE NEC&SUFF RECURSIVE QUICK [COND
          ((EQUAL BA1 '(OSET )) T)
          ((CDDR BA1) (APPLYB 'OSET 'DEFN (RPLACD BA1 (CDDR BA1)))
          (T NIL])
     (TYPE NEC&SUFF NONRECURSIVE QUICK (MATCH BA1 WITH ('OSET &)))
GENL (ORD-STRUC NO-MULT-ELES-STRUC)
WORTH (400)
IN-DOM-OF (OSET-JOIN OSET-INTERSECT OSET-DIFF OSET-INSERT OSET-DELETE)
IN-RAN-OF (OSET-JOIN OSET-INTERSECT OSET-DIFF OSET-INSERT OSET-DELETE)
VIEW (STRUCTURE (RPLACA BA1 'OSET))
```

Compare this with the way that the "Osets" concept appeared, on page 214 of Appendix 2.1:

> **Name(s): Oset, Oset-structure, Ordered-set, sometimes: Set.**
> **Definitions:**
> **Recursive: λ (S) (S=[ ] or Oset.Definition(Oset-Delete.Alg(Member.Alg(S),S)))**
> **Recursive quick: λ (S) (S=[ ] or Oset.Definition (CDR(S)))**
> **Quick: λ (S) (Match S with [...])**
> **Generalizations: Ordered-Structure, No-multiple-elements-Structure**
> **Worth: 400**
> **In-domain-of: Oset-union, Oset-intersect, Oset-difference, Oset-insert, Oset-delete**
> **In-range-of: Oset-union, Oset-intersect, Oset-difference, Oset-insert, Oset-delete**
> **View: To view any structure as a Oset, do: λ (x) Enclose-in-square-brackets(x)**

Appendix 2.4. Concepts created by AM

The list below is meant to suggest the range of AM's definitions; it is far from complete, and most of the omissions were real losers. The concepts are listed in the order in which they were defined.[70] In place of the (usually-awkward) name chosen by AM, I have given either the standard math/English name for the concept, or else a short description of what it is.

Sets with less than 2 elements (singletons and empty sets).
Sets with no atomic elements (nests of braces).
Singleton sets.
Bags containing (multiple occurrences of) just one kind of element.
Superset (contains).
Doubleton bags and sets.
Set-membership.
Disjoint bags.
Subset.
Disjoint sets.
Singleton osets.
Same-length (same number of elements).
Same number of left parentheses, plus identical leftmost atoms.
Count (find the number of elements of a given structure).
Numbers (unary representation).
Add.
Minimum.
SUB1 (λ (x) x-1).
Insert x into a given Bag-of-T's (almost ADD1, but not quite).
Subtract (except: if x<y, then the result of x-y will be zero[71]).
Less than or equal to.
Times.
Union of a *bag* of structures.
& (the ampersand represents the creation of several real losers.)
Compose a given operation F with itself (form F∘F).
Insert structure S into itself.
Try to delete structure S from itself (a loser).
Double (add 'x' to itself).
Subtract 'x' from itself (as an operation, this is a real zero[72]).
Square (TIMES(x,x)).
Union structure S with itself.
Coalesced-replace2: replace each element s of S by F(s,s).
Coalesced-join2: append together F(s,s), for each member s∈S.
Coa-repeat2: create a new op which takes a struc S, op F, and repeats F(s,t,S) all along S.
Compose three operations: λ(F,G,H) F∘(G∘H).
Compose three operations: λ(F,G,H) (F∘G)∘H.

---

[70] See Appendix 5.2, p. 294, for a detailed trace of how these concepts were discovered. Or see Section 6.1, p. 115, for a briefer version of the same development.

[71] This is "natural-number subtract", in the same spirit of naming as we find for "integer division".

[72] a natural zero?

& (lots of losing compositions created, e.g. Self-insert∘Set-union.)
$ADD^{-1}(x)$: all ways of representing x as the sum of a bunch of nonzero numbers.
G∘H, s.t. H(G(H(x))) is always defined (wherever H is), and G and H are interesting.
Insert∘Delete.
Delete∘Insert.
Size∘ADD^{-1}. (λ (n) The number of ways to partition n)
Cubing
&
Exponentiation.
Halving (in natual numbers only; thus Halving(15)=7).
Even numbers.
Integer square-root.
Perfect squares.
Divisors-of.
Numbers-with-0-divisors.
Numbers-with-1-divisor.
Primes (Numbers-with-2-divisors).
Squares of primes (Numbers-with-3-divisors).
Squares of squares of primes.
Square-roots of primes (a loser).
$TIMES^{-1}(x)$: all ways of representing x as the product of a bunch of numbers (>1).
All ways of representing x as the product of just one number (a trivial notion).
All ways of representing x as the product of primes.
All ways of representing x as the sum of primes.
All ways of representing x as the sum of two primes.
Numbers uniquely representable as the sum of two primes.
Products of squares.
Multiplication by 1.
Multiplication by 0.
Multiplication by 2.
Addition of 0.
Addition of 1.
Addition of 2.
Product of even numbers.
Sum of squares.
Sum of even numbers.
& (losers: various compositions of 3 operations.)
Pairs of perfect squares whose sum is also a perfect square ($x^2+y^2=z^2$).
Prime pairs (p,p+2 are prime).

• • •

Appendix 3. AM's Heuristics

Infallible rules of discovery leading to the solution of all possible mathematical problems would be more desirable than the philosophers' stone, vainly sought by the alchemists. Such rules would work magic; but there is no such thing as magic. To find unfailing rules applicable to all sorts of problems is an old philosophical dream; but this dream will never be more than a dream.

-- Polya

To the extent that a professor of music at a conservatoire can assist his students in becoming familiar with the patterns of harmony and rhythm, and with how they combine, it must be possible to assist students in becoming sensitive to patterns of reasoning and how they combine. The analogy is not far-fetched at all

-- Dijkstra

This appendix lists all the heuristics with which AM is initially provided. They are organized by concept, most general concepts first. Within a concept, they are organized into four groups:

- Fillin: rules for filling in new entries on various facets.
- Check: rules for patching up existing entries on various facets.
- Suggest: rules which propose new tasks to break AM out of stagnant loops.
- Interest: criteria for estimating the interestingness of various entities.

Each heuristic is presented in English translation. Whenever there is a very tricky, non-obvious, or brilliant translation of some English clause into LISP, a brief note will follow about how that is coded. Also given (usually) are some example(s) of its use, and its overall importance. Concepts which have no heuristics are not present in this appendix.

Hundreds of heuristics were planned on paper but never coded (e.g., those dealing with proof techniques, those dealing with the drives and rewards of generalized message senders/receivers), and whole classes of rules were coded but never used by AM during any of its runs (e.g., how to deal with contradictions, how to deal with Intu's facets). Such superfluous rules will not be included here. They would raise the total number of heuristic rules from 242 to about 500.

The rule numbering in this Appendix is referred to occasionally in other appendices. The total number of rules coded in AM is actually higher, since many rules are present but never used, and since many rules listed with one number here are really *several* rules in LISP (e.g., see rules 97 and 129).

It would be advantageous to have a cross-indexing of the body of heuristics along several dimensions (a multiple sorting by a small set of key parameters): sorted by interest, by relevance (the current arrangement), by cost, by payoff, by frequency of usage, etc. This is left as a starred excercise for the interested reader.

Appendix 3.1. Heuristics for dealing with Anything

All these rules deal with any item X, be it concept, atom, event, etc. These rules are about as general – and as *weak* – as one can imagine.

Anything . Suggest

1. If AM has recently referenced entity X,
. Then boost the priority of any tasks involving X.

2. If the user has recently referred to X,
Then boost the priority of any tasks involving X.

The above two rules simply reaffirm the idea of "focus of attention". The boost in ratings is only slight, and only temporary (it decays toward zero exponentially with time). Besides this gradual decline in task ratings, the rule below explicitly modulates this boosting, so that infinite loops can be avoided.

3. If AM has recently dealt with X with poor results,
Then lower the priority rating of all tasks involving X.

4. If AM just referenced X and almost succeeded, but not quite,
Then look for a very similar entity Y, and retry the activity with Y in place of X.

There is a separate precise meaning for "almost succeed", "similar entity", and "retry" for each kind of entity and activity that might be involved. For example, if the activity were a *task* (say to fill in examples of Odd-primes) and the entity X were a *concept* (in this case, Odd-primes), then a 'similar entity' might be the concept Odd-numbers, and in that case the result of this rule would be *a new task* (to fill in examples of Odd-numbers). If the failure occurred while AM was trying to access the examples facet of Primes, with X=Examples, then a 'similar entity' might be the Boundary-examples facet, and the above rule would suggest that AM access instead the Boundary-examples facet of Primes. Of course, this rule is so weak that it is not often of much help.

5. If space is running out, and AM has not referenced X for a long time, and X is taking up a lot of space, and no important conjectures reference X,
Then X may be forgotten and its space liberated. Probably the user should be informed of this, at least tersely.

Just a general-purpose directive for emergency garbage-collection.

Anything . Interest

6. Any entity X is interesting if it is referred to in several interesting conjectures.

7. Any entity X is interesting if it is related (via a rare, interesting relation) to another entity which arose in a very different way and is not obviously tied to X.

Unexpected connections are worth closer examination, typically. X might be 'related to' Y because F(X)=Y (for some very interesting operation F), because Y(X) is true (for some rarely-satisfied predicate Y), because some conjecture involving X is syntactically identical to the same conjecture involving Y, etc.

8. Entity X is (tentatively) interesting if there is an analogy in which X corresponds to Y, and Y has turned out to be very interesting.

**9. If entity X is an example of concept C, and X satisfies some features on C.Int,
Then X is interesting, and C's interestingness features will indicate a numeric rating for X.**

This is practically the definiton of the Int facet. Below is a much more ususual rule:

**10. If entity X is an example of concept C, and X satisfies absolutely none of the features on C.Int, and X is just about the only C which doesn't satisfy something,
Then X is interesting because of its unusual boringness.**

Since most singletons are interesting because all pairs of their elements are Equal, the above rule says it would be interesting actually to find a singleton for which *not* all pairs of its members were equal. While it would be interesting, AM has very little chance of finding such a critter.

Appendix 3.2. Heuristics for dealing with Any-concept

This concept has a huge number of heuristics. For that reason, I have partitioned off – both here and in AM itself[1] – the heuristics which apply to each kind of facet.

Appendix 3.2.1. Heuristics for any facet of Any-concept

The first set of heuristics we'll look at are very general, applying to no particular facet exactly.

Any-concept . Fillin

11. When trying to fill in facet F of concept C, for any C and F,
If C is analogous to concept X, and X.F has some entries,
Then try to construct the analogs of those entries, and see if they are really valid entries for C.F.

Recall that "C.F" is shorthand for "facet F of concept C". This rule simply says that if an analogy exists between two concepts C and X, then it may be strong enough to map entries on X.F into entries for C.F. Note that F can be any given facet. There is an analogy between Sets and Bags, and AM uses the above rule to turn the extreme example of Sets – the empty set – into the extreme kind of bag.

Any-concept . Suggest

12. If the F facet of concept X is blank,
Then consider trying to fill it in.

The above super-weak rule will result in a new task being added to the agenda, for every blank facet of every concept. It is more of a legal move generator than a plausible move proposer. The rating of each such task will depend on the Worth of the concept X and the overall worth of the type F facet, but in all cases will be *very small*. The "emptiness" of a facet is always a valid reason for trying to fill it in, but never an a priori important reason. So the net effect of the rule is to slightly bias AM toward working on blank – rather than non-blank – facets.

13. While trying to fill in facet F of concept C, for any C and F, if C is known to be similar to some other concept D, except for difference d,
Then try to fill in C.F by selecting items from D.F for which d is nonexistent.

This rule is made more specific when F is actually known, and hence the format of d is actually determined. For example, if C=Reverse-at-all-levels, F=examples, then (at one particular moment) a note is found on the Conjecs facet of concept C which says that C is

---

[1] Thus the LISP program has a separate concept called "Examples-of-any-concept", another concept called "Definitions-of-any-concept", etc.

just like the concept D=Reverse-top-level, except C also recurs on the nonatomic elements of its arguments, whereas D doesn't. Thus d is made null by choosing examples of D for which there are no nonatomic elements. So an example like 'Reverse-top-level(<a b c>)=<c b a>' will be selected and will lead to the proposed example 'Reverse-at-all-levels(<a b c>)=<c b a>', which is in fact valid.

14. After dealing with concept C,
Slightly, temporarily boost the priority value of each existing task which involves an Active concept whose domain or range is C.

This is done efficiently using the In-dom-of and In-ran-of facets of C. A typical usage was after checking the just-filled-in examples of Bags, when AM slightly boosted the rating of filling in examples of Bag-union, and this task just barely squeaked through as the next one to be chosen. Note that the rule reinforced that task twice, since both domain and range of Bag-union are bags.

Any-concept . Check

15. When checking facet F of concept C, (for any F and C,)
Prune away all the entries there until the facet's size is reduced to the size which C merits.

The algorithm for doing this is as follows: The Worth of C is multiplied by the overall worth of facet type F. This is normalized in two ways, yielding the maximum amount of list cells that C.F may occupy, and also yielding the maximum number of separate entries to keep around on C.F. If either limit is being exceeded, then an entry is plucked at random (but weighted to favor selection from the rear of the facet) and excised. This repeats as long as C.F is oversized. As space grows tight, the normalization weights decline, so each concept's allocation is reduced.

16. When checking facet F of concept C,
Eliminate redundant entries.

Although it might conceivably mean something for an entry to occur twice, this was never desirable for the set of facets which each AM concept possessed.

Any-concept . Interest

The interest features apply to tell how interesting a concept is, and are rarely subdivided by relevant facet. That is, most of the reasons that Any concept might be interesting will be given below.

17. A concept X is interesting if X.Conjecs contains some interesting entries.

18. A concept is interesting if its boundary accidentally coincides with another, well-known, interesting concept.

The boundary of a concept means the items which just barely fall into (or just barely miss satisfying) the definition of that concept. Thus the boundary of Primes might include 1,2,3,4. If the boundary of Even numbers includes numbers differing by at most 1 from an even number, then clearly their boundary is *all* numbers. Thus it coincides with the already-known concept Numbers, and this makes Even-nos more interesting. This expresses the property we intuitively understand as: no number is very far from an even number.

19. A concept is interesting if its boundary accidentally coincides with the boundary of another, very different, interesting concept.

Thus, for example, Primes and Numbers are both a little more interesting since the extreme cases of numbers are all boundary cases of primes. Even numbers and Odd numbers both have the same boundary, namely Numbers. This is a tie between them, and slightly raises AM's interest in both concepts.

20. A concept is interesting if it is -- accidentally -- precisely the boundary of some other, interesting concept.

In the case mentioned for the above rule, Numbers is raised in interest because it turns out to be the boundary for even and odd numbers.

21. A concept is boring if, after several attempts, only a couple examples are found.

Another rule indicates, in such situations, that the concept may be forgotten and replaced by some conjecture.

22. Concept C is interesting if some normally-inefficient operation F can be efficiently performed on C's.

Thus it is very fast to perform Insert of items into lists because (i) no pre-existence checking need be done (as with sets and osets), and (ii) no ordered merging need be done (as with bags). So "Lists" is an interesting concept for that reason, according to the above rule.

23. Concept C is interesting if each example of C accidentally seems to satisfy the otherwise-rarely satisfied predicate P, or (equivalently) if there is an unusual conjecture involving C.

This is almost a primitive affirmation of intererestingness.

24. Concept C is interesting if C is closely related to the very interesting concept X.

This is intererestingness by association. AM was interested in Divisors-of because it was closely related to TIMES, which had proven to be a very interesting concept.

25. Concept C is interesting if there is an analogy in which C corresponds to Y, and the analogs of the interest features of Y indicate that C is interesting.

This might have been a very useful rule, if only there had been more decent analogies floating around the system. As it was, the rule was rarely used to advantage. It essentially says that the analogs of interest criteria are themselves (probably) valid criteria.

26. A concept C is interesting if one of its generalizations or specializations turns out to be unexpectedly very interesting.

"Unexpected" means that the interesting property hadn't already been observed for C. If C is interesting in some way, and then one of its generalizations is seen to be interesting in exactly the same way, then that is "expected". It's almost more interesting if the second concept unexpectedly *lacks* some fundamental property about C. At least in that case AM might learn something about what gives C that property. In fact, AM has this rule:

27. If concept C possesses some very interesting property lacked by one of its specializations S, Then both C and S become slightly more interesting.

In the LISP program, ths is closely linked with rule 104.

28. If a concept C is re-derived in a new way, that makes it more interesting.
If concepts C1 and C2 turn out to be equivalent concepts, then merge them. The combined concept is now more interesting than either of its predecessors.

The two conditionals above are really the same rule, so they aren't given separate numbers. C1 and C2 might be conjectured equivalent because their examples coincide, each is a generalization of the other, their definitions can be formally shown to be equivalent, etc. This rule is similar in spirit to rule number 114.

Appendix 3.2.2. Heuristics for the Examples facets of Any-concept

The following heuristics are used for dealing with the many kinds of examples facets which a concept can possess: non-examples, boundary examples, Isa links, etc.

Any-concept . Examples . Fillin

29. To fill in examples of X, where X is a kind of Y (for some more general concept Y),
Inspect the examples of Y; some of them may be examples of X as well.
The further removed Y is from X, the less cost-effective this rule is.

For the task of filling in Empty-structures, AM knows that concept is a specialization of Structures, so it looks over all the then-known examples of Structures. Sure enough, a few of them are empty (satisfy Empty-structures.Defn). Similarly, for the task of filling in examples of Primes, this rule would have AM notice that Primes is a kind of Number, and therefore look over all the known examples of Number. It would not be cost-effective to look for primes by testing each example of Anything, and the third and final clause in the above rule recognizes that fact.

30. To fill in non-examples of concept X,
Search the specializations of X. Look at all their non-examples. Some of them may turn out to be non-examples of X as well.

This rule is the counterpart of the last one, but for *non*-examples. As expected, this was less useful than the preceding positive rule.

31. If the current task is to fill in examples of any concept X,
Then one way to get them is to symbolically instantiate a definition of X.

That rule simply says to use some known tricks, some hacks, to wring examples from a declarative definition. One trick AM knows about is to plug already-known examples of X into the recursive step of a definition. Another trick is simply to try to instantiate the base step of a recursive definition. Another trick is to take a definition of the form "λ (x) x isa P, and *<sub-expression>*", work on instantiating just the *sub-expression*, and then pop back up and see which of those items are P's.

32. If the current task is to fill in non-examples of concept X,
Then one fast way to get them is to pick any random item, any example of Anything, and check that it fails X.Defn.

This is an affirmation that for any concept X, most things in the universe will probably not be X's. This rule was almost never used to good advantage: non-examples of a concept X were never sought unless there was some reason to expect that they might not exist. In those cases, the presumption of the above rule was wrong, and it failed. That is, the rule succeeded iff it was not needed.[2]

33. To fill in examples of concept X,
If X.View tells how to view a Z as if it were an X, and some examples of Z are known,
Then just run X.View on those examples, and check that the results really are X's.

Thus examples of osets were found by viewing other known examples of structures (e.g., examples of sets) as if they were osets.

34. To fill in examples of concept X,
Find an operation whose range is X,[3] and find examples of that operation being applied.

To fill in examples of Even-nos, this rule might have AM notice the operation 'Double'. Any example of Double will contain an example of an even number as its value: e.g., <3→6> contains the even number 6.

35. If the current task is to fill in examples of concept X,
One bizarre way is to specialize X, adding a strong constraint to X.Defn, and then look for examples of that new specialization.

Like the classical "insane heuristic"[4], this sounds crazy but works embarassingly often. If I ask you to find numbers having a prime number of divisors, the rate at which you find them will probably be lower than if I'd asked you to find numbers with precisely 2 divisors. The *variety* of examples will suffer, of course. The converse of this heuristic – for non-examples – was deemed too unaesthetic to feed to AM.

---

[2] Catch-22?

[3] or at least INTERSECTS X. Use the In-ran-of facets and the rippling mechanism to find such an operation.

[4] A harder task might be easier to do. A stronger theorem might be easier to prove. This is called "The Inventor's Paradox", on page 121 of [Polya 57].

36. To fill in examples of X,
One inefficient method is to examine random examples of Anything, checking each by running X.Defn to see if it is an X. Slightly better is to ripple outward from X in all directions, testing all the examples of the concepts encountered.

This is blind generate-and-test, and was (luckily) not needed much by AM.

37. To find more examples of X (or: to find an extreme example of X), when a nice big example is known, and X has a recursive definition,
Try to plug the known example into the definition and produce a simpler one. Repeat this until an example is produced which satisfies the base-step predicate of the definition. That entity is then an extreme (boundary) example of X.

For example, AM had a definition of a set as

"Set(S) if S={} or if Set(Remove-random-element(S))." When AM found the big example {A,B,{{C},D},{{{E}}},F} by some other means, it used the above rule and on he recursive definition to turn this into {A,B,{{{E}}},F} by removing the randomly-chosen third element. {A,B,F} was produced next, followed by {B,F} and {F}. After that, {} was produced and the rule relinquished control.

38. To find examples of X, when X has a recursive definition,
One method with low success rate but high payoff is to try to invert that definition, thereby creating a procedure for generating new examples.

Using the previous example, AM was able to turn the recursive definition of a set into the program "Insert-any-random-item(S)", which turns any set into a (usually different and larger) new set. Since the rules which AM uses to do these transformations are very special-purpose, they are not worth detailing here. This is one very managable open problem, where someone might spend some months and create a decent body of definition-inversion rules. A typical rule AM has says:
"Any phrase matching '*Removing an x and ensuring that P(x)*' can be inverted and turned into this one: '*Finding any random x for which P(x) holds, then inserting x*'." The class of definitions which can be inverted using AM's existing rules is quite small; whenever AM needed to be able to invert another particular definition, the author simply supplied whatever rules would be required.

39. While filling in examples of C,
if two constructs x and y are found which are very similar yet only one of which is an example of the concept C,
Then one is a boundary example of C, and the other is a boundary non-example,
and it's worth creating more boundary examples and boundary non-examples by slowly transforming x and y into each other.

Thus when AM notices that {a} and {a,b,a} are similar yet not both sets, it creates {a,b}, {b,a}, {a,a} and sees which are and are not examples of sets. In this way, some boundary items (both examples and non-examples) are created. The rules for this slow transformation are again special purpose. They examine the difference between the items x and y, and suggest operators (e.g., Deletion) which will reduce that difference. This GPS-like strategy has been well studied by others, and its inferior implementation inside AM will not be detailed.

40. If the main task now is to fill in examples of concept C,
Consider all the examples of "first cousins" of C. Some of them might be examples of C as well.

By "first cousins", we mean all direct specializations of all direct generalizations of a concept, or vice versa. That is, going up once along a Genl link, and then down once along a Spec link (or going down one link and then up one link).

41. If the main task now is to fill in boundary (non-)examples of concept C,
Consider all the boundary (non-)examples of "first cousins" of C. Some of them might lie on the boundary of C as well.

If they turn out not to be boundary examples, they can be recorded as boundary non-examples, and vice versa.

42. To fill in Isa links of concept X, (that is, to find a list of concepts of which X is an example),
Just ripple down the tree of concepts, applying a definition of each concept. Whenever a definition fails, don't waste time trying any of its specializations. The Isa's of X are then all the concepts tried whose definitions passed X.

When a new concept is created, e.g., a new composition, this rule can ascertain the most specific Isa links that can be attached to it. Another use for this rule would be: If the Isa link network ever got fouled up (contained paradoxes), this rule could be used to straighten everything out (with a logarithmic expenditure of time).

Any-concept . Examples . Suggest

43. If some (but not most) examples of X are also examples of Y (for some concept Y),
and some (but not most) examples of Y are also examples of X,
Create a new concept defined as the intersection of those two concepts (X and Y). This will be a specialization of both concepts.

If you hapen to notice that some primes are palindromic, this rule would suggest creating a brand new concept, defined as the set of numbers which are both palindromic and prime. AM never actually noticed this, since it represented all numbers in unary. If pushed, AM will define Palindrome(n) to mean that the sequence of exponents of prime factors is symmetric; thus $2^3 9^8 5^1 7^1 11^8 13^3$ is palindromic in AM's sense because the sequence of its exponents (3 8 1 1 8 3) is unchanged upon reversal. In this sense, the only Prime palindromes are the primes themselves (or: just '2', depending upon the precise definition).

44. If very few examples of X are found,
Then add the following task to the agenda: "Generalize the concept X", for the following reason: "X's are quite rare; a slighlly less restrictive concept might be more interesting".

Of course, AM contains a precise meaning for the phrase "very few". When AM looks for primes among examples of already-known kinds of numbers, it will find dozens of non-examples for every example of a prime it uncovers. "Very few" is thus naturally implemented as a statistical confidence level. AM uses this rule when very few examples of Equality are found readily.

**45. If very many examples of X are found in a short period of time,
Then try to create a new, specialized version of X.**

This is similar to the preceding rule. Since numbers are easy to find, this might cause us to look for certain more interesting subclasses of numbers to study.

**46. If there are no known examples for the interesting concept X,
Then consider spending some time looking for such examples.**

I've heard of a math student who defined a set of number which had quite marvelous properties. After the 20th incredible theorem about them he'd proved, someone noticed that the set was empty. The danger of unwittingly dealing with a vacuous concept is even worse for a machine than for a human mathematician. The above rule explicitly prevents that.

**47. If the totality of examples of concept C is too small to be interesting,
Then consider these reactions: (i) generalize C; (ii) forget C completely; (iii) replace C by one conjecture.**

This is a good example of when a task like "Fill in generalizations of Numbers-with-1-divisors" might get proposed with a high-priority reason. The class of entities which C encompasses is simply too small, too trivial to be worth maintaining a separate concept. When C is numbers-with-1-divisor, C is really just another disguise for the singleton set {1}. The above rule might cause a new task to be added to the agenda, Fill in generalizations of Numbers-with-1-divisor. When that task is executed, AM might create the concept Numbers-with-odd-no-of-divisors, Numbers-with-prime-number-of-divisors, etc. Besides generalizing that concept, the above rule gives AM two other alternatives. AM may simply obliterate the nearly-vacuous concept, perhaps leaving around just the statement "*1 is the only number with one divisor*". That conjecture might be tacked onto the Conjecs facet of Divisors-of. The actual rule will specify criteria for deciding which of the three alternatives to try. In fact, AM really starts all three activities: a task will always be created and added to the agenda (to generalize C), the vacuous concept will be tagged as "forgettable", and AM will attempt to formulate a conjecture (the only items satisfying C.Defn are C.Exs).

**48. If the totality of examples of concept C is too large to be interesting,
Then consider these three possible reactions: (i) specialize C; (ii) forget C completely; (iii) replace C by one conjecture.**

This is analogous to the preceding rule, but is used far less frequently. One common use is when a disjunction of two concepts has been formed which is accidentally large or already-known (e.g., "Evens u Odds" would be replaced by a conjecture).

**49. After filling in examples of C, if some examples were found,
Look at all the operations which can be applied to C's (that is, access C.In-dom-of), find those which are interesting but which have no known examples, and suggest that AM fill in examples for them, because some items are now known which are in their domain, namely C.Exs.**

This rule had AM fill in examples of Set-insertion, as soon as some examples of Sets had been found.

**50. After filling in examples of C, if some examples were found,
Consider the task of Checking the examples facet of concept C.**

This was very frequently used during AM's runs.

51. After checking examples of C, if many examples remain,
Consider the task of 'Filling in some Conjecs for C'.

This was used often by AM. After checking the examples of C, AM would try to empirically formulate some interesting conjecture about C.

52. After successfully filling in non-examples of X, if no examples exist,
If AM has not recently tried to find examples of X, then it should do so.
If AM has recently tried and failed to find examples, consider the conjecture that X is vacuous, empty, null, always-False. Consider generalizing X.

53. After trying in vain to find some non-examples of X, if many examples exist,
Consider the conjecture that X is universal, always-True. Consider specializing X.

54. After successfully filling in examples of X, if no non-examples exist,
If AM has not recently tried to find non-examples of X, then it should consider doing so.
If AM has recently tried and failed to find non-examples, consider the conjecture that X is universal, always-True. Consider specializing X.

55. After trying in vain to find some examples of X,
If many non-examples exist,
Consider the conjecture that X is vacuous, null, empty, always-False. Consider generalizing X.

Any-concept . Examples . Check

56. If the current task is to Check Examples of concept X,
and (Forsome Y) Y is a generalization of X with many examples,
and all examples of Y (ignoring boundary cases) are also examples of X,
Then conjecture that X is really no more specialized than Y,
and Check the truth of this conjecture on boundary examples of Y,
and see whether Y might itself turn out to be no more specialized than one of *its* generalizations.

This rule caused AM, while checking examples of odd-primes, to conjecture that *all* primes were odd-primes.

57. If the current task is to Check Examples of concept X,
and (Forsome Y) Y is a specialization of X,
and all examples of X (ignoring boundary cases) are also examples of Y,
Then conjecture that X is really no more general than Y,
and Check the truth of this conjecture on boundary examples of X,
and see whether Y might itself turn out to be no more general than one of *its* specializations.

This rule is analogous to the preceding one for generalizations.

58. When checking boundary examples of a concept C,
ensure that every scrap of C.Defn has been used.

It is often the tiny details in the definition that determine the precise boundary. Thus we must look carefully to see whether Primes allows 1 as an example or not. A definition like "numbers divisible only by 1 and themselves" includes 1, but this definition doesn't: "numbers having precisely 2 divisors". In the LISP program, this rule contains several hacks (tricks) for checking that the definition has been stretched to the fullest. For example: if the definition is of the form "all x in X such that...", then pay careful attention to the boundary of X. That is, take the time to access X.Boundary-exs and X.Boundary-non-exs, and check them against C.Defn.

59. When checking examples of C,
Ensure that each example satisfies C.Defn, and each non-example fails it. The precise member of C.Defn to use can be chosen depending on the example.

As described earlier in the text, definitions can have descriptors which indicate what kinds of arguments they might be best for, their overall speed, etc.

60. When checking examples of C,
If an entry e is rejected (i.e., it is seen to be not an example of C after all), then remove e from C.Exs and consider inserting it on the Boundary non-examples facet of C.

There is a complicated[5] algorithm for deciding whether to forget e entirely or to keep it around as a close but not close enough kind of example.

61. When checking examples of C,
After an entry e has been verified as a bona fide example of C,
Check whether e is also a valid example of some direct specialization of C.
If it is, then remove it from C.Exs, and consider adding it to the examples facet of that specialization, and suggest the task of Checking examples of that specialization.

62. When checking examples of C,
If an entry e is rejected,
Then check whether e is nevertheless a valid example of some generalization of C.
If it is, consider adding it to that concept's boundary-examples facet, and consider adding it to the boundary non-examples facet of C.

This is similar to the preceding rule.

63. When checking non-examples of C, including boundary non-examples,
Ensure that each one fails a definition of C. Otherwise, transfer it to the boundary examples facet of C.

---

[5] Not necessarily sophisticated. First, AM accesses the Worth of C. From this it determines how many boundary non-examples C deserves to keep around (and how many total list cells it merits). AM compares these quotas with the current number of (and size of) entries already listed on C.bdy-non-exs. The degree of need of another entry there then sets the "odds" for insertion versus forgetting. Finally a random number is computed, and the odds determine what range it must lie in for e to be remembered.

64. When checking non-examples of C, including boundary non-examples,
After an entry e has been verified as a bona fide non-example of C,
Check whether e is also a non-example of some direct generalization of C.
If it is, then remove it from C.Non-Exs, and consider adding it to the non-examples facet of that generalization, and suggest the task of Checking examples of that generalization.

65. When checking (boundary) non-examples of C,
If an entry e is rejected, that is if it turns out to be an example of C after all,
Then check whether e is nevertheless a non-example of some specialization of C.
If it is, consider adding it to that concept's boundary non-examples facet.

This is similar to the preceding rule.

Appendix 3.2.3. Heuristics for the Conjecs facet of Any-concept

Any-concept . Conjecs . Fillin

When the task is to look around and find conjectures dealing with concept C, the following general rules may be useful.

66. If there is an analogy from X to C, and a nice item in X.Conjecs, formulate and test the analogous conjecture for C.

Since an analogy is not much more than a set of substitutions, formulating the 'analogous conjecture' is almost a purely syntactic transformation.

67. Examine C.Exs for regularities.

What mysteries are lurking in the LISP code for *this* rule, you ask? Nothing but a few special-purpose hacks and a few ultra-general hacks. Here is a slightly more specific rule for you seekers:

68. Look at C.Exs. Pick one element at random. Write down statements true about that example e. Include a list of all concepts of which it is an example, all interests features it satisfies, etc.
Then check each conjecture on this list against all other known examples of C. If any example (except a boundary example) of C violates a conjecture, discard it.
Take all the surviving conjectures, and eliminate any which trivially follow from other ones.

This is a common way AM uses: induce a conjecture from one example and test it on all the rest. A more sophisticated approach might be to induce it by using a few examples simultaneously, but I haven't thought of any nontrivial way to do that. The careful reader will perceive that most of the conjectures AM will derive using this heuristic will be of the form "X is unexpectedly a specialization of Y", or "X is unexpectedly an example of Y", etc. Indeed, most of AM's conjectures are really that simple syntactically.

69. Formulate a parameterized conjecture, a "template", which gets slowly specialized or instantiated into a definite conjecture.

AM has only a few trivial methods for doing this (e.g., introduce a variable initially and find the constant value to plug in there later). As usual, they will be omitted here, and the author encourages some research in this area, to turn out a decent set of general rules for accomplishing this hypothesis template instantiation. The best effort to date along these lines, in one specific sophisticated scientific field, is that of META-DENDRAL [Buchanan].

Any-concept . Conjecs . Check

70. If a universal conjecture (For all X's, ...) is contradicted by empirical data, gather the data together and try to find a regularity in those exceptions.
If this succeeds, give the exceptions a name N (if they aren't already a concept), and rephrase the conjecture (For all X's which are not N's...). Consider making X-N a new concept.

Again note how "active" this little checking rule can be. It can patch up nearly-true conjectures, examine data, define new concepts, etc.

71. After verifying a conjecture for concept C,
See if it also holds for related concepts (e.g., a generalization of C).

There are of course bookeeping details not explicitly shown above, which are present in the LISP program. For example, if conjecture X is true for all specializations of C, then it must be added to C.Conjecs and removed from the Conjecs facets of each specialization of C.

Any-concept . Conjecs . Suggest

72. If X is probably related to Y, but no definite connection is known,
It's worthwhile looking for a specific conjecture tying X and Y together.

How might AM know that X and Y are only *probably* related? X and Y may play the same role in an analogy (e.g., the singleton bag "(T)" and "any typical singleton bag" share many properties), or they may both be specializations of the same concept Z (e.g., two kinds of numbers), or they may both have been created in the same unusual way (e.g., Plus and Times and Exponentiation are all creatable by *repeating* another operation).

Any-concept . Conjecs . Interest

73. A conjecture about X is interesting if X is very interesting.

74. A nonconstructive existence conjecture is interesting.

Thus the unique factorization theorem is judged to be interesting because it merely guarantees that some factoring will be into primes. If you give an algorithm for that factoring, then the theorem actually loses its mystique and (according to this rule) some of its value. But it increases in value due to the next rule.

75. A constructive existence conjecture is interesting if it is frequently used.

76. A conjecture C about X is interesting if the origin and the verification of C for each specialization of X was quite independent of each other, and preceded C's being noticed applicable to all X's.

This would be even more striking if *proof* techniques were known, and each specialized case had a separate kind of proof. Many number theory results are like this, where there exists a general proof only for numbers bigger than 317, say, and all smaller numbers are simply checked individually to make sure they satisfy the conjecture. Category theory is built upon practically nothing but this heuristic.

Appendix 3.2.4. Heuristics for the Analogies facet of Any-concept

Any-concept . Analogies . Fillin

77. To fill in conjectures involving concept C, where C is analogous to D, Consider the analogue of each conjecture involving D.

78. If the current task involves a specific analogy, and the request is to find more conjectures, Then consider the analog of each interesting conjecture about any concept involved centrally in the analogy.

That is, this rule suggests applying the preceding rule to each concept which is central to the given analogy. The result is a flood of new conjectures. There is a tradeoff (explicitly taken into account in the LISP version of this rule) between how interesting a conjecture has to be, and how centrally a concept has to fit into the analogy, in order to determine what resources AM should be willing to expend to find the analogous conjecture. Note that this is not a general suggestion of what to do, but a specific strategy for enlarging the analogy itself. If the new conjecture is verified, then not only would it be entered under some Conjecs facet, but it would also go to enlarging the data structure which represents the analogy.

79. Let the analogy suggest how to specialize and generalize each concept into what is at least the analog of a known, very interesting concept.

Like the last rule, this one simply says to use the analogy itself as the "reason" for exploring certain new entities, in this case new concepts. When the Bags↔Numbers analogy is made, AM notices that Singleton bags and Empty bags are two interesting, extreme specializations of Bags. The above rule then allows AM to construct and study what we know and love as the numbers one and zero. The analogy is flawed because there is only one "one", although there are many different singleton bags. But just as singletons and empty bags have special properties under bag operations, so do 0,1 under numeric operations. This was one case where an analogy paid off handsomely.

80. If it is desired to have an analogy between concepts X and Y, then look for two already-known analogies between X↔Z and Z↔Y, for any Z.
If found, compose the two analogies and see if the resultant analogy makes sense.

Since the analogies are really just substitutions, composing them has a familiar, precise meaning. This rule was never really used by AM, due to the paucity of analogies. The user can push AM into creating more of them, and ultimately using this rule. A chain from X↔Z↔Y↔X can be found which presents a new, bizarre analogy from X to itself.

Any-concept . Analogies . Suggest

81. If an analogy is strong, and one concept has a very interesting universal conjecture C (For all x in B...), but the analog conjecture C' is false,
Then it's worth constructing the set of items in B' for which the conjecture holds. It's perhaps even more interesting to isolate the set of exceptional elements.

With the Add↔Times analogy, it's true that all numbers n>1 can be represented as the sum of two other numbers (each of them smaller than n), but it is *not* true that all numbers (with just a couple exceptions) can be represented as the product of other (hence smaller) numbers. The above rule has AM define the set of numbers which can/can't be so represented. These are just the composite numbers and the set of primes. This second way of encountering primes was very unexpected – both by AM and by the author. It expresses the deep fact that one difference between Add and Times is the presence of primes only for multiplication. At the time of its discovery, AM didn't appreciate this fully of course.

82. If space is tight, and no use of the analogy has ever been made, and it is very old, and it takes up a lot of space,
Then it is permissable to forget it without a trace.

83. If two valuable conjectures are syntactically identical, and can be made identical by a simple substitution, then tentatively consider the analogy which is that very substitution.

Thus the associative/commutative property of Add and Times causes them to be tied together in an analogy, because of this rule.

84. If an analogy is very interesting and very complete,
Then spend some time refining it, looking for small exceptions. If none are found, see whether the two situations are genuinely isomorphic.

85. If concepts X and Y are analogous, look for analogies between their specializations, or between their generalizations.

This rule is not used much by AM, although the author thought it would be.

Any-concept . Analogies . Interest

86. An analogy which has no discrepancies whatsoever is not as interesting as a slightly flawed analogy.

87. An analogy is interesting if it associates (for the first time) two concepts which are each unusually fully filled out (having many conjectures, many examples, many interest features, etc.).

Appendix 3.2.5. Heuristics for the Genl/Spec facets of Any-concept

Any-concept . Genl/Spec . Fillin

88. To fill in specializations of X, if it was very easy to find examples of X,
Grab some features which would indicate than an X was interesting (some entries from X.Interest, or more remote interest predicates garnered by rippling), and conjoin them onto the definition of X, thereby creating a new concept.

Here's one instance where the above rule was used: It was so easy for AM to produce examples of sets that it decided to specialize that concept. The above rule then plucked the interestingness feature "all pairs of members satisfy the same rare predicate" and conjoined it to the old definition of Sets. The new concept, Interesting-sets, included all singletons (because all pairs of members drawn from a singleton satisfy the predicate Equal) and empty sets.

89. To fill in generalizations of concept X,
Take the definition e, and replace it by a generalization of e. If e is a concept, use e.Genl; if e is a conjunction, then remove a conjunct or generalize[6] a conjunct; if e is a disjunction, then add a disjunct or generalize a disjunct; if e is negated, then specialize the negate; if e is an example of E, then replace e by "any example of E"; if e satisfies any property P, then replace e by "anything satisfying P"; if e is a constant[7], then replace e by a new variable (or an existing one) which could assume value e; if e is a variable, then enlarge its scope of possible bindings.

This rule contains a bag of tricks for generalizing any LISP predicate, the definition of any concept. They are all *syntactic* tricks, however.

90. To fill in generalizations of concept X, If some conjecture exists about "all X's and Y's" or "in X or Y", for some other concept Y,
Create a new concept, a generalization of both X and Y, defined as their disjunction.

---

[6] ie. recur.

[7] Of course it's unlikely that a concept is defined simply as a constant, but the preceding footnote shows that this little program can be entered recursively, being fed a sub-expression of the definition.

This rule contains another trick for generalizing any concept, although it is more meaningful, more semantic than the previous rule's tricks. Many theorems are true about numbers with 1 or 2 divisors, so this might be one reasonable way to generalize Numbers-with-1-divisor into a new useful[8] concept.

91 To fill in generalizations of concept X,
If other generalizations G1, G2 of X exist but are TOO general,
Create a new concept, a generalization of X and a specialization of both G1 and G2, defined as the conjunction of G1 and G2's definitions.

Thus when AM generalizes Reverse-all-levels into Reverse-top-level and Reverse-first-element, the above rule causes AM to create a new operation, which reverses the top level and which reverses the CAR[9] of the original list. While not particularly useful, the reader should observe that it is in fact midway in generality between the original Reverse function and the first two generalizations.

92 To fill in specializations of concept X,
Take the definition e, and replace it by a specialization of e. If e is a concept, use e.Genl; if e is a disjunction, then remove a disjunct or specialize a disjunct; if e is a conjunction, then add a conjunct or specialize a conjunct; if e is negated, then generalize the negate; if e is "any example of E", then replace e by a particular example of E; if e is "anything satisfying P", then replace e by a particular satisfier of P; if e is a variable, then replace it by a well-chosen constant or restrict its scope.

This rule contains a bag of tricks for specializing any LISP predicate, the definition of any concept. They are all *syntactic* tricks, however. Note that the Lisp code for this rule will typically call itself (recur) in order to specialize small pieces of the original definition.

93. To fill in specializations of concept X, if some conjecture exists about "all X's which are also Y's" or "in X and Y", for some other concept Y,
Create a new concept, a specialization of both X and Y, defined as their conjunction.

This rule contains another trick for specializing any concept, although it is more meaningful, more semantic than the previous rule's tricks. Many theorems about primes contain the condition "p>2"; i.e., they are really true about primes which are odd. So this might be one reasonable way to specialize Primes into a new concept: by conjoining the definitions of Primes and Odd-numbers, into the new concept Odd-primes. Here's another usage of this rule: If AM had originally defined Primes to include '1', then the frequency of conjectures where 1 was an exception would trigger this rule to define Primes more normally (p≥2).

94. To fill in specializations of concept X,
If other specializations S1, S2 of X exist but are TOO restrictive to be interesting,
Create a new concept, a specialization of X and a generalization of both S1 and S2, defined as the disjunction of S1 and S2's definitions.

---

[8] at least, several theorems will be stated more concisely using this new concept: Numbers with 1 or 2 divisors.

[9] also the CAR of the CAR, etc., until a non-list is encountered.

95. To fill in generalizations of concept X, when a recursive definition of X exists,
If the definition contains two conjoined recursive calls, replace them by a disjunction or eliminate one call entirely.
If there is only one recursive call, disjoin a second call, this one on a different destructive function applied to the original argument. If the original destructive function is one of {CAR,CDR}, then let the new one be the other member of that pair.

AM uses the first part of this rule to turn Equal-lists into two variants of Same-length-as. The second part, while surprisingly unused, could work on this definition of MEMBER: "λ (x,L) LISTP(L) and: [x=CAR(L) or MEMBER(x,CDR(L))]", which is just "membership at the top level of", or ∈, and transform it into this one of MEM, which represents membership at *any* depth: "λ(x,L) LISTP[10](L) and: [x=CAR(L) or MEM(x,CDR(L)) or MEM(x,CAR(L))]". The rule noticed a recursive call on CDR(L), and simply disjoined a recursive call on CAR(L).

96. To fill in specializations of concept X, when a recursive definition of C exists,
If the definition contains two disjoined recursive calls, replace them by a conjunction or eliminate one call entirely.
If there is only one recursive call, conjoin a second on another destructive function applied to the original argument. Often the two recursions will be on the CAR and the CDR of the original argument to the predicate which is the definition for X.

This is closely related to the preceding rule. Just as it turned the concept of 'element of' into the more general one of 'membership at any depth', the above rule can specialize the definition of MEMBER into this one, called AMEM: "λ (x,L) LISTP(L) and: [x=CAR(L) or: [AMEM(x,CDR(L)) and AMEM(x,CAR(L))]]".[11]

97. To fill in specializations of concept X,
Find,, within a definition of X (at even parity of NOT's), an expression of the form "For some x in X, P(x)", and replace it either by "For all x in X, P(x)", or by $P(x_0)$.

Thus "sets, all pairs of whose members satisfy SOME interesting predicate" gets specialized into "sets, all pairs of whose members satisfy Equality". The same rule, with "even parity" replaced by "odd parity", is useful for *generalizing* a definition. This rule is really 4 separate rules, in the LISP program. The same rule, with the transformations going in the opposite direction, is also used for generalizing. The same rule, with the transformations reversed and the parity reversed, is used for specializing a definition. Here is that doubly-switched rule:

98. To fill in specializations of concept X,
Find within a definition of X (at odd parity of NOT's) an expression of the form "For all x in X, P(x)", and replace it either by "For some x in X, P(x)", or by $P(x_0)$. Or replace "P(∝)", where ∝ is a constant, by "For some x in A, P(x)" where A is a concept of which ∝ is one example.

---

[10] The Interlisp function LISTP(L) tests whether or not L is a (nonnull) list.

[11] This operation is almost impossible to explain verbally. AMEM(x,L) means that x is an element of L, and for each member M of L before the x, M is an ordered structure and x is an element of M, and for each member N of M before the x which is inside M, etc. E.g. <[x] [<<x a b> <x> x d e> <x f> x g h] [<x i> x j] x k [l] m>.

99. When creating in a specialization S of concept C,
Note that S.Genl should contain C, and that C.Spec should contain S.

The analogous rule exists, in which all spec and genl are switched.

Any-concept . Genl/Spec . Suggest

100. After creating a new specialization S of concept C,
Explicitly look for ties between S and other known specializations of C.

For example, after AM defines the new concept of Numbers-with-3-divisors, it looks around for ties between that kind of number and other kinds of numbers.

101. After creating a new generalization G of concept C,
Explicitly look for ties between G and other close generalizations of C.

For example, AM defined the new predicates Same-size-CARs and Same-size-CDRs[12] as two generalizations of Equality. The above rule then suggested that AM explicitly try to find some connection between these two new predicates. Although *AM* failed, Don Knuth (using a similar heuristic, perhaps) also looked for a connection, and found one: it turns out that the two predicates are both ways of defining the relation we intuitively understand as "having the same length as".

102. After creating a new specialization S of concept C,
Consider looking for examples of S.

This has to be said explicitly, because all too often a concept is specialized into vacuousness.

103. After creating a new generalization G of concept C,
Consider looking for non-examples of G.

This has to be said explicitly, because all too often a concept is generalized into vacuous universality. This rule is less useful to AM than the preceding one.

104. If concept C possesses some very interesting property lacked by one of its specializations S,
Then considering creating a concept intermediate in specialization between C and S, and see whether that possesses the property.

This rule will trigger whenever a new generalization or specialization is created.

105. If concept S is now very interesting, and S was created as a specialization of some earlier concept C,
Give extra consideration to specializing S, and to specializing concept C again (but in different ways than ever before).

---

[12] Two lists satisfy Same-size-CDRs iff they have the same number of members. Two lists satisfy Same-size-CARs iff (when written out in standard LISP notation) they have the same number of initial left parentheses and also have the same first identifier following that last initial left parenthesis.

The next rule is the analog of the preceding one. They incorporate tiny bits of the strategies of hill-climbing and learning from one's successes.

106. If concept G is now very interesting, and G was created as a generalization of some earlier concept C,
Give extra consideration to generalizing G, and to generalizing C in other ways.

The analogous rules exist, for concepts that have become so boring they've just been discarded:

107. If concept X proved to be a dead-end, and X was created as a generalization of (specialization of) some earlier concept C,
Give less consideration to generalizing (specializing) X, and to generalizing (specializing) C in other ways in the future.

Any-concept . Genl/Spec . Check

108. When checking a generalization G of concept C,
Specifically test to ensure that G is not equivalent to C.
The easiest way is to examine the non-examples (especially boundary non-examples) of C, and look for one satisfying G; or examine the examples of G (esp. boundary) and look for one failing to satisfy C.
If they appear to be the same concept, look carefully at G. Are there any specializations of G whose examples have never been filled in? If so, then by all means suggest looking for such concepts' examples before concluding that G and C are really equivalent.
If they are the same, then replace one by a conjecture.
If they are different, make sure that some boundary non-example of C (which is an example of G) is explicitly stored for C.

This rule makes sure that AM is not deluding itself. When AM generalizes Numbers-with-1-divisor into Numbers-which-equal-their-no-of-divisors, it still hasn't gotten past the singleton set {1}. The conjecture in this case would be *"The only number which equals its own number of divisors is 1"*. Typically, when a generalization G of C turns out to be equivalent to C, there is theorem lurking around, of the form "All G's also satisfy this property...", where the property is the "extra" constraint present in C's definition but absent from G's. This rule also was used when AM had just found some examples of Sets. AM almost believed that all Unordered-Structures were also Sets, but the last main clause of the rule had AM notice that Bags is a specialization of Unordered-structures, and that the latter concept had never had any of its examples filled in. As a result, AM printed out this message: "Almost concluded that Unordered-structures are also always Sets. But will wait until examples of Bags are found. Perhaps some Bags will not be Sets." In fact, examples of Bags are soon found, and they aren't sets.

109. When checking a specialization S of concept C,
Specifically test to ensure that S is not equivalent to C.
If they are the same, then replace one by a conjecture.
If they are different, make sure that some boundary example of C (which is not an example of S) is explicitly stored for C.

This rule is similar to the preceding one. If adding a new constraint P to the definition doesn't change the concept C, then there is probably a theorem there of the form "All C's also satisfy constraint P".

110. When checking a specialization S of a specialization X of a concept C,
if there exist other specs. of specs. of C,
then ensure that none of them are the same as S. This is especially worthwhile if the specializing operators in each case were the same but reversed in order.

Thus we can add a constraint to C and collapse the first two arguments, or we can collapse the arguments and add the constraint; either way, we get to the same very specialized new concept. The above rule helps detect those accidental duplicates. E.g., Coalesced-Dom=Ran-Compositions are really the same as Dom=Ran-Coalesced-Compositions, and this rule would suspect that they might be.

111. When checking the Genl or Spec facet entries for concept C,
ensure that C.Genl and C.Spec have no common member Z. If they do, then conjecture that C and Z are actually equivalent.

In fact, this rule actually ensures that Generalizations(C) does not intersect Specializations(C). If it does, a whole 'cycle' of concepts exists which can be collapsed into one single concept. See also rule 114, below.

Any-concept . Genl/Spec . Interest

112. A generalization of X is interesting if all the previously-known boundary non-examples are now boundary examples of the concept.

A check is included here to ensure that the new concept was not simply defined as the closure of the old one.

113. A specialization of X is interesting if all the previously-known boundary examples are now boundary non-examples of the new specialized concept.

A check is included here to ensure that the new concept was not simply defined as the interior of the old one.

114. If C1 is a generalization of C2, which is a generalization of C3,..., which is a generalization of Cj, and it has just been learned that Cj is a generalization of C1,
Then all the concepts C1,...,Cj are equivalent, and can be merged, and the combined concept will be much more interesting than any single one, and the interestingness of the new composite concept increases rapidly with j.

The Lisp code has the new interest value be computed as the maximum value of the old concepts, plus a bonus which increases like the square of j. This is similar to rule number 28. A rule just like the preceding one exists, with 'Specialization' substituted everywhere for 'Generalization'. Thus a closed loop of Spec links constitutes a demonstration that all the concepts in that loop are equivalent. These rules were used more frequently than expected.

Appendix 3.2.6. Heuristics for the View facet of Any-concept

Any-concept . View . Fillin

115. To fill in View facet entries for X,
Find an interesting operation F whose range is X,
and indicate that any member of Domain(F) can be viewed as an X just by running F on it.

While trying to fill in the View facet of Even-nos, AM used this rule. It located the operation Doubling, whose domain is Numbers and whose range is Even-nos. Then the rule created a new entry: "to view any number as if it were an even number, double it". This is a twisted affirmation of the standard correspondence between natural numbers and even natural numbers.

Appendix 3.2.7. Heuristics for the In-dom/ran-of facets of Any-concept

Any-concept . In-dom-of/In-ran-of . Fillin

116. To fill in entries for the In-dom-of facet of concept X,
Ripple down the tree of concepts, starting at Active, to empirically determine which active concepts can be run on X's.

This can usually be decided by inspecting the Domain/range facets of the Active concepts. Occasionally, AM must actually try to run an active on sample X's, to see whether it fails or returns a value.[13]

117. To fill in the In-ran-of facet of concept X,
Ripple down the tree of concepts, starting at Active, to empirically determine which active concepts can be run to yield X's.

This can usually be decided by inspecting the Domain/range facets of the Active concepts. Occasionally, AM inspects known examples of some Active concept, to see if any of the results are X's.

118. While filling in entries for the In-dom-of facet of X,
Look especially carefully for Operations which transform examples and non-examples into each other;
This is even better if the operation pushes boundary exs/non-exs 'across the boundary'.

This was used to note that Insert and Delete had a lot to do with the concept of Singleton.

---

[13] One key feature of Lisp which permits this to be done is the Errorset feature.

Appendix 3.2.8. Heuristics for the Definition facet of Any-concept

Any-concept . Defn . Suggest

119. If there are no known definitions for concept X,
Then it is crucial that AM spend some time looking for such definitions.

This situation might occur if only an Algorithm is present for some concept. In that case, the above rule would suggest a new, high-priority task, and AM would then twist the algorithm into a (probably very inefficient) definition. A much more serious situation would occur if a concept were specified only by its Intuition entries (created, e.g., by modifying another concept's intuitions). In that case, rapidly formulating a precise definition would be a necessity. Of course, this need never arose, since all the intuitions were deleted.

Any-concept . Defn . Check

120. When checking the Definition facet of concept C,
Ensure that each member of C.Exs satisfies all definitions present, and each non-example fails all definitions. If there is one dissenting definition, modify it, and move the offending example to the boundary.

There is little real "checking" that can be done to a definition, aside from internal consistency: If there exist several suposedly-equivalent definitions, then AM can at least ensure they agree on the known examples and non-examples of the concept. If the Intuitions facets were permitted, then each definition could be checked for intuitive appeal.

121. When checking the Definition facet of concept C,
Try to find and eliminate any redundant constraints, try to find and eliminate any circularity, check that any recursion will terminate.

Here are the other few tricks that AM knows for "checking" a definition. For each clause in the rule above, AM has a very limited ability to detect and patch up "bugs" of that sort. Checking that recursion will terminate, for example, is done by examining the argument to the recursive call, and verifying that it contains (at some level before the original argument) an application of a LISP function on Destructive-LISP-functions-list. There is no intelligent inference that is going on here, and for that reason the process is not even mentioned within the body of this document.

Appendix 3.3. Heuristics for dealing with any Active concept

All the rules below are applicable to tasks which involve operations, predicates, relations, functions, etc. In short, they apply to all the concepts AM knows about which involve *doing* something, which involve action.

Active . Fillin

122. If the current task is to fill in examples of the activity F,
One way to get them is to run F on randomly chosen examples of the domain of F.

Thus, to find examples of Equality, AM repeatedly executed Equality.Alg on randomly chosen pairs of objects. AM found examples of Compositions by actually picking a pair of operations at random and trying to compose them. Of course, most such "unmotivated" compositions turned out to be uninteresting.

123. While filling in examples of the activity F, by running F.Algs on random arguments from F.Domain,
It is worth the effort to specifically include extreme or boundary examples of the domain of F, among the arguments on which F.Algs is run.

124. To fill in a Domain entry for active concept F,
Run F on various entities, rippling down the tree of concepts, to determine empirically where F seems to be defined.

This may shock the reader, as it sounds dumb and explosive, but the concepts are arranged in a tree (using Genl links), so the search is really quite fast. Although this rule is rarely used, it always seems to give surprisingly good results.

125. To fill in generalizations of active F,
Consider just extending F, by enlarging its domain. Revise F.Defn as little as possible.

Although Equality is initially only for structures, AM extends it (using the same definition, actually) to a predicate over all pairs of entities.

126. To fill in specializations of active F,
Consider just restricting F, by shrinking its domain. Check F.Defn to see if some optimization is possible.

127. After an algorithm is known for F, if AM wants a better one,
AM is permitted to ask the user to provide a fast but opaque algorithm for F.

This was used a few times, especially for inverse functions. A nontrivial open-ended research problem (⋆)[14] is to collect a body of rules which transform an inefficient algorithm

[14] Following Knuth, we shall reserve a star (⋆) for those problems which are quite difficult, which should take the reader roughly 3 full lifetimes to master. Readers not believing in reincarnation should therefore skip such problems.

into a computationally acceptable one.

128. If the current task is to fill in boundary (non-)examples of the activity F,
One way to get them is to run F on randomly chosen boundary examples and (with proper safeguards) boundary non-examples of the domain of F.

Proper safeguards are required to ensure that F.Algs doesn't loop or cause an error when fed a slightly-wrong (slightly-illegal) argument. In LISP, a timer and an ERRORSET suffice as crude safeguards.

129. If the current task is to fill in (boundary) non-examples of the activity F,
One low-interest way to get them is to run F on randomly chosen examples of its domain, and then replace the value obtained by some other (very similar) value. Also, be sure to check that the resultant i/o pair doesn't accidentally satisfy F.Defn.

The parentheses in the above rule mean that it is really two rules: for *boundary* non-examples, just change the final value slightly. For *typical* non-examples, change the result significantly. If you read the words inside in the parentheses in the IF part, then read the words inside the parentheses in the THEN part as well, *or* omit them in both cases.

Active . Check

130. When checking an algorithm for active F,
run that algorithm and ensure that the input/output satisfy F.Defn.

131. When checking a definition d for active concept F,
Run one of its algorithms and ensure that the input/output satisfy d.

This is the converse of the preceding rule. They simply say that the definition and algorithm facets must be mutually consistent.

132. While checking examples or boundary examples of the active concept F,
Ensure that each input/output pair is consistent with F.Dom/range.

If the domain/range entry is <D1 D2... Dk → R>, and the i/o pair is <d_1 d_2... d_k , r>, then each component d_i of the input must be an example of the corresponding Di, and the output r must be an example of R.

133. When checking examples of the active concept F,
If any argument(s) to F were concepts, tag their In-domain-of facets with 'F'.
If any values produced by F are concepts, tag their In-range-of facets with 'F'.

For example, Restrict(Union) produced Add, at one time in AM's history. Then the above rule caused 'Restrict' to be inserted as a new entry on Union.In-dom-of and also on Add.In-ran-of.

Active . Suggest

**134. If there are no known algorithms for active concept F,
Then AM should spend some time looking for such algorithms.**

This situation might occur if only a Definition is present for some operation. In that case, the above rule would suggest a new, high-priority task, and AM would then twist the definition into a (probably very inefficient) algorithm. The rule below is similar, for the Domain/range facet:

**135. If the Domain/range facet of active concept F is blank,
Then AM should spend some time looking for specifications of F's domain and range.**

**136. If a Domain of active concept F is encountered frequently, either within conjectures or as the domain or range of other operations and predicates,
Then define that Domain as a separate concept, and raise the Worth of F slightly.**

The 'Domain' here refers to the sequence of components, whose cartesian product is what is normally referred to in mathematics as the domain of the operation. This led to the definition of "Anything x Structures", which is the domain of several Insertion and Deletion operations, Membership testing predicates, etc.

137. It is worthwhile to explicitly calculate the value of F for all distinguished (extreme, boundary, interesting) members of and subsets of its domain.

**138. If some domain component of F has a very interesting specialization,
Then consider restricting F (along that component) to that smaller domain.**

Note that these last couple rules deal with the image of interesting domain items. The next rule deals with the inverse image (pre-image) of unusual range items. We saw earlier in this document (Chapter 2) how this rule led to the definition of Prime numbers.

**139. If the range of F contains interesting items, or an interesting specialization,
Then it is worthwhile to consider their inverse image under F.**

**140. When trying to fill in new Algorithms for Active concept F,
Try to transform any conjectures about F into (pieces of) new algorithms.**

This is one place where a sophisticated body of transformation rules might be inserted. Others are working on this problem [Burstall & Darlington 75], and AM only contains a few simple tricks for turning conjectures into procedures. For example, "All primes are odd, except '2'", is transformed into a more eficient search for primes: a separate test for x=2, followed by a search through only Odd-numbers.

**141. After trying in vain to fill in examples of active concept F,
Locate the domain of F, and suggest that AM try to fill in examples for each component of that domain.**

Thus after failing to find examples for Set-union, AM was told to find examples of Sets, because that could have let the previous task succeed. There is no recursion here: after the sets are found, AM will not automatically go back to finding examples of Set-union. In practice, that task was eventually proposed and chosen again, and succeeded this time.

142. After working on an Active concept F,
Give a slight, ephemeral boost to tasks involving Domain(F): give a moderate size boost to each task which asks to fill in examples of that domain/range component, and give a very tiny boost to each other task mentioning such a concept.

This is both a supplement to the more general "focus of attention" rule, and a nontrivial heuristic for finding valuable new tasks. It is the partial converse of rule 14.

Active . Interest

143. An active concept F is interesting if there are other operations with the same domain as F, and if they are (on the average) fairly interesting. If the other operations' domain is only similar, then they must be very interesting and have some valuable conjectures tied to them, if they are to be allowed to push up F's interestingness rating.

The value of having the same domain/range is the ability to compose with them. If the domain/range is only similar, then AM can hope for analogies or for partial compositions.

144. An active concept is interesting if it was recently created.

This is a slight extra boost given to each new operation, predicate, etc. This bonus decays rapidly with time, and thus so will the overall worth of the concept, unless some interesting property is encountered quickly.

145. An active concept is interesting if its domain is very interesting.

An important common case of this rule is when the domain is interesting because all its members are equal to each other. The corresponding statement about *ranges* does exist, but only operations can be said to have a specific range (not, e.g. Predicates). Therefore, the 'range' rule is listed under Operation.Interest, as rule number 165.

Appendix 3.4. Heuristics for dealing with any Predicate

Each of these heuristics can be assumed to be prefaced by a clause of the form "If the current task is to deal with concept X, where X isa Predicate,...". This will be repeated below, for each rule.

Predicate . Fillin

146. If the current task was (Fill-in examples of X),
and X is a predicate,
and more than 100 items are known in the domain of X,
and at least 10 cpu seconds were spent trying to randomly instantiate X,
and the ratio of successes/failures is both >0 and less than .05
Then add the following task to the agenda: (Fill-in generalizations of X), for the following reason:
"X is rarely satisfied; a slightly less restrictive concept might be more interesting".
This reason's rating is computed as three times the ratio of nonexamples/examples found.

This rule says to generalize a predicate if it rarely succeeds (returns T). One use for this was when Equality was found to be quite rare; the resultant generalizations did indeed turn out to be more valuable (numbers). A similar use was found for predicates which tested for identical equality of two angles, of two triangles, and of two lines. Their generalizations were also valuable (congruence, similarity, parallel, equal-measure). Most rules in this appendix are not presented with the same level of detail as the preceding one, as the reader has no doubt observed.

147. To fill in Domain/range entries for predicate P,
P can operate on the domain of any specialization of P,
P can operate on any specialization of the domain of P,
P can operate on some restriction of the domain of any generalization of P,
P may be able to operate on some enlargement of its current domain,
The range of P will necessarily be the doubleton set {T,F},
P is guaranteed return T if any of its specializations do, and F if any of its generalizations do.

This contains a compiled version of what we mean when we say that one predicate is a generalization or specialization of another. Viewed as relations, as subsets of a Cartesian-product of spaces, this notion of general/special is just that of superset/subset. The last line of the rule is meant to indicate that adding new constraints onto P can only make it return True less frequently, while relaxing P's definition can only make it return True more often.

Predicate . Suggest

148. If all the values of Active concept F happen to be Truth-values, and F is not known to be a predicate,
Then conjecture that F is in fact a predicate.

This rule is placed on the Suggest facet because, if placed anywhere else on this concept, it could only be seen as relevant by AM if AM already knew that F were a predeicate. On the other hand, the rule can't be placed, e.g., on Active.Fillin, since just forgetting (deleting)

this "Predicate" concept should be enough to delete all references to predicates anywhere in the system.

Predicate . Interest

149. A predicate P is interesting if its domain is Any-concept (the space of all known concepts). This is especially true if there is a significant positive correlation (theoretical or empirical) between concepts' worths and their P-values.

This very high level heuristic wasn't really used by AM during its runs.

Appendix 3.5. Heuristics for dealing with any Operation

Operation . Fillin

150. To fill in examples of operation F (with domain A and range B),
when many examples α of A are already known,
and F maps some of those examples α into distinguished members (esp: extrema) b of B,
Then (for each such distinguished member "b"$\in$B) study $F^{-1}(b)$ as a new concept. That is, isolate those members of A whose F-value is the unusual item b$\in$B.

This rule says to investigate the inverse image of an unusual item b, under the interesting operation f. When b=2 and f=number-of-divisors-of, this rule leads to the definition of prime numbers. When b=Phi[15] and f=Intersection, the rule led to the discovery of the concept of disjointness of sets.

151. To fill in Domain/range entries for operation F,
F can operate on the domain of any specialization of F,
F can operate on the specialization of the domain of any specialization of F (including F itself),
F can operate on some restriction of the domain of any generalization of F, at least on its current domain and perhaps even on a bigger space,
F may be able to operate on some generalization of (some component(s) of) its current domain,
F can only (and will always) produce values lying in the range of each generalization of F,
F can -- with the proper arguments -- produce values lying in the range of any particular specialization of F.

There are only a few changes between this rule and the corresponding one for Predicates. Recall that Operations can be multi-valued, and those values are not limited to the set {T,F}.

152. To fill in Domain/range entries for operation F, when some exist already,
Take an entry of the form <D1 D2... Dn → R> and see if DixR is meaningful for some i (especially: i=n).
If so, then remove Di from the left side of the entry, and replace R by DixR, and modify the definition of F.

In LISP, "meaningful" is coded as: either DixR is equivalent to an already-known concept, or else it is found in at least two interesting conjectures. This is probably an instance of what McDermott calls natural stupidity[16]. This rule is tagged as being explosive, and is not used very often by AM.

153. To fill in a Range entry for operation F,
Run F on various domain examples, especially boundary examples, to collect examples of the range. Then ripple down the tree of concepts to determine empirically where F seems to be sending its values.

---

[15] the empty set, NIL, {}

[16] See [McDermott 76] for natural stupidity. He criticizes the use of very intelligent-sounding names for otherwise-simple program modules. But consider "Homo sapiens", which means "wise man". Now there's a misleading label.

This may shock the reader, as it sounds dumb and explosive, but the concepts are arranged in a tree (using Genl links), so the search is really quite fast. Although this rule is rarely used, it always seems to give surprisingly good results.

154. If operation F has just been applied, and has yielded a new concept C as its result,
Then carefully examine F.Dom/range to try to find out what C.Isa should be. C.Isa will be all legal entries listed as values of the range of F.

When F=Compose, say AM has just created C=Empty•Insert.[17] What is C? It is a concept, of course, but what else? By examining the Domain/range facet of Compose, AM finds the entry <Active Active → Active>. Aha! So C must be an Active. But AM also finds the entry <Predicate Active → Predicate>. Since "Empty" is a predicate, the final composition C must also be a predicate. So C.Isa would be filled in with "Predicate". AM thus used the above rule to determine that Empty•Insert was a predicate. Even if this rule were excised, AM could still determine that fact, painfully, by noticing that all the values were truth-values.

155. If operation F has just been applied to A1,A2,..., and has yielded a new concept C as its result,
Then add F to C.In-ran-of; add F to the In-dom-of facet of all the Ai's which are concepts; add <A1... → C> to F.Exs.

There is some overlap here with earlier rules, but there is no theoretical or practical difficulty with such redundancy.

156. When filling in examples of operation F, if F takes some existing concepts A1, A2,... and (may) produce a new concept,
Then only consider, as potential A_i's, those concepts which already have some examples. Prefer the A_i's to be interesting, to have a high worth rating, to have some interesting conjectures about them, to have several examples and several non-examples, etc.

The danger here is of, e.g., Composing two operations which turn out to be vacuous, or of Conjoining an empty concept onto another, or of proliferating variants of a boring concept, etc.

Operation . Check

Below are rules used to check existing entries on various facets of operations.

[17] i.e., insert x into a structure S and then see if S is empty. This leads AM to realize that inserting can never result in an empty structure.

157. To check the domain/range entries on the operation F,
IF a domain/range entry has the form (D D D... → R),
and all the D's are equal, and R is a generalization of D (or, with less enthusiasm: if R and D have a significant overlap),
THEN it's worth seeing whether (D D D... → D) is consistent with all known examples of the operation:
If there are no known examples, add a task to the agenda requesting they be filled in.
If there are examples, and (D D D... → D) is consistent, add it to the Domain/range facet of this operation.
If there are some contradicting examples, create a new concept which is defined as this operation restricted to (D D D... → D).

When AM restricts Bag-union to numbers (bags of T's), the new operation has a Domain/range entry of the form (Numbers Numbers → Bag). The above rule has AM investigate whether the range specification mightn't also be narrowed down to Number. In this case it is a great help. The rule often fails, of course: the sum of two primes is rarely a prime, the cross-product of two lists-of-atoms is not a list-of-atoms, etc. Since this rule is almost instantaneous to execute, it's cost-effective overall.

158. When checking the domain/range entries on the operation F,
IF a domain/range entry has the form (D D D... → R),
and all the D's are equal, and R is a specialization of D,
THEN it's worth inserting (D D D... → D) as a new entry on F.Dom/ran, even though that is redundant.

This shows that symmetry and aesthetics are sometimes preferable to absolute optimization. That's why we program in Lisp, instead of machine language. On the other hand, this rule wasn't really that useful to AM. Now, by analogy,...?

159. When checking the Domain/range entries for operation F,
Ensure that the boundary items in the range can actually be reached by F. If not, see whether the range is really just some known specialization of F.

This rule is a typical checking rule. Note that it is active, not passive: it might alter the Domain/range facet of F, it it finds an error there.

160. When checking examples of the operation F, for each such example,
If the value returned by F is a concept C, add 'F' to C.In-range-of.

Operation . Suggest

161. Whenever the domain of operation F has changed,
check whether the range has also changed. Often the range will change analogously to the domain, where the operation itself is the Analogy.

162. After working on Operation F,
Give a slight, ephemeral boost to tasks involving Range(F).

This will be a moderate size boost for each task which asks to fill in examples of that range concept, and a very tiny boost for each other task mentioning such a concept. This is both a supplement to the more general "focus of attention" rule, and a nontrivial heuristic for finding valuable new tasks. It is an extension of rule number 142, and a partial converse to rule 14.

Operation . Interest

163. An operation F is interesting if there are other operations with the same domain and range, and if they are (on the average) fairly interesting.

164. An operation F is interesting if it is the first operation connecting its domain concept to its range concept, and if these domain/range components are themselves valuable concepts, and there is no analogy between them, and there are some interesting conjectures involving the domain of F.

The above two rules say that F can be valuable because it's similar to other, already-liked operations, or because it is totally different from any known operation. Although these two criteria are nonintersecting, their union represents only a small fraction of the operations that get created: typically, *neither* rule will trigger.

165. An operation F is interesting if its range is very interesting.

Range here refers to the concept in which all resuks of F must lie. It is the R in the domain/range facet entry <D → R> for concept F. The corresponding rule for 'domains' is applicable to any Active, not just to Operations, hence is listed under Active.Interest, as rule number 145.

166. An operation F is interesting if the values of F satisfy some unusual property which is not (in general) satisfied by the arguments to F.

Thus doubling is interesting because it always returns an even number. This is one case where the interesting property can be deduced trivially just by looking at the domain and range of the operation: Numbers→Even-nos.

167. An operation is interesting if its values are interesting.

This can mean that each value is interesting (e.g., Compose is well-received because it produces many new, valuable concepts as its values). Or, it can mean that the operations' values, gathered together into one big set, are interesting as a set. Unlike the preceding rule, this one has no mention whatsoever of the domain items, the arguments to the operation. This rule was used to good advantage frequently by AM. For example, Factorings of numbers are interesting because (using rule 232) for all x, Factorings(x) is interesting in exactly the same way. Namely, Factorings(x), viewed as a set, always contains precisely one item which has a certain interesting property (see rule 233). Namely, all its members are primes (see rule 232 again). This explains one way in which AM noticed that all numbers seem to factor uniquely into primes.

168. An operation is interesting if its values are interesting, ignoring the images of boundary items from the domain.

That is, if the image of the domain – minus its boundary – is interesting.

169. An operation is interesting if its values on the boundary items from the domain are very interesting. Ignore the non-boundary parts of the domain.

That is, if the image of the boundary of the domain is interesting.

170. An operation is interesting if it leaves intact any interesting properties of its argument(s). This is even better if it eliminates some undesirable properties, or adds some new, desirable ones.

Thus a new, specialized kind of Insertion operation is interesting if, even though it stuffs more items into a structure, the nice properties of the structure remain. The operation "Merge" is interesting for this very reason: it inserts items into an alphabetized list, yet it doesn't destroy that interesting property of the list.

171. An operation is interesting if its domain and range are equal. If there is more than one domain component, then at least one of them should equal the range. The more components which are equal to the range, the better.

Thus "Insertion" qualifies here, since its domain/range entry is <Anything Structures → Structures>. But "Union" is even better, since both domain components equal the range, namely Structures.

172. An operation is mildly interesting if its range is related somehow (e.g. specialization of) to one or more components of its range. The more the better.

A weakened form of the preceding rule.

**173. If the result of applying operation F is a new concept C,
Then the interestingness of F is weakly tied to that of C.**

If the new concept C becomes very valuable, then F will rise slightly in interest. If C is so bad it gets forgotten, F will not be regarded quite as highly. When Canonize scores big its first time used, it rises in interest. This caused AM to form poorly-motivated canonizations, which led to dismal results, which gradually lowered the rating of Canonize to where it was originally.

Appendix 3.6. Heuristics for dealing with any Composition

Composition . Fillin

174. To fill in algorithms for operation F, where F is a composition GoH,
One algorithm is: apply H and then apply G to the result.

Of course this rule is not much more than the definition of what it means to compose two operations.

175. To fill in Domain/range entries for operation F, where F is a composition GoH,
Tentatively assume that the domain is Domain(H), and range is Range(G). More precisely, the domain will be the result of substituting Domain(H) for Range(H) wherever Range(H) appears (or: just once) in Domain(G).

Thus for F=DividesoCount, where Divides:<Number,Number → {T,F}>, and Count:<Bag → Number>, the above rule would say that the domain/range entries for F are gotten by substituting 'Bag' for 'Number' once or twice in Domain(Divides). The possible entries for F.Dom/range are thus: <Bag,Bag → {T,F}>, <Number,Bag → {T,F}>, and <Bag,Number → {T,F}>.

176. To fill in Domain/range entries for operation F, where F is a composition GoH, But Range(H) does not occur as a component of Domain(G),
The range of F is still Range(G), but the domain of F is computed as follows: Ascertain the component X of Domain(G) having the biggest (fractional) overlap with Range(H). Then substitute Domain(H) for X in Domain(G). The result is the value to be used for Domain(F).

This rule is a second-order correction to the previous one. If there is no absolute equality, then a large intersection will suffice. Notice that F may no longer be defined on all of its domain, even if G and H are. If identical equality is taken as the maximum possible overlap betwen two concepts, then this rule can be used to replace the preceding one completely.

177. When trying to fill in the Isa entries for a composition F=GoH,
Examine G.Isa and H.Isa, and especially their intersection. Some of those concepts may also claim F as an example. Run their definition facet to see.

To see how this is encoded into LISP, turn to page 219.

178. When trying to fill in the Genl or Spec entries for a composition F=GoH,
Examine the corresponding facet on G and on H.

This rule is similar to the preceding one, but wasn't as useful or as reliable.

179. A satisfactory initial guess at the Worth value of composition F=GoH is the root-sum-of-squares of G.Worth and H.Worth.

180. To fill in examples of F, where F=GoH, and both G and H are time-consuming, but where many examples of both G and H exist,
Seek an example x→y of H, and an example y→z of G, and then return x→z as a probable example of F.

Above, 'seek' is done in a tight, efficent manner. The examples are H are hashed into an array, based on the values y of each one. Then the arguments of the examples of G are hashed to see if they occur in this array. Those that do will generate an example of the new composition.

181. To fill in examples of F, where F=GoH, and G is timeconsuming, but many exemples of G exist, and it is not known whether H is time-consuming or not,
Spend a moment trying to access or trivially fill in examples of H.
If this succeeds, apply the preceding rule.
If this fails, then formally propose that AM fill in examples of H, with priority equal to that of the current task, for these two reasons: (i) if examples of H existed, then AM could have used the heuristic preceding this one, to fill in examples of F, and (ii) it is dangerous to spend a long time dealing with GoH before any examples at all of H are known.

This rule is of course tightly coupled to the preceding one. The same rule exists for the case where just H is time-consuming, instead of G.

182. When trying to fill in Conjecs about a composition F=GoH,
Consider that F may be the same as G (or the same as H).

It was somewhat depressing that this 'stupid' heuristic turned out to be valuable, perhaps even necessary for AM's top performance.

Composition . Check

183. Check that FoG is really not the same as F, or the same as G. Spend some time checking whether FoG is equivalent to any already-known active concept.

This happens often enough to make it worth stating explicitly. Often, for example, F will not even bother looking at the result of G! For example,
Proj2oSquare(x,y) = Proj2(Square(x),y) = y = Proj2(x,y).

184. When checking the Algorithms entries for a composition F=GoH,
If range(H) is not wholly contained in the domain of G,
then the algorithm must contain a "legality" check, ensuring that H(x) is a valid member of the domain of G.

Composition . Suggest

185. Given an interesting operation $F:A^n \rightarrow A$,
consider composing F with itself.

This may result in more than one new operation. From F=division, for example, we get the two operations (x/y)/z and x/(y/z). AM quickly realizes that such variants are really equivalent, and (if prodded) eventually realizes that F(F(x,y),z)=F(x,F(y,z)) is a common situation (which we call associativity of F).

186. If the newly-formed domain of the composition F=GoH contains more than one occurrence of the concept D, and this isn't true of G or H,
Then consider creating a new operation, a specialization of F, by Coalescing the domain/range of F, by eliminating one of the D components.

Thus when InsertoDelete is formed, the old Domain/range entries were both of the form <Anything Structure → Structure>. The newly-created entry for InsertoDelete was <Anything Anything Structure → Structure>; i.e., take x, delete it from S, then insert y into S. The above rule had AM turn this into a new operation, with domain/range <Anything Structure → Structure>, which deleted x from S and the inserted the very same x back into S.

Composition . Interest

187. A composition F=GoH is interesting if G and H are very interesting.

188. A composition F=GoH is interesting if F has an interesting property not possessed by either G or H.

189. A composition F=GoH is interesting if F has most of the interesting properties which are possessed by either G or H. This is slightly reduced if both G and H possess the property.

190. A composition F=GoH is interesting if F lacks any undesirable properties true of G or H. This is greatly increased if both G and H possess the bad property, unless G and H are very closely related to each other (e.g., H=G, or $H=G^{-1}$).

The numeric impact of each of these rules was guessed at initially, and has never needed tuning. Here is an area where experimentation might prove interesting.

191. A composition F=GoH is interesting if F maps interesting subsets of domain(H) into interesting subsets of range(G).
F is to be judged even more interesting if the image was not thought to be interesting until after it was explicitly isolated and studied because of part 1 of this very rule.

Here, an "interesting" subset of domain(H) is one so judged by Interests(domain(H)). A completely different set of criteria will be used to judge the interestingness of the resultant

image under F. Namely, for that purpose, AM will ask for range(G).Interest, and ripple outwards to look for related interest features.

192. A composition F=GoH is interesting if F^{-1} maps interesting subsets of range(G) into interesting subsets of domain(F).
This is even better if the preimage wasn't hitherto realized as interesting.

This is the converse of the preceding rule. Again, "interesting" is judged by two different sets of criteria.

193. A composition F=GoH is interesting if F maps interesting elements of domain(H) into interesting subsets of range(G).

194. A composition F=GoH is interesting if F^{-1} maps interesting elements of range(G) into interesting subsets of domain(F).
This is even better if the subset is only now seen to be interesting.

This is the analogue of an earlier rule, but for individual items rather than for whole subsets of the domain and range of F.

195. A composition F=GoH is interesting if range(H) is equal to, not just intersects, one component of domain(G).

196. A composition F=GoH is mildly interesting if range(H) is a specialization of one component of domain(G).

This is a weakened version of the preceding feature. Such a composition is interesting because it is guaranteed to always be applicable. If Range(H) merely intersects a domain component of G, then there must be an extra check, after computing H(x), to ensure it lies within the legal domain of G, before trying to run G on that new entity H(x).

197. A composition F=GoH is more interesting if range(G) is equal to a domain component of H.

This is over and above the slight boost given to the composition because it is an *operation* whose domain and range coincide (see rule 171).

Appendix 3.7. Heuristics for dealing with any Insertions

Insertion . Check

198. When checking an example of any kind of insertion of x into S,
Ensure that x is a member of S.

The only types of insertions known to AM are *unconditional* insertions, so this rule is valid. It is useful for ensuring that a particular new operation really is an insertion-operation after all!

Appendix 3.6. Heuristics for dealing with the operation Coalesce

Coalesce . Fillin

199. When coalescing F(a,b,c,...), whose domain/range is <A B C... → R>,
A good choice of two domain components to coalesce is a pair of identically equal ones. Barring that, choose a pair related by specialization (eliminate the more general one). Barring that, choose a pair with a common specialization S, and replace both by S.

Thus to coalesce the operation "Insert•Delete" [which takes two items and a structure, deletes the first argument from the structure and then inserts the second argument], AM examines its Domain/range entry: <Anything Anything Structure → Structure>. Although it would be legal to collapse the second and third arguments, the above rule says it makes more sense in general to collapse the first and second. In fact, in that case, AM gets an operation which tells it something about multiple elements structures.

200. When filling in Algorithms for a coalesced version G of active concept F,
One natural algorithm is simply to call on F.Alg, with two arguments the same.

Of course the two identical arguments are those which have been decided to be merged. This will be decided before the definition and algorithm facets are filled in. Thus a natural algorithm for Square is to call on TIMES.Alg(x,x). The following rule is similar:

201. When filling in Definitions for a coalesced version G of active concept F,
One natural Definition is simply to call on F.Defn, with two arguments the same.

202. When filling in the Worth of a new coalesced version of F,
A suitable value is 0.9x(Worth of F) + 0.1x(Worth of Coalesce).

This is a compromise between (i) the knowledge that the new operation will probably be less interesting than F, and (ii) the knowledge that it may lead to even more valuable new concepts (e.g., *its* inverse may be more interesting than F's). The formula also incorporates a small factor which is based on the overall value of coalescings which AM has done so far in the run.

Coalesce . Check

203. If G and H are each two coalescings away from F, for any F,
Then check that G and H aren't really the same, by writing their definitions out in terms of F.Defn.

Thus if R(a,b,c) is really F(a,b,a,c), and S(a,b,c) is really F(a,b,c,c), and R and S get coalesced again, into G(a,b) whch is R(a,b,a) and into H(a,b) which is S(a,b,a), then both G and H are really F(a,b,a,a). The order of coalescing is unimportant. This is a boost to the more general impetus for checking this sort of thing, rule 110. This rule is faster, containing a special-purpose program for untangling argument-calls rapidly. If the concept of Coalesce is

excised from the system, one can easily imagine it being re-derived by a more general 'coincidence' strategy, but how will these specific, high-powered, tightly-coded heuristics ever get discovered and tacked onto the Coalesce concept? This is an instance of the main meta-level research problem proposed earlier in the thesis (Chapter 7).

Coalesce . Suggest

204. If a newly-interesting active concept F(x,y) takes a pair of N's as arguments,
Then create a new concept, a specialization of F, called F-itself, taking just one N as argument, defined as F(x,x), with initial worth Worth(F).
If AM has never coalesced F before, this gets a slight bonus value.
If AM has coalesced F before, say into S, then modify this suggestion's value according to the current worth of S.
The lower the system's interest-threshhold is, the more attactive this suggestion becomes.

AM used this rule to coalesce many active concepts. Times(x,x) is what we know as squaring; Equality(x,x) turned out to be the constant predicate True; Intersect(x,x) turned out to be the identity operator; Compose(f,f) was an interesting "iteration" operator[18]; etc. This rule is really a bundle of little meta-rules modifying one suggestion: the suggestion that AM coalesce F. The very last part of the above rule indicates that if the system is desparate, this is the least distasteful way to "take a chance" on a high-payoff high-risk course of action. It is more sane than, e.g., randomly composing two operations until a nice new one is created.

205. If concept F takes only one argument,
Then it is not worthwhile to try to coalesce it.

This rule was of little help cpu-timewise, since even if AM tried to coalesce such an active concept,it would fail almost instantaneously. The rule did help make AM appear smarter, however.

Appendix 3.9. Heuristics for dealing with the operation Canonize

Canonize . Fillin

206. If the task is to Canonize predicates P1 and P2 (over AxA)[19], and the difference between a definition of P1 and definition of P2 is just that P2 performs some extra check that P1 doesn't,
Then F should convert any a∈A into a member of A which automatically satisfies that extra constraint.

Thus when P1=Same-length, P2=Equality, A=Lists, the extra constraint that P2 satisfies is just that it recurs in the CAR direction: the CARs of the two arguments must also satisfy

---

[18] e.g., Compose(Compose,Compose) is an operator which takes 3 operations f,g,h and forms "f ∘ g ∘ h"; i.e., their joint composition.

[19] That is, find a function F such that P1(x,y) iff P2(F(x),F(y)).

P2. P1 doesn't have such a requirement. The above rule then has AM seek out a way to guarantee that the CARs will always satisfy Equality. A special hand-crafted piece of knowledge tells AM that since "T=T" is an example of Equality, one solution is for all the CARs to be the atom T. Then the operation F must contain a procedure for converting each ember of a structure to the atom "T". Thus (A C {Z A B} Q Q) would be converted to (T T T T T). This rule is a specialized, "compiled" version of the idea expressed in rule number 13.

207. If the task is to Canonize P1 and P2 over AxA, trying to synthesize F, where A=Structure,
Then perhaps there is a distinguished type of structure B which the argument to F should always be converted into. In that case, consider P1 and P2 as two predicates over BxB.

This special-purpose rule is used to guide a series of experiments, to determine whether P1 is affected by adding multiple copies of existing elements into its arguments, and whether its value is affected by rearranging some of its arguments' elements. In the case of P1=Same-size, the answers are: multiple elements do make a difference, but rearrangement doesn't. So the canonical type of structure for F=Size must be one which is Mult-eles-allowed and also one which is Unordered. Namely, a Bag. Thus F is modified so that it first converts its argument to a Bag. Then Equality and Same-size are viewed as taking a pair of Bags, and Size is viewed as taking one Bag and giving back a canonical bag.

208. After F is created from P1 and P2, as Canonize(P1,P2),
an acceptable value for the worth of F is the maximum of the worths of P1 and P2.

In the actual Lisp code, an extra small term is added which takes into account the overall value of all the Canonizations which AM has recently produced.

209. IF the current task has just created a canonical specialization B of concept A, with respect to predicates P1 and P2, [i.e., two members of B satisfy P1 iff they satisfy P2],
THEN add the following entry to the Analogies facet of B:

<A	P1	Operations-on-and-into(A)>
<B	P2	Those operations restricted to B's>

This rather incoherent rule says that it's worth taking the trouble to study the behavior of each operation when it is restricted to working on standard or "canonical" items. Moreover, some of the old relationships may carry over – or at least have analogues – in this restricted world. When numbers are discovered as canonical bags, all the bag operations are restricted to work on only canonical bags, and they magically turn into what we know and love as numeric operations. Many of the old bag-theoretic relationships have numeric analogues. Thus we knew that the bag-difference of x and x was the empty bag; this is still true for x a canonical bag, but we would word it as "x minus x is zero". This is because the restriction of Bag-difference to canonical bags (bags of T's) is precisely the operation we call subtraction.

210. When Canonize works on P1, P2 (two predicates), and produces an operation, F,
Both predicates must share a common Domain, of the form AxA for some concept A, and the new operation F can have <A → A> as one of its Domain/range entries.
If a canonical specialization (say 'B') of A is defined, then the domain/range of F can actually be tightened to <A → B>, and it is also worth explicitly storing the redundant entry <B → B>.

211. In the same situation as the last rule,
One conjecture is that P1 and P2 are equal, when restricted to working on pairs of B's [i.e., pairs of Canonical A's, A's which are in F(A), range items for F, items x which are the image F(a) of some a∈A].

After canonizing Equal and Same-size into the new operation Length, AM conjectures that two canonical bags are equal iff they have the same size.

Canonize . Suggest

212. When Canonize works on P1, P2, both predicates over AxA, and produces an operation F:A→A,
It is worth defining and studying the image F(A); i.e., the totality of A's which are canonical, already in standard form. When this new concept Canonical-A is defined, suggest the task "Fillin Dom/range entries for Canonical-A".

Thus AM studied Canonical-Bags, which turned out to be isomorphic to the natural numbers. What we've called 'Canonical-A' in this rule, we've referred to simply as 'B' in earlier Canonizing rules.

213. If P1 is a very interesting predicate over AxA, for some interesting concept A,
Then look over P1.Spec for some other predicate P2 which is also over AxA, and which has some interesting properties P1 lacks. For each such predicate P2, consider applying Canonize(P1,P2).

214. After producing F as Canonize(P1,P2) [both predicates over AxA], and after defining Canonical-A,
It is worth restricting operations in In-dom-of(A) to Canonical-A. Some new properties may emerge.

Thus after defining Canonical-Bags, AM looked at Bags.In-dom-of. In that list was the operation "Bag-union". So AM considered the restriction of Bag-union to Canonical-bags. Instead of Bag-union mapping two bags into a new bag, this new operation took two canonical-bags and mapped them into a new bag. AM later noticed that this new bag was itself always canonical. Thus was born the operation we call "Addition".

215. After Canonical-A is produced,
It is marginally worth trying to restrict operations in In-range-of(A) to map into Canonical-A.

This gives an added boost to picking Union to restrict, since it is in both Bags.In-dom-of and Bags.In-ran-of. This rule is much harder to implement, since it demands that the range be restricted. There are just a few special-purpose tricks AM knows to do this. Restricting the domain is, by comparison, much cleaner. AM simply creates a new concept with the same definition, but with a more restricted domain/range facet. For restricting the range, AM must insert into the definition a check to ensure that the final result is inside Canonical-A, not just in A. This leads to a very inefficent definition.

216. After Canonical-A is produced,
It is worthwhile to consider filling in examples (especially boundary) of that new concept.

This is above and beyond the slight push which rule 12 gives that task.

Appendix 3.10. Heuristics for dealing with the operation Substitute

Note that substitution operations are produced via the initial concepts called Parallel-replace and Parallel-replace2. The following rules are tacked on there.

Parallel-replace . Suggest

217. If two different variables are used to represent the same entity,[20] then substitute one for the other.
This is very important if the two occurrences are within the same entry on some facet of a single concept, and less so otherwise.
The dominant variable should be the one typed out previously to the user; barring that, the older usage; barring that, the one closest to the letter "a" in the alphabet.

This heuristic was used less often – and proved less impressive – than was originally anticipated by the author. Since most concepts were begotten from older ones, they always assumed the same variable namings, hence there were very few mismatches. A special test was needed to explicitly check for "x=y" occurring as a conjunct somewhere, in which case we removed it and y substituted for x throughout the conjunction.

218. If two expressions (especially: two conjectures) are structurally similar, and appear to differ by a certain substitution,
Then if the substitution is permissable we have just arrived at the same expression in various ways, and tag it as such,
But if the substitution is not seen to be tautologous, then a new analogy is born. Associate the constituent parts of both expressions. This is made interesting if there are several concepts involved which are assigned new analogues.

The similar statements of the associativity of Add and Times led to this rule's identifying them as analogous. If AM had been more sophisticated, it might have eventually formulated some abstract algebra concepts like "semigroup" from such analogies.

Appendix 3.11. Heuristics for dealing with the operation Restrict

Restrict . Fillin

219. When filling in definitions (algorithms) for a restriction R of the active concept F,
One entry can simply be a call on F.Defn (F.Algs).

Thus one definition of Addition will always be as a call on the old, general operation 'Bag-union.' Of course one major reason for restricting the domain/range of an activity is that it can be performed using a faster algorithm! So the above rule was used frequently if not dramatically.

---

[20] When we say that x and y represent the same entity, what we really mean is that they have the same domain of identity (e.g., ∀x(Bags)) and they are equally free (there is a precise logical definition of all this, but there is little point to presenting it here.)

220. When creating a restriction R of the active concept F,
Note that R.Genl should contain F, and that F.Spec should contain R.

221. When creating in a restriction R of the active concept F, by restricting the domain or range to some specialization S of its previous value C,
A viable initial guess for R.Worth is F.Worth, augmented by the difference in worth between S and C. Hopefully, S.Worth is bigger than C.Worth!

Appendix 3.12. Heuristics for dealing with the operation Invert

Invert . Fillin

222. When filling in definitions for an inverse F^{-1} of the active concept F,
One "Sufficent Defn" entry can simply be a blind search through the examples of F.

If we already knew that 4→16 is an example of Square, then AM can use the above rule to quickly notice that Square-Inverse.Defn(16,4) is true. This is almost the "essence" of inverting an operation, of course.

Invert . Suggest

223. After creating an inverted form F^{-1} of some operation F,
If the only definition and algorithm entries are the "obvious" inefficient ones,
Then consider the task: "Fill in algorithms for F^{-1}", because the old blind search is just too awful to tolerate.

Appendix 3.13. Heuristics for dealing with Logical combinations

Eventually, there may be separate concepts for each kind of logical connective. For the moment, all Boolean operators are lumped together here. Their definition is too trivial for a 'Fillin' heuristic to be useful, and even 'Check' heuristics are almost pointless.

Logical-combine . Check

224. The user may sometimes indicate 'Conjunction' when he really means 'Repeating'.

Logical-combine . Suggest

225. If there is something interesting to say about entities satisfying the disjunction (conjunction) of two concepts' definitions,
Then consider creating a new concept defined as that logical combination of the two concepts' definitions.

226. Given an implication,
Try to weaken the left side as much as possible without destroying the validity of the whole implication. Similarly, try to strengthen the right side of the implication.

Logical-combine . Interest

227. A disjunction (conjunction) is interesting if there is a conjecture which is very interesting yet which cannot be made about any one disjunct (conjunct).

In other words, their logical combination implies more than any consituent.

228. An implication is interesting if the right side is more interesting than the left side.

229. An implication is interesting if the right side is interesting yet unexpected based only on assuming the left side.

Appendix 3.14. Heuristics for dealing with Structures

Structure . Fillin

230. To fill in examples of a kind of structure S,
Start with an empty S, pluck any other member of Examples(Structure), and transfer members one at a time from the random structure into the embryonic S. Finally, check that the resultant S really satisfies S.Defn.

This is useful, e.g., to convert examples of lists into examples of sets.

231. To fill in specializations of a kind of structure,
add a new constraint that each member must satisfy, or a constraint on all pairs of members, or a constraint on all pairs of distinct members, or a constraint which the structure as a whole must satisfy. Such a constraint is often merely a stipulation of being an example of an X, for some interesting concept X.

Thus AM might specialize Bags into Bags-of-primes, or into Bags-of-distinct-primes, or into Bags-containing-a-prime.

Structure . Interest

232. Structure S is mildly interesting if all members of S satisfy the interesting predicate P, or (equivalently) if they are all accidentally examples of the interesting concept C, or (similarly) if all pairs of members of S satisfy the interesting binary predicate P, etc.
Also: a KIND of structure is interesting if it appears that each instance of such a structure satisfies the above condition (for a fixed P or C).

Thus a singleton is interesting because all pairs of members satisfy Equal. The concept "Singletons" is interesting because each singleton is mildly interesting in just that same way. Similarly, AM defines the concept of a bag containing only primes, because the above rule says it might be interesting.

233. A structure is mildly interesting if one member is very interesting. Even better: exactly one member.
Also: a KIND of structure is interesting if each instance satisfies the above condition in the same way.

Thus the values of ADD^{-1} are interesting because they always contain precisely one bag which is a Singleton.

Appendix 3.15. Heuristics for dealing with Ordered-structures

Ordered-struc . Fillin

234. To fill in some new examples of the ordered structure S, when some already exist,
Pick an existing example and randomly permute its members.

235. To fill in specializations of a kind of ordered structure,
add a new constraint that each pair of adjacent members must satisfy, or a constraint on all ordered pairs of members in the order they appear in the structure. Such a constraint is often merely a stipulation of being an example of an X, for some interesting concept X.

Thus Lists-of-numbers might be specialized into Sorted-lists-of-numbers, assuming AM has discovered "≤" and assuming it is chosen as the 'constraint' relationship between adjacent members of the list.

Ordered-struc . Check

236. If the structure is to be accessed sequentially until some condition is met, and if the precise ordering is superfluous,
Then keep the structure ordered by frequency of use, the most useful element first.

This is a simple data-structure management trick. If you have several rules to use in a certain situation, and rule R is one which usually succeeds, then put R first in the list of rules to try. Similarly, in a pattern-matcher, try first the test most likely to detect non-matching arguments.

237. If structure S is always to be maintained in alphanumeric order,
Then AM can[21] actually maintain it as an unordered structure, if desired.

Luckily this heavily implementation-dependent rule was never needed by AM.

Ordered-struc . Interest

238. An ordered structure S is interesting if each adjacent pair of members of S satisfies predicate P (for some rare, interesting P).

When AM discovers the relation "≤", it immediately thinks that any *sorted* list of numbers is more interesting, due to the above rule.

Appendix 3.16. Heuristics for dealing with Unordered-structures

Unord-struc . Check

239. To check an example of an unordered structure,
Ensure that it is in alphanumerically-sorted order. If not, then SORT it.

All unordered objects are maintained in lexicographic order, so that two of them can be tested for equality using the LISP function EQUAL. Because of this convention, any two structures can therefore be tested for equality using this fast list-structure comparator.

Appendix 3.17. Heuristics for dealing with Multiple-eles-structures

Mult-ele-struc . Fillin

240. To fill in some new examples of the structure S, where S is a structure admitting multiple occurrences of the same element, when some examples already exist,
Pick an existing example and randomly change the multiplicity with which various members occur within the structure.

Appendix 3.18. Heuristics for dealing with Sets

Sets . Suggest

241. If P is a very interesting predicate over X,
Then study these two sets: the members of X for which P holds, and the ones for which P fails.

---

[21] due to the current LISP implementation of data-structures

While we humans know that this partitions X into two pieces, AM is never explicitly told this. It would occasionally be surprised to discover that the union of two such complements "accidentally" coincided with X. Incidentally, this rule is really the key linkage between predicates and sets; it is close to the entry on Set.View which tells how to view any predicate as a set.

Sets . Interest

242. A set S is interesting if, for some interesting predicate P, whose domain is X, S accidentally appears to be related strongly to {x∈X | P(x)}, i.e., to those members of X satisfying P.

To the surprise of the author, this rule never found application in any of AM's runs.

Appendix 4. Maximally-Divisible Numbers

The honor of your machine is preserved.

Erdos[1]

This appendix discusses a discovery motivated by AM: a new little bit of mathematics that was discovered. It is presented as if it were a math journal article, in a fairly formal way, almost unmotivated.

After the concept was discovered, the author learned that Ramanujan, a self-taught Indian mathematician, had worked on similar issues early in this century. The final subsection contains a relaxed summary of what AM did, what the author did, and what Ramanujan did.

Appendix 4.1. A Meaningful Question

We begin by asking the question, "What is the *converse* concept to prime numbers?" If we define "primeness" to mean that a natural number has as few divisors as possible (namely, just two of them: 1 and itself), then the converse kind of number would be one which had an abnormally *large* number of divisors.

One could consider the following set M of maximally-divisible numbers:

$$M = \{x \in N^* \mid (\forall y < x)\ (\ d(y) < d(x)\ )\ \}$$

where d(n) is the number of divisors of n,[2] N^* is the set of positive integers, and the vertical bar, "|" is read 'such that'.

In words, this says that M is the set of all positive integers satisfying the property that every smaller number has fewer divisors. That is, we throw into the set M a number x if (and only if) it has more divisors than any smaller number. So 1 gets thrown in (the smallest number with 1 divisor), 2 (having 2 divisors), 4 (3 divisors, namely 1, 2, and 4), 6 (4 divisors), 12 (6 divisors), etc. Another way to specify M is as the set containing (for all n) the smallest number having at least n divisors. Notice that we are *not* going to include "the smallest number with *precisely* 5 divisors", since this number (which happens to be 2^4 or 16) is bigger than 12 (which has 6 divisors). So no number in M has precisely five divisors.

---

[1] Remarked by Paul Erdos, after examining some of this appendix's material.

[2] E.g., d(12) = ||{1,2,3,4,6,12}|| = 6.

One of the questions at the heart of our study is "Given d, what is the smallest number with at least d divisors?"

How can we even start on this question? The most powerful tool we have is the following combinatorially-proved theorem:

(T1) If we write n as $2^{a_1}3^{a_2}\cdots p_k^{a_k}$, then $d(n) = (a_1+1)(a_2+1)\cdots(a_k+1)$.

Our central question could be answered if we could somehow invert this formula into one which expressed n as a function of d, and then found the minima of that function n(d). Coupled with the knowledge that each number can be factored uniquely into prime factors, T1 provides a closed-form way of manipulating d(n). That is, n is really a function of the sequence of exponents when written as $2^{a_1}3^{a_2}\ldots$, so we can consider $n = n(a_1, a_2,\ldots)$. Then T1 is really a way of expressing $d(n) = d(a_1, a_2,\ldots)$.

Appendix 4.2. Special Case: $n = 2^a3^b$

Let's consider a special case. We'll restrict our attention to numbers n which are of the form 2^a3^b. So T1 says that d(n)=(a+1)(b+1). Consider fixing d, and asking how n varies with a and b. Notice that we are now saying that (a+1)(b+1)=d=constant. So we can say that (b+1)=d/(a+1), so b=(d/(a+1))-1. So our number n is really $2^a3^{[(d/(a+1))-1]}$. Aha! This is an expression for n simply as a function of a. We can use calculus to find the minima of this function. That will correspond to the optimal exponent a, from which we can compute the optimal b.

$dn/da = 2^a(3^b(-d/(a+1)^2)\log(3)) + 3^{(d/(a+1))-1}(2^a\log(2))$

$= [(a+1)\log(2) - (b+1)\log(3)](n/(a+1))$.

This is zero when (a+1)log(2) = (b+1)log(3).

So we have two equations now:

(a+1) = (b+1)log(3)/log(2)

(a+1) = d/(b+1)

Together they say that (b+1)log(3)/log(2) = d/(b+1), from which we derive $(b+1)^2$ = log(2)d/log(3). Substituting this back in, we also get that $(a+1)^2$ = log(3)d/log(2).

If real-valued exponents were allowed, our optimal n(d) would be:

$2^{SQRT[d\log(3) / \log(2)]} \cdot 3^{SQRT[d\log(2) / \log(3)]}$.

Three observations we can make from intuition – and justify from reality – are (i) this optimal real value is better than (i.e., ≤) any integral n (divisible only by 2 and 3) with at least d divisors, (ii) the ideal n is very close to the best such integral n, (iii) the best such integral n will have exponents a and b which are close to our theoretical real-valued "ideal" a and b.

For example, if we choose to ask for a number with at least 8 divisors, our theoretical values for a and b are about 2.6 and 1.2; the ideal n is then about 23. In actuality, the first number with 8 or more divisors is 24, and it is factored into $2^3 3^1$ (i.e., the best integral values for a and b are 3 and 1, respectively).

Appendix 4.3. Special Case: $n = 2^a3^b5^c$

Let's consider a second special case. We'll restrict our attention to numbers n which are of the form $2^a3^b5^c$. So T1 says that d(n)=(a+1)(b+1)(c+1). Consider fixing d, and asking how n varies with a, b, and c. Notice that we are now saying that (a+1)(b+1)(c+1)=d=constant. So we can say that (c+1)=d/(a+1)(b+1), so c=(d/(a+1)(b+1))-1. So our number n is really $2^a3^b5^{[(d/(a+1)(b+1))-1]}$.

Viewing c as a function of a and b, we can compute the differential

$dc = d(d/(a+1)(b+1))$

$= d*[d(1/(a+1)(b+1))]$

$= d*[(1/(a+1))(-1/(b+1)^2)db + (1/(b+1))(-1/(a+1)^2)da]$

$= (-(c+1)/(b+1))db + (-(c+1)/(a+1))da$

We want to minimize this function n=n(a,b). It will turn out to be easier to find the minima of log(n), viewed as a function of a and b. The minima of n will occur precisely at the minima of log(n). So to find the solutions to $dn = 0$, we just find the solutions to $d\text{LOGn} = 0$. Now log(n) = log(2)a + log(3)b + log(5)c. So the differential $d\text{LOGn} = \log(2)da + \log(3)db + \log(5)dc$. Substituting in the value we obtained for dc, we get

$d\text{LOGn} = \log(2)da + \log(3)db + \log(5)[(-(c+1)/(b+1))db + (-(c+1)/(a+1))da]$

$= [\log(2)-(c+1)\log(5)/(a+1)]da + [\log(3)-(c+1)\log(5)/(b+1)]db$

One nice way to make this identically zero is if the coefficients of da and db become zero. That is, n will have minima when both log(2) = (c+1)log(5)/(a+1) and log(3) = (c+1)log(5)/(b+1) are true. That is, when (a+1)log(2) = (b+1)log(3) = (c+1)log(5).

This is a generalization of the earlier result that minima occur when (a+1)log(2) = (b+1)log(3). We can easily see that the general pattern of the constraints are: $(a_i+1)/(a_j+1) = \log(p_j)/\log(p_i)$.

What are the explicit formulae for the exponents in the k=3 case? We can solve for them in terms of d by using T1. Namely,

(a+1)(b+1)(c+1) = d

(a+1) = (c+1)log(5)/log(2)

(b+1) = (c+1)log(5)/log(3)

Substituting the last two equations into the first, we get $(c+1)^3 (\log(5))^2 = d \log(2)\log(3)$. Hence c+1 = CUBEROOT[d log(2) log(3) / $\log^2(5)$]

For reasons of symmetry, we will transform this slightly into

c+1 = CUBEROOT[d log(2) log(3) log(5)] / log(5)

The nicely symmetric equations for a+1 and b+1 turn out to be:

a+1 = CUBEROOT[d log(2) log(3) log(5)] / log(2)

b+1 = CUBEROOT[d log(2) log(3) log(5)] / log(3)

Viewed in this way, we can rewrite our equation from the k=2 case into the same kind of expression, namely:

a+1 = SQUAREROOT[d log(2) log(3)] / log(2)

b+1 = SQUAREROOT[d log(2) log(3)] / log(3)

Again the general pattern seems to be evident:

a_i+1 = K^{th}ROOT[d log(2) log(3)...log(p_k)] / log(p_i)

As in the k=2 case, the equations for a,b,c have real correspondence to the optimal integral values of the exponents.

Appendix 4.4. The General Case

We are now ready to consider the most general case, namely when $n = 2^{a1}3^{a2}...p_k^{ak}$. By T1, we know that $d(n) = (a_1+1)(a_2+1)...(a_k+1)$. One generalization of our earlier work would indicate that minima of n (for a given value of d) occur whenever

(T2) [for all i and j between 1 and k] $(a_i+1)/(a_j+1) = \log(p_j)/\log(p_i)$.

This is really a set of k-1 equations in the k different variables $a_1,...,a_k$. Using the formula for d which T1 provides, we can solve this system of equations for each a_i in terms only of d. The resulting formulae are:

(T3) [$\forall i \le k$] a_i+1 = K^{th}ROOT[d log(2) log(3)...log(p_k)] / log(p_i)

The deriviation of T3 from T2 is straightforward. Below is the proof of T2, due to Knuth. He uses Lagrange multipliers.

> The task is to minimize n, for a given d. It suffices to find the minima of log(n). Thus we wish to minimize $a_1\log(p_1)+...+a_k\log(p_k)$, for a given value of $d=(a_1+1)*...*(a_k+1)$.

But this latter constraint means that $\lambda*[(a_1+1)...(a_k+1) - d]$ is identically zero for any value of λ. Thus we may view our problem as the minimization of $a_1\log(p_1)+...+a_k\log(p_k) + \lambda*[(a_1+1)...(a_k+1) - d]$. For any i, the partial derivative of this with respect to a_i is $\log(p_i) + \lambda*[(a_1+1)...(a_k+1)]/(a_i+1)$. At an extremum, all such partial derivatives vanish. That is, for any i, $\log(p_i) + \lambda*[(a_1+1)...(a_k+1)]/(a_i+1) = 0$. This says that $(a_i+1)\log(p_i) = -\lambda*[(a_1+1)...(a_k+1)]$. So, for any i, $(a_i+1)\log(p_i) = -\lambda*d$. Since λ and d are independent of i, this proves T2.

Now that we know T2 and T3 to be true, we can actually compute the optimal[3] values for n. It will simplify matters again to consider only log(n) for the moment. [note: $SIGMA_i(...)$ means "the sum, from i=1 to i=k, of ..."] Now

$$\log(n) = a_1\log(2) + a_2\log(3) +...+ a_k\log(p_k)$$

$$= SIGMA_i[\log(p_i)\, a_i]$$

$$= SIGMA_i[\log(p_i)X(K^{th}ROOT[\, d\ \log(2)\ \log(3)...\log(p_k)\ ]/\log(p_i)) - 1)]$$

$$= SIGMA_i[K^{th}ROOT(\, d\ \log(2)\ \log(3)...\log(p_k)\ ) - \log(p_i)]$$

$$= k[K^{th}ROOT(\, d\ \log(2)\ \log(3)...\log(p_k))] - SIGMA_i[\log(p_i)]$$

This then gives the nice result:

$$(*)\quad n = \{_e[k\ K^{th}ROOT(d)\ K^{th}ROOT(\log(2)\log(3)...\log(p_k))]\} \ /\ \{\log(2)\ \log(3)...\log(pk)\}$$

If we let F_k represent the product of the first k primes, then this says

$$n = \{_e[k\ K^{th}ROOT(d)\ K^{th}ROOT(F_k]\} \ /\ F_k$$

If we let G_k be $_e[k\ K^{th}ROOT(\log(2)...\log(p_k))]$, then this becomes

---

[3] Real, not integral. The exponents a_i are assumed to be allowed to have real values.

$$(T4)_n = \{G_k[K^{th}ROOT(d)]\} \;/\; F_k$$

So by tabulating G_k and F_k, we can efficiently compute the ideal value for n (and for each a_i) given a desired d and allowable k.

Notice that if we are allowed more and more distinct prime factors, that is, as k grows, the real-valued exponents a_i get smaller and smaller, until finally they become negative, and we have broken all ties to reality. Empirically, the ideal value seems to be obtained when no exponent is allowed to be below 0.5; in those cases, the ideal real value for n is both close to, and slightly lower than, any intergral solution.

Of course this is not satisfactory; what we now need is a formula which tells us, for a given d, how many distinct prime factors any n must have, in order to have d divisors. That is, we would like to know k as a function of d, or k as a function of n. Luckily, k(n) is a very slowly changing function.

For the numbers of form $n=2\cdot3\cdot5\cdot\ldots\cdot p_k$, we can see from T1 that $d\cdot 2^k$. For maximally-divisibles, it seems likely that d will in general be larger; say it is of the form $d=\beta^k$ (where β is trivially seen to be ≥ 2). : Then we can plug this into (*):

$$n = e^{\{[\log(d)/\log(\beta)]\beta\ K^{th}ROOT[\log 2\ \log 3\ldots\log p_k]\}} / 2\cdot3\cdot5\ldots p_k$$

But the geometric mean is roughly $\log(p_k)$, which is about log(log(d)). Also, the product of the first k primes is roughly k^k, which is about [log(d)/log(β)][loglog(d)-loglog(β)]. Putting these into the last equation, we get:

$$n \geq e^{\{[\log(d)/\log(\beta)](\beta-1)\log(\log(d))\}}$$

$$n \geq d^{\{\log\log(d)\ (\beta-1)\ /\ \log(\beta)\}}$$

If the best we can do is the trivial result that $\beta\geq 2$, then we obtain the already-known relation[4] that

$$n \geq d^{\{\log\log(d)\ /\ \log 2\}}$$

If we can show that k is at least 3, then these n's jump to the *squares* of their former values.

---

[4] This is in fact the sharpest bound hitherto known for n(d). It was previously derived much more tortuously, using methods not related to the calculus.

This would be a much better bound, of course. In general, the sharpest bound will be found by determining β sharply.

Appendix 4.5. An even stronger claim

A very constructive answer to this whole development could be provided if the following were true:

(T5) The set M of maximally divisible numbers coincides precisely with the set of integers obtained in the following manner:

> (1) For each natural number d, use T3 to compute the optimal exponents for n(d,k), with k as large as possible such that no a_i is below 0.5
>
> (2) Round each exponent to the nearest integer, and compute the corresponding n. Add this n to the set.

There is probably a nice closed-form formula for such numbers, a sort of "compiled" version of T3 and T5. This is then the desired characterization of M. Exhaustive search has in fact confirmed T5 for all d below 1500.[5] T5 has the advantage of being intuitively clear. Perhaps its proof will turn out to be nontrivial or nonexistent. I leave it as "AM's conjecture". This is so far the only piece of interesting mathematics, previously unknown, that was directly motivated by AM.

For example: consider d=1344. The largest that k can be without T3 calling for $a_k < 0.5$ is k=7. For this d and k, T3 predicts exponents 5.9, 3.3, 2.0, 1.4, 1.0, 0.9, and 0.7. Rounding this off, we get 6, 3, 2, 1, 1, 1, 1. Next we compute $2^6 3^3 5^2 7^1 11^1 13^1 17^1$. This is 735,134,400. T1 tells us that this has in fact precisely 1344 divisors (coincidence). Exhaustive search tells us that no smaller n has that many divisors (this is a verification of T5). Incidentally, the ideal real value for n (for this k and d value, using (*)) is 603,696,064. Notice that it is close to, and less than, the best possible integral n with 1344 divisors.

Another such "rounded-exponent" number is
$n=2^8 3^5 5^3 7^2 11^2 13^1 17^1 19^1 23^1 29^1 31^1 37^1 41^1 43^1 47^1 53^1$. The progression of its exponents+1 (9 6 4 3 3 2 2 2 2 2 2 2 2 2 2 2) is about as close as one can get to satisfying the "logarithm" constraint. By that I mean that 9/6 is close to log(3)/log(2); that 2/2 is close to log(47)/log(43), etc. Changing any exponent by plus or minus 1 would make those ratios worse than they are. This number n is about 25 billion, and has about 4 million divisors. The AM conjecture is that there is no smaller number with that many divisors. Incidentally, the average number in the neighborhood of n has about $2^{\log\log n}$ divisors (about 9 divisors, for numbers near this n).

---

[5] The only fault is that the number 4 is in M, yet isn't found by this procedure. This may be due to errors occurring near small integers, or it may portend the occasional (perhaps infinitely often) failure of this procedure T5.

Appendix 4.6. AM and Ramanujan

I then adopt a different point of view [from Dirichlet, Wigert, and other mathematicians who have studied d(n)]. I define a highly composite number as a number whose number of divisors exceeds that of all its predecessors.

-- Ramanujan

What AM did: AM defined the set M, of maximally-divisible numbers. It thought the set might prove interesting. AM found several members of M. It had recently learned about unique factorization, so it factored each member: each number $n=2^{a1}...p_k^{ak}$ was replaced by the sequence $<a_1,...,a_k>$. While factored in this form, a rough kind of regularity was noticed. AM didn't have the concepts of logarithms, exponentiation, *e*, analyticity, reals, continuity, etc., so it couldn't possibly carry this work much further.

What the author did (aided and abetted by Randall Davis): Noticing the general pattern in these sequences[6], the author developed the results which were described in the past few subsections. The major results are as follows:

1. The smallest possible number n with d or more divisors (where n is of the form $n=2^{a1}...p_k^{ak}$) is $e^{k*K^{1th}ROOT\{d*\log2*\log3*...*\log(p_k)\}/2*3*...*p_k}$. This is a real number, which is a good lower bound on n(d) (the smallest n with d or more divisors). If k is approximable as log(d)/log(β), for some β (we know β is bigger than 2), then the preceding formula can be simplified into:

   $$n \geq d^{\{\log\log(d)*(\beta-1)/\log(\beta)\}}.$$

   T' : higher one can prove β (>2) is, the better this result.

2. For such "idealized" real values of n(d), the exponents a_i of the prime factors of n can be computed by the formulae: $a_i+1 = K^{1th}ROOT\{d*\log(2)*\log(3)*...*\log(p_k)\}/\log(p_i)$. These exponents satisfy the well-known relationship that the product of the (a_i+1)'s is equal to d. They also satisfy the lesser-known[7] relation that $(a_i+1)*\log(p_i)$ is a constant (the same for all values of the index *i*).

---

[6] Namely, they seemed to be describable as: <big no, medium no, medium-small-no,.., 2, 2, 1, 1,.., 1>

[7] I thought this was "unknown", but later found that Ramanujan had found a very similar relationship.

3. The elements of M appear to be those same numbers, but with the above real-valued exponents (the a_i's) rounded to the nearest integer.

What Ramanujan did: Very recently, the author was directed to the work of Srinivasa Ramanujan Aiyangar. This Indian mathematician was essentially self-taught. Receiving little formal education, he had almost no contact with Western number theory. Yet he became interested in number theoretic issues, and re-derived much of that field all by himself. In that way, he is perhaps the closest human analogue to AM: he had ability, techniques, background knowledge, and he was left to explore and redevelop much elementary mathematics on his own. Let me quote from G. H. Hardy's final[8] sketch of this genius:

> *"The limitations of his knowledge were as startling as its profundity... Here was a man who...had found the dominant terms of many of the most famous problems in the analytic theory of numbers, and yet...his ideas of mathematical proof were of the most shadowy description. All his results, new or old, right or wrong, had been arrived at by...intuition and induction from numerical examples."*

It was exciting to learn that Ramanujan had also defined the very same set M, which he called *highly-composite* numbers. His great interest in them has been almost unique[9] within mathematics circles – until AM was led to consider them. In an article published in 1915, Ramanujan derives the relationship: $a_i \cdot \log(p_i) = \text{const}$, which he says holds approximately, for values of i which are much smaller than k. He establishes this using inequalities (and using the distribution of prime numbers $\pi(x)$). Thus it has a different flavor from the similar relationship derived using calculus (•2 above, and also found as statement T2 a couple pages ago). Also, Ramanujan at this point went off on a different path, and missed the other two results listed above (•1 and •3). Instead, he defined a specialization of this concept, which he called '*superior* highly-composite numbers', and investigated them in detail. Neither AM nor the author had the sophistication to make that definition and to find the results which Ramanujan subsequently uncovered about superior highly-composite numbers.

---

[8] Taken from Ramanujan's obituary notice, 1921. Found in the preface to [Ramanujan 27].

[9] recently, Paul Erdos has been studying these highly-composite numbers.

Appendix 5. Traces of AM in Action

There are three types of traces which are represented in this appendix. First comes a high-level prose desription, a commentary on AM as it goes through a long, successful run. This is an expanded version of the historian's description of AM as a mathematician, found in Section 1.3, on page 10.

Next comes a detailed description of what AM did, on a numbered task-by-task basis. This is an expanded version of the task-by-task trace given in Section 6.1, on page 115. For each task, a paragraph is provided explaining what AM did, why, and what happened. These task summaries start on page 294. The task numbers listed there correspond to the numbering in Section 6.1.

Finally, several pages of traces are presented in totally undoctored form, so the reader can see that (i) it *is* harder to follow than the slightly rephrased ones, and yet (ii) those earlier, "doctored" traces didn't enhance (or alter the the spirit of) what AM printed out.

Appendix 5.1. Prose Traces

In this section are sketched many of the paths which AM explored during its runs. Besides describing what AM did, this section will explain why, and indicate where each path led. Along the way, some concepts were created which were interesting to *us* (in the smug wisdom of millenia of hindsight) but which AM never bothered to develop. These will be noted, and a stab will be made to apologize for AM[1]. A few exciting moments occurred when AM became interested in a concept which had been ignored by humans – at least, unknown to the author. Very often the "new discovery" was never shown to be anything other than cute (e.g., the concept of Timberline; see page 133 for a definition and diagram of it).

AM began by exploring structures and structural operations. After it was started, with the base of knowledge outlined in the previous chapter, the main activity AM did for the first several minutes was to fill in examples of various kinds of structures (e.g., Sets), structural operations (e.g., Insert), and create new concepts of this type (e.g., Singleton). In more detail, here is what was done:

Trying to fill in examples of set-operations, AM kept failing because there were no known examples of Sets to "run" those operations on. So it turned to filling in examples of Sets. Some of these came from the recursive definition of a set: "S is a set if S={} or if both (i) we can pull an element y out of S using Some-member, and (ii) Set-delete(y,S) is a set". A heuristic rule knew to extract the base case from such a definition, yielding {} as one example of a set. Another heuristic said to run operations whose range is 'Sets'. One of these is Set-insert. So a procedure for getting a new set is to take the given set S, and anything y, and run Set-insert(y,S). When this was done, using S={}, a bunch of singletons

[1] The typical excuse is that AM was distracted at that moment by some even more interesting task.

were created. AM literally collected Examples(Anything) and randomly chose members y from that big and varied assortment. Other set-operations were run just to provide some additional examples of Sets. Not every attempt was successful, of course: one heuristic said to find some examples of Lists, and then use the View facet of Sets to transform them into sets. Unfortunately, there were no examples of any other kinds of structures at the moment, so this rule failed. After about 20 cpu seconds, the time and space quanta were both just about exhausted. Roughly 30 examples of sets were found.

In similar ways, examples were found for Bags, Lists, Osets, and Ordered-pairs. Examples of structural operations were found "incidentally" as above – to aid in producing examples of a certain kind of structure. Occasionally, the primary task (the one selected from the agenda) was to find examples of a given operation. In that case, AM spent a great deal of time just on that one operation. For example, consider Set-union. When AM got around to filling in some examples for it, many examples already existed under the concepts of Sets, Bags, and Bag-union. One way to get examples of Set-union was by analogy to Bag-union. This caused some slightly erroneous entries to be found (e.g., {a,b,c}∪{a,c,e}={a,a,b,c,c,e}). Such errors were soon caught and corrected when the task **"Check examples of Set-union"** was chosen from the agenda. Similar errors and corrections occurred when AM viewed lists as if they were osets, in order to find examples of osets.

The simple development described above was broken frequently by some new concept being created and found to be very interesting. In fact, if left to its own judgment, AM would never finish the above process, because of these interruptions. The user must interrupt and tell it to ignore new concepts, if he really wants AM to finish finding examples of all those structures and operations.

What kinds of concepts were created, and why? What did AM do to investigate them? In general, what happened was this: As AM collected examples of a concept C, the characteristics of its efforts (how easy/hard to find examples, how varied they were, etc.) caused certain heuristic rules to trigger. Those rules then caused some new concepts to be created, for a particular reason. AM would find examples of them, then compare the results with already-known concepts.

The first instance of this process was when AM found many examples of sets so easily. A rule said that in such cases, it was worthwhile specializing the concept Sets. That is, make a new concept which would have fewer examples. One way AM did this was to look over the Interest features of all generalizations of Sets, pluck a couple of them, and conjoin them onto the definition of Sets, thereby getting a definition for a brand new concept, called interesting-sets or INT-Sets for short. The feature selected first was the following: each pair of elements of the structure satisfy the same rare predicate P, for some P. This was previously tacked onto the Interest facet of Structures. Initially, there were very few predicates known: Constantly-True, Constantly-False, Object-Equality. The following three types of INT-Sets were therefore eventually found:

(i) Sets – the same concept but in a new disguise (for any pair of members from *any* set, Constantly-True would return True),
(ii) Empty-sets – an already known concept, but now with a new definition (for any pair of members from *any* set, Constantly-False would never return True), and
(iii) Singletons '{a}' (sets for which all pairs are Equal to each other).

This immediately catapulted the empty set to stardom. Another heuristic rule had AM restrict its attention to the predicates which were not trivial. In this case, both Constant-True and Constant-False were no longer eligible. So the only remaining INT-Sets were those for which all pairs of elements were Equal. AM decided to explicitly define a new kind of set having just that definition. This became a specialization of INT-Sets, called Equality-sets or Esets for short. Since the empty set was already distinguished, it was decided not to include it as a valid Eset. So Esets were precisely the sets we would call Singletons.

Unfortunately, the set-operation Union, when applied to Singletons, didn't always yield singletons. By isolating the kind of sets they *did* yield, and not counting the few extreme cases when they yielded singletons, AM discovered the concept of Doubleton: a set with precisely two members. Typically, the union of Singletons was a doubleton, their intersection was the empty set, their Set-difference was a singleton, inserting anything into a singleton was a doubleton, removing something left a singleton, etc. The exceptions were all related to when the arguments were equal.

Several more structural concepts were created and explored in this way, besides Singleton: Doubleton, Tripleton, Function (an operation, all of whose values were singleton sets: i.e., a single-valued operation),... In general, these occurred because the intial primitive concepts were too general, too easy to satisfy.

During its investigation of Set-Intersection, AM noticed that sometimes $X \cap Y = X$, and formalized that relationship between two sets. This is just the relation we normally call Superset. The notion of Subset also was discovered, but AM never had much interest in either of these notions.

When AM looked for examples satisfying the predicate Object-Equality, abbreviated Equal, the situation was just the opposite. A heuristic rule attached to 'Active' indicated that examples could be found by randomly instantiating the two arguments of Equal with a pair of objects. There were about 100 known examples of structures. AM gathered them into one set, and then repeatedly chose a pair of them to run Equal on. Thus only about 1% of the time did it succeed (did Equal return the value T). Another heuristic triggerred, and said that in such cases, it was worthwhile trying to generalize the predicate Equal. A new task to this effect was added to the agenda.

Soon, AM selected this task, and tried to create new concepts which were generalizations of Equal. One definition of Equal was a transparent recursive one, which essentially said that x and y were Equal iff their Cars and their Cdrs were, plus it had a base step that asked if both arguments were empty (in which case Equal returned True), or if precisely one argument was empty (in which case Equal returned False), or if both arguments were identical atoms (True), or if they were nonidentical atoms or only one was an atom (False).

```
λ (x,y)
  IF x and y are identical atoms, THEN return True;
  ELSE IF either x or y is not a list, THEN return False;
      ELSE IF both x and y are Null lists, THEN return True;
          ELSE IF only one of x or y is Null, THEN return False;
              ELSE both of these must be true:
                    Equal( CAR(x), CAR(y) )
                    Equal( CDR(x), CDR(y) )
```

Base
Cases

One heuristic rule that AM possessed suggested that this could be generalized in a small way by weakening the base step. A couple new concepts were created this way, but they all turned out to be trivial: either constantly returning T, or no different than Equal was.

Another heuristic suggested weakening the recursive step. One way to do this, since the recursive step is a conjunction, is to remove one of the conjuncts. The rule checks to ensure that the definition is still recursive: one of the remaining conjuncts must call on the function itself. In this case, both conjuncts call on Equal, so AM can remove either one. Two new concepts were generated in this manner. For example, here is one of them, which AM named "Equ0":

```
λ (x,y)
  IF x and y are identical atoms, THEN return True;
  ELSE IF either x or y is not a list, THEN return False;
      ELSE IF both x and y are Null lists, THEN return True;
          ELSE IF only one of x or y is Null, THEN return False;
              ELSE Equ0( CDR(x), CDR(y) )
```

Base / *Cases*

Note that the base cases were unchanged; the recursive step is a recursion in the CDR direction, but no longer in the CAR direction. A note to that effect is placed inside the definition of Equ0 itself, as a comment. Other parts of Equ0 can be filled in easily: it is a generalization of Equal, it is an example of a Predicate, its domain/range is the same as Equal, its worth is initially set a little higher than Equal's, etc.

This predicate maps down two lists, one element at a time, and returns True iff they both become empty at the same moment. That is, iff they have the same length. In fact, we can assume that the user interrupts and orders AM to rename Equ0 as "Same-length".

The other new generalization, called "Equ1", examines the CAR's (i.e., the first elements) of a pair of lists, and returns True if they were identical atoms; if they were both lists, it recurses on those two lists. A human LISP programmer's interpretation is as follows: when the two lists were written out in standard notation (using parentheses), all the initial left parentheses match, and the first non-left-parenthesis entity of each list also matches. Although this is isomorphic to numbers again[2], AM didn't pursue this concept very far. Most of the examples of lists AM knew about were only 1-level deep, although they varied significantly in length. If it had been otherwise, AM might have developed Equ1 into Same-length, and lost interest in Equ0. As it was, the Worth of this concept Equ1 slowly declined, and very few tasks involving it bubbled up to the top of the agenda.

The concept of Same-length will form the springboard for the development of cardinality, a tale which is related in a little while. Before moving on, let's mention a couple more set-theoretic activities that AM did.

Several moderately interesting compositions and coalescings were done to set-operations. (Actually, to structure-operations). First let's discuss the coalescings of operations f(x,y) into new operations f(x,x) – where a pair of arguments is made to coincide. Coalescing Insert (insert S into itself) led to an operation which always seemed to give a bigger set than it

---

[2] Two list structures were treated as equivalent if they have the same number of left parentheses; zero is the list NIL; adding 1 is consing with NIL; subtracting 1 is CAR.

started with. This led AM to the finitely-true conjecture that a set is never a member of itself. The conjecture was rediscovered when the coalescing of Delete seemed to produce the identity operation (Deleting S from S never seemed to change the value of S).

Coalescing Intersect also gave the identity operation; this showed that SnS=S (apparently). Similarly for Union. Coalescing "Compose" gave a new operation of some value: the notion of composing f with f. When this was applied to, e.g., Intersect, it created the new operation IntersectoIntersect, which took 3 arguments and formed their common intersection. By forming this in two ways – (xny)nz and also xn(ynz) – AM noticed that they were the same, that the order didn't matter. Since xnx had already been shown to be the identity operation, multiple occurrences of an element in an intersection don't make any difference. Finally, AM had constructed xny and ynx as two separate operations, and then found them to be equivalent. Taking all these results together, it was possible to view n as taking a *set* of sets as its argument, and forming the intersection of all those sets. Thus n({ {a,b,c},{c,g,h},{a,c,e} })={c}.

Some valuable compositons were formed. By forming xn(yuz) and (xny)u(xnz) as two separate compositions, AM found their equivalence via experimental data. This was one case where the Intuition functions had given AM an unfair advantage, since the Venn-diagram abstract representation for sets suggested this relationship explicitly. When the intuition was removed, the discovery was made much more valid. All of de Morgan's laws were discovered in this manner, incidentally. Several special cases were singled out, involving empty sets, singletons, and doubletons.[3]

The compositon DeleteoInsert is not quite so trivial as one might think: it takes a structre S, inserts an element e, and then removes element e. This simple operation can be used to test the type of structure that S is: it *never* alters a Bag or a List, but it does alter Sets and Osets. Orthogonally, InsertoDelete always alters Lists and Osets, but can leave Bags and Sets unchanged. The first composition tests for multiple elements, and the second composition tests for order. AM defined both of these, and (at least partially) perceived their abilities to distinguish structural types.

Operations formed by switching the two arguments of an old operation were marginally interesting. They helped pin down the true nature of what kind of argument the operation should be considered as taking. For example, Union(x,y)=Union(y,x), so Union can take an unordered collection of sets as its argument, and form their union. Similarly, we see that Insert(x,y) is in general quite different from Insert(y,x). When AM tries to see what invariants exist for this operation, it can notice only that the trivial constraint x=y is sufficient to cause the order of arguments not to matter. If it had the concept of the LISP function "COUNT", which counts up all the storage space used, or "FLATTEN", which removes all parentheses from a list structure, then AM would notice that the COUNT's of both forms of Inserting were equal in number, and that their FLATTEN's were equal sets of elements.

Looking for invariants is one favorite pastime of AM. In general, two operations f and g

---

[3] E.g., if x is a singleton, then xn(yUz) is equal to either xny or to xnz; if both those sets were the same, then of course xn(yUz) is equal to their common value; if the two sets differ, then one is empty and the other is x, and the ultimate intersection is equal to x. Or: that intersection is always either x or the empty set.

will not coincide. AM wants to find a unifying operation U, such that U(f(x))=U(g(x)); or, AM tries to find a U such that f(U(x))=g(U(x)). This is of course the idea mathematicians normally refer to as homomorphism. AM calls this process Canonizing. Canonizing the two orders of Insert (x into y, and y into x) would hopefully find the operation U=FLATTEN or U=COUNT. Canonizing Same-length will cause the operation of Length to be synthesized. Canonizing the notion of angles-equal-in-measure were the operations we normally call rigid motions in the plane.

Let's pick the main thread of development again, before we lose it entirely. Earlier, the operation "Same-length" was synthesized, as a generalized form of the predicate which told when two structures were equal. Same-length(x,y) is True iff x and y are two list structures which have the same length (i.e., the same number of elements). This new predicate was examined by AM, and sure enough it let many more pairs of random objects return True than Equality did, yet it didn't let too high a percentage through: just about 10%. This made AM want to find an invariant, a canonizing function f, which put any given list structure x into a standard form f(x), satisfying:

Same-length(x,y) iff Equal(f(x),f(y))

That is, x and y would have the same length precisely when the standard forms of x and y were identically equal to each other.

AM had to synthesize this function f, step by step. First it performed some experiments, and found that Same-length didn't care what the type of its arguments were. If a certain Set S did/didn't satisfy Same-length(R,S), then the same result would obtain if S were replaced by Viewing S as a list, or as a bag, or as an oset. Thus the standard form of a structure could be fixed as one specific type. But which should it be (bag, set, list, oset)? To find out, AM ran two more experiments. The first experiment was to see whether Same-length(R,S) was affected when the order of elements inside R were changed. This turned out to be negative. So R might as well be an unordered structure: bag or set. The next experiment had AM decide whether or not multple elements inside R would affect the value of Same-length(R,S). This turned out positive, so multiple elements had to be taken into account. The canonical type of argument was thus either bag or list. Together, the two experiments indicated that the type had to be Bag. So the canonizing function f would first convert any argument R to a bag. A note tacked onto the Same-length concept's definition said that this concept didn't look at the Car's or value-cells of its arguments. That would mean that they should all be replaced by some fixed value, like T. This checked out experimentally. So f should replace each element in the structure R by the constant T. The final operation f synthesized was:

f(R) = Replace-each-element-by-T (Convert-to--bag (R)).

This operation would convert {a, (b,c,{d},e,e), q} into (T,T,T). The set of standard forms for bags was called Canonical-bags, and renamed by the user as Numbers. The canonizing operation f was called Length, by the user, since it converts any structure into a "number" which represents the length of that structure:

Same-length(R,S) iff Equal(Length(R),Length(S))

AM now made explicit an important analogy: bags↔numbers, equal↔same-length, bag-

operations↔[those same operations, restricted to canonical-bags]. Several minutes were spent developing these restricted bag-operations. The old operation of inserting a bag S into itself provided a cute way to "add 1" to any number. The Bag-union operation union turned into what we call Addition:

Bag-union((T T T) (T T)) = (T T T T T) means "3+2=5", in unary.

TIMES was discovered in four ways, as discussed previously in the thesis.

The intersection of two "numbers" is the operation we call MINIMUM:

Intersect((T T T) (T T T T)) = (T T T) just says "Minimum(3,4)=3".

AM likes to find inverses, and the inverse of these operations turned out to be the ones we call subtraction, factoring, division, less-than, etc.

AM likes coalescing, and some important operations were created that way, too: Add(x,x) is Doubling; Times(x,x) is Squaring; the inverses of those were Halving and Square-rooting (both restricted to natural numbers).

AM worried about which numbers could be halved or square-rooted, and this led to the creation of the concepts Even-numbers and Perfect-squares. When asking when a number z can be the result of a multiplication involving x, AM derived the notion of Divides; analogously, AM defined the relation which meant that Add of x and something else could yield z. That relation is just "≤", and was called LEQ by the user. AM noticed many simple properties of inequalities.

AM likes composing and reversing args, and thereby figured out that most arithmetic operations like Add and Times take a *bag* of numbers to work on.

$TIMES^{-1}$ was, as we said, factoring: given n, find all possible bags of numbers (>1) whose product yielded n. The identity of Times ("1") was intentionally excluded, since its presence in any quantity would not affect the result of Times. AM soon asked itself about numbers with extreme or unusual factorings.

Primes were found in this way, as was Goldbach's conjecture. The example in chapter 2 went into this in great detail. A large number of spurious analogous concepts were created and studied, to no avail. For example, AM asked itself about numbers which could *uniquely* represented as the sum of two primes. As another example, AM defined the concept of Square-roots-of-primes, which of course was not a winner.

The unique factorization of any number into primes was perceived quite naturally, but many seemingly elementary concepts were left undiscovered. AM never defined gcd (the greatest common divisors) onits own; it was, however, possible to guide it to discovering that concept.

The task-by-task trace in the next section closes with the restriction of addition to primes; i.e., p+q=r for primes p,q,r. This situation only occurs when p (say) is 2, and q,r form a prime pair. That trace will of course delve into concepts not mentioned above, and

conversely AM happended to miss, onthat run, some of those mentioned in this section. Finally, both sections omit mention of several interesting acitivites AM performed: maximally-divisibles and all the geometric concepts, for example. The line must be drawn somewhere. The frustrated reader should try AM himself, and watch its progress.

Appendix 5.2. A 'Nice' Task-by-task Trace

Now that we've discussed this development at a fairly high level, let's list what AM did task by task. The lines below give a highly compressed "trace" of AM's sequence of behavior. The tasks in the "best run"[4] are listed in order, and numbered. For each, I have condensed AM's printout, usually retaining some of the reasons AM had for atempting the task, the methods AM used, the final outcome, and occasionally a bit of commentary (in italics). The task numbers below correspond to the numbering used in Section 6.1, starting on page 115.

** Task 1 ** Fill in examples of Compose, because Compose is highly-rated and no examples of Compose are known yet. Several heuristics relevant, but none succeeded. One of them, heuristic rule number 122, failed because no operations could be found which had examples. Also important at this moment was heuristic rule number 156; see Appendix 3.

** Task 2 ** Fill in examples of Set-union, because Set-union is highly-rated, and no examples of Set-union are known yet, and if some examples had been known earlier then AM would have been able to Fill in examples of Compose. Several heuristics relevant, but again none succeeded.

** Task 3 ** Fill in examples of Sets, because Sets is highly-rated, and no examples of Sets are known yet, and if some examples had been known earlier then AM would have been able to Fill in example of Set-union, and AM was recently working on Domain(Set-union), and AM was recently working on Range(Set-union). Several heuristic rules are relevant. After rule 31 succeeded, so could many other techniques (e.g., rule 38). In fact, it was so easy for AM to crank out one example of a set after another, that rule 45 triggered.

** Task 4 ** Fill in specializations of Sets, because it was very easy to find examples of Sets, and no specializations of Sets exist yet, and Focus of Attention. Many ways of creating new concepts, new specialized forms of Sets, were relevant. AM created INT-Sets, defined as "λ (S) S is a set, and all pairs of members of S satisfy the rare predicate P". AM also created BI-Sets, defined as " λ (S) S={}, or else both CAR(S) and CDR(S) are BI-Sets." *The former specialization will lead to Singletons, the latter deals with nests of braces (sets with no atomic elements).*[5]

** Task 5 ** Fill in examples of INT-Sets, because any such item will automatically be an interesting example of a Set, and INT-Sets was just created, and no examples of it are known yet. Very slowly, 6 examples were found via inference, and then 7 more were produced quickly via more brutish methods. After eliminating duplicates, only 3 remain: {}, {A}, and {B}. In each case, the predicate P was "Equal", and the worth of the concept Equal was raised slightly.

---

[4] Actually, a couple "very good" runs have been appended, but NOT spliced together to the benefit of AM.

[5] Recall that *Italics* signify the author's comments, and contain information which AM -- and probably the user as well -- could not have known at the time.

** Task 6 ** Check all examples of INT-Sets, because many unchecked examples are present there, and Focus of Attention. All three examples were confirmed. No surprises were encountered. One of the three INT-Sets turned out to be an Empty-structure, but no conjecture was actually formulated at this time.

** Task 7 ** Check all examples of Sets, because many were recently found and never checked, and the previous task dealt with a specialization of the Sets concept. One small modification had to be made to one of the sets (namely, {b,a,b}→{a,b}). Based on empirical evidence, AM hypothesizes that Sets may really be no more specialized than Unordered-strucs. AM will reserve judgment until after it has tried to find examples of Bags. (See heuristic rule 108, page 248.) Similarly, AM considers the possibility that all Non-multiple-elements-strucs are really Sets as well. Before relying on this flimsy conjecture, AM must first look for examples of Osets, and see if *they* are really also Sets.

** Task 8 ** Fill in examples of Bags, because no examples of Bags exist yet, and to help settle the question about all unordered structures being sets. 10 examples found. None are sets.

** Task 9 ** Fill in specializations of Bags, because it was very easy to find examples of Bags, and no specializations of Bags are known about yet, and Focus of Attention. Many ways of creating new concepts, new specialized forms of Bags, were relevant. AM created INT-Bags, defined as "λ (S) S is a Bag, and all pairs of members of S satisfy the rare predicate P". AM also created BI-Bags, defined as " λ (S) S=(), or else both CAR(S) and CDR(S) are BI-Bags."

** Task 10 ** Fill in examples of Osets, because none exist yet, and to help settle the question about all nonmult-strucs being sets. 13 distinct examples found. None are sets.

** Task 11 ** Check examples of Osets, because many unchecked examples of Osets exist on Osets.Exs, and Focus of Attention. All confirmed. The priority rating of this task is not high enough to inspire the creation of any new concepts. One weak conjecture made: all ordered structures are Osets. Will settle this by:

** Task 12 ** Fill in examples of Lists, because none exist yet, and to help settle the question about all ord-strucs being osets. 29 examples found. None are osets.

** Task 13 ** Check examples of Lists, because many unchecked examples of Lists exist, and Focus of Attention. All confirmed. Nothing special noted or motivated.

** Task 14 ** Fill in examples of All-but-first, because no such examples exist yet, and AM was just working on Domain(All-but-first), and AM was recently working on Domain(All-but-first). *The similarity of the last two reasons escaped AM, and points up one of its flaws. AM is swayed by the presence of a slightly-different reason just as much as by a very-different supporting reason. There is no processing done on the reasons.* Choosing this task represents a radical shift of attention for AM. 32 examples found, mostly by applying All-but-first to the examples of Lists and Osets already known.

** Task 15 ** Fill in examples of All-but-last, because none exist yet, and AM was recently working on Domain(All-but-last). Another poorly-motivated task. *Luckily for AM, the user erroneously believes that this task is also chosen to be symmetric with the last task.* 45 examples found.

** Task 16 ** Fill in specializations of All-but-last, because it's so easy to find examples of it, and no specializations exist yet, and Focus of Attention. The syntactic methods AM can bring to bear are not enough to produce any meaningful new concepts, and this task Fails. *Failure of a task causes 'Focus of Attention' to go away for one cycle.*

** Task 17 ** Fill in examples of List-union, because none exist yet. Another wild shift of attention. 3 examples derived by symbolic manipulation of the definition facet of this concept, then 22 more derived using less inferential techniques (some were garnered by running List-union.Alg itself on the early examples!).

** Task 18 ** Fill in examples of Proj1, because none exist yet. 26 found.

** Task 19 ** Check examples of All-but-first, because many were recently found but not yet confirmed. All check out. This task has no repercussions (new concepts, tasks, etc.).

** Task 20 ** Check examples of All-but-last, because many unchecked examples exist on the Examples facet of All-but-last. All confirmed, with no repercussions. *If the initial Worth values of All-but-first and All-but-last are set high enough, AM defines a new concept at this point, a new kind of ordered structure: λ (S) All-but-first(S) = All-but-last(S). In fact, the only kind of Osets included herein are those which are singletons or empty. Among lists, it also includes those which contain just one kind of element.*

** Task 21 ** Fill in examples of Proj2, because none exist yet. 26 found. AM will typically quit looking for examples if (i) the time allocated runs out, or (ii) the space allocated is used up, or (iii) 26 examples are found, or (iv) 151 attempts to find examples fail. *The cosmic significance of 151 has rarely been recognized in print. Perhaps 151 is more comic than cosmic. Seriously, these numbers must be changed by almost an order of magnitude before any gross change in AM's behavior is noticed.*

** Task 22 ** Fill in examples of Empty-structures, because none exist yet, and the Worth of this concept was increased recently (during task 6). Just by looking at the examples of Structures (i.e., using heuristic rule number 29), AM is able to get four empty ones: {}, [], <>, (); i.e., the empty set, oset, list, and bag. Although some of these are rederived in other ways, there are no other examples ever found. This paucity triggers a rule which suggests this task:

** Task 23 ** Fill in generalizations of Empty-structures, because it was very hard – but not impossible – to find examples of that concept, and Focus of Attention. AM examines the definitions of Empty-structure, but can't find any recognizable syntactic pattern it knows about. It does find (NOT (SOME-MEMBER S)), and tries to replace SOME-MEMBER by a specialization of same, but none is known to exist. *If the user initially tells AM that First-member and Last-member are specializations of Some-member, then AM can generalize Empty-structures. In fact, it then comes up with what is equivalent to 'Empty-struc ∪ Unord-struc'. In the current setup, however, this task fails. If AM had a better understanding of constructive/destructive operations, it might have defined Structures-with-empty-CARs, or perhaps Structures-with-empty-CDRs (i.e., Singletons again).*

** Task 24 ** Check examples of List-union, because many were recently added but not yet confirmed. This shows the mechanical patience that a 'stack' gives you. Since no higher-priority task has been suggested, AM attends to a task which has been on there for a while.

** Task 25 ** Check examples of Bags, because many examples and a couple specializations exist. A few small modifications had to be made (e.g., (A C B A) → (A A B C)).

** Task 26 ** Fill in examples of Bag-union, because none exist yet, and AM was just working on Domain(Bag-union), and AM was just working on Range(Bag-union). Note the influence of heuristic rule numer 14. This task proceeded smoothly, with 26 examples being generated.

** Task 27 ** Check examples of Proj2, because several were recently found and not yet checked. All confirmed, with no new suggestions generated.

** Task 28 ** Fill in examples of Set-union, because none exist yet. Again we see rule 14 in action. 26 examples found.

** Task 29 ** Check examples of Set-union, because many examples have recently been found, but not yet checked, and Focus of Attention (AM just worked on Set-union). A few patches had to be made. Often, Set-union(x,y) was equal to one of its arguments. AM therefore defined a new predicate Eq-union(x,y) which is True iff Set-union(x,y)=x. The user later renamed this "Superset-of", since this is the relationship of containment. *In typical math texts, containment is introduced long before union, and then this is a theorem: "A⊃B iff A∪B=A". But AM used "∪" to form the definition of "⊃".*

** Task 30 ** Fill in examples of Bag-insert, because none exist yet, and AM was recently working on Domain(Bag-insert), and AM was recently working on Range(Bag-insert). *A saddeningly stupid move for AM. It should have rated Superset higher, and continued working on it.* AM has no trouble finding many examples of insertions into bags.

** Task 31 ** Check examples of Bag-insert, because many examples have recently been found, but not yet checked, and AM just worked on Bag-insert. All examples were confirmed. This operation always seemed to result in Nonempty bags. The Domain/range facet was so amended. The value is never either of its arguments, but there is no concrete action AM must take in such a situation, no compact way of noting that anti-relationship (short of creating a full-blown conjecture). Restricted to singletons, Bag-insert never produces a singleton or an empty bag. AM defines the concept of a bag gotten by doing a Bag-insert on a singleton; i.e., the notion of a doubleton bag.

** Task 32 ** Fill in examples of Bag-intersect, because none exist yet, and AM was recently working on Domain(Bag-intersect), and AM was recently working on Range(Bag-intersect). 26 found without trouble.

** Task 33 ** Fill in examples of Set-insert, because none exist yet. Just another data-gathering task, building up AM's store of empirical results.

** Task 34 ** Check examples of Set-insert, because many examples have recently been found, but not yet checked, and AM just worked on Set-insert. Applying this operation always seems to result in Nonempty sets. Domain/range so amended. The value is sometimes just one of its arguments. AM defines what will eventually be called Set-membership in this way: λ (x,S) Set-insert(x,S)=S. This is not the only important result here. AM notices that Set-insert(x,S) is always related to S by Superset-of. That is, Superset-of(Set-insert(x,S), S) [for any x]. So conjectured. This doesn't actually suggest anything marvelous for AM to do next, however. Incidentally, during this task AM also defines the concept of doubleton set.

** Task 35 ** Fill in examples of Bag-delete, because none exist yet. *Note that 'working on bags' is so long past that it is no longer given as a reason for this task.* Able to generate two examples by manipulating definitions supplied with Bag-delete, then a couple dozen more were generated. Some were generated by accessing already-known examples of the domain (i.e., the entries on the Examples facet of the Bags concept) and then running Bag-delete.Alg on them.

** Task 36 ** Fill in examples of Bag-difference, because none exist yet. 26 found. *This is a good vantage point to look back and ahead, and notice that while the surrounding tasks are all reasonable data-gathering forays, the order in which they're performed is unimportant. Later on, AM will follow threads of discoveries, and the order of tasks then will appear very important.*

** Task 37 ** Check examples of Bag-intersect, because many examples have recently been found, but not yet checked. So many examples were found that AM will entertain the idea of creating a specialized new concept. Since the intersection of two randomly-chosen bags was often empty, AM defined the concept of disjoint bags: λ (x,y) Bag-intersect(x,y)=().

** Task 38 ** Fill in examples of Set-intersect, because none exist yet. Many found very easily.

** Task 39 ** Check examples of Set-intersect, because many examples have recently been found, but not yet checked, and Focus of Attention. 3 small modifications had to be made. This time, AM noticed that the intersection of two sets was often empty, and defined the Disjoint-sets concept. AM also noted that xny was often one of those very same arguments, so it defined the relation which is eventually renamed Subset: λ (x,y) Set-intersect(x,y)=x. At the moment, AM didn't realize that there was any connection between Subset and Superset. This tie came much, much later (Task number 227 (qv.) in this run).

** Task 40 ** Fill in examples of List-intersect, because none exist yet, and the interest of Intersect (the general concept of which this is a specialization) has recently been increased a few times. 26 found without incident.

AM is bored[6]! Will look at Suggest-type heuristics for new things to do.

> *If "Equality" is excised at this moment, AM continues the preceding line of inquiry for a while, and then defines Singleton-osets, as Osets all of whose members are equal to each other. AM notices that All-but-first and All-but-last, restricted to Singleton-osets, always yield the same result, namely the empty oset. AM then "generalizes" this into the concept which is all the osets for which All-but-first(x)=All-but-last(x). AM then turns to relationships involving Subset and Superset, followed by a flurry of compositions and coalescings, and their investigation. But Equality is present, so AM goes off on another course of exploration.*

** Task 41 ** Fill in examples of Equal, because Equal has recently become more interesting, and there are no examples known yet. Equal became more interesting gradually, as INT-Sets were define and liked, Eq-union defined and liked, etc. Once chosen, this task does not go smoothly. By inference methods, only a couple examples were derived. Later, when heuristic rule number 122 was run, 151 failures were encountered and only 2 successes. This is so small a success rate that a heuristic rule strenuously proposed this next task, with a high enough rating to be chosen right away:

---

[6] Of course "bored" is just what AM types out. It reflects the low value of the top task on the agenda, and the meager results obtained recently. Please forgive the anthropomorphism; it is meant only to be "cute", not misleading. AM has no internal model which could be called its "emotional state", as, e.g., PARRY [Colby 73] claims.

** Task 42 ** Fill in generalizations of Equal, because Equal is apparently quite rarely satisfied, and there are no entries yet on Equal.Genl. Removing one of the two conjoined recursive calls in the recursive definition given for Equal caused the creation of Equal-except-CARs and Equal-except-CDRs. *The first predicate tests whether x and y have the same number of elements; the second tests whether x and y have the same number of leading left parens and the same first atom after that final leading left parenthesis. As Knuth pointed out, both of these concepts are valid ways of defining "numbers": one counts the number of elements, the other counts the number of leading left parentheses. But most structures which AM knows about are just simple 1-level affairs. Therefore, Equal-except-CDRs was almost always the same as "CAR(x)=CAR(y)". So AM never realized this duality. If it had pushed Bl-Sets and Bl-Bags further, it might have.* Another concept created here is far more bizarre. Instead of eliminating one of the two conjoined recursive calls, AM replaced the AND with an OR. The new concept Genl-Eq3 was defined: 'λ(x,y) if x or y are non-lists, then EQ(x,y), else [Genl-Eq3(CAR(x),CAR(y)) OR Genl-Eq3(CDR(x),CDR(y)].' This is true if x and y have the same length, or if the jth element of each is the same (for any j), or if the jth element of each has the same length (>0), or if the ith element of the jth element of each is the same or has the same length (for any i,j), or...

** Task 43 ** Fill in examples of Equal-except-CARs, because this is a new generalization of Equal which must be examined, and because no examples exist yet. Only 10 examples were found before the time quantum was exhausted, but this was still many more than were found for Equal before. The user now renamed this concept "Same-size". *A whirlwind of discovery is about to sweep the other two generalizations of Equal out of the top spot on AM's agenda for quite a while. If AM accidentally picked another of these to work on before Same-size, only a small amount of time would have been spent before moving on. For example, AM is unable to perform Canonize.Algs(Genl-Eq3,Equal), so that would be a dead-end right there.*

** Task 44 ** Apply an Algorithm for Canonize to the args Same-size and Equal, because a heuristic rule[7] explicitly suggested that, and there are no known examples of Canonize yet, and AM was just working on Same-size, and AM was recently working on Equal, and Same-size was recently created, and Same-size was just renamed by the user. AM performs several experiments, and eventually synthesizes this canonizing function: f(S) takes a structure S, converts it to a Bag, and replaces each element by "T". This function is later renamed "Size" by the user. AM also defines the set of canonical structures: bags of T's. The user renames Bags-of-T's as "Numbers".

** Task 45 ** Restrict the Domain/range facet of Bag-union, because Bags-of-T's (called Numbers now) is a new, interesting specialization of Bags, and a heuristic rule explicitly suggested this, and Focus of Attention, and many examples of Bag-union exist. A new operation is defined, Number-union, with domain/range entry <Number Number → Bag>. This task used less than one cpu second.

** Task 46 ** Fill in examples of Number-union, because it is recently-interesting, and it was just created, and AM was recently working on Domain(Number-union). Several examples are found. *At this point, the author turned on a tricky LISP printing function, which converted each bag of T's to base-10 exponential notation before allowing it to be typed out.*

---

[7] The rule referred to is number 213, on page 270.

** Task 47 ** Check the domain/range of Number-union, because a heuristic rule explicitly suggested that the range might really be "Number", and AM was just working on Number-union, and AM was recently working on Domain(Number-union) [i.e., working on 'Numbers']. In fact, what the heuristic rule suggested purely from symmetry desires turned out empirically to be true: the value of Number-union(x,y) *did* always appear to be a Number (a bag of all T's). The result of this was to amend the Domain/range facet of Number-union. Although AM regards this uniformity as very interesting, it has no direct suggestion for what to do next. The user renames this operation "Add2", since it takes precisely 2 arguments (unary numbers) and adds them.

** Task 48 ** Restrict the domain/range of Bag-intersect, to Numbers, *for similar reasons as above.* After again noticing that the intersection of 2 numbers always seems to be a number, this leads to the operation which the user renames "Minimum". *Since the pattern of tasks is Restrict → Fill in examples → Check examples, there is not much point in listing all three tasks for all of these simple restrictions. Each one will only get a single number in this listing. Also, since the reasons for these restrictings are pretty much the same, they won't be repeated for each task below.*

** Task 49 ** Restrict the domain/range of Bag-delete, to Numbers. The user renames this operation "SUB1", although this is not quite accurate. If x is not 'T', then applying this operation to x and N (for some number N represented as a bag of T's) will not alter N at all. AM does not possess the reasoning abilities needed to anticipate this.

** Task 50 ** Restrict the domain/range of Bag-insert, to Numbers. The domain/range entry is changed to <Anything Number → Bag>. Renamed Number-insert. Although this new operation will in fact change a number N, it may not necessarily change it into a number. The last operation, SUB1, would always produce a number, though it might sometimes fail to change N at all. Here is the sad discovery of that asymmetry about Number-insert:

** Task 51 ** Check the domain/range of Number-insert, because a heuristic rule explicitly suggested that the range might really be "Number". In fact, its quickly seen *not* to be. This operation is lowered in worth, and never touched again. Due to AM's imperfect heuristics, the worth of SUB1 is slightly higher still than this concept's.

** Task 52 ** Restrict the domain/range of Bag-difference, to Numbers. After again noticing that the difference of 2 numbers always seems to be a number, this leads to the operation which the user renames "Subtract".

** Task 53 ** Fill in examples of Subtract, because none exist yet, and Subtract was just created. Many examples are found. If a larger number is "subtracted" from a smaller, the result is zero, according to this operation. Thus about half of these examples have the value zero (empty bag). AM explicitly defines the set of ordered pairs of numbers having zero difference. It turns out that (in modern terminology) <x,y> is in this new set iff x is less than or equal to y. So the user renames this relation "LEQ".

** Task 54 ** Fill in examples of LEQ, because none exist yet, and LEQ was just created. 26 examples found. When random numbers are chosen, the success rate is (as we wise observors know) a little over 50%. This is very nice and AM's estimate of the worth of LEQ rises.

** Task 55 ** Check examples of LEQ, because many examples have recently been found, but not yet checked, and the worth of LEQ has recently risen. All confirmed. *Unfortunately, AM has derived Subset but not Subbag, else it might have noticed that (for all numbers x and y, represented as bags of T's) x≤y iff Subbag(x,y). Then AM could simply observe that LEQ was just Subbag restricted to Numbers. Looked at another way, AM has here discovered a restricted version of the concept Subbag.*

** Task 56 ** Apply algorithm of Coalesce to LEQ, because LEQ is an interesting operation, recently created, many examples already exist for it, AM just worked on LEQ, LEQ takes two of the same argument (Numbers), and no examples of Coalesce are known yet. The new predicate is defined as λ(x) x≤x. But this is Always-True, so AM conjectures that each number is LEQ itself, and forgets the new coalesced version of LEQ.

** Task 57 ** Fill in examples of Parallel-join2, because none exist yet. Included is Parallel-join2(Bags,Bags,Proj2) (initially called MJ2-BBP2), which turns out to be multiplication of two numbers and is renamed "TIMES2" by the user. Also included is Parallel-join2(Structures,Structures,Proj1), which is a generalized kind of Union operation (renamed "G-Union" by the user). Many losers are also created, however, like Parallel-join2(Bags,Sets,Set-difference)).

** Task 58 ** , - 69. Fill in and check examples of the operations just createad. Nothing out of the ordinary is done here, just the routine legwork of gathering empirical data for later use. No startling conjectures made.

** Task 70 ** Fill in examples of Coalesce, because none exist yet. One shining example is Self-compose, which takes any operation F (whose range is also a domain component) and forms FoF. Another example is Self-Insert, which takes a structure S and inserts S into S. Also created were: Self-Delete, Self-Add, Self-Times, Self-Union, etc. A different kind of coalescing was done for Parallel-replace2, Parallel-join2, and Repeat2; the two structural arguments (the first and second arguments for each) were merged, creating three new operations: Coa-repeat2, Coa-join2, Coa-replace2. Coa-replace2, for example, takes a single structure S and an operation F, and replaces each member x of S by the value F(x,S).

** Task 71 ** Fill in examples of Self-Delete, because none exist yet, and Self-delete was just created. Many examples are found quite easily, of course, except:

** Task 72 ** Check examples of Self-Delete, because many examples have recently been found, but not yet checked. Since trying to delete S from S will never work, the value of Delete(S,S) is just S all the time. Self-delete is the same as the identity operation. AM is able to discover this and state it as a conjecture, obviating the need for bothering with this concept ever again.

** Task 73 ** Fill in examples of Self-Member, because none exist yet, and Self-member was recently created. Only negative instances are found.

** Task 74 ** Check examples of Self-Member, because many examples have recently been found, but not yet checked. This predicate is Always-False, empirically. Replace by a conjec: Self-Member is the same as the predicate Always-False; Member(S,S)=False for any S. Also, an extra algorithm entry is added to Member.Alg: Once Early Quick: λ (x,y) if x=y then False. Also, a new task is proposed, to generalize Self-Member. This never quite rises to the top of theagenda, so it is never attempted.

** Task 75 ** Fill in examples of Self-Add, because none exist yet. Many found. User renames this "Doubling", after he observes the many examples which are produced.

** Task 76 ** Check examples of Coalesce, because many examples have recently been found, but not yet checked. All were confirmed. Some were already proving to be interesting, so the value of Coalesce was raised.

** Task 77 ** Check examples of Add2, because many examples have recently been found, but not yet checked. All were confirmed. Somewhat disappointingly, AM didn't notice anything special about Add2 at the time.

** Task 78 ** Fill in examples of Self-Times, because none exist yet, and AM recently worked on Isa(Self-Times) [namely, worked on Coalesce]. Renamed "Squaring" by the user. 20 examples found before quitting due to lack of alotted space.

** Task 79 ** Fill in examples of Self-Compose, because none exist yet. Created Add2∘Add2 (two versions: Add21 which is λ (x,y,z) (x+y)+z, and Add22 which is x+(y+z)). Similarly, two versions of TIMES2∘TIMES2, called TIMES21 and TIMES22. Also, two versions of Compose∘Compose. Some losers were defined as well, like Member(Member(x,y),z) and Parallel-join2(S,R,Parallel-join2(P,Q,F)) – the latter of which accepts as arguments four kinds of structures and a function name. Many minimally-acceptable concepts were created: Coalesce∘Coalesce, Squaring∘Squaring, Doubling∘Doubling, etc.

** Task 80 ** Fill in examples of Add21, because none exist yet, and Add21 was just created. This operation is defined as λ (x,y,z) (x+y)+z. It is easy to find examples.

** Task 81 ** Fill in examples of Add22, because none exist yet, and Add22 was recently created. This operation is defined as λ (x,y,z) x+(y+z). It is easy to find examples. Most of these examples are gotten from the "cousin" concept Add21.

** Task 82 ** Check examples of Squaring, because many examples have recently been found, but not yet checked. All confirmed. *It is unfortunate that this task intruded into AM's line of development.*

** Task 83 ** Check examples of Add22, because many examples have recently been found, but not yet checked. During this process, AM notices that Add21 and Add22 seem to be equivalent. Before conjecturing this, though, AM will do this next task:

** Task 84 ** Check examples of Add21, because many examples have recently been found, but not yet checked, and this task was specifcally suggested while AM was trying to check examples of Add21. After checking these examples, Add21 and Add22 still appear equivalent. AM conjectures this and merges the two operations. One consequence is the boosting of the worth of the new, combined operation. The most important aftereffect of this is that AM now knows that the "proper" argument for a generalized kind of addition will be a Bag, not a List, of numbers. This new kind of addition is called Add, to distinguish it from Add2. Add2 takes a pair of numbers and adds them, but Add accepts a *bag* of numbers and forms their sum.

** Task 85 ** Apply algorithm for Invert to argument 'Add', because Add is interesting, recently worked on, and has never been inverted, and there are no examples yet for Invert, and the worth of Add has recently risen, and Add was just created. *By looking at those reasons, we see why some semantic processing should be available. There is tremendous overlap there, and the task is not really supported by as many reasons as AM thinks.* AM defines Inv-add(x) (also called Add^{-1}) as the set of all bags of numbers (>0) whose sum is x.

** Task 86 ** Fill in examples of TIMES21, because none exist yet. Defined as (x•y)•z. Many are found.

** Task 87 ** Fill in examples of TIMES22, because none exist yet. Defined as x•(y•z). Many are found.

** Task 88 ** Check examples of TIMES22, because many examples have recently been found, but not yet checked. As with Add, earlier, TIMES21 and TIMES22 now appear equivalent. Before saying this, AM must do this task:

** Task 89 ** Check examples of TIMES21, because many examples have recently been found, but not yet checked, and this task was specifically suggested while AM was trying to check examples of TIMES22. After checking these examples, TIMES21 and TIMES22 still appear equivalent. AM conjectures this and merges the two operations. One consequence is the boosting of the worth of the new, combined operation. The most important aftereffect of this is that AM now knows that the "proper" argument for a generalized kind of product will be a Bag, not a List, of numbers. This new kind of multiplication is called TIMES, to distinguish it from TIMES2. Notice the same property held true for Add2, earlier. AM sets up an analogy between TIMES and ADD, because of this common fact. *The analogy itself is close to what mathematicians call the concept of Semigroups.*

** Task 90 ** Apply algorithm for Invert to argument 'TIMES', because TIMES contains a new, promising analogy, and the analog of TIMES has been inverted, and TIMES has never been inverted, and the worth of TIMES has recently risen, and TIMES was just created. AM defines Inv-TIMES(x) (also called $TIMES^{-1}$) as the set of all bags of numbers (>1) whose product is x. AM noted that $TIMES^{-1}$ should probably be analogic to Add^{-1}.

** Task 91 ** Fill in examples of Parallel-replace2, because none exist yet. *I could kick AM for doing this now!! The priority rating of this task happened to place it above all the others, including those with extra bonusses because of focus of attention. This task is merely diverting, not harmful in any lasting sense, but it does degrade the apparent level of purposefulenss of the system.* Several examples of Parallel-replace2 are found. Included are Parallel-replace2(Bags,Bags,Proj2) (called MR2-BBP2), and many losers. MR2-BBP2(S1,S2) replaces each element in S1 by a full copy of the whole of S2. *This is the way that Skemp suggests developing the notion of multiplication -- and in fact AM will (in task 109) derives an operation which is equivalent to TIMES2 just by unioning the results of this operation. In task 127 AM realizes that this is in fact just multiplication, and merges those two operations, concurrently boosting the worth of that combined concept greatly.*

** Task 92 ** , - 107. Fill in and check examples of the operations just created. *Nothing really worth our time (or AM's). Sigh.*

** Task 108 ** Fill in examples of Compose, because none exist yet. It is very easy to create new compositions -- most of them losers. Some of the concepts produced (e.g., Size∘Add^{-1}) were valuable but were lost amid the mass of losers (e.g., Insert∘Equal). Because of this flood of poorly-motivated new concepts, a heuristic triggers which has AM create a new specialization of Compose, called Int-Compose, by conjoining onto Compose.Defn a few of the features from Compose.Interest. The Worths of the new compositions just created are all lowered, so that the (future) examples of Int-Compose will predominate. *The task first considered in TASK 1 has finally bubbled back up to the top of the agenda, and has proved to be quite worthwhile.*

** Task 109 ** Fill in examples of Int-Compose, because none exist yet, and Int-Compose was just created, and any example of Int-Compose is automatically an interesting example of Compose, and the worth of Int-Compose is very high. The two chosen operations G,H must be such that ran(H)∈dom(G), and ran(G)∈dom(H); both G and H must be interesting or at least newly-created. Well, two operations recently dealt with are G-Union and MR2-BBP2. Since the range of MR2-BBP2 is 'Bags of Bags', it is *precisely* equal to the domain of the newly-synthesized operation G-Union. So one composition considered is G-Union∘MR2-BBP2. This is an alternate derivation of the operation of multiplication. Also included are: TIMES∘Squaring, Coalesce∘Compose, Insert∘Delete, Delete∘Insert, Add2∘Times2, etc. Although most of these operations were never investigated very much, they are much better than the random compositions produced during the previous task. This seems clear even to AM, even before studying them very much.

** Task 110 ** , - 126. Fill in and check examples of the compositions just createad. Nothing of great interest until...:

** Task 127 ** Check examples of G-Union∘MR2-BBP2, because many examples have recently been found, but not yet checked. AM discovers that this operation is equivalent to MJ2-BBP2 (i.e., TIMES2). Since they arose in very different ways, the worth of the new, merged concept module is greatly increased, as is that of the more general operation TIMES.

** Task 128 ** Fill in examples of Coa-repeat2, because none exist yet. 26 operations synthesized. Foremost among them was Coa-repeat2(Bags-of-Numbers,Add2), which turned out to be yet another derivation of multiplication! Also produced was Coa-repeat2(Bags-of-Numbers,Times) – a definition of exponentiaion. Others included Coa-repeat2(Structures,Proj1), which turns out to be the same as First-element-of, i.e., CAR, and Coa-repeat2(Structures,Proj2), which turns out to be Last-element-of.

** Task 129 ** Check the examples of Coa-repeat2, because many examples have recently been found, but not yet checked, and Focus of Attention. All confirmed.

** Task 130 ** Apply algorithms for Invert to 'Doubling', Doubling is interesting, and it has never been inverted. The result is called "Halving" by the user. AM decided to isolate the domain of Halving (the range of Doubling). Such numbers are renamed by the user as "Evens". *Although pleased with the result of this task, it was somewhat jarring in the context of the preceding development.*

** Task 131 ** Fill in examples of Self-Insert, because none exist yet. *AM has apparently lost the "thread" of a development and is wandering around, taking care of only moderately promising tasks.* Many examples of this operation are found.

** Task 132 ** Check examples of Self-Insert, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing special found. The result is never the same as the argument.

** Task 133 ** Fill in examples of Coa-repeat2-Add2, because none exist yet. Many found quickly, but at a large cost in terms of storage space.

** Task 134 ** Check examples of Coa-repeat2-Add2, because many examples have recently been found, but not yet checked, and Focus of Attention. Confirmed. AM noticed it's the same as TIMES. Boost the worth of TIMES even higher, far above that of any other concept. *AM will stay interested in TIMES for most of the future of this run.*

** Task 135 ** Apply algorithm for Invert to argument 'Squaring', Squaring is interesting, has never been inverted, is related to the very interesting concept TIMES, is related to the very interesting concept Coalesce, is analogic to the already-inverted concept Doubling. AM defines Inv-square(x) as the number[8] whose square is x. Renamed by user as "Square-root".

** Task 136 ** Fill in examples of Square-root, because none exist yet, and Square-root was just created, and Square-root is related to the very interesting concept TIMES. AM spent quite a while on this task, and only about 10 examples were found (discounting duplicates).

---

[8] Actually: the set of all numbers.

** Task 137 ** Fill in new algorithms for Square-root, because Square-root.Algs are all too slow, and have been called on a great deal recently. *AM has some zeroth order rules for improving algorithms, backed up by a marvelous tactic: ask the user. In this case, AM asks the user for a better algorithm, and he supplies one. Of course, the new algorithm is completely opaque to AM. The user never tells AM how to do something unless it had a (slow) way to do that thing already.* One fast new algorithm now exists.

** Task 138 ** Check examples of Square-root, because many examples have recently been found, but not yet checked, and Focus of Attention. AM is plagued by the frequency of numbers not having square-roots, so it isolates those that do. It defined the set of numbers having a square-root, and this concept was renamed "Perfect-squares" by the user.

** Task 139 ** Fill in examples of Coa-repeat2-Times, because none exist yet, and this concept is related to the very interesting concept TIMES. *A moderately rational thing to investigate.* Examples are easily found, but they take up a lot of space.

** Task 140 ** Check examples of Coa-repeat2-Times, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing special noticed, unfortunately (this is exponentiation, folks). *If the user interrupts and tells AM that this is really interesting, AM soon creates the specialization of it defined as Expon(x,2), and then AM notices that this is just squaring. I.e., $x^2=x*x$: the base tie between exponentiation and multiplication. On its own, AM doesn't rate Coa-repeat2-Times high enough to start this chain of discoveries.*

** Task 141 ** Fill in examples of Inv-TIMES, because none exist yet, and Inv-TIMES is related to the very interesting concept TIMES. Many found.

** Task 142 ** Fill in new algorithms for Inv-TIMES, because Inv-TIMES.Algs are all too slow, and have been called on a great deal recently, and $TIMES^{-1}$ is related to the very interesting concept TIMES, and Focus of Attention. AM asks the user, who supplies a decent recursive algorithm for this function.

** Task 143 ** Check examples of Inv-TIMES, because many examples have recently been found, but not yet checked, and Focus of Attention. This proceeds along, and all are confirmed. A heuristic rule notices that the domain/range is <Number → Sets-of-Bags-of-Numbers>; it searchs for an operation whose Domain/range facet contains an entry (compatible with) <Sets-of-Bags-of-Numbers → Number>, and fails; next it looks for an operation whose dom/range is <Sets-of-Bags → Set or Bag>, and finds that G-Union fills the bill. Therefore, the rule suggests the following task (with a high rating):

** Task 144 ** Apply Compose algorithm to G-Union and Inv-TIMES, because the three concepts involved are interesting, related to TIMES, and this task was specifically suggested by the preceding one. The composition is created, as specified in the task. This new operation has domain/range <Number → Set-of-Numbers>, and is thus given a higher rating than either of its constituents. It is renamed "Divisors" by the user.

** Task 145 ** Fill in examples of Divisors, because none exist yet, and Divisors is related to the very interesting concept TIMES, and divisors was just created. Many examples are found, but only after much inefficient searching amid the set of all (known examples of) numbers.

** Task 146 ** Fill in new algorithms for Divisors, because Divisors.Algs are all too slow, and have been called on a great deal recently. AM asks the user, who supplies a decent iterative algorithm for this function.

** Task 147 ** Fill in examples of Perfect-squares, because none exist yet, and Perfect-squares is related to the very interesting concept TIMES. 15 found, after which the space allocation was exhausted.

** Task 148 ** Fill in specializations of TIMES, because TIMES is very interesting, has very few known specializations, and it was very easy to find examples of TIMES. AM now allocates a huge chunk of cpu time and space to this task. A few specializations of TIMES are gotten by plugging in a distinguished value for one argument: Times1(x)=1•x, Times0(x)=0•x, etc. Other new operations are simply TIMES with its domain restricted to a bag of special numbers: Times-sq has its domain a bag of perfect squares, Times-ev takes only even arguments, etc. Others (inefficient to compute) are TIMES with its range restricted: Times-to-evens requires that the result be even, Times-to-sq for square results, etc.

** Task 149 ** Check examples of Divisors, because many examples have recently been found, but not yet checked, and Divisors is related to the very interesting concept TIMES. Often, Divisors(x) is interesting (to AM) as a set; AM isolates the cases by defining 0-Div, 1-Div, 2-Div, and 3-Div, the sets of numbers whose Divisors value is the empty set, a singleton, a doubleton, and a tripleton, respectively. AM will gradually partition the examples of Divisors into these categories, as AM tries to fill in examples of each kind of number.

This is the point where the example in Chapter 2 begins, and is also roughly the point where the unadulterated LISP trace (Appendix 5.3) ends. Both this section and the earlier condensed task-by-task trace, found in Chapter 6, go further.

** Task 150 ** Fill in examples of 1-Div, because none exist yet, and 1-Div was just created, and is related to the very interesting concept TIMES. Only one example found: "1". This causes the Worth of 1-Div to be lowered.

** Task 151 ** Fill in examples of 0-Div, because none exist yet, and 0-Div was recently created, and is related to the very interesting concept TIMES. None found. Lower the worth of this concept.

** Task 152 ** Fill in examples of 2-Div, because none exist yet, and this concept is related to the very interesting concept TIMES. About 19 are found (out of about 170 attempts). This is a nice ratio, a nice density within the natural numbers – not too many nor too few to be interesting. As a result, 2-Div.Worth is slightly raised.

** Task 153 ** Check examples of 2-Div, because many examples have recently been found, but not yet checked, and the worth of 2-div has just increased, and Focus of Attention. The existing examples were confirmed, but no pattern noticed. *This was heart-stopping, since 2-Div is the notion of prime numbers; here AM is tossing it off as non-interest-catching!*

** Task 154 ** Fill in examples of 3-Div, because none exist yet, and 3-Div is related to the very interesting concept TIMES. As with 2-Div, a nice number of examples were found (albeit on the scarce side of nice).

** Task 155 ** Check examples of 3-Div, because many examples have recently been found, but not yet checked, and 3-Div is related to the very interesting concept TIMES., and Focus of Attention. All confirmed. All are perfect squares! Very unexpected (both by AM and the user). AM greatly increased the worth of 3-Div. One suggestion, due to the fact that 3-Div was now In-dom-of Square-root, was:

** Task 156 ** Restrict Square-root to numbers which are in 3-Div, Square-root is interesting, 3-Div is very interesting, and the preceding task specifically requested this action. AM calls the new concept Root3.

** Task 157 ** Fill in examples of Root3, because none exist yet, and Root3 was just created, and Root3 is related to the very interesting concept 3-Div. Many examples found. In fact, it was easy to take the square-root of each known example of 3-Div.

** Task 158 ** Check examples of Root3, because many examples have recently been found, but not yet checked. All confirmed. Each result turned out to be a 2-Div type of number. Very surprising. Conjecture: the square-root of a number with 3 divisors is a number with 2 divisors. AM raised the worths of all the concepts involved. At this point, the user renamed 2-Divs as "Primes".

** Task 159 ** Restrict Squaring to Primes, Squaring is interesting, Primes is recently interesting, and the preceding task specifically suggested this action. AM calls the result Square2.

** Task 160 ** Fill in examples of Square2, because none exist yet, and Square2 was just created. Many found.

** Task 161 ** Check the domain/range facet of Square2, it has been specifically suggested that the range of Square2 may be 3-Div, and 3-Div is very interesting, and Square2 is related to the interesting concept Primes, and Focus of Attention. As hoped for, all are 3-Divs. Conjecture: x is a Primes iff its square is a 3-Div iff it is the square-root of a 3-Div.

** Task 162 ** Restrict Squaring to 3-Divs, Squaring is interesting, 3-Div is interesting, and an earlier task specifically suggested this action. The result is called Square3. *AM's past few successful tasks have now incremented the Worths of certain activities above their true value: AM will now be tied up with restricting Squaring and Square-rooting to all the concepts involved. The net effect will be to lower those inflated worth ratings, and to lower the user's — and the reader's — opinion of AM.*

** Task 163 ** Restrict Square-rooting to Primes. Square-root is interesting, Primes is interesting, and an earlier task specifically requestd this action. Call the result Root2.

** Task 164 ** Fill in examples of Square3, because none exist yet, and Square3 is related to the interesting concept 3-Div, and Square3 was recently created. Only 9 examples found before running out of space. By analogy, since Divisors-of∘Square2 was interesting, AM considers:

** Task 165 ** Compose Divisors-of and Square3. Analogic to tripletons, and Divisors-of is interesting, and Square3 is interesting, and the preceding task specifically suggested this action. AM calls the result Div-Sq3.

** Task 166 ** Fill in examples of Div-Sq3, because none exist yet, and Div-Sq3 was just created. 9 examples found right away, by simply running Divisors-of.Algs on the 9 known examples of Square3.

** Task 167 ** Check examples of Div-Sq3, because many examples have recently been found, but not yet checked. All such examples are Same-size. Although AM doesn't have the notion of '5-ness' explicitly, they each have 5 members. A specialized hack heuristic observes the general pattern: Divisors∘Primes are all of the same size; Divisors∘Squaring∘Primes are all of the same size; Divisors∘Squaring∘Squaring∘Primes are all of the same size; A new conjecture is formulated and typed to the user: Divisors∘Repeat(Squaring)∘Primes will all be the same size.

> *This expresses the fact that, for a given n, p^{2^n} has the same number of divisors for each prime p. AM was not able to figure out that number of divisors (it is 2n+1). This would be a trivial sequence extrapolation problem, but AM of course had no heuristics for dealing with numbers, hence no sequence extrapolation techniques. Deriving the concept of sequence extrapolation itself would have been quite astounding, but never occurred. Discovering the concept of inductive inference and studying it explicitly in isolation is quite a sophisticated achievement — that's what AI researchers spend much of their time trying to accomplish. This is one time when AM was much further from discovering a theorem than it appeared to the casual observor.*

** Task 168 ** , - 175. More confirmations and explorations of the above conjecture. Gradually, all its ramifications lead to dead-ends (as far as AM is concerned).

** Task 176 ** Fill in examples of Root2, because none exist yet, and Root2 is related to the interesting concept Primes. But no examples at all are found. This is not surprising, since very few primes are also perfect squares. AM conjectures that there are none. Worth of Root2 is lowered.

** Task 177 ** Check examples of Inv-TIMES, because many examples have recently been found, but not yet checked. This is a break in the previous smooth line of development. Inv-TIMES appears to always contain a singleton bag; in fact, Inv-TIMES(x) always contains the singleton bag (x). Another conjecture AM makes: Inv-TIMES(x) always contains a bag of primes. This last hypothesis suggests the following two tasks:

** Task 178 ** Restrict the range of Inv-TIMES to bags of primes, because Inv-TIMES is interesting, is related to the very interesting concept TIMES, primes are more interesting than numbers at the moment, Focus of Attention, and the previous task specifically suggested this action. AM calls the new result Prime-Times.

** Task 179 ** Restrict the range of Inv-TIMES to singletons, because Inv-TIMES is interesting, is related to the very interesting concept TIMES, singletons are more interesting than sets at the moment, a recent task specifically suggested this action, and Focus of Attention. AM calls the new result Single-Times.

** Task 180 ** Fill in examples of Prime-times, because none exist yet, and Prime-times was recently defined, and Prime-times is related to the interesting concept Primes, and Prime-times is related to the interesting concept TIMES. Many examples are found.

** Task 181 ** Check examples of Prime-times, because many examples have recently been found, but not yet checked, and Focus of Attention, and Prime-times is related to the very interesting concept TIMES. The value of Prime-times(x) is always a singleton set. Conjecture: Inv-TIMES(x) contains precisely one bag of primes. User renames this conjecture "The unique factorization theorem". AM prints out that this will probably be very natural and important. The reason for this is that Primes was itself derived from TIMES, so any conjecture connecting them is quite natural. Any *unexpected* such natural conjecture will probably be useful.

** Task 182 ** Fill in examples of Single-TIMES, because none exist yet, and this concept is related to the very interesting concept TIMES. Many found.

** Task 183 ** Check examples of Single-TIMES, because many examples have recently been found, but not yet checked, and it is related to the very interesting concept TIMES, and Focus of Attention. The value of Single-times(x) is always a singleton set. Conjecture: Inv-TIMES(x) contains precisely one singleton bag. Single-TIMES is actually the same as Bag-insert, in the sense that both Single-TIMES(x) and Bag-insert(x) give the value (x) – the bag containing only x. In the latter case, this is because Bag-insert is "smart" enough to supply an empty bag as the second argument S to Bag-insert(x,S), if S is missing.

** Task 184 ** Fill in examples of Self-set-union, because none exist yet. *AM has dropped the momentum of its previous whirlwind of discovery, and is simply marking time, gathering evidence.* Many examples are found.

** Task 185 ** Check examples of Self-set-union, because many examples have recently been found, but not yet checked, and Focus of Attention. Apparently, this concept is the same as Identity (but with domain/range restricted to Sets). Replace by a conjecture.

** Task 186 ** Fill in examples of Self-bag-union, because none exist yet. *On the same track, but boring.* Many found.

** Task 187 ** Check examples of Self-bag-union, because many examples have recently been found, but not yet checked, and Focus of Attention. All are confirmed. Nothing interesting noticed.

** Task 188 ** Fill in examples of Inv-ADD, because none exist yet. Slowly, examples were found.

** Task 189 ** Check examples of Inv-ADD, because many examples have recently been found, but not yet checked, and Focus of Attention. Inv-ADD(x) always contains a singleton bag (x), a doubleton bag, a tripleton, a bag of 1's,... So many conjectures that:

** Task 190 ** Restrict the domain of Inv-ADD, because Inv-Add is interesting, related to the interesting concept Add, Focus of Attention, and the previous task specifically suggested this action. When the domain is restricted to primes, AM defines 'Inv-Add-primes'. When it restricts Inv-Add to work only on evens, AM thereby defines the operation it calls 'Inv-Add-evens'.

** Task 191 ** Fill in examples of Inv-add-primes, because none exist yet, and this concept was just defined, and Focus of Attention. Many found.

** Task 192 ** Check examples of Inv-add-primes, because many examples have recently been found, but not yet checked, and Focus of Attention. All were confirmed, but nothing special noticed.

** Task 193 ** Fill in examples of Inv-add-evens, because none exist yet, and this concept was recently defined. Many examples found.

** Task 194 ** Check examples of Inv-add-evens, because many examples have recently been found, but not yet checked, and Focus of Attention. Confirmed. Inv-Add-evens(x) always contains a bag of primes. This is mildly surprising, and prompts:

** Task 195 ** Restrict the range of Inv-Add-evens to bags of primes, because Inv-Add-evens is recently interesting, and Primes is more interesting than Numbers, and the previous task specifically requested this action (hence Focus of Attention). AM names the new operation Prime-ADD.

** Task 196 ** Restrict the range of Inv-ADD to singletons, because Inv-Add is interesting, singletons are more interesting than sets, AM just worked on Inv-Add, AM recently worked on Inv-Add, and an earlier task specifically suggested this action. Thus Single-Add is born.

** Task 197 ** Fill in examples of Prime-ADD, because none exist yet, and Prime-add was recently defined. Many found.

** Task 198 ** Check examples of Prime-ADD, because many examples have recently been found, but not yet checked, and Focus of Attention. The value of Prime-ADD(x) is always a nonempty set (of bags of primes). So conjectured (domain/range changed). User renames this conjecture "Goldbach's conjecture".

** Task 199 ** Fill in examples of Single-ADD, because none exist yet, and this concept was recently defined. Many found.

** Task 200 ** Check examples of Single-ADD, because many examples have recently been found, but not yet checked, and Focus of Attention. The value of Single!ADD(x) is always a singleton set. Conjecture: Inv-ADD(x) contains precisely one singleton bag. Single-ADD is actually the same as Bag-insert (and Single-TIMES).

** Task 201 ** Restrict the range of Prime-ADD to singletons, because analogic to Prime-TIMES, and Prime-Add is interesting, and Singletons is more interesting than Sets. This was initiated by analogy, not by an earlier task specifically suggesting that the restriction be done. In this case, AM is asking which numbers are uniquely representable as the sum of two primes. The new operation is Prime-ADD-s.

** Task 202 ** Fill in examples of Prime-ADD-s, because none exist yet, and Prime-ADD-s was just defined. Many examples are found, but after a nontrivial processing effort.

** Task 203 ** Check examples of Prime-ADD-s, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing special noticed.

** Task 204 ** Fill in examples of Times-sq, because none exist yet. *Losing the thread of discovery, moving back to data-gathering blindness.* Recall that Times-sq is just TIMES restricted to operate on perfect squares. Many examples found.

** Task 205 ** Check domain/range of Times-sq, because the range of this operation may actually be Perfect-squares, and examples of Times-sq were just filled in, and this concept is related to the very interesting concept TIMES, and Focus of Attention. The range really does seem to be as hoped for. Conjecture: the product of perfect squares is a perfect square.

** Task 206 ** Fill in examples of Times1, because none exist yet. Recall that Times1(x)=TIMES(1,x). Many found.

** Task 207 ** Check examples of Times1, because many examples have recently been found, but not yet checked, and Focus of Attention. Apparently Times1 is just a restriction of Identity. Times1 is therefore replaced by a lone conjecture: Times(x,1)=x.

** Task 208 ** Check examples of Times-sq, because many examples have recently been found, but not yet checked. Confirmed.

** Task 209 ** Fill in examples of Times0, because none exist yet, and Times0 is related to the very interesting concept TIMES. Many found.

** Task 210 ** Fill in examples of Times2', because none exist yet, and this operation is related to the very interesting concept TIMES. Many found. Recall that Times2(x) is defined as 2*x.

** Task 211 ** Check examples of Times2', because many examples have recently been found, but not yet checked, and Focus of Attention. Apparently, Times2' is the same as Doubling. That is, x+x=2*x. A very powerful tie between Add and Times! This was highly unexpected. It is not predicted by the existing analogy. By analogy, AM now defines Ad2(x) as x+2, and will invesitigate that.

** Task 212 ** Fill in examples of Ad2, because none exist yet, and Ad2 was just created. Many examples found.

** Task 213 ** Check examples of Ad2, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing interesting noticed. *AM didn't have the notion of Add1∘Add1 at that moment, or it could have derived the analogic conjecture between successor/addition that it found between addition/multiplication. The same lack of knowledge about exponentiation inhibited the perception the times/exponent analogic relationship. Every little bit of knowledg about operations involving Add served to raise the worth of Add slightly. Finally, the following task rises to the top:*

** Task 214 ** Fill in specializations of Add, because there are very many examples of Add, and Add has recently risen in interest. Among those created are: Add0 (x+0), Add1, Add3, Add-sq (addition restricted to perfect squares), Add-ev (sum of even numbers), Add-pr (sum of primes), etc. The techniques used were the same ones used to specialize TIMES earlier.

** Task 215 ** Check examples of Times0, because many examples have recently been found, but not yet checked. The value always seems to be 0. So conjectured. Times0 goes away.

** Task 216 ** Fill in examples of Times-ev, because none exist yet. Recall that Times-ev is just like TIMES restricted to operating on even numbers. Many examples found.

** Task 217 ** Check examples of Times-ev, because many examples have recently been found, but not yet checked, and Focus of Attention. Apparently all the results are all Evens. Conjecture: the product of even numbers is even. *Note that no explicit task of the form "Check D/R" had to be executed in order to notice and correct this. The net result here is the same: a conjecture and an alteration of the Domain/range facet of Times-ev.*

** Task 218 ** Fill in examples of Times-to-ev, because none exist yet. That is, consider bags of numbers which multiply to give an even number. Many found.

** Task 219 ** Fill in examples of Times-to-sq, because none exist yet. Find bags of numbers whose product is a perfect square. A few found right away, then great difficulty in locating more examples.

** Task 220 ** Check examples of Times-to-sq, because many examples have recently been found, but not yet checked, and Focus of Attention. All arguments always seem to be squares. Conjec: Times-to-sq is really the same as Times-sq. Merge the two. *This is a false conjecture, since, e.g., the product of the numbers in the bag (2 2 3 3) is a perfect square, but none of those numbers is itself a square. This did AM no harm, and AM never detected its mistake.*

** Task 221 ** Check examples of Times-to-ev, because many examples have recently been found, but not yet checked. The domain always seems to contain an even number. So conjectured. *I.e., if the product of some numbers is even, then so is one of those numbers. This conjecture is true, at least.*

** Task 222 ** Fill in examples of Self-Union, because none exist yet. *Reaching way back in time. Ugh.* Many found.

** Task 223 ** Check examples of Self-Union, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing special noticed.

** Task 224 ** Fill in examples of SubSet, because none exist yet. Many found.

** Task 225 ** Check example of SubSet, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing special noticed.

** Task 226 ** Fill in examples of SuperSet, because none exist yet. Many found.

** Task 227 ** Check examples of SuperSet., because many examples have recently been found, but not yet checked, and Focus of Attention. AM notices that if <x,y> are related by SubSet, then Reverse-ord-pair(<x,y>) are related by SuperSet, and conversely. This is the base connection between union and intersection (see Tasks 29 and 39, where these two concepts are defined). That is, $x \subset y$ iff $y \supset x$.

** Task 228 ** Fill in examples of Compose∘Compose-1, because none exist yet. AM creates some poor combinations (e.g., Square∘Count∘ADD^{-1}), some explosive ones (e.g., (Compose∘Compose)∘(Compose∘Compose)∘(Compose∘Compose)), and even a few – very few – winners (e.g., SUB1∘Count∘Self-Insert). *This is too much like throwing "flying, hooked atoms" up into the air, and hoping that three of them collide fortuitously. While a little guidance may help you to find good collisions of 2 such fliers, the combinatorial explosion swamps the poor researcher when he takes them on three at a time. As St. Augustine observed, the Latin 'cogito' derives from 'shake together', but 'intelligo' derives from 'select among'.*

** Task 229 ** Check examples of Compose∘Compose-1, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing interesting to find.

** Task 230 ** Fill in examples of Compose∘Compose-2, because none exist yet. Recall that the difference between this operation and the last one is merely in the order of the composing: F∘(G∘H) versus (F∘G)∘H. AM recreates many of the previous tasks' operations.

** Task 231 ** Check examples of Compose∘Compose-2, because many examples have recently been found, but not yet checked, and Focus of Attention. Nothing noticed yet. *Later on, AM finds that one after another of the operations created in the preceding task as, say, Compose∘Compose-1(F,G,H), is really the same as the corresponding operation created as Compose∘Compose-2(F,G,H). Eventually, AM conjectures that those two Compose∘Compose operations are really the same; that is, Compose is associative.*

** Task 232 ** , - 252. Fill in and check examples of the losing compositions just created.

** Task 253 ** Fill in examples of Add-sq, because none exist yet, and Add-sq is related to the interesting concept Add. Recall that Add-sq is just addition, restricted to perfect squares. Many examples found.

** Task 254 ** Check domain/range entries of Add-sq, because the range may also be Perfect-squares, and examples of Add-sq were just filled in (hence Focus of Attention), and Add-sq is related to the interesting concept Add. The range isn't Perfect-squares (e.g., 4+9 is not square), but some values are, so AM defines the predicate Add-sq-sq(x,y), which is True iff x and y are perfect squares and their sum is a perfect square as well (e.g., Add-sq-sq(16,9)). Add-sq-sq.Defn is a predicate which is true if its 3 arguments are squares, say x^2, y^2, z^2, and if the sum of the first two is equal to the third: $x^2+y^2=z^2$.

** Task 255 ** Fill in examples of Add-pr, because none exist yet, and Add-pr is related to the interesting concept Add. That is, the sum of a pair of primes.

** Task 256 ** Check Domain/range entries of Add-pr, because many examples have recently been found, but not yet checked. AM defines the set of pairs of primes whose sum is also a prime (e.g., Add-pr-pr(2,5)). *In a rather bizarre way, AM has defined prime pairs. The sum of two primes can be a prime iff one of them is 2. So Add-pr-pr can really be considered a predicate on one prime argument x, which returns True iff x+2 is a prime; i.e., iff x is the lower member of a prime pair. There is something at once awful and sublime about this derivation of prime pairs. Perhaps this captures the spirit of AM's actions as a whole, so let's stop this trace right here.*

Appendix 5.3. An 'Unadulterated' Trace

Here is the way that the AM program begins. The human user's typing will appear in italics[9]. He first types *(START)* to start the system, after which AM asks him some questions. Finally, the main Select&Execute-a-TASK loop of AM is entered.

The careful reader will notice several small changes in this transcript, compared to the nicely doctored ones which preceded it. For one thing, the task numbering here is not precisely the same as in the rest of this document. A task is called a "Cand", and the agenda is called "CANDS". Only some of the reasons are printed out, and they are not as "chatty" as the reasons in. e.g., Chapter 2's example trace. The user has asked AM to type out the top three tasks on the agenda at each "cycle". In a better hardware environment, the user could dynamically watch the top hundred tasks bubbling around on one side of a CRT screen. To interrupt AM, the user types CONTROL-I. At that moment he has a very limited syntax of questions he may ask. See (α) below (page 319).

An approximate level of familiarity of the user with the AM program is maintained by AM, as a numeric variable. Initially, its value is determined by the number of times the human user has used AM in the past.[10] It gradually changes in value as a single session proceeds. Many print statements use this variable to determine the necessary level of detail to type. For example, contrast the line pointed to by an arrow labelled (β) below with the line labelled (ε). In between, the variable increased to the point where a detailed message

---

[9] This is not a doctoring: I have written an i/o routine for AM which prints Σ4 before everything the user types, and Σ1 afterwards. The 'PUB' documentation program interprets this to mean "switch to font 4" and "switch back to font 1". This document was PUBBed with font 4 defined as italics.

[10] I shall resist the temptation to call this a simple "user model", even in a footnote.

was thought to be superfluous. The level of detail needed for clarity should not be confused with the level of verbosity of output that is desired. Should the user see every function call, or the results of each task, or just monitor the overall character of what AM is doing? A separate variable is maintained for this verbosity indication. Finally, another partially independent dimension is the amount of control the user wishes to have on AM. Must he approve each move AM makes, occasionally redirect AM, be able to ask questions? Another variable indicates the level of user control in effect at the moment. The first several questions deal with setting up these parameters.

INTERLISP-10 4-JAN-75 ...

Good afternoon, Doug.

←*(START)*

Please type in your last name (then carriage-return):*FEIGENBAUM*

Almost ready to enter AM's main loop, Ed.

Ed, you have used AM once before.
Do you feel like you need some refreshing about how to work with it? *Yes*

AM has 113 concepts to start with, each with only about 5 of its potential 30 facets (parts) filled in.
Repeatedly, AM selects a part of a concept, and tries to fill it in or check it. In this process, new concepts may emerge and be granted full status; in those cases, almost all their parts will be empty at the time of their creation.

CANDS is a list of suggested future activities for AM. Repeatedly, AM picks a Candidate from CANDS and does what it says. A typical Cand might be:
(Fill in examples of interesting compositions).
Each Cand also has a list of reasons explaining why it was proposed, and a numeric rating of its overall value.

DO-THRESHhold is a numeric variable that indicates the lowest rating a Cand may have and still be executed by AM. If no Cand on CANDS measures up, then all the concepts try to suggest new candidates, which are merged into CANDS.

DISCLAIMER: The user interface is still unfinished, and in fact the only polished part is the following message describing it! Try it at your own risk!

You, Ed, can affect this concept growing process.
At any time, you may hit ↑I, which will interrupt me.
Once interrupted, I will answer one question or perform one task;
a typical question I can answer is WHY;
a typical task is (Raise the Interest Level of the Frobnate Concept).
A second way to interact with me is to help me decide

which Cand to do next each time. You can see my top choices, their reasons, and overrule me if you want. The variable Seencands is the number of Candidates you see each time, and Ucontrol indicates the amount of control you have over my choosing.

To keep you informed, I will periodically print out messages. The level of verbosity can be changed by interrupting me.

More details can be obtained when you interrupt with ↑I.

Verbosity Level (1 - 10, or ?) .... (Please terminate your response with a carriage-return) ...?

Verbosity = the amount of data that AM spews out to Ed.
In this system, Verbosity is just a numeric-valued variable
which the Printing functions look at.
Verbosity level 0 suppresses all messages.
Verbosity level 5 lets most important messages get printed.
Verbosity level 10 dumps out enough to actually get some
feeling for the inner workings of AM.

The suggested value for Ed is 8.

Verbosity Level (1 - 10, or ?) .... (Please terminate your response with a carriage-return) ...*10*

User-control Level (0 - 10, or ?) ....?

User-control = the degree to which you supervise AM's activities
In this system, User-control is just a numeric-valued variable
which the top-level control functions look at.
User-control level 0 lets you gain control only via ↑I.
User-control level 5 lets you see a few of AM's alternatives,
and waits 6 seconds for you to OK its choice.
User-control level 10 displays several alternative Candidates,
and waits indefinitely until you to select one.

User-control Level (0 - 10, or ?) *4*

Before deciding which new Cand to work on, I'll print my top choices.

How many Candidates would you like to see each time?
(I suggest 4): *3*

Should I tell you my reasons for each Cand I am considering selecting each time? *No*

Should I tell you my reasons for the Cand I actually select each time? *Yes*

If you have not typed anything within 4 seconds after a prompt, then AM will fill in a default answer for you.
A space will suffice to keep AM from defaulting on you,
while you think about what to reply to any question AM asks you.

In general, your response should be terminated by a carriage return.

Would you like to reset this waiting time? *No*

Entering AM's main loop now.

No Cand on CANDS is good enuf.
Do-thresh reduced from 500 to 333
Must find new candidates and merge them into CANDS.

The top 3 Cands are:
1: Fill in some examples of Set-struc-intersect
2: Fill in some examples of Set-struc-join
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (We have no examples for SET-STRUC-INTERSECT yet)

Beginning 1st cycle.

Failed. Tried to fill in new examples of SET-STRUC-INTERSECT.

<At this moment, the user hit control-I and interrupted AM.>
?: (W, I, E, M, N, ?, Q) ? ←(α)

Here are more detailed explanations of your options:
W Why: AM gives Ed the explanation behind its last printed message.
I Interest: Ed can modify the interest ratings of concepts and Candidates.
E Evaluate: Ed types in an expression and AM runs EVAL on it.
M Message: What was the last message that AM did NOT type out because the verbosity was too low?
N Name: Rename some concept to whatever you want to call it.
Q Quit: resume execution.

In general, AM will automatically resume execution after answering one query. You must hit ↑I again to interrupt.

?: *W*

Why: (No examples of SET-STRUC-INTERSECT were found; there is no reason to even consider specializing it further)

This Cand used 11.159 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Set-struc
2: Fill in some examples of Coalesce
3: Fill in some examples of Nonempty-struc

I choose first Cand. OK? *yes.*

The 2 reasons for considering this Cand are:
(Active-exs specifically asked for some examples of SET-STRUC while trying to Fill in some Set-struc-intersect examples)
(We have no examples for SET-STRUC yet)

Beginning 2nd cycle.

Creating new Being, similar to SET-STRUC, named INT-SET-STRUC, but restricted so as to make it more interesting.
An INT-SET-STRUC is any SET-STRUC for which (Each pair of elements satisfies the same interesting predicate P (for some P)).

Filled in examples of SET-STRUC.
0 examples existed originally on SET-STRUC.
11 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
(CLASS)
(CLASS DOUG CORDELL BRUCE)
(CLASS R0-7 R1-7 R2-7 R3-7 R4-7 R5-7 R6-7 R7-7)
(CLASS A)
(CLASS B)
(CLASS A B)
(CLASS 0 D F I M)
After eliminating duplicate and already-known entries, AM finds that only 7 new, distinct examples of SET-STRUC had to be added.

Do-thresh raised from 332 to 346 because this last Cand succeeded, so we raise our hopes-- and our standards-- temporarily. ←(6)

This Cand used 23.743 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Int-set-struc
2: Fill in some examples of Coalesce
3: Check all examples of Set-struc

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (Any example of INT-SET-STRUC is automatically an interesting example of SET-STRUC)

Beginning 3rd cycle.

Won't try to create a restricted interesting version of INT-SET-STRUC.

Filled in examples of INT-SET-STRUC.
0 examples existed originally on INT-SET-STRUC.
13 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
(CLASS)
(CLASS A)
(CLASS B)
After eliminating duplicate and already-known entries, AM finds that.
only 3 new, distinct examples of INT-SET-STRUC had to be added.

Do-thresh raised from 346 to 358. ←(6)

This Cand used 11.881 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Obj-equal
2: Check all examples of Int-set-struc
3: Check all examples of Set-struc

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (We have no examples for OBJ-EQUAL yet)

Beginning 4th cycle.

Record of attempts to find examples:----------
An ex (sought) is: ((CLASS A),(CLASS A) → T) +----------
----+----------+---
--------+----------+------+---

Found 6 examples (and 151 non-exs), in 11.644 secs.
Ratio of exs to non-exs is too low (6 / 151); Exs are too sparse.
AM will sometime try to generalize OBJ-EQUAL.
Won't try to create a restricted interesting version of OBJ-EQUAL.

Filled in examples of OBJ-EQUAL.
0 examples existed originally on OBJ-EQUAL.
6 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
((CLASS A) (CLASS A) → T)
((CLASS O D F I M) (CLASS O D F I M) → T)
(FALSE FALSE → T)

After eliminating duplicate and already-known entries, AM finds that only 3 new, distinct examples of OBJ-EQUAL had to be added.

Do-thresh raised from 358 to 359.

This Cand used 17.886 cpu seconds.

No Cand on CANDS is good enuf.
Do-thresh reduced from 359 to 239
Must find new candidates and merge them into CANDS.

The top 3 Cands are:
1: Fill in some examples of Set-struc-intersect
2: Check all examples of Int-set-struc
3: Fill in some generalizations of Obj-equal

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (We have no examples for SET-STRUC-INTERSECT yet)

AM recently tried this same Cand, so let's skip it now.

The top 3 Cands are:
1: Check all examples of Int-set-struc
2: Fill in some generalizations of Obj-equal
3: Check all examples of Set-struc

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (Some new , unchecked examples of INT-SET-STRUC have recently been added)

Beginning 5th cycle.

AM is forgetting the entire SUGG facet of the INT-SET-STRUC concept.
Because: (No sense using this suggestion more than once).

Checked examples of INT-SET-STRUC and all entries were confirmed

This Cand used 11.362 cpu seconds.

The top 3 Cands are:

1: Check all examples of Set-struc
2: Fill in some generalizations of Obj-equal
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (Some new , unchecked examples of SET-STRUC have recently been added)

Beginning 6th cycle.

Based on empirical experiments, AM believes that SET-STRUC may really be no more specialized than UNORD-OBJ.

Closer inspection reveals that the evidence for this was quite flimsy. AM will wait until some examples of any of these have been found: (BAG-STRUC), and then see if they truly also are SET-STRUC's.

Based on empirical experiments, AM believes that SET-STRUC may really be no more specialized than NONMULT-STRUC.

Closer inspection reveals that the evidence for this was quite flimsy. AM will wait until some examples of any of these have been found: (OSET-STRUC), and then see if they truly also are SET-STRUC's.

Checked examples of SET-STRUC.
5 entries were there initially.
1 small modifications had to be made.
5 entries are present now.

This Cand used 8.008 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Bag-struc
2: Fill in some examples of Oset-struc
3: Fill in some generalizations of Obj-equal

I choose first Cand. OK? *yes.*

The reason for considering this Cand is: (We have no examples for BAG-STRUC yet)

Beginning 7th cycle.

Filled in examples of BAG-STRUC.

0 examples existed originally on BAG-STRUC.
19 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
(BAG)
(BAG A)
(BAG B)
(BAG A B)
(BAG A A)
(BAG A A B)
(BAG 0 D F I M)
(BAG A B (BAG B) (CLASS))
(BAG BRUCE CORDELL DOUG)
(BAG R0-7 R1-7 R2-7 R3-7 R4-7 R5-7 R6-7 R7-7)
After eliminating duplicate and already-known entries, AM finds that.
only 10 new, distinct examples of BAG-STRUC had to be added.

XEQ-CAND

Do-thresh raised from 239 to 264.

This Cand used 17.692 cpu seconds.

The top 3 Cands are:
1: Fill in some generalizations of Obj-equal
2: Fill in some examples of Oset-struc
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The reason is: (The ratio of examples to non-examples of
OBJ-EQUAL is too low ; OBJ-EQUAL is too specialized , too narrow)

Beginning 8th cycle.

Considering genlizing a recursive defn of OBJ-EQUAL
Will try to remove a conjunct.
2 possible conjuncts to choose from.
AM generalizes OBJ-EQUAL into the new concept GENL-OBJ-EQUAL, by
not recursing on the CAR of each arg.
i.e., GENL-OBJ-EQUAL will not have a recursive check
like this one, which is present in OBJ-EQUAL:

APPLYB
(QUOTE OBJ-EQUAL)
(QUOTE DEFN)
(CAR BA1)
(CAR BA2)
AM generalizes OBJ-EQUAL into the new concept GENL-OBJ-EQUAL-1,
by not recursing on the CDR of each arg.
i.e., GENL-OBJ-EQUAL-1 will not have a recursive check

like this one, which is present in OBJ-EQUAL:

APPLYB
(QUOTE OBJ-EQUAL)
(QUOTE DEFN)
(CDR BA1)
(CDR BA2)

If any of (GENL-OBJ-EQUAL GENL-OBJ-EQUAL-1) ever seems to be too specialized, AM will consider disjoining it with other members of that set.

Filled in generalizations of OBJ-EQUAL.
0 generalizations existed originally on OBJ-EQUAL.
2 potential new entries were just proposed.

Eliminating duplicates, the newly constructed generalizations are:
GENL-OBJ-EQUAL
GENL-OBJ-EQUAL-1
After eliminating duplicate and already-known entries, AM finds that all 2 new, distinct generalizations of OBJ-EQUAL had to be added.

Do-thresh raised from 264 to 335.

This Cand used 6.667 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Genl-obj-equal-1
2: Fill in some examples of Genl-obj-equal
3: Fill in some examplls of Oset-struc

I choose first Cand. OK? *yes.*

The reason is: (The generalization GENL-OBJ-EQUAL-1 of OBJ-EQUAL is relatively new and has no exs of its own yet , excepting those of OBJ-EQUAL)

Beginning 9th cycle.

?: *N*

Rename which existing concept? *GENL-OBJ-EQUAL*

What is its new name? *SAME-SIZE*

Done.

Record of attempts to find examples:

An ex (sought) is: ((VECTOR BAG) (VECTOR B (BAG B) (CLASS) A))

Found 26 examples (and 105 non-exs), in 8.037 secs.
A nice ratio of exs/non-exs was encountered for GENL-OBJ-EQUAL-1
Won't try to create a restricted interesting version of
GENL-OBJ-EQUAL-1.

Filled in examples of GENL-OBJ-EQUAL-1.
0 examples existed originally on GENL-OBJ-EQUAL-1.
26 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
((VECTOR BAG) (VECTOR B (BAG B) (CLASS) A) → T)
((OSET 0 D F I M) (OSET 0 D F I M) → T)
((BAG) (BAG DON ED) → T)
((OSET D M I F 0) (OSET D M I F 0) → T)
((PAIR DOUG BRUCE) (PAIR DOUG BRUCE) → T)
((VECTOR BAG) (VECTOR D M I F 0) → T)
((VECTOR B) (VECTOR D M I F 0) → T)
((BAG B) (BAG B) → T)
((VECTOR D M I F 0) (VECTOR A A B) → T)
((BAG A) (BAG A B) → T)
((VECTOR) (VECTOR B (BAG B) (CLASS) A) → T)
((OSET BRUCE DON) (OSET B A) → T)
((PAIR COMPOSE-EXS COMPOSE-EXS) (PAIR LIST-STRUC-INTERSECT
ANYB-SPEC) → T)
((OSET R2-1 R2-2 R2-3 R2-4 R2-5 R2-6 R3-1 R3-2 R3-3 R3-4 R3-5
R3-6 R4-1 R4-2 R4-3 R4-4 R4-5 R4-6 R5-1 R5-2 R5-3 R5-4 R5-5 R5-6 R6-1
R6-2 R6-3 R6-4 R6-5 R6-6) (OSET 0 D F I M) → T)
((OSET A B (BAG B) (CLASS)) (OSET B (BAG B) (CLASS) A) → T)
((OSET 0 D F I M) (OSET B) → T)
((VECTOR A A) (VECTOR A B) → T)
((OSET DON ED) (OSET BAG) → T)
((BAG A A B) (BAG) → T)
((OSET B) (OSET BRUCE DON) → T)
((CLASS DON ED) (CLASS A) → T)
((PAIR LIST-STRUC-INSERT CANONIZE) (PAIR LIST-STRUC-INTERSECT
ANYB-SPEC) → T)
((VECTOR) (VECTOR BAG) → T)
((OSET A) (OSET D M I F 0) → T)
((VECTOR BAG) (VECTOR BAG) → T)
After eliminating duplicate and already-known entries, AM finds that.
only 25 new, distinct examples of GENL-OBJ-EQUAL-1 had to be added.

Do-thresh raised from 335 to 367.

This Cand used 29.095 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Same-size
2: Check all examples of Genl-obj-equal-1
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The 2 reasons are:
(Interestingness of SAME-SIZE has changed recently)
(The generalization SAME-SIZE of OBJ-EQUAL is relatively new and has no exs of its own yet , excepting those of OBJ-EQUAL)

Beginning 10th cycle.

Record of attempts to find examples:
-
An ex (sought) is: ((VECTOR A) (OSET B))+---+--+--------+----+++-----
----+-+---+-+-----+-------+--+-+---+-+---------+-+---------+---+-----+
----+---+--------------+

Found 26 examples (and 102 non-exs), in 8.032 secs.
A nice ratio of exs/non-exs was encountered for SAME-SIZE
Won't try to create a restricted interesting version of SAME-SIZE.

Filled in examples of SAME-SIZE.
0 examples existed originally on SAME-SIZE.
36 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
((OSET 0 D F I M) (OSET 0 D F I M) → T)
((OSET D M I F 0) (OSET D M I F 0) → T)
((PAIR DOUG BRUCE) (PAIR DOUG BRUCE) → T)
((BAG B) (BAG B) → T)
((OSET BRUCE DON) (OSET B A) → T)
((PAIR COMPOSE-EXS COMPOSE-EXS) (PAIR LIST-STRUC-INTERSECT ANYB-SPEC) → T)
((OSET A B (BAG B) (CLASS)) (OSET B (BAG B) (CLASS) A) → T)
((VECTOR A A) (VECTOR A B) → T)
((PAIR LIST-STRUC-INSERT CANONIZE) (PAIR LIST-STRUC-INTERSECT ANYB-SPEC) → T)
((VECTOR BAG) (VECTOR BAG) → T)
((VECTOR A) (OSET B) → T)
((BAG A B) (OSET B A) → T)
((CLASS 0 D F I M) (BAG 0 D F I M) → T)
((VECTOR B) (BAG A) → T)
((PAIR LIST-STRUC-INTERSECT ANYB-SPEC) (PAIR DOUG BRUCE) → T)
((OSET DON ED) (PAIR LIST-STRUC-INTERSECT ANYB-SPEC) → T)
((BAG 0 D F I M) (VECTOR D M I F 0) → T)
((VECTOR B) (BAG B) → T)
((OSET BAG) (OSET A) → T)
((VECTOR A A) (BAG A A) → T)
((CLASS A) (VECTOR BAG) → T)
((CLASS A B) (OSET A B) → T)
((PAIR COMPOSE-EXS COMPOSE-EXS) (OSET DON ED) → T)

((VECTOR A) (OSET A) → T)
((OSET BAG) (CLASS A) → T)
((OSET A) (CLASS A) → T)
((OSET B) (OSET A) → T)
((BAG O D F I M) (OSET O D F I M) → T)
((OSET DON ED) (OSET ED CORDELL) → T)
((OSET ED CORDELL) (OSET B A) → T)
((OSET A) (BAG B) → T)
((OSET B A) (OSET A B) → T)
((VECTOR B A) (OSET ED CORDELL) → T)
((OSET A) (VECTOR BAG) → T)
((OSET B A) (OSET DON ED) → T)
After eliminating duplicate and already-known entries, AM finds that. only 35 new, distinct examples of SAME-SIZE had to be added.

Do-thresh raised from 367 to 406.

This Cand used 21.725 cpu seconds.

The top 3 Cands are:
1: Check all examples of Same-size
2: Check all examples of Gen!-obj-equal-1
3: Check all things which just barely miss being examples of Same-size

I choose first Cand. OK? *yes.*

The reason is: (Some new , unchecked examples of SAME-SIZE have recently been added)

Beginning 11st cycle.

Checked examples of SAME-SIZE.
35 entries were there initially.
1 had to be completely discarded.
4 had to be transferred elsewhere.
30 entries are present now.

Do-thresh raised from 406 to 421.

This Cand used 6.917 cpu seconds.

The top 3 Cands are:
1: Check all examples of Gen!-obj-equal-1
2: Check all things which just barely miss being examples of

Same-size
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The reason is: (Some new , unchecked examples of GENL-OBJ-EQUAL-1 have recently been added)

Beginning 12nd cycle.

Checked examples of GENL-OBJ-EQUAL-1.
25 entries were there initially.
1 had to be completely discarded.
4 had to be transferred elsewhere.
20 entries are present now.

This Cand used 4.711 cpu seconds.

No Cand on CANDS is good enuf.
Do-thresh reduced from 421 to 333
Must find new candidates and merge them into CANDS.

The top 3 Cands are:
1: Canonize these 2 arguments: Genl-obj-equal-1 and Obj-equal
2: Canonize these 2 arguments: Same-size and Obj-equal
3: Fill in some examples of Coalesce

I choose first Cand. OK? *yes.*

The reason is: (It would be nice to find a canonical (with respect to Genl-obj-equal-1 and Obj-equal) representation C for any Object X ; that is ,
(GENL-OBJ-EQUAL-1 x y) iff
(OBJ-EQUAL (C x) (C y)).
)

Beginning 13rd cycle.

Experiments indicate that GENL-OBJ-EQUAL-1 is affected by the varying the type of structure of its arguments.

GENL-OBJ-EQUAL-1 doesn't look at any elements of OBJECT except possibly the car of the structure which denotes its type, so AM replaces the tail of OBJECT by a canonical distinguished tail, say NIL.

Succeeded!

Some conjectures that AM considers believable:

OBJ-EQUAL, restricted to canonical OBJECT's, is indistinguishable from GENL-OBJ-EQUAL-1.

There is a powerful analogy between

GENL-OBJ-EQUAL-1.....OBJ-EQUAL
OBJECT.....CANONICAL-OBJECT
operators on and into those operators restricted to
OBJECT.....CANONICAL-OBJECT
statements involving these.....statements involving these

Do-thresh raised from 333 to 341.

This Cand used 9.02 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Canonical-object
2: Restrict the following: Genl-obj-equal-1 Canonical-object Domain
3: Canonize these 2 arguments: Same-size and Obj-equal

I choose first Cand. OK? *yes.*

The reason is: (Any example of CANONICAL-OBJECT is a canonical example of OBJECT)

Beginning 14th cycle.

AM will now try to produce examples of CANONICAL-OBJECT by running the following operations:
(CANONIZE-GENL-OBJ-EQUAL-1&OBJ-EQUAL).

Won't try to create a restricted interesting version of CANONICAL-OBJECT.

Filled in examples of CANONICAL-OBJECT.
0 examples existed originally on CANONICAL-OBJECT.
165 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
(VECTOR)
(BAG)
(CLASS)
(OSET)
FALSE
T
TRUE
(PAIR)
(T)

(NIL)
(TRUE)
(FALSE)
After eliminating duplicate and already-known entries, AM finds that. only 12 new, distinct examples of CANONICAL-OBJECT had to be added.

Do-thresh raised from 341 to 391.

This Cand used 23.827 cpu seconds.

The top 3 Cands are:
1: Restrict the following: Genl-obj-equal-1 Canonical-object Domain
2: Canonize these 2 arguments: Same-size and Obj-equal
3: Fill in examples of Coalesce

I choose first Cand. OK? *yes.*

The reason is: (GENL-OBJ-EQUAL-1 was one of the predicates which defined the new concept CANONICAL-OBJECT , so it is worth considering the restriction of GENL-OBJ-EQUAL-1 to that subset of OBJECT 's)

Beginning 15th cycle.

Succeeded!

Do-thresh raised from 391 to 431.

This Cand used 3.562 cpu seconds.

The top 3 Cands are:
1: Canonize these 2 arguments: Same-size and Obj-equal
2: Fill in some examples of Coalesce
3: Restrict the following: Obj-equal Canonical-object Domain

I choose first Cand. OK? *yes.*

The reason is: (It would be nice to find a canonical (with respect to Same-size and Obj-equal) representation C for any Object X ; that is ,
(SAME-SIZE x y) iff
(OBJ-EQUAL (C x) (C y)).
)

Beginning 16th cycle.

Experiments indicate that SAME-SIZE is not affected by varying the type of structure of its arguments.

Experiments indicate that SAME-SIZE is not affected by reordering elements of its structural arguments.
So any canonical arguments can be Bags and Sets.

Experiments indicate that SAME-SIZE is affected by the presence of multiple elements in its structural arguments.
So any canonical arguments can be Bags and Lists.

SAME-SIZE doesn't look at the specific elements in OBJECT, like OBJ-EQUAL does, so AM can replace them all by a single distinguished element, say T.

Succeeded!

Some conjectures that AM considers believable:

OBJ-EQUAL, restricted to canonical BAG-STRUC's, is indistinguishable from SAME-SIZE.

There is a powerful analogy between

SAME-SIZE.....OBJ-EQUAL
BAG-STRUC.....CANONICAL-BAG-STRUC
operators on and into.....those operators restricted to
BAG-STRUC..... CANONICAL-BAG-STRUC
statements involving these.....statements involving these

Do-thresh raised from 431 to 457.

This Cand used 17.297 cpu seconds.

The top 3 Cands are:
1: Fill in some examples of Canonical-bag-struc
2: Restrict the following: Same-size Canonical-bag-struc Domain
3: Restrict the following: Bag-struc-join Canonical-bag-struc Domain

I choose first Cand. OK? *yes.*

The reason is: (Any example of CANONICAL-BAG-STRUC is a canonical example of BAG-STRUC)

Beginning 17th cycle.

AM will now try to produce examples of CANONICAL-BAG-STRUC by running the following operations:
(CANONIZE-SAME-SIZE&OBJ-EQUAL).

Filled in examples of CANONICAL-BAG-STRUC.
0 examples existed originally on CANONICAL-BAG-STRUC.
211 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
(BAG)
(BAG T T)
(BAG T T T)
(BAG T)
(BAG T T T T T)
(BAG T T T T)
(BAG T T T T T

Rename which existing concept? *CANONICAL-BAG-STRUC*

What is its new name? *NUMBER*

Done.

?: (W, I, E, M, N, ?, Q) *N*

Rename which existing concept? *CANONIZE-SAME-SIZE&OBJ-EQUAL*

What is its new name? *SIZE*

Done.

The top 3 Cands are:
1: Check all examples of Number
2: Restrict the following: Obj-equal Canonical-object Domain
3: Check all examples of Canonical-object

I choose first Cand. OK? *yes.*

The 2 reasons are:
(Interestingness of NUMBER has changed recently)
(Some new , unchecked examples of NUMBER have recently been added)

Beginning 19th cycle.

Checked examples of NUMBER and all entries were confirmed

This Cand used 1.909 cpu seconds.

The top 3 Cands are:
1: Check all examples of Canonical-object
2: Check all things which just barely miss being examples of Number
3: Restrict the following: Bag-struc-join Number Domain

I choose first Cand. OK? *yes.*

The reason is: (Some new , unchecked examples of CANONICAL-OBJECT have recently been added)

Beginning 20th cycle.

CANONICAL-OBJECT has 7 examples which occupy 11 list cells, but is not interesting enough to warrant taking up that much space; so about 2 will be selected at random and forgotten.
Checked examples of CANONICAL-OBJECT.
12 entries were there initially.
10 were never confirmed or rejected.
2 had to be completely discarded.
5 entries are present now.

This Cand used 16. 626 cpu seconds.

No Cand on CANDS is good enuf.
Do-thresh reduced from 495 to 340
Must find new candidates and merge them into CANDS.

The top 3 Cands are:
1: Fill in some examples of Size
2: Fill in some examples of Coalesce
3: Restrict the following: Bag-struc-join Number Domain

I choose first Cand. OK? *yes.*

The reason is: (We have no examples for SIZE yet)

Beginning 21st cycle.

Record of attempts to find examples:
An ex (sought) is: (BAG T T)++++++++++++++++++++++++++

Found 26 examples (and 0 non-exs), in .996 secs.
A nice ratio of exs/non-exs was encountered for SIZE
Won't try to create a restricted interesting version of SIZE.

Filled in examples of SIZE.
13 examples existed originally on SIZE.
26 potential new entries were just proposed.

Eliminating duplicates, the newly constructed examples are:
((BAG T T) → (BAG T T))
((BAG T T T T T) → (BAG T T T T T))
((BAG B) → (BAG T))
((BAG A A) → (BAG T T))
((BAG T T T) → (BAG T T T))
((BAG T T T T) → (BAG T T T T))
((BAG A B) → (BAG T T))
((BAG R2-1 R2-2 R2-3 R2-4 R2-5 R2-6 R3-1 R3-2 R3-3 R3-4 R3-5 R3-6 R4-1 R4-2 R4-3 R4-4 R4-5 R4-6 R5-1 R5-2 R5-3 R5-4 R5-5 R5-6 R6-1 R6-2 R6-3 R6-4 R6-5 R6-6) → (BAG T T T T T T T T T T T T T T T T T T

T T T T T T T T T T T T))
((BAG A A B) → (BAG T T T))
((BAG 0 D F I M) → (BAG T T T T T))
((BAG A) → (BAG T))
((BAG T T T T T T T T T T T T

Appendix 6. Bibliography

Of all the articles, books, and memos which were read as background for AM, I have selected those which had some impact on that work (or at least, on this document). While numerous, they form a far from comprehensive list of publications dealing with automated theory formation, with AI in general, and with how mathematicians do research.

Preceding the listing of these references, Appendix 6.1 will provide some pointers to real-world documentation, the AM program itself, etc.

Appendix 6.1. Documentation

Below are listed some references to earlier articles, to on-line documentation about AM, to the AM program itself, etc.

The AM representation is a variant of the "Beings" ideas, a modular representation for knowledge. In his summary of the state of Automatic Programming [Bierman 76], Bierman compares Beings with Frames, Actors, etc., and gives a nice example of Beings in action.

History buffs may be interested in perusing the original thesis proposal for AM (about 50 pages long). It is kept as SYS4[tlk,dbl]@SU-AI.

The full body of knowledge provided to AM is found in English translation on file GIVEN[tlk,dbl]@SU-AI. This is a longer, fuller treatment than the one found in this document, in Appendix 2.1 and Appendix 3. The knowledge *as used* is of course the AM program itself. Needless to say, it is *much* longer than the excerpts shown in Appendix 2.3.

Said running AM program is stored at SUMEX, on directory <LENAT>. From Interlisp, one need only load in the file <lenat>LT. This in turn will load in three files: TOP6, CON6, and UTIL6. So if you want to steal AM, take all four files!

Once loaded, the program is self-explanatory. It will instruct the user to type *(START)* to begin AM itself. Once he does this, AM will ask him some questions, and then enter the select-and-execute-a-task loop.

A crude "user's manual" is stored as file MANUAL[am,dbl]@SU-AI. The reader may wish to glance over it before running AM, since much of the actual LISP code is more complicated than this thesis made it seem (e.g., there are two dynamically-adjusted variables, Verbosity-level and Expert-level. The former variable determines which events generate a message, and the latter variable affects the terseness of each of those printed messages.)

Appendix 6.2. References

Adams, James L., *Conceptual Blockbusting*, W.H. Freeman and Co., San Francisco, 1974.

Amarel, Saul, *On Representations and Modelling in Problem Solving and On Future Directions for Intelligent Systems*, RCA Labs Scientific Report No. 2, Princeton, 1967.

Atkin, A. O. L., and B. J. Birch, eds., *Computers in Number Theory*, Proceedings of the 1969 SRCA Oxford Symposium, Academic Press, New York, 1971.

Badre, Nagib A., *Computer Learning From English Text*, Memorandum No. ERL-M372, Electronics Research Laboratory, UCB, December 20, 1972. Also summarized in *CLET — A Computer Program that Learns Arithmetic from an Elementary Textbook*, IBM Research Report RC 4235, February 21, 1973.

Berliner, H., *Chess as Problem Solving: The Development of a Tactics Analyzer*, Carnegie-Mellon University Computer Science Department Thesis, March, 1974.

Beth, E. W., and J. Piaget, *Mathematical Epistemology and Psychology*, Gordon and Breach, New York, 1966.

Beveridge, W. I., *The Art of Scientific Investigation*, Vintage Books, N. Y. 1950.

Biermann, A. W., *Approaches to Automatic Programming*, in Advances in Computers, v. 15, Academic Press, 1976.

Black, M., *Margins of Precision*, Cornell University Press, Ithaca, New York, 1970.

Blalock, H. M., *Theory Construction*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

Bledsoe, W. W., *Splitting and Reduction Heuristics in Automatic Theorem Proving*, Artificial Intelligence 2, 1971, pp. 55-77.

Bledsoe, W. W., and Bruell, Peter, *A Man-Machine Theorem-Proving System*, Artificial Intelligence 5, 1974, 51-72.

Bobrow, D., and A. Collins, editors, *Representation and Understanding*, Academic Press, S.F., 1975.

Bobrow, D., and D. Norman, *Some Principles of Memory Schemata*, XEROX PARC Memo CSL 75-4, Palo Alto, July, 1975.

Bobrow, D. G., and T. Winograd, *An Overview of KRL, A Knowledge Representation Language*, Journal of Cognitive Science, Vol 1, No 1, January 1977.

Bourbaki, N., *The Architechture of Mathematics*, American Mathematics Monthly, v. 57, pp. 221-232, Published by the MAA, Albany, N. Y., 1950.

Boyer, R. S., and J S. Moore, *Proving Theorems about LISP Functions*, JACM, v. 22, No. 1, January, 1975, pp. 129-144.

Brotz, D. K., *Embedding Heuristic Problem Solving Methods in a Mechanical Theorem Prover*, Ph.D. dissertation published as Stanford Computer Science Report STAN-CS-74-443, August, 1974.

Bruijn, N. G. de, *AUTOMATH, a language for mathematics*, Les Presses de L'Universite de Montreal, Montreal, 1973.

Buchanan, B. G., G. Sutherland, and E. Feigenbaum, *Heuristic Dendral: A Program for Generating Explanatory Hypotheses in Organic Chemistry*, in (Meltzer and Michie, eds.) Machine Intelligence 4, American Elsevier Pub., N. Y., 1969, pp. 209-254.

Buchanan, B. G., E. Feigenbaum, and Sridharan, *Heuristic Theory Formation*, Machine Intelligence 7, 1972, pp. 267-290.

Buchanan, B. G., *Scientific Theory Formation by Computer*, NATO Advanced Study Institute on Computer Oriented Learning Processes, Bonas, France, 1974.

Buchanan, Bruce G., *Applications of Artificial Intelligence to Scientific Reasoning*, Second USA-Japan Computer Conference, Tokyo, August 26-28. Published by AFIPS and IPSJ, Tokyo, 1975, pp. 189-194.

Bundy, A., *Doing Arithmetic with Diagrams*, 3rd International Joint Conference on Artificial Intelligence (3rd IJCAI), Stanford, 1973, pp. 130-138.

Burstall, R., and J. Darlington, *A Transformation System for Developing Recursive Programs*, University of Edinburgh AI Research Report, March, 1976.

Church, A., *The calculi of Lambda-conversion*, Princeton University Press, Princeton, 1941.

Cohen, P. J., *Set Theory and the Continuum Hypothesis*, W.A.Benjamin, Inc., New York, 1966.

Colby, K. M., "*Simulations of belief systems*", in [Schank and Colby 73].

Copeland, R. W., *How Children Learn Mathematics*, The MacMillan Company, London, 1970.

Courant, R., and H. Robins, *What is Mathematics*, Oxford University Press, New York, 1941.

Dahl, O., et. al., *SIMULA-67: A Common Base Language*, Norwegian Computing Center Publication No. S-2, Oslo, 1968.

Darlington, J., and R. Burstall, *A System Which Automatically Improves Programs*, 3rd IJCAI, 1973, pp. 479-485.

Davis, R., and J. King, *An Overview of Production Systems*, Stanford AI Lab Memo 271, October, 1975.

Davis, R., *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*, Stanford AI Lab Memo 283, July, 1976.

Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

Eddington, Sir A. S., *New Pathways in Science*, Macmillan Co., N. Y., 1935.

Engelman, C., *MATHLAB: A Program for On-Line Assistance in Symbolic Computation*, in Proceedings of the FJCC, v. 2, Spartan Books, 1965.

Engelman, C., *MATHLAB '68*, in IFIP, Edinburgh, 1968.

Evans, T. G., *A Program for the Solution of Geometric-Analogy Intelligence Test Questions*, in [Minsky 68], pp. 271-353.

Eynden, C. V., *Number Theory: An Introduction to Proof*, International Textbook Company, Scranton, Pennsylvania, 1970.

Feigenbaum, E. A., *EPAM: The Simulation of Verbal Learning Behavior*, in [Feigenbaum & Feldman 63], Part 2, Section 2, pp. 297-309.

Feigenbaum, E. A., and J. Feldman, editors, *Computers and Thought*, McGraw-Hill Book Company, New York, New York, 1963.

Feigenbaum, E., B. Buchanan, and J. Lederberg, *On Generality and Problem Solving: A Case Study Using The DENDRAL Program*, in (Meltzer and Michie, eds.) Machine Intelligence 6, 1971, pp 165-190.

Fogel, L., A. Owens, and M. Walsh, *Artificial Intelligence Through Simulated Evolution*, John Wiley & Sons, Inc., N. Y., 1966. Fuller, R. B., *Synergetics*, Macmillan Co., N. Y., 1975.

Gardner, M., *Mathematical Games*, Scientific American [numerous columns, including especially: February, 1975.]

Gelernter, H., *Realization of a Geometry-Theorem Proving Machine*, in [Feigenbaum & Feldman 63], Part 1, Section 3, pages 134-152.

Goldstein, I., *Elementary Geometry Theorem Proving*, MIT AI Memo 280, April, 1973.

Goodstein, R. L., *Fundamental Concepts of Mathematics*, Pergamon Press, New York, 1962.

Green, C.C., R. Waldinger, D. Barstow, R. Elschlager, D. Lenat, B. McCune, D. Shaw, and L. Steinberg, *Progress Report on Program-Understanding Systems*, Memo AIM-240, CS Report STAN-CS-74-444,Artificial Intelligence Laboratory, Stanford University, August, 1974.

Guard, J. R., Eastman et al., *Semi-Automated Mathematics*, JACM 16, January, 1969, pp. 49-62.

Hadamard, J., *The Psychology of Invention in the Mathematical Field*, Dover Publications, New York, 1945.

Halmos, P. R., *Innovation in Mathematics*, in [Kline 68]. Originally in Scientific American, September, 1958.

Hardy, G. H., and E. M. Wright, *An Introduction to the Theory of Numbers*, Oxford U. Press, London, 1938. (Fourth edition, 1960)

Hayes-Roth, F., and V. R. Lesser, *Focus of Attention in a Distributed Speech Understanding System*, Computer Science Dept. Memo, Carnegie-Mellon University, Pittsburgh, Pa., January 12, 1976.

Hempel, C. G., *Fundamentals of Concept Formation in Empirical Science*, University of Chicago Press, Chicago, 1952.

Hewitt, C., *A Universal Modular ACTOR Formalism for Artificial Intelligence*, Third International Joint Conference on Artificial Intelligence, 1973, pp. 235-245.

Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages*, MIT AI Lab Working Paper 92, April, 1976.

Hilpinen, R., *Rules of Acceptance and Inductive Logic*, Acta Philosophica Fennica, Fasc. 22, North-Holland Publishing Company, Amsterdam, 1968.

Hintikka, J., *Knowledge and Belief*, Cornell U. Press, Ithaca, N. Y., 1962.

Hintikka, J., and P. Suppes, editors, *Aspects of Inductive Logic*, North-Holland Publishing Company, Amsterdam, 1966.

Iberall, A. S., *Toward a General Science of Viable Systems*, McGraw-Hill Book Co., N. Y. 1972.

Kershner, R.B., and L.R.Wilcox, *The Anatomy of Mathematics*, The Ronald Press Company, New York, 1950.

Kline, M. (ed), *Mathematics in the Modern World: Readings from Scientific American*, W.H.Freeman and Co., San Francisco, 1968.

Kling, R. E., *Reasoning by Analogy with Applications to Heuristic Problem Solving: A Case Study*, Stanford Artificial Intelligence Project Memo AIM-147, CS Department report CS-216, August, 1971.

Knuth, D. E., *Fundamental Algorithms*, v. 1 of The Art of Computer Programming, Addison-Wesley Publishing Company, Menlo Park, 1968.

Knuth, D. E., *Surreal Numbers*, Addison-Wesley Publishing Company, Reading, Mass., 1974.

Knuth, D. E., *Ancient Babylonian Algorithms*, CACM 15, July, 1972, pp. 671-677.

Koestler, A., *The Act of Creation*, New York, Dell Pub., 1967.

Koppelman, E., "Progress in Mathematics", in the proceedings of the Workshop on the Historical Development of Modern Mathematics, July, 1975.

Lamon, W. E., *Learning and the Nature of Mathematics*, Science Research Associates, Palo Alto, 1972.

Lederberg, J., *DENDRAL-64: A System for Computer Construction, Enumeration, and Notation of Organic Molecules as Tree Structures and Cyclic Graphs*, Parts I-V of the Interim Report to NASA, 1964.

Lederberg, J.; New York Times review of [Weizenbaum 76]; 1976.

Lefrancois, G. R., *Psychological Theories and Human Learning*, Wadsworth Publishing, Belmont, Ca., 1972.

Lenat, D., *Synthesis of Large Programs from Specific Dialogues*, Proceedings of the International Symposium on Proving and Improving Programs, Le Chesnay, France, July, 1975a.

Lenat, D., *BEINGs: Knowledge as Interacting Experts*, 4th IJCAI, Tbilisi, Georgian SSR, USSR, 1975b.

Lesser, V., R.D. Fennell, L. D. Erman, and D. R. Reddy, *Organization of the Hearsay-II Speech Understanding System*, in IEEE Transactions on Acoustics, Speech, and Signal Processing, v. ASSP-23, 1975, pp. 11-23.

Linderholm, C. E., *Mathematics Made Difficult*, World Publishing Co., N. Y. 1972.

Lombardi, L.A., and B. Raphael, *LISP as the language for an incremental computer*, in (E. C. Berkeley and D. G. Bobrow, eds.) The Programming Language LISP: Its Operation and Applications, Information International Inc., 1964.

McDermott, D., *Artificial Intelligence Meets Natural Stupidity*, in Sigart Newsletter, No. 57, April, 1976, pp. 4-9.

Martin, W., and R. Fateman, *The MACSYMA System*, in (S. Petrick, ed.) Second Symposium on Symbolic and Algebraic Manipulation, ACM SIGSAM, N. Y. (conference was held in Los Angeles), 1971, pp. 59-75.

Minsky, M., editor, *Semantic Information Processing*, The MIT Press, Cambridge, Massachusetts, 1968.

Minsky, M., *Frames*, in [Winston 75].

Mirsky, L., *Studies in Pure Mathematics*, Academic Press, New York, 1971.

Moore, J S., *Introducing Iteration into the Pure LISP Theorem Prover*, XEROX PARC report CSL-74-3, Palo Alto, 1975.

Moore, J. and A. Newell, "How can MERLIN understand?," in (Gregg, ed.) Knowledge and Cognition, Lawrence Erlbaum Associates, 1973.

Moore, R. C., *D-SCRIPT: A Computational Theory of Descriptions*, MIT AI Memo 278, February, 1973.

Neumann, J. von, *The Mathematician*, in R.B. Heywood (ed), The Works of the Mind, U. Chicago Press, pp. 180-196, 1947.

Nevins, Arthur J., *A Human Oriented Logic for Automatic Theorem Proving*, MIT AI Memo 268, October, 1972.

Nevins, Arthur J., *Plane Geometry Theorem Proving Using Forward Chaining*, Artificial Intelligence 6, Spring 1975, pp. 1-23.

Newell, A. *Heuristic Programming: Ill-Structured Problems*, in (A. Aronofsky, ed.) Progress in Operations Research III, John Wiley and Sons, 1969.

Newell, A., *Production Systems: Models of Control Structures*, May, 1973 CMU Report, also published in (W.G. Chase, ed.) Visual Information Processing, N. Y.: Academic Press, Chapter 10, pp. 463-526.

Newell, A., J. Shaw, and H. Simon, *Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristics*, RAND Corp. Report P-951, March, 1957.

Newell, A., and H. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

Newell, A. and H. Simon, *Computer Science as Empirical Inquiry: Symbols and Search*, the 1975 ACM Turing Award Lecture, printed in CACM 19, No. 3, March, 1976, pp. 113-126.

Nilsson, N. J., *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill Book Company, New York, New York, 1971

Norman, D., and D. Bobrow, *On Data-limited and Resource-limited Processes*, Journal of Cognitive Psychology, v. 7, 1975, pp. 44-64.

Norman, D., and D. Rumelhart, *Explorations in Cognition*, W. H. Freeman & Co., S.F., 1975.

Ore, O., *Number Theory and its History*, McGraw-Hill, New York, 1948.

Papert, S., *Teaching Children to be Mathematicians Versus Teaching About Mathematics*, in the International Journal of Mathematical Education in Science and Technology, v. 3, No. 3, July-September, 1972, pp. 249-262.

Piaget, J., *The Language and Thought of the Child*, The World Publishing Co., N. Y., 1955.

Pietarinen, J., *Lawlikeness, Analogy, and Inductive Logic*, North-Holland, Amsterdam, published as v. 26 of the series Acta Philosophica Fennica (J. Hintikka, ed.), 1972.

Pitrat, J., *Heuristic Interest of using Metatheorems*, Symposium on Automatic Demonstration, Springer-Verlag, 1970.

Poincare', H., *The Foundations of Science: Science and Hypothesis, The Value of Science, Science and Method*, The Science Press, New York, 1929.

Polya, G., *Mathematics and Plausible Reasoning*, Princeton University Press, Princeton, Vol. 1, 1954; Vol. 2, 1954.

Polya, G., *How To Solve It*, Second Edition, Doubleday Anchor Books, Garden City, New York, 1957.

Polya, G., *Mathematical Discovery*, John Wiley & Sons, New York, Vol. 1, 1962; Vol. 2, 1965.

Ramanujan, S. A., *Collected Papers*, (Hardy, Aiyar, and Wilson, eds.), Chelsea Publishing Company, N. Y., 1927.

Rulifson, J., J. Derksen, and R. Waldinger, *QA4: A Procedural Calculus for Intuitive Reasoning*, SRI Project 8721, Technical Note 73, Artificial Intelligence Center, SRI, Menlo Park, California, November, 1972.

Saaty, T. L., and F. J. Weyl, editors, *The Spirit and the Uses of the Mathematical Sciences*, McGraw-Hill Book Company, New York, 1969.

Samuel, A., *Some Studies in Machine Learning Using the Game of Checkers II. Recent Progress*, in the IBM Journal of Research and Development, vol. 11, no. 6, pp. 610-617, November, 1967.

Schank, R. and K. Colby, *Computer Models of Thought and Language.* , W. H. Freeman, 1973.

Schminke, C. W., and W. R. Arnold, editors, *Mathematics is a Verb*, The Dryden Press, Hinsdale, Illinois, 1971.

Simon, H. A., *The Heuristic Compiler*, in [Simon & Siklossy 72], Part 1, Chapter 1, pp. 9-43, 1972.

Simon, H. A., *Does Scientific Discovery Have a Logic?*, Philosophy of Science, v. 40, No. 4, December, 1973, pp. 471-480.

Simon, H. A., and L. Siklossy, editors, *Representation and Meaning: Experiments with Information Processing Systems*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1972.

Skemp, R. R., *The Psychology of Learning Mathematics*, Penguin Books, Ltd., Middlesex, England, 1971.

Sloman, A., *Interactions Between Philosophy and Artificial Intelligence: The Role of Intuition and Non-Logical Reasoning in Intelligence*, Artificial Intelligence 2, 1971, pp. 209-225.

Smith, N. W., *A Question-Answering System for Elementary Mathematics*, Stanford Institute for Mathematical Studies in the Social Sciences (IMSSS), Technical Report 227, April 19, 1974.

Spivak, M., *Calculus on Manifolds*, W.A.Benjamin, Inc., N. Y. 1965.

Stein, S. K., *Mathematics: The Man-Made Universe: An Introduction to the Spirit of Mathematics*, Second Edition, W. H. Freeman and Company, San Francisco, 1969.

Teitelman, W., *INTERLISP Reference Manual*, XEROX PARC, 1974.

Tullock, G., *The Organization of Inquiry*, Duke U. Press, Durham, N. C., 1966.

Venn, J., *The Principles of Empirical or Inductive Logic*, MacMillan and Co., London, 1889.

Waismann, F., *Introduction to Mathematical Thinking*, Frederick Ungar Publishing Co., New York, 1951.

Wang, H., *Toward Mechanical Mathematics*, IBM Journal of Research and Development, v. 4, Number 1, January, 1960, pp. 2-22.

Weizenbaum, J., *Computer Power and Human Reason*, W. H. Freeman, S.F., 1976.

Wickelgren, W. A., *How to Solve Problems: Elements of a Theory of Problems and Problem Solving*, W. H. Freeman and Co., Sanf Francisco, 1974.

Wilder, R. L., *Evolution of Mathematical Concepts*, John Wiley & Sons, Inc., N. Y., 1968.

Winograd, T., *Understanding Natural Language*, Academic Press, Inc., New York, New York, 1972.

Winston, P., *Learning Structural Descriptions from Examples*, Ph.D. thesis, Dept. of Electrical Engineering, TR-76, Project MAC, TR-231, MIT AI Lab, September, 1970.

Winston, P., editor, *New Progress in Artificial Intelligence*, MIT AI Lab Memo AI-TR-310, June, 1974.

Winston, P., editor, *The Psychology of Computer Vision*, McGraw Hill, N. Y. 1975.

Wittner, G. E., *The Structure of Mathematics*, Xerox College Publishing, Lexington, Mass., 1972.