Invention
And
Exploration
In
Discovery

by

Kenneth William Haase Jr.

S.B. in Philosophy, Massachusetts Institute of Technology (1984)
S.M. in Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1986)

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
February 1990

Signature of Author _____
Department of Electrical Engineering and Computer Science
February 1, 1989

Certified by _____
Marvin L. Minsky, Professor EECS, MIT
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Invention and Exploration in Discovery

Kenneth W. Haase Jr.

This thesis explores the processes of discovery and creativity by constructing computer programs which invent interesting constructions and discover significant regularities in their own isolated worlds. It suggests that discovery is best understood as the interaction of two distinct processes: the *exploration* of possibilities in a given vocabulary and the *invention* of new vocabularies where such explorations will prove fruitful. The 'vocabularies' explored or invented by these processes are not merely the words and definitions one might find in a dictionary or glossary, but are the essential categories, criteria of identity and simplicity, 'technical' practices, and descriptive biases involved in both the exploration of possibilities and the reduction of possibilities to practice. Extension or modification of any of these elements may be part of vocabulary invention; in the most radical cases, criteria of identity, simplicity, or reference may be drastically reformulated in the transformation to a new vocabulary.

A program called "Cyrano" is used to experiment with the interaction of exploration and invention in discovery; beginning from a base vocabulary of operations on simple recursive structures, Cyrano proceeds to the construction and description of simple arithmetic systems. For Cyrano, a vocabulary is a set of data structures and combining operations together with an opaque interpreter which generates 'examples' from data structures; Cyrano's processes of vocabulary invention may either transform the space of data structures it acts upon (by making composites into primitives or abstracting common patterns of combination) or construct new 'interpeters' which codify different standards for identity or behavior than those which it was initially given. The invention of new vocabularies is driven by the history of explorations within an existing vocabulary; invented vocabularies are then subject to the same sort of exploration leading (presumably) to the invention of yet other vocabularies.

To the Memory of
Dorothy Peterson Haase
"Her Life Taught Love"

Acknowledgements

A central intellectual element of this thesis is that a given result is as much a product of the framework that produced it is a statement or assertion within that framework. The same can be said of this work; I have much to owe for the framework whose threads I attempt to tie together in this document. Parts of the framework are the intellectual environment in which I have had the honor and pleasure to be immersed over the past ten years; other parts are the framework of friends and family who have helped me keep my perspective on my work and my life.

First and foremost, I must thank my thesis committee: Marvin Minsky, my advisor and mentor who showed me how to be playful and critical at the same time; Patrick Winston, who kept my feet on the ground as we stared cloud-wards; Doug Lenat who insisted that my work be both fun and exciting as well as intellectually interesting; and Thomas Kuhn, who listened so carefully to my awkward descriptions as to teach me important lessons about both what I was saying and about the importance — and difficulty — of trying to figure out what someone means and not meerely what they say.

Of equal importance has been the cadre of faculty, graduate students, undergraduates, and otherw who have created the intellectual ambience in which these ideas and this author matured. In no particular order, Phil Agre, David Chapman, Gary Drescher, Ken Forbus, Dan Weld, Dan Huttenlocher, John Batali, Jim Davis, Margaret Minsky, Pattie Maes, Walter van der Velde, Luc Steels, Rick Lathrop, Jonathan Amsterdam, John Mallery, David Wallace, Gerry Roylance, Tom Knight, Ramesh Patil, Gerry Sussman, Ian Horswill, Marty Hiller, and many I must have forgotten have all shaped me and this work with discussions, debates, and humorous exchanges.

Many of the individuals above have, along with me, been participants in the adventure known as the MIT Artificial Intelligence Laboratory. In helping to maintain a maximally hassle-free, relatively egalitarian, and — most importantly — comfortable place, I'd like to thank Priscilla Cobb, Marilyn Melithoniotes, Cody Curtis, Rita Boyland, Cathleen Krebs, and (again) many others. My work in particular has been funded by the Office of Naval Research through either the auspices of Marvin Minsky or Patrick Winston; their support is gratefully acknowledged.

My friends and family, too many to be thanked individually, have been a support during all the process of graduate school. Thank you and I love you all.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

How do we discover things? Where do new ideas come from? This thesis explores processes of discovery and creativity by constructing computer programs which propose significant constructions and discover interesting regularities in their own isolated worlds. Experience with these programs suggests that discovery is best understood as the interaction of two distinct processes: the *exploration* of possibilities in a given descriptive space and the *invention* of new descriptive spaces where such explorations are likely to prove fruitful. I refer to the descriptive spaces which are explored or invented as "vocabularies" to emphasize that they are not merely sets of possible syntactic constructions, but include standards of simplicity and identity, practices of reference and verification, and significant generative and descriptive biases. Extension or modification of any of these elements may be part of inventing a new vocabulary; in the most radical cases, criteria of identity, simplicity, or reference may be drastically reformulated by the transformation to a new vocabulary.

A program called "Cyrano" is presented as an experiment in the interaction of exploration and invention in discovery; beginning from an initial set of operations on simple recursive structures, Cyrano proceeds to invent simple arithmetic systems and explore their properties. As implemented on the computer, the vocabularies explored by Cyrano are collections of data structures and operations among them (the system's 'domain'), a simple programming language in which to form combinations of these domain operations (the system's 'space of definitions'), and an opaque interpreter which generates examples of domain level operations or constructed definitions (the system's 'empirical grounding'). Cyrano methodically explores its space of definitions based on a criterion of syntactic simplicity with respect to its given primitives. At intervals during this exploration, the program pauses to reflect on the history of its activity and consider either the reorganization of its current vocabulary or the introduction of new primitive structures and operations which capture the regularities of its own exploration process.

Historically, there have been a host of qualitative accounts of discovery [Poincaré, 1929], [Polya, 1962], but only in this century have we seen *technical* theories of discovery. But most such work has focussed either on exploration of initially given representational spaces [Langley *et al.*, 1987] or has attempted to 'mainstream' the invention of representations with their exploration[Lenat, 1983]. To focus on particularly radical sorts of vocabulary invention, experiments with Cyrano began by separating the exploration and invention of descriptive spaces. Though this separation was initially introduced for purposes of experimental and pedagogical isolation, it turns out that some of the processes of vocabulary invention and exploration *must* be separated; exploration of any particular space will be effective because of its local standards of

simplicity, convenience, and accuracy; but these standards are the very properties whose variations the invention of vocabularies must be free to explore.

This need to separate exploration from invention arises from the *entrenchment* of vocabularies in their own observational and descriptive biases. A good vocabulary for problem solving or discovery is *dense* with solutions; their is a high probability that constructions in the vocabulary will be useful or interesting. To ensure such a probability, the vocabulary imposes a 'generative bias' which makes some constructions more likely than others.[1]

But the density of a vocabulary is as much a result of those things which it makes *hard* to say as of those thing which it expresses easily. This bias makes it possible to find increasingly dense and powerful representations which succeed only by making it more difficult to construct certain sorts of solutions or ask certain sorts of questions. While these vocabularies may be useful and locally desirable, their 'artificial enhancements' will limit them when they must evolve in those areas which they find difficult to describe.

The space of potential vocabularies is riddled with 'local maxima' of this sort, where the incremental improvement of vocabularies has led to enormously useful and effective 'dead ends' that succeed as much by virtue of what they make hard to express as by virtue of what they express easily. While vocabularies such as these are immensely useful, their existence suggests that the processes of vocabulary invention themselves should proceed in a space of weaker constraint and lesser effectiveness. Put another way, the 'strong methods' of Artificial Intelligence succeed by exploiting structure known to exist in a domain; but the space of strong methods — because of the variety of ways in which knowledge and structure can be imposed on the world — does not itself have such strong methods.

Since there is no 'simple vocabulary' for vocabulary invention or 'easy road' to better vocabularies, the mechanisms of invention applied by Cyrano are heuristic and local at best. But this same description applies to any computer program, human individual, intellectual community, or social culture. In view of this, I believe that much is to be gained by examining the activity of such local and heuristic mechanisms *in vitro* with computer programs which explore and invent vocabularies for their own isolated worlds.

1.1 Theoretical Conclusions

Work with Cyrano and the examination of other programs which seek to explore or invent vocabularies has revealed several constraints on the structure of discovery

[1]Minsky [Minsky, 1986, Page 146] has suggested the term 'investment' for the tendency to persist in using a well developed vocabulary rather than a newwer, less developed one. I prefer the term entrenchment because of the connotation that the power of our representation can block us from seeing both its inadequacies and the advantages of alternative representations.

systems both designed and evolved:

- **Constructive Closure.** Discovery processes which generate new constructions from existing constructions should be able to use their new results to yield further constructions. The processes of construction and conjecture must form a connected *cycle* so that syntactic 'dead-ends' can be avoided.

- **Representational Density.** A discovery system's representation must concisely express the 'interesting' definitions in its domain. Such a representation is said to be 'dense' with respect to the space of definitions regarded as valuable; i.e. a random definition in the representation is likely to be significant. Given the constraint of *constructive closure*, any representations which the system generates should also tend to be dense with interesting constructions.

- **Parsimonious Generation.** In order to take advantage of dense representations, the system's control strategy must apply some sorts of *simplicity criteria* to order the consideration of potential constructions. Such criteria, however, must be open to change and modification; one important component of vocabulary invention is the reformulation of the standards of simplicity for a vocabulary.

- **Basis Independence** The mechanisms of discovery we propose or construct must not depend on the particular properties of their initial domain; otherwise, the shift to a new representation would obsolete that particular expertise and diminish the system's future effectiveness.

1.2 Implementation Highlights

The Cyrano program has emerged from a process of design under these constraints; in its current implementation, these constraints are realized by several implementational components:

- **Definition by Combination.** New definitions are constructed by Cyrano as combinations of existing definitions; this assures the syntactic closure of the cycle of discovery and also ensures that Cyrano will never 'lose an option' by mutating away an important distinction.

- **Typing of Definitions.** Every definition and combining operation produced or used by Cyrano is annotated with information about its domain (the objects it acts upon) and its range (the objects it produces). These annotations constrain the generation of new definitions. Other than these annotations, Cyrano never examines the internal structure of the definitions it produces. This regimen ensures that Cyrano's mechanisms will not depend superficially on syntactic or implementational details of particular representational vocabularies.

- **Parsimonious Search.** Each definition is assigned a *simplicity* between zero and one; the simplicity of a combination is the product of the simplicity of its components. Cyrano's underlying control regimen generates simpler definitions before less simple ones; however, one important element of vocabulary invention is the reassessment and modification of these standards of simplicity.

- **Empirical Examination.** Constructed definitions are judged and selected based on their empirical behavior; this requires some effort in generating examples of definitions to examine, but ensures that the system is not limited only to those syntactic vocabularies in which its predictions are reliable.

- **Polygenic Behaviors.** Cyrano's only *a priori* criterion of interest is the identification of *polygenic behaviors*: operations which can be defined in a variety of different ways. In addition to their pragmatic utility — having operations which are known to be acheivable in various ways — such families of definitions usually indicate underlying regularities not explicit in the current representation. Many of the interestingness criteria of other systems — predictability, regularity, operationality, correlation, etc — are reducible to the criterion of polygenicity with respect to a body of definitions.

- **Behavioral Reification.** New vocabularies are formed by taking the operations in one vocabulary and constructing a new vocabulary whose objects are the operations in the first vocabulary and whose operations are the combination of those operations. This is called *reifying* the operations of the first vocabulary. For instance, Cyrano constructs the vocabulary of 'numbers' by reifying the operations of removing n elements from a list and defining new operations on objects corresponding to combinations of these operations.

Figure 1.1 is a block diagram of the Cyrano system. It consists of an exploration loop and an invention loop; the invention loop operates upon the pool of definitions within which the exploration loop explores. This pool of definitions contains both *domain definitions*: simple computer programs take particular inputs and yield particular results; and *fragments*: computer programs with 'blank spaces' where domain operations may be inserted. Cyrano constructs new definitions by combining a fragmentary definition with a domain definition to yield either a domain definition or a fragmentary definition to be further combined. This method of combination is used to impose a simplicity metric on the space of definitions; every definition has an associated simplicity between zero and one; a combination of a fragment and a definition yeilds a definition whose simplicity is the arithmetic product of each individual definition's simplicity.

The exploration loop considers new combinations of existing definitions in order of decreasing simplicity, filtering those definitions based on superficial syntactic criteria.

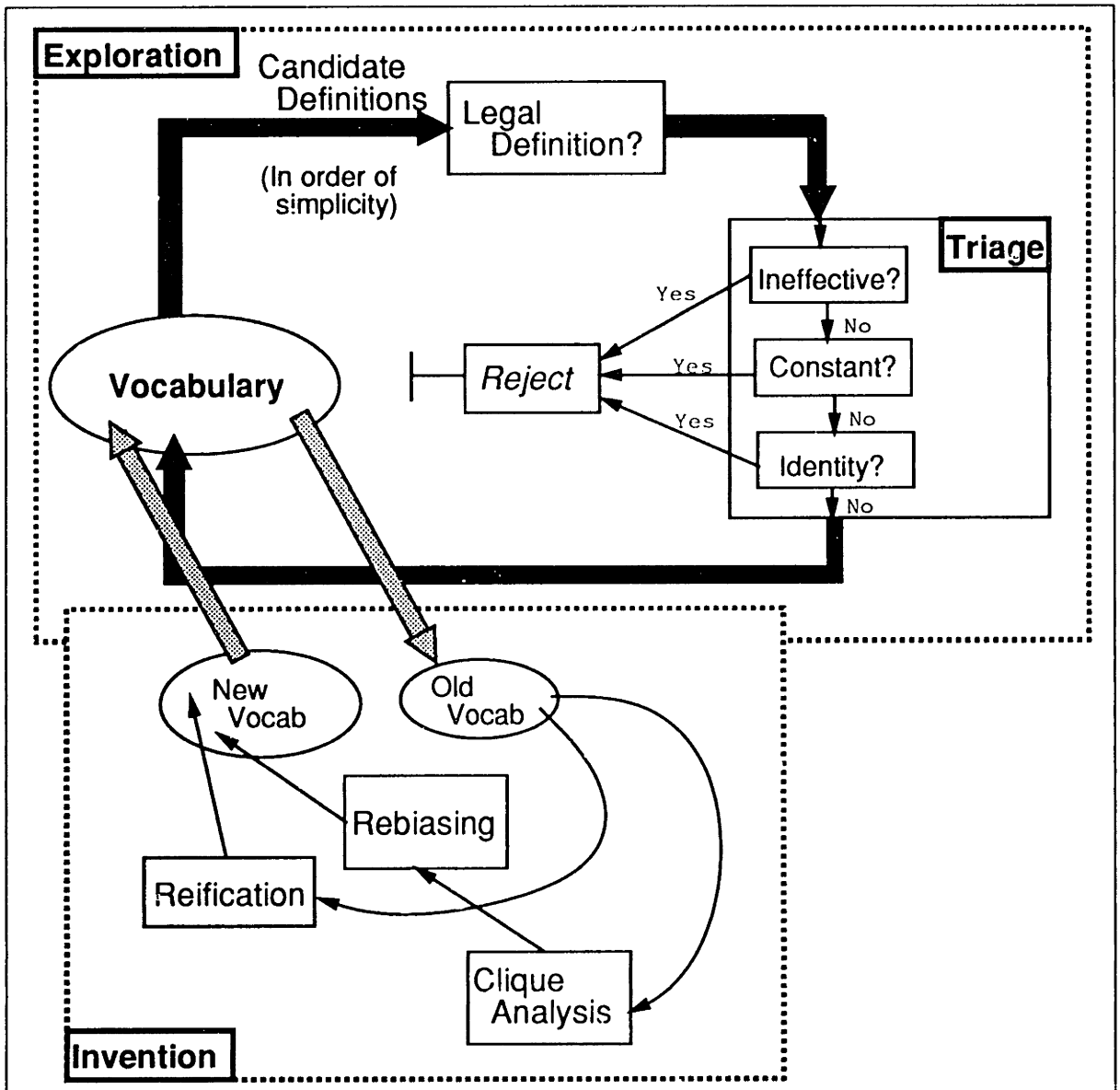


Figure 1.1: Cyrano's basic control loop generates possibilities in order of decreasing simplicity; this generation is constrained by syntactic correctness and empirical effectiveness. This control loop is part of a larger structure which intermittently evaluates the progress in a vocabulary and defines a new vocabulary whose standards of simplicity and identity may differ from those in which exploration has occured thus far.

For instance, Cyrano will not consider the composition of an operation f with an operation g (which will apply g to an input to yield a result to pass to f) unless there are expected to be some results of g which will in fact be acceptable as inputs to f.

Definitions which appear to be reasonable to the type system are subject to empirical examiniation by a 'triage process'; this process attempts to generate examples of the definition and then examine these examples for particular structure or regularities. Triage checks whether a definition is never defined, always specifies a constant value, or always specifies a particular one of its arguments; if any of these is true, the definition is flagged as uninteresting for further combination. If a definition reveals a reasonable diversity of behaviors, it passes through the triage process and becomes a candidate for inclusion in future combinations.

The invention loop operates on the results of this exploration process. The first part of the invention loop takes all of the constructed domain definitions and organizes them into 'cliques' based on their behavior; definitions which behave identically are placed in the same clique. Cyrano's primary criterion of interest is isolating non-singular cliques and biasing the generation process to produce more definitions related to these convergent definitions. This rebiasing considers the fragmentary definitions which play recurring roles in 'popular cliques' and artifically declaring these to be 'simpler' than less common fragments.

In addition to clique analysis, Cyrano considers the *reification* of definitions in the domain. A reification of a definition organizes the partial evaluations of the definition into classes; for instance, reifying the addition operation 'add x and y' yields the family of operations 'add x and 2', 'add x and 3', etc. These families of operations become objects at a new level of description where the operations between objects are the combining operations which first yielded the definitions in the exploration cycle.

By a combination of rebiasing and reification, Cyrano constructs a new vocabulary (or more precisely, a new set of vocabularies) based on the structure and regularities revealed in the previous vocabulary; within this vocabulary, it begins anew the process of exploration.

1.3 Some of Cyrano's Inventions

In this section, I sketch several of Cyrano's discoveries in order to describe how the processes of invention and exploration interact in the program. The descriptions are meant to be explanatory of Cyrano's mechanisms rather then exhaustive of Cyrano's performance. Chapter 3 presents voluminously annotated fragments of Cyrano's actual execution for interested readers.

The discoveries described here are meant to illustrate the elements of the implementation described above. In particular, it describes: the exploration of an initial vocabulary guided by standards of parsimony, effectiveness, and diversity; the retrospective analysis of the vocabulary for definitions of interest and the subsequent

transformation of the vocabulary which bias the generation of similar definitions; and the construction of a new set of primitives based on reifying the vocabulary's operations.

The examples given here describe how Cyrano — starting from a vocabulary of operations over natural numbers — develops a vocabulary of operations over the natural numbers modulo 3.

The purpose of these examples is to illustrate Cyrano's central mechanisms; in fact, many interesting events are left undescribed to enhance the clarity of the descriptions.

1.3.1 Constructing Simple Compositions

Cyrano begins with a set of domain operations and a collection of *combiners* which yield new domain operations from existing ones. We begin our story at a point where Cyrano has constructed the domain of natural numbers and the operations of predecessor and successor upon them; given these, Cyrano proceeds to explore the various combinations and combinations of combinations it can generate from them.

Cyrano's combiners can be understood as small computer programs with empty places into which particular domain operations can be placed. For instance, the combiner **prefix** might be look like this in pseudo code:

```
To PREFIX left to right on x
  LET y be right(x)
   if y is defined,
      RETURN left(y)
      ELSE RETURN 'undefined'
```

where *left* and *right* can be arbitrary domain definitions. Combining **prefix** with the arithmetic operation **predecessor** would yield the following fragment:

```
To PREFIX  predecessor to right on x
  LET y be right(x)
   if y is defined,
      RETURN  predecessor(y)
      ELSE RETURN 'undefined'
```

Cyrano considers new definitions in order of decreasing *simplicity*. Simplicity assessments are assigned as follows: suppose that the fragment **PREFIX** has a simplicity measure of .1 and the definition **predecessor** has simplicity .2; the simplicity measure for the fragmentary definition above (which still has *right* to be specified) would be .02, the product of the simplicity assessments of the components which combined to create it. If the fragmentary definition above were combined with **predecessor** again, the resulting domain definition

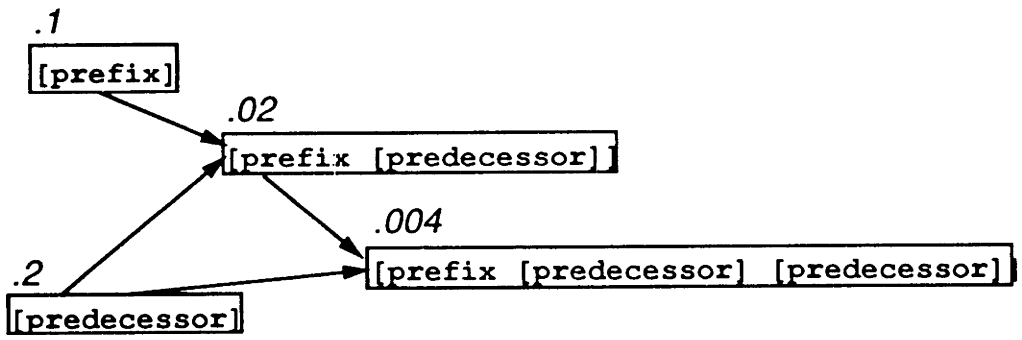


Figure 1.2: New definitions are constructed by combining existing definitions; such definitions are either fragments like `prefix` or domain definitions like `predecessor`. Definitions occur in order of decreasing simplicity, where the simplicity of a definition (the italic annotations above) is the product of the simplicity of those definitions which combined to create it.

```
To PREFIX  predecessor to  predecessor on x
  LET y be  predecessor(x)
   if y is defined,
      RETURN  predecessor(y)
      ELSE RETURN 'undefined'
```

with a simplicity measure of .004 (See Figure 1.2). Empirically, this final domain definition subtracts two from whatever number it is given as input; if the number it is given is one or zero, the definition's result is undefined.

In the descriptions of this chapter and the chapters to follow, I use a bracketed notation to describe domain definitions and fragments: `[prefix]` refers to the program fragment with spaces for *left* and *right*; `[prefix [predecessor]]` indicates the result of replacing *left* with `predecessor` and `[prefix [predecessor] [predecessor]]` describes the domain definition which subtracts two from its input.

Cyrano also has a `postfix` combiner which accepts *left* and *right* in opposite order:

```
To POSTFIX right by left on x
  LET y be right(x)
   if y is defined,
      RETURN left(y)
      ELSE RETURN 'undefined'
```

In any domain, Cyrano begins by exploring the simplest combinations, which include the simplest compositions. It turns out that many of the simplest compositions of

arithmetic primitives are in fact identical; they yield operations like the identity and incrementing or decrementing by one, two, or three; objectively, we can understand that these convergences occur because successor and predecessor are strict functions as well as strict inverses. But from Cyrano's point of view, the coincidence of definitions is unexpected and brings it to focus on the definitions which yield these convergent behaviors. This focussing occurs by enhancing the assigned simplicity assessments for the definitions; this enhancement does not exclude other definitions, so the system is not losing any options, but merely focussing on operations which reveal unexpected regularities.

Cyrano's generation of new definitions is constrained by the triage process; this means that its reinventions of the identity — definitions like `[prefix [predecessor] [successor]]` — are not further combined; however the invention of the identity counts as an intersting regularity which nonetheless contributes to the focussing of attention upon compositions involving successor or predecessor.

To the story we are telling here, one important convergent operation defined and noticed by Cyrano is `minus3` — subtracting three from a number — computed by (for instance) the definition `[prefix [predecessor] [prefix [predecessor] [predecessor]]]` (Figure 1.3). This will play a role in Cyrano's construction of the numbers modulo 3.

1.3.2 Iterative Combinations

Another of Cyrano's combiners is the fragment `TERMINATE` which determines the point at which a domain definition becomes undefined.

```
To TERMINATE next from x
 IF next(x) is undefined
    return x
 OTHERWISE
   TERMINATE next from next(x)
```

The definition `[terminate [successor]]` (using the bracket notation above) never terminates and is thus never defined; the `[terminate [predecessor]]` definition, on the other hand, is always defined and always zero (see Figure 1.4). Cyrano notices both of these properties during its triage process; they both keep the definitions from being used in further constructions but the constant nature of `[terminator [predecessor]]` is noted for use in later rebiasing.

Another iterative combiner is the `meet` combiner which combines two domain operations into a single operation on two inputs; it repeatedly applies each operation to a corresponding input and returns the point (if it exists) where the values returned are identical. It might be defined in pseudocode thus:

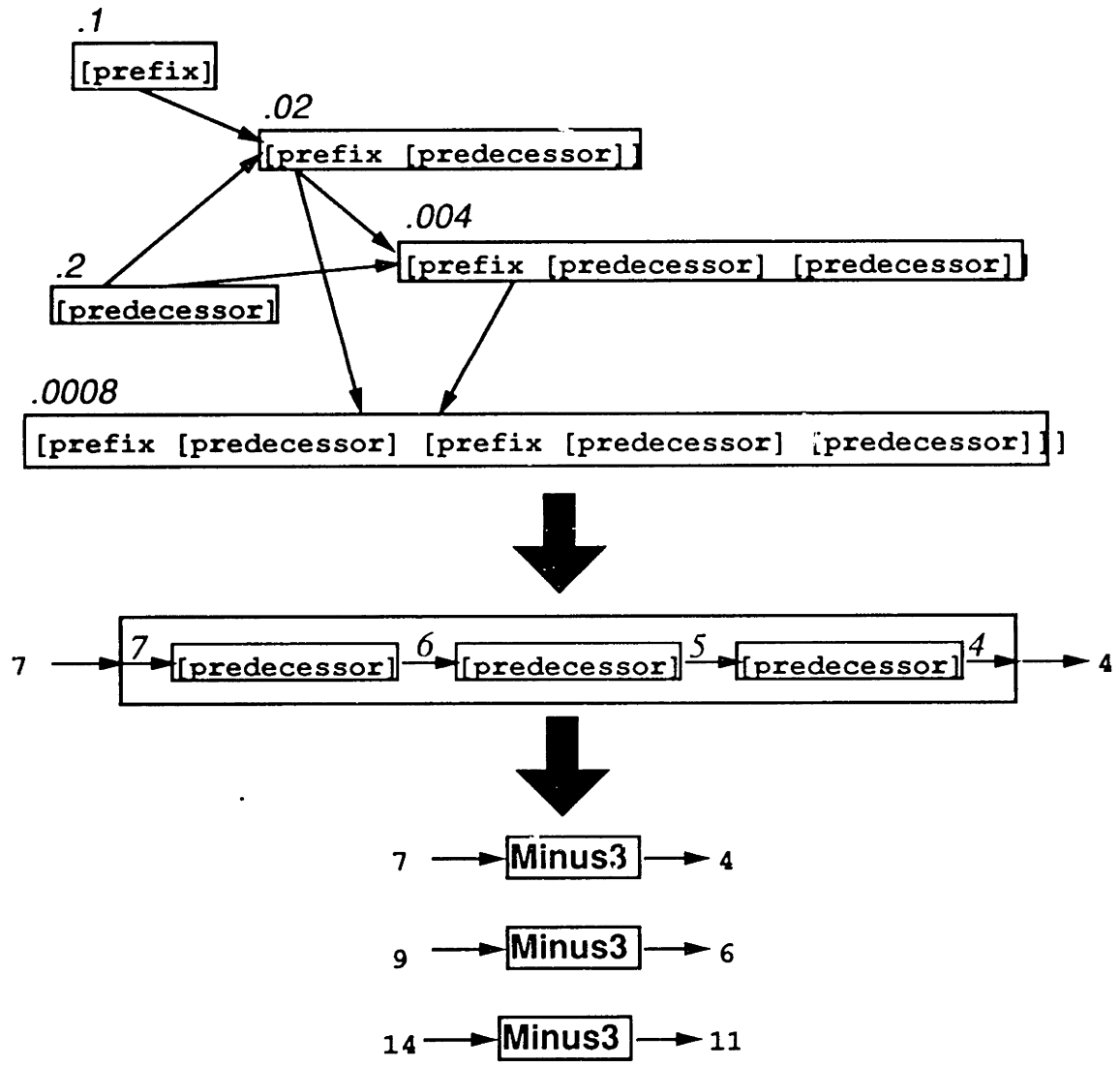
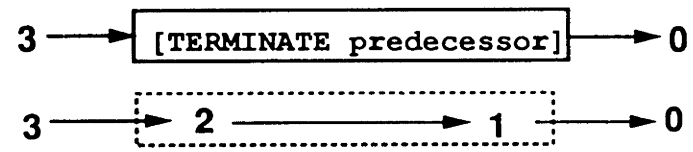


Figure 1.3: Cyrano defines the operation of subtracting three from a number in several different ways: primarily by combinations of **prefix** and **predecessor** operations as pictured here.

```
To MEET f AND g FROM x AND y
   LET X be a set of  x's, initially empty
   LET Y be a set of  y's, initially empty
   ADVANCE ON x AND y BY:
     IF f(x) is in Y
        RETURN f(x)
     IF g(y) is in X
        RETURN g(y)
     OTHERWISE
       ADD f(x) to X and
       AND g(y) to Y and
       ADVANCE ON f(x) AND g(y)
```

```
To TERMINATE predecessor from n
   IF predecessor(n) is undefined
      RETURN n
   OTHERWISE
      TERMINATE predecessor from predecessor(n)
```




```
To TERMINATE successor from n
   IF successor(n) is undefined
      RETURN n
   OTHERWISE
      TERMINATE successor from successor(n)
```

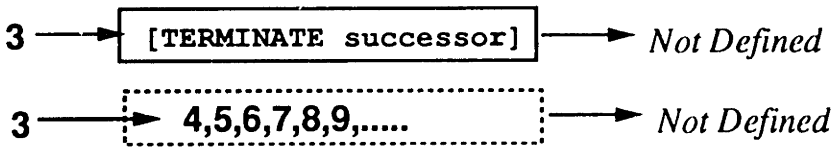


Figure 1.4: Cyrano defines the terminator of operations as an operation which repeatedly applies an operation and returns the input for which it becomes undefined.

The `meet` combiner yields several interesting definitions (Figure 1.5): `[meet [successor] [successor]]` returns the minimum of two numbers; `[meet [predecessor] [successor]]` returns the arithmetic mean of two numbers providing the first is smaller than the second; otherwise it fails to terminate. The definition `[meet [minus3] [minus3]]` (using the composition `minus3` described above) returns the minimum of two numbers if the numbers are equal modulo 3 and is undefined otherwise (Figure 1.6).

1.3.3 Checking Definitions

The distinction between defined and undefined results can emerge in two ways: from definitions whose application does not terminate or from type restrictions on the domains of definitions. For instance, predecessor is not defined for zero, so `predecessor` of zero is undefined; on the other hand `[meet [predecessor] [successor]]`

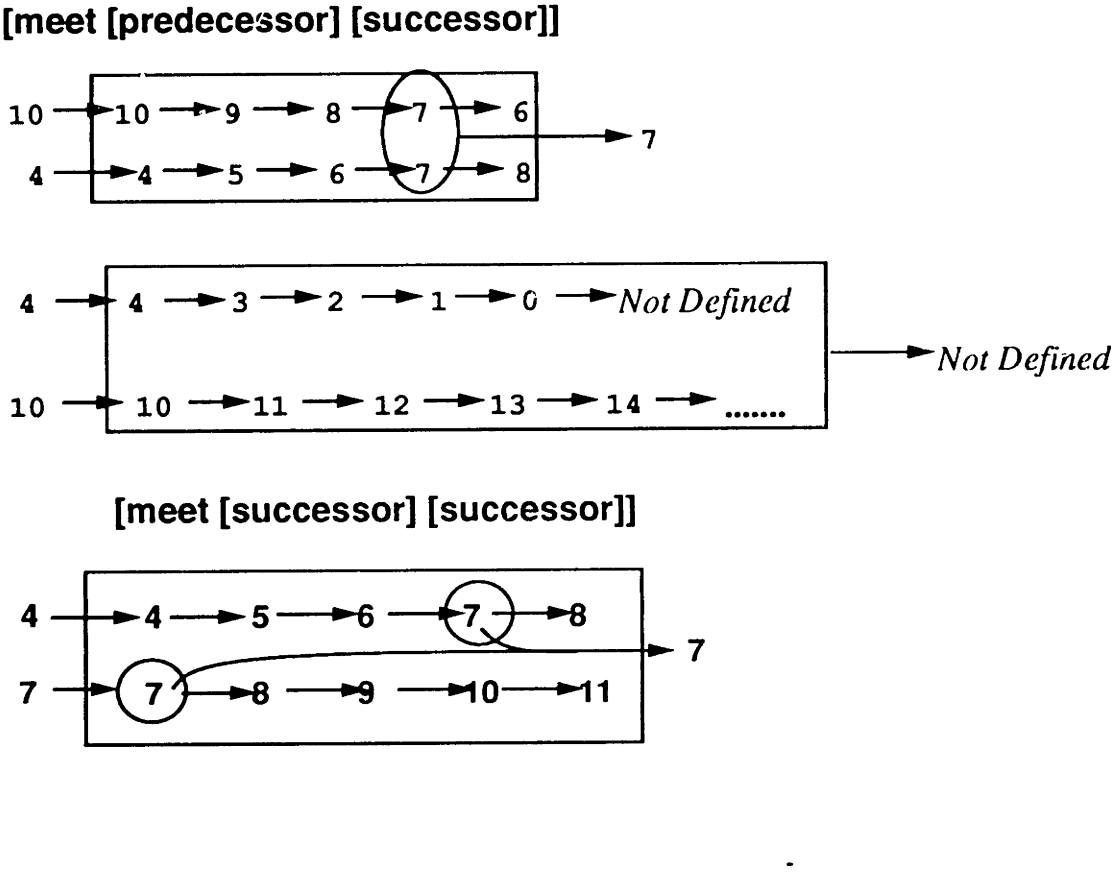


Figure 1.5: The meet combiner returns the values at which a series of results intersects. The definition [meet [predecessor] [successor]] computes an arithmetic mean when its first input is larger than its second and is undefined otherwise. The definition [meet [predecessor] [predecessor]] is always defined and returns the minimum of its two inputs.

computes a mean for $\langle x, y \rangle : x > y$ yet fails to terminate (and is thus undefined) otherwise. Cyrano has a combiner check which converts definition or undefinition into logical values:

```
To CHECK f on x1, x2, x3, ..., xn
   IF f(x1, x2, x3, ..., xn) is undefined
      RETURN False;
   OTHERWISE
      RETURN True
```

The definition [check [predecessor]] returns true for all the 'counting numbers' and false otherwise; the definition [check [successor]] is always true (for num-

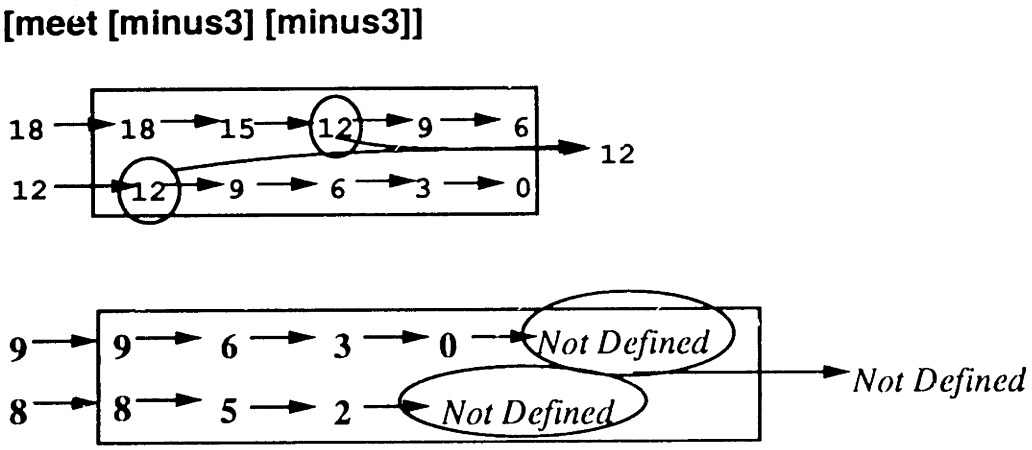


Figure 1.6: When meet combines the minus3 compositions defined above, it yields a version of the minimum operation which is only defined when its inputs are equivalent modulo 3.

bers). For more complicated definitions, more interesting predicates are yielded: [check [meet [predecessor] [successor]]] is the comparison 'greater than or equal to' (Figure 1.7) while [check [meet [minus3] [minus3]]] is the comparison 'equal modulo 3' (Figure 1.8).

This last definition is also computed by the definition [compare [terminator [minus3]] [terminator [minus3]]] which uses the compare combiner:

```
To COMPARE f and g of x and y
   IF f(x) = g(y)
      RETURN True
   OTHERWISE
      RETURN False
```

in combination with the terminator combiner .

This convergence of definitions, like the convergence of compositions above, focusses Cyrano's attention (to a limited degree); this brings Cyrano to consider the *reifications* of the comparison 'equal modulo 3'.

1.3.4 Reifications

A reification of a definition is an organization of the 'partial evaluations' of the operation into classes based on their behaviors. For instance, a reification of addition 'add x and y' consists of the procedures 'add 3 to x', 'add 4 to x', etc; the classes produced by reifications are called 'canons' and provide a canonical representation

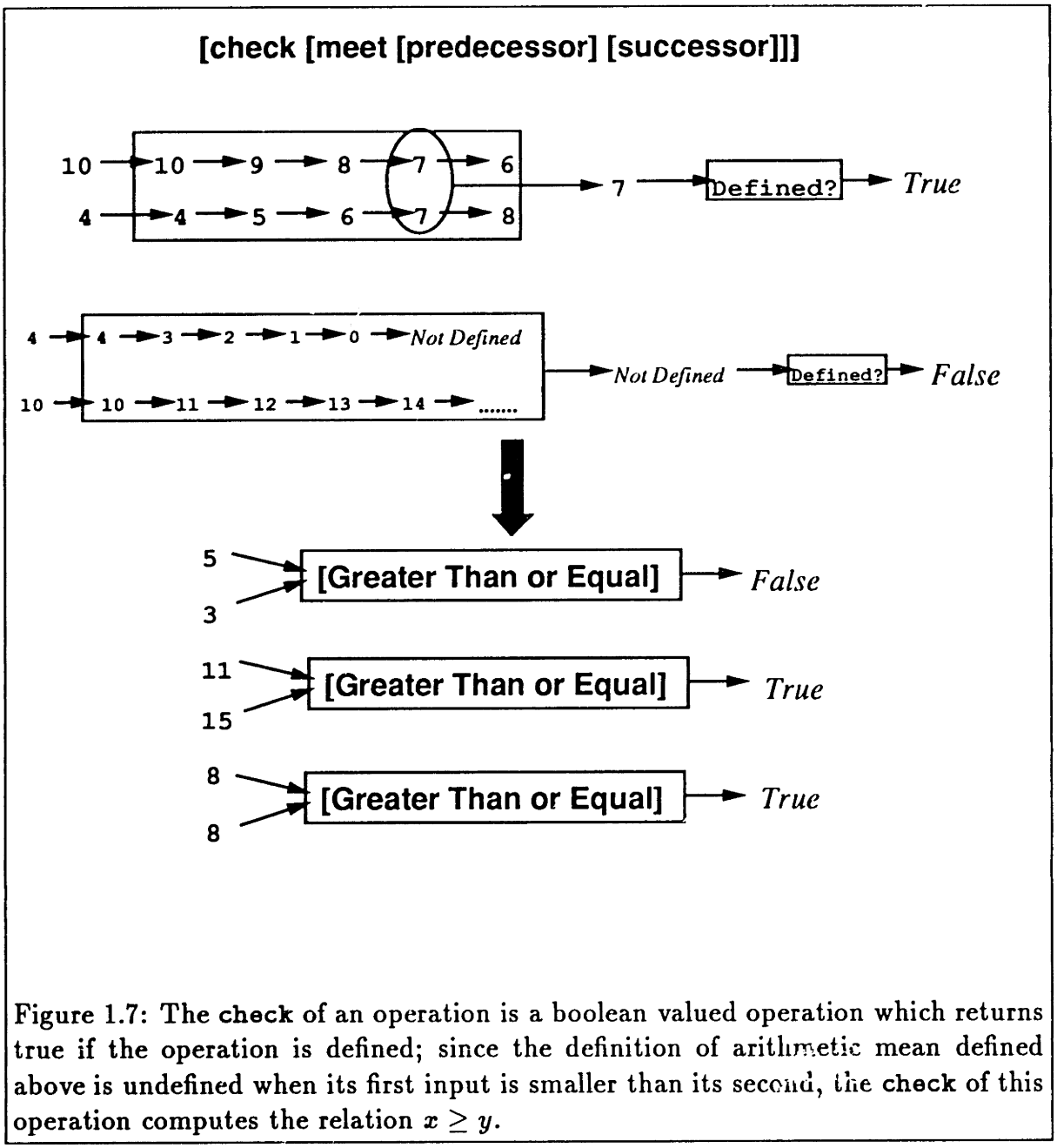


Figure 1.7: The **check** of an operation is a boolean valued operation which returns true if the operation is defined; since the definition of arithmetic mean defined above is undefined when its first input is smaller than its second, the **check** of this operation computes the relation $x \geq y$.

for the partial evaluations of the definition. When a collection of canons has been defined, it is possible to take fragmentary definitions (like `[prefix [predecessor]]` or `[postfix [successor]]` and consider whether they consistently map the procedures of a canon into the procedures of another canon; for instance, `[prefix [successor]]` consistently maps the canon 'add 4' into 'add 5' and the canon 'add 3' to the canon 'add 4'.

Cyrano reifies the operation of comparing two numbers modulo 3 (it also does the

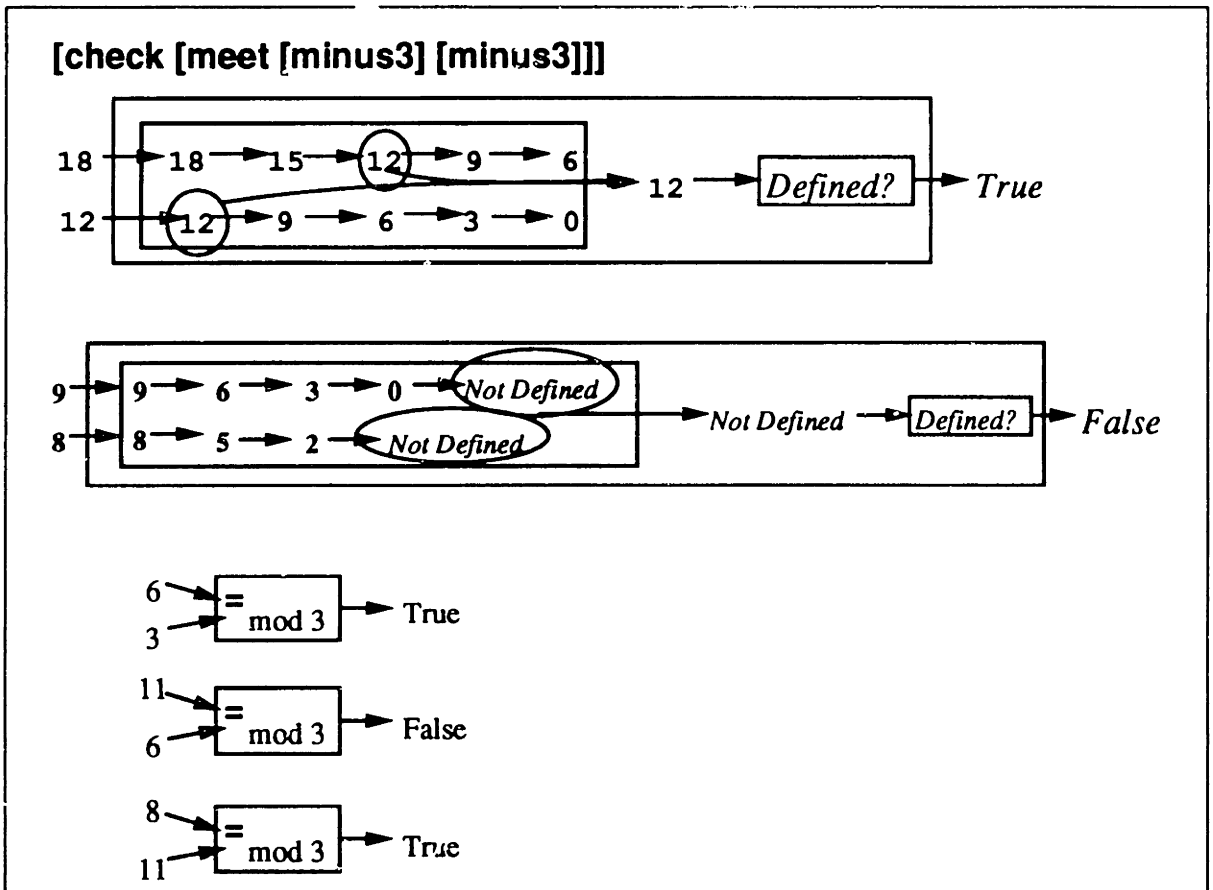


Figure 1.8: The comparison $=_{(\text{mod } 3)}$ is computed by checking whether there is a defined point at which a series of **minus3** operations — applied to two numbers — coincide.

same thing with numbers modulo 2, but that's another story) to yield the canons 'equal to 0 modulo 3', 'equal to 1 modulo 3', and 'equal to 2 modulo 3'. Fragmentary definitions like `[postfix [successor]]` become operations over these canons;

1.4 The Limits of Invention

In the previous section, I sketched how the Cyrano program invents and explores some particular representational vocabularies which are isomorphic to the human vocabularies of arithmetic over the natural numbers and other algebraic fields. It was impressive — and should perhaps be somewhat disturbing — how quickly Cyrano acheived these results in its given and invented vocabularies. This was possible because the vocabularies were *dense* with significant results; many such results could be

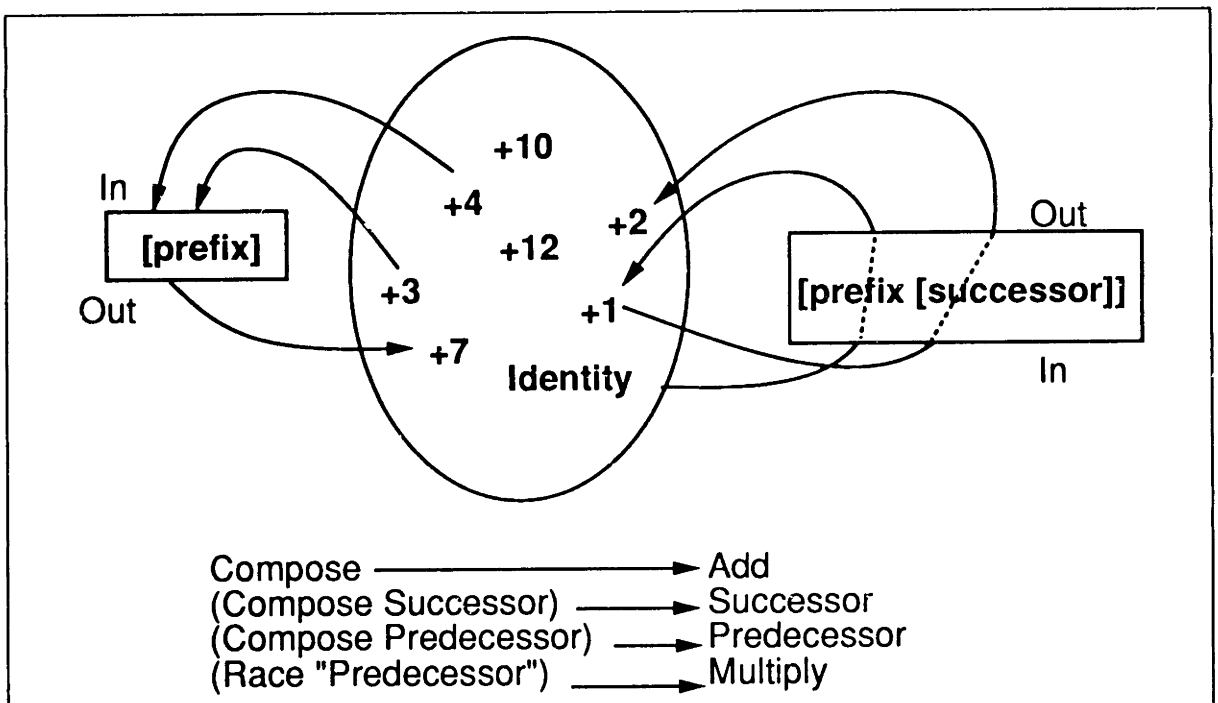


Figure 1.9: Reification takes a class of operations and turns them into objects called 'canons' operated upon by meta-level operations. For instance, the reification of addition consists of procedures like 'add 5', 'add 6', etc (represented by '+5' and '+6' in the figure). Operations on operations — like 'compose with successor' — or just 'compose' become operations on these reified objects.

found among the vocabulary's simplest constructions. This density is characterisic of all mature representations; they make the 'important' things easy to express or encounter.

Other discovery systems like BACON [Langley *et al.*, 1987] or AM [Lenat, 1976] also yielded impressive results due to the density of their representational vocabularies. The BACON program looked for qualitative laws describing systems of experimental variables which already had significant correlations; the AM program developed the structures of arithmetic and elementary number theory by mutating LISP procedures whose recursive formulations already lent themselves easily to describing

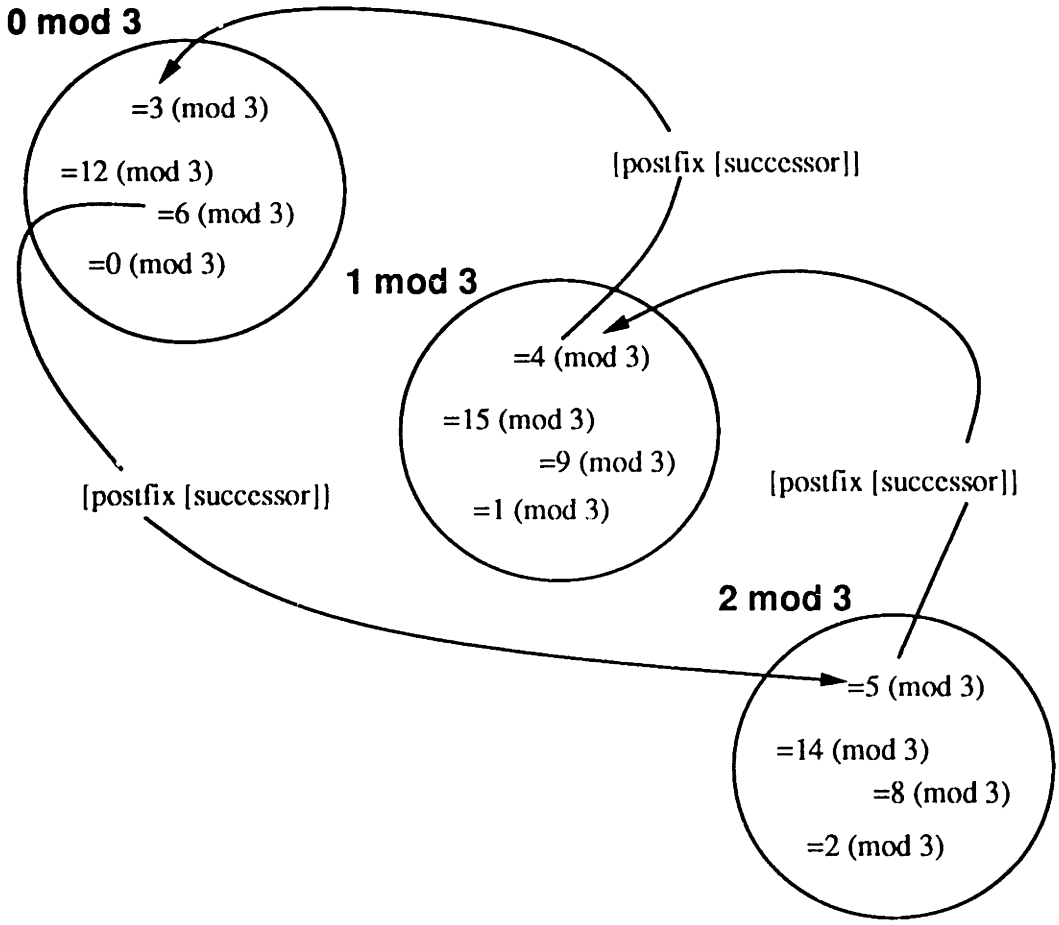


Figure 1.10: Applied to modular equality, reification yields a set of terms and operations isomorphic to arithmetic over the numbers modulo 3; the canons generated by reficiation consist of comparisons with particular numbers modulo 3. There are three such comparisons corresponding to zero, one, and two modulo 3. When these comparisons operations have successor postfixed to them, they are moved into the an operation of the 'next' sort.

mathemtical systems. In both these cases, discovery was able to proceed rapidly because 'early pickings' were so rich; in the descriptions of the previous section, Cyrano's quick progress in individual vocabularies utilizes the same source of power.

A concern often levied against 'creative programs' like these criticizes the density of the representations with which they begin. Characterizing them as 'wind-up toys' whose initial representation prescribes a particular sequence of discoveries, critics question the generality and scope of the program's architectural and representational decisions. But any simple or well-designed system — computational, mathematical

or scientific — is exactly such a 'windup toy' where the simplest descriptions are of great interest; the real innovations in projects like AM, BACON, or Cyrano are the way in which they invent new systems of description which retain the density — the 'wind-up-ness' — of their initial configuration.

For instance, the Bacon program [Langley *et al.*, 1987] searches for quantitative physical laws describing systems of experimental parameters; it does so by considering simple arithmetic combinations (ratios, products, etc) of the parameters and looking for global or local regularities among these derived terms and primitive parameters together. Bacon's considerations are guided by the empirical correlations between the terms it manipulates; it will construct the ratio terms $\frac{a}{b}$ and $\frac{b}{a}$ from two terms a and b if it notes that a and b rise or fall together[Langley *et al.*, 1987, Page ??].

For the goal of discovering qualitative laws, a dense representation is one where combinations yield terms whose derivatives move towards zero (any qualitative equality is equivalent to a function whose derivative is zero). Bacon's consideration of $\frac{a}{b}$ and $\frac{b}{a}$ is reasonable exactly because the derivative of one of them will be less than the derivative of a or b alone.

In looking at any discovery program, we can examine the way it initially provides and progressively maintains or enhances the density of its representations. Lenat's AM program (for instance), focussed on extrema or transformations of operations with intersting emprical properties; constructions around such extrema or transformations were likely to yield interesting results. In Cyrano, the identification of polygenic behaviors and the isolation of 'coherent' reifications both pick out vocabularies where distinctions are sharp enough to yield significant results.

Looking at the various mechanisms for producing dense representations used by programs like BACON, AM, or Cyrano, we might hope that some basic set of operations will enumerate the space of dense representations in a way that is itself dense. Unfortunately, the space of representations is filled with 'garden paths' where theories grow denser and denser while in fact losing breadth and flexibility. Commonly, the cost of power in a representation is flexibility and when a system 'gets stuck', it must sometimes sacrifice both simplicity and power to regain its footing.

A good representation makes important things simple to say; in a representation which is 'good' for generating assertions or claims of type P, these assertions or claims are syntactically simple; but this simplification will make other sorts of assertions or claims more complicated. In particular it may make awkward the description or construction of counterexamples to its coverage or effectiveness.

Any representation imposes particular generative biases and it is both the strength and weakness of representations that they can win effectiveness (being good at generating particular sorts of results) at the cost of coverage (becoming worse at generating other sorts of results). Consider the representation a driver uses in getting from point to point in a sprawling modern city; a useful representation — which makes it easy to figure out how to get between any two points quickly — would focus on the system

of highway routes and describe paths in terms of 'exit to exit' descriptions. In generating routes, the driver would quickly produce a chain of 'highway numbers' which would be effective for getting between places. While the driver may sometimes miss nice short-cuts in unfamiliar areas, such a representation is both effective and — in principle — complete, since it need not be confined to highway exits.

But a hidden cost of this representation is the lost of a crucial adaptivity; as new 'local roads' emerge or possible short cuts are introduced, the driver is unlikely to notice them because such routes are so seldom travelled. Starting from scratch, a new driver might take into account shifts and changes in the local 'back roads' and produce an ideal merger of highway and city street routes; yet the more experienced driver is less likely to see this 'big picture' because the routes she takes are those which are familiar and reliable.

This property of representations bodes ill for attempts at general and uniformly reliable mechanisms for the progressive refinment of representations; while we may improve our representations along particular dimensions, these improvements come at the cost of a loss of flexibility in other dimensions. Thus, the space of representations is dotted with hilltops where one part of the world is elegantly and parsimoniously described, but from which the rest of the world recedes into the distance. While there may be a 'perfect representation', we can never know if it is the mountain we are actually climbing or yet another mountain shrouded in the distance by the choices of path and perspective we have made. It is well known that a representation is only 'good' with respect to certain goals, but even if those goals are clear, the judgement of the representations effectiveness is still limited by the things it makes easy or difficult to say.

What are we to make, then, of systems like Cyrano and its predecessors? They are attempts at devising general mechanisms for 'climbing' the hilltops of any given representation; but these systems cannot hope (and neither can we) to have an unerring compass that will consistently improve their representations. Because our chances of entrenchment increase with the effectiveness of our representations, to switch representations we must step back to a weaker and less effective representation which does not allow the same 'easy solutions' as our specialized domain representations.

Discovery systems are studies of general mechanisms which — given a set of simplicity criteria — will explore the space those criteria dictate and suggest some transformations of those criteria based on experience with their consequences. We can then evaluate a discovery system by asking whether its transformations agree with our own standards and expecations: is the program devising simpler ways to say things which we consider interesting?

1.5 Relation to other Work

The work described here has two distinct components of different lineage: the Cyrano program has both historic and inspirational links to discovery programs like Lenat's AM [Lenat, 1976] and Eurisko [Lenat, 1983] or the seminal work of Langley, Simon, and their colleagues [Langley *et al.*, 1987]; on the other hand, the view of discovery as vocabulary formation has its roots in the thought of Jean Piaget and Thomas Kuhn. This section sketches those roots briefly.

Readers are also directed to Chapter 5's discussion of the the AM and Eurisko programs, Chapter 6's discussion of the systems of Herbert Simon, his students and grand-students, and Chapter 7's discussion of the relation of this viewpoint to constructivist positions in developmental psychology and the history and philosophy of science. Finally, an overview of the connection of this work with other work in the field is found among the conclusions of Chapter 8.

1.5.1 Technical Roots

As a discovery program, Cyrano's performance domain is closest to that of Lenat's now-classic AM program. Cyrano surpasses AM in some respects while not matching AM in others; Cyrano's forays into other varieties of arithmetic are new while Cyrano did not reproduce AM's discoveries in primality and simple number theory. But the intent of Cyrano was to focus on the 'revolutionary' changes of representation characteristic of (for instance) AM's discovery of the natural numbers. In this area, Cyrano makes significant strides by both its initial invention of numbers (in various ways) and its subsequent inventions of modular fields.

A notable milestone in computational accounts of scientific discovery is the work of Simon and his colleagues, particularly in programs like BACON and GLAUBER which recapitulate versions of notable discoveries from the history of science. In contrast, the Cyrano program works in a very different domain: the invention of arithmetic systems and the exploration of their properties. But in viewing discovery as vocabulary formation, we note that systems like BACON and GLAUBER are given most of the vocabulary (identity criteria, experimental setups, relevant measurements, etc) for the discoveries they make. Indeed, BACON's developers scoured the history of science for moments where the terms and methods were already present and most of the work of discovery (at least from the vocabulary formation standpoint) had already been done. Insofar as Cyrano attempts to be a principled approach to discovery as vocabulary formation, it could not make discoveries in any domain to which it did not have direct access. New vocabularies are constructed from the degrees of freedom of old vocabularies, and without access to the materials of experimentation such construction is impossible.

Cyrano's connection to other work in Machine Learning is best explained by invok-

ing the wisdom of William Martin: "You can't learn anything until you almost know it". Rather than focussing on particular accounts of learning, Cyrano has focussed on "how you get to almost know something"; its model of learning is trivial: keep trying combinations until you find one which fits. While future work will undoubtedly focus on improving this element of Cyrano,[2] this will be limited by the degree to which such learning methods can really be freed from dependence on particular domains and representational schemes.

Finally, the work of Ray Solomonoff on discovery systems has provided a significant contribution to the ideas and implemenation of Cyrano. The rough organization of Cyrano's control structure is a version of the framework suggested by Solomonoff in [Solomonoff, b]. His work in discovery builds on his seminal work in algorithmic probability [Solomonoff, a].

1.5.2 Philosophical Roots

The idea that our programs might be seen as *constructing* their vocabularies for interpreting the world is my inheritance from the work of Jean Piaget and his philosophical forbearers. These forebearers, beginning with Immanuel Kant , attempted to balance empiricism and idealism by an account of how the very act of perception imposed structure on our sensory understanding. But in experiments that would have profoundly disturbed Kant, Piaget and his colleagues revealed that the structures imposed by children are very different from those imposed by adults. The phenomena of a child's *developing* understanding demonstrates both how much of our understanding is constructed and how much that construction emerges from the child's interaction with the world.

On the other side of the world's hierarchies from the children are the institutions and practices of science; these have provided another rich example of the evolution of ideas. A vocabulary bears some similarity to what Thomas Kuhn calls a paradigm [Kuhn, 1961]. More immediately, Kuhn's recent work has focussed on the role that vocabulary plays in the formation and evolution of scientific theories[Kuhn, 1983, Kuhn, 1989, Kuhn, 1987]; his examples from the history of science have convinced me even more of the importance of distinguishing vocabulary invention in any account of creative process.

Finally, yet another philosophical root of this work lies in those theories of Marvin Minsky which describe understanding and the growth of understanding as emerging

[2]Actually, earlier versions of Cyrano had more interesting learning models, but they were abandoned out of my concern that they were too domain depdendent; described in [Haase, 1986a, Haase, 1987], these systems learned by trying to identify 'cognitive cliches' in the phenomena under investigation. These cognitive cliches were largely mathematical in nature and I was concerned that such 'interestingness criteria' might be biasing the system too heavily in the direction I wished it to go. It is interesting that Cyrano's current criterion of interestingness — the identification of polygenic behaviors — brings the program to identify many important cliches on its own.

from the combination of simple elements in complicated patterns of activity. In both the design of the Cyrano system and the consideration of how any system's knowledge might evolve, I have returned to these ideas about building complicated systems from simpler pieces.

1.6 Structure of the Thesis

This thesis describes both the implementation and operation of the program Cyrano and the perspective of invention and exploration which underlies its design. The description of Cyrano is comprised of Chapter 2's overview of Cyrano's implementation and Chapter 3's annotated transcripts of fragments of Cyrano's explorations.

The discussion of invention and exploration in discovery commences in Chapter 4 with a discussion of the distinction between *generative* and *descriptive* efficacy in a representation and the problem of *entrenchment* arising from the tension between these properties. After this is introduced, a discussion of Lenat's AM program [Lenat, 1976] is presented in the form of an 'exegesis' describing the 'good run' of AM from the standpoint of vocabulary invention. The description of AM is succeeded by a critical description of the BACON program which argues that BACON's underlying thesis — that discovery is a species of problem solving — is somewhat misguided; in particular, the processes of invention occur in a sufficiently weak space that the applying the label of problem solving to them is inappropriate without making its definition so broad as to be virtually meaningless.

Chapter 2
Cyrano's Implementation
An Overview

This chapter offers an overview of the design and operation of the Cyrano program. The descriptions here provide the context for the detailed description of a 'full run' in Chapter 3.

In addition to believing that technical issues and their resolution may be of interest to some readers, this detail is intended to support researchers interested in reproducing or building upon the Cyrano program. A common methodological criticism of AI research points to the irreproducibility of reported results. This has particularly been the case in domains like discovery [Ritchie and Hanna, 1984] where the success of the program is measured by particular results starting from a particular situation. Though I will argue in Chapter 7 that this criterion of judgement may be inappropriate for discovery systems, the focus on reproducibility remains important.

2.1 The Program

Cyrano is implemented in Scheme [Abelson and Sussman, 1985, Rees and Clinger, 1986] on top of a package of utilities described in [Haase, 1990] and a type representation language called TYPICAL described in [Haase, 1987]. The size of the system in December 1989 breaks down something like this:[1]

Component	*Lines*	*Symbols*	*Definitions*	*Image*
Utilities	2491	11336	500	10000
TYPICAL	4618	20209	1000	100000
Cyrano	1992	7687	500	10000
Total	10000	4000	1000	100000

For the most part, Cyrano (and its sizable substrate) are implemented in the Scheme standard described in [Rees and Clinger, 1986]. Development of Cyrano has been done — at various times — under MIT-Scheme on a variety of Unix™ workstations, under Jonathan Rees' *PseudoScheme* in both Symbolics™ and Kyoto Common LISP, and under Texas Instrument's PC-Scheme™ for IBM-compatible computers. But most of the significant development work was done under MIT Scheme on Hewlett-Packard workstations.

[1]The column *Symbols* refers to the number of unique symbols in the source code; the column *Definitions* is the number top-level definitions; and the the column *Image* refers to the number of 32 bit words in 68030-compiled version of the system.

Scheme was chosen because transparence and transportability were goals from Cyrano's initial conception. The definitions produced by Cyrano are procedures which are combinations of other procedures. It is important that such procedure-combining procedures be easy to write. While other languages or LISP dialects might support the same sorts of definitions with nearly equivalent ease, the particular transparence of such definitions in Scheme provided conceptual as well as technical support for the implementation of the ideas behind Cyrano.

To apply the vocabulary of vocabulary invention to the development of Cyrano, the Scheme language has provided a vocabulary which makes perspicuous the description of significant computational ideas. My ideas about definitions and combinations have undoubtedly been influenced by the community of ideas which maintains (and restrains!) the Scheme language.

2.2 Typical

Cyrano is implemented on top of TYPICAL, a type representation language described in great detail by [Haase, 1987]. Readers seeking to understand the depths of Cyrano (or trying to unravel either a copy of the sources or a running version of the program) are advised to examine both [Haase, 1987] and [Haase, 1990] for an understanding of the 'vocabulary' in which Cyrano is embedded.[2] This section sketches the operations of TYPICAL in sufficient depth to support the explanations of the forthcoming chapters; interested readers are referred to these other documents for further elaboration.

TYPICAL is a combinator language for boolean predicates in a subsumption lattice. A predicate or *type* is placed above or below other types in this lattice; the types above a type are its *generalizations* while those below are its *specializations.* An object satisfies a type if the corresponding predicate procedure returns true for the object; if an object satisfies one type, the object satisfies all of the types above it in the lattice. TYPICAL provides a collection of *type combinators* for constructing new types from existing types and automatically placing these types in the lattice; these combinators are procedures which take existing TYPICAL types (and an assortment of other objects) as inputs and return types as results. TYPICAL also provides a mechanism for introducing new primitive types and placing them in the lattice.

TYPICAL might be considered the 'inference component' of Cyrano; many of Cyrano's definitions refer to types in the lattice and the relation of these types to other types sometimes constitute important results about the relations between definitions. In particular, TYPICAL is used to 'type' the definitions produced and used

[2]Many of the questions often asked about the Eurisko program [Lenat, 1983] were answered in descriptions of the representation language language RLL-1 [Greiner, 1980]. Unfortunately, in published accounts, the importance of this connection was not adequately indicated; this lead to some portion of the confusion and skepticsm with which Eurisko was received.

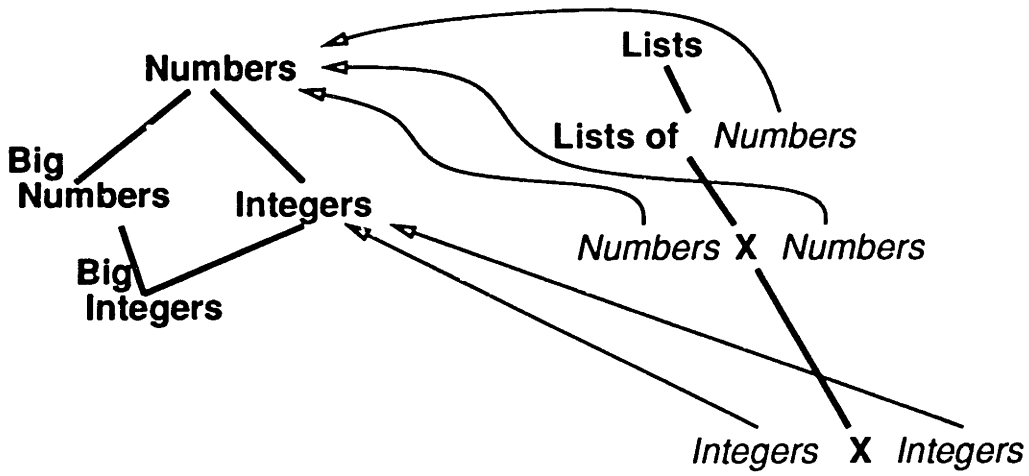


Figure 2.1: This is a fragment of the Typical lattice constructed by the definitions in the text.

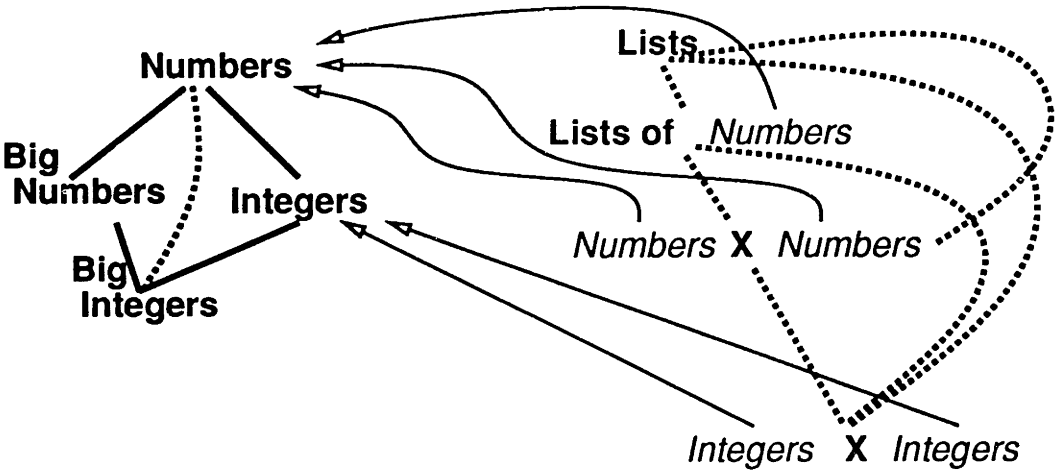


Figure 2.2: These are the inferences made by Typical about the definitions given in the text and diagrammed in the previous figure.

by Cyrano; each of Cyrano's definitions is a procedure whose inputs and outputs are characterized by TYPICAL types.

An earlier version of Cyrano used the lattice more extensively, constructing each invented definition as a type in TYPICAL's lattice. However, the interesting properties of such definitions were usually determined empirically rather than by inference in the lattice and it became clear that the overhead of representing all definitions in the lattice was not justified by the inferences the lattice was able to make about the new

definitions. This became especially evident when Cyrano moved to domains of its own definition and every domain object was a different TYPICAL type to be maintained in the lattice.

As a module, transactions with TYPICAL take two forms: type declarations and type queries. Type declarations create new types based on either existing types or other lisp objects (e.g. 'bachelors' are the intersection of 'men' and 'unmarried'; 'numbered-lists' are all lists whose first element is a number); type queries are either satisfaction queries ('does x satisfy the type T?') or subsumption queries ('are all instances of S also instances of T?'). Due to a heavily engineered implementation, TYPICAL is able to provide time complexity guarantees for these operations: subsumption queries are guaranteed to be constant time operations and type declaration is at worst case an $O(n^2)$ operation (where n is the total number of types in the lattice).[3]

TYPICAL's combinators provide logical combinations of existing types, constraints based on the images of procedures, and recursive definitions of potentially infinite objects; an extension of TYPICAL allows the description of types of procedures based on their domains and ranges. Some examples of TYPICAL definitions (whose representation as a fragment of the lattice is show in Figure 2.1) might be:

Numbers are a sort of thing.
Big Numbers are *Numbers* larger than 20.
Integers are a sort of *Numbers*.
Big Integers are *Integers* which are also *Big Numbers*.

Figure 2.2 shows (with shaded lines) the inferences which TYPICAL makes about the definitions diagrammed in Figure 2.1. It can tell that big integers are a sort of number and that pairs of integers are also pairs of numbers and that both are lists of numbers. These sorts of inferences allows Cyrano to tell something about the definitions it constructs without needing to painstakingly explore their empirical behavior.

2.3 Representing Definitions

Cyrano's definitions are Scheme procedures which accept a single input and yield a single result; in certain cases — if the input is of an inappropriate type or the procedure's evaluation fails to terminate in some alloted time — the result is said to be *undefined.* Operations which require more than one input (such as addition) are implemented by *curried* definitions; a curried definition of addition, for instance, would accept an input such as '3' and yield a procedure which adds 3 to a number.

[3]These guarantees are made at a cost in completeness; since subsumption is NP-complete (see [Haase, 1987]), only by sacrificing completeness is time complexity kept in the polynomial domain.

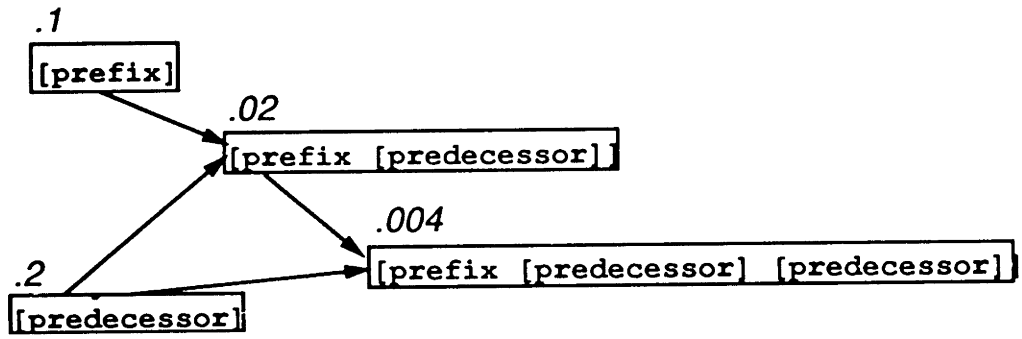


Figure 2.3: New definitions are constructed by combining existing definitions; such definitions are either fragments like `prefix` or domain definitions like `predecessor`. Definitions occur in order of decreasing simplicity, where the simplicity of a definition (the italic annotations above) is the product of the simplicity of those definitions which combined to create it.

The domain and range of all procedural definitions are *typed* by TYPICAL definitions; for instance, Cyrano knows that the `predecessor` operation accepts a number as input and yields a number as a result. In the case of curried definitions, the range of the definition is a type of definition; `plus` operation accepts a number as input and produces a new definition which also takes a number as input but (in this case) yields a number as a result.

Based on the types of inputs and outputs, Cyrano's definitions are further divided into *domain definitions* and *fragments*. Domain objects for Cyrano are a collection of datatypes which *do not* include definitions; a domain definition accepts domain objects as input and yields either a domain object or a domain definition (if the definition is curried) as a result. Fragments are definitions which either accept other definitions as inputs or accept objects as inputs and yield fragments as results. Fragments can be understood as small computer programs with empty places for certain steps; when applied to an input they return a program with some of the places replaced by the input.

This process of application is called *combination*; most of Cyrano's definitions, outside of a small space of root definitions, are produced by combination of fragments with domain definitions. The fragment applied to yield a particular definition is called the 'mother' of the definition; the object or definition to which it was applied is called the 'father' of the definition. An example of combination is shown in Figure 2.3 where the fragment `prefix` is twice combined with the domain definition `predecessor` to yield the definition `[prefix [predecessor] [predecessor]]`.

Combination may yield fragments for further combination instead of immediately producing domain definitions. For instance the `postfix` fragment takes a domain

definition as input but yields a fragment which transforms domain definitions by first processing their inputs through the first domain definitions. Thus, [postfix [predecessor]] (produced by combining postfix with predecessor) is a fragment which takes another definition (call it D) and yields a third definition which takes an input x and calls D on the predecessor of x. If D were successor, the resulting combination [postfix [predcessor] [successor]] would be the identity, decrementing and the incrementing its input.[4]

Both fragments and domain definitions are annotated with domain and range information. For fragments, these annotations are used to constrain the consideration of possible combinations; for domain definitions, the annotations are used to generate examples of the domain definition 'in action' over the domain.

2.4 Control in Cyrano: Growing Pools

Cyrano organizes definitions in collections called *pools*. Within each pool, every definition is assigned a 'simplicity' between zero and one. One definition is simpler than another if its simplicity is numerically greater. In the case of a combination (like [terminator [predecessor]] the simplicity of the definition is the product of the mother and father definitions which combined to create it. In Figure 2.3, the simplicity is denoted by the number above the upper right corner of the definition. The root definitions (terminator and predecessor) in this case) are assigned initial probabilities by either the user or the vocabulary invention process which produced them.[5]

Pools are incrementally *grown* by the consideration and addition of new definitions from the combination of definitions already in the pool. The order in which such combinations are considered is determined by their relative simplicity; simpler definitions are always considered bfeore less simple ones. Since the simplicity of a combination is a function of simplicities already determined for the pool, control preferences can be implemented by altering the simplicity measures for fragments or domain definitions within the pool.

Simplicity measures are used to order the consideration of combinations. The combinations actually performed are further constrained by the typing information on fragments and domain definitions. A combination only occurs if the potential father satisfies the domain type of the potential mother; other combinations are rejected.

After a combination has been considered and accepted it is added back into the pool from which it was created; if it is a fragmentary definition it will then be able to combine with other definitions already in the pool. On the other hand, if it is a domain definition, it also goes through an additional process, called *triage*, which

[4]Actually with predecessor and successor defined over the natural numbers, the operation would be undefined for zero, distinguishing it from the identity operation.

[5]Speaking broadly, we could characterize the user as a vocabulary invention process.

determines its suitability for combination with fragments already in the pool. This determination is based on an empirical analysis of the definition which places it in one of three categories:

- **Ineffective.** It is impossible to generate any examples of this definition, so its 'simplicity' is reduced to zero. (It will never be used in future definitions).
- **Trivial.** The behavior of the definition is syntactically trivial (for instance, always returning a constant or one of its arguments). This fact is annotated and the definition's simplicity is reduced to zero. While this means that it will not be used in any future definitions, it may suggest directions for future reformulations of the vocabulary.
- **Acceptable.** The generated examples of the definition show some interesting variety. The *a priori* simplicity and content of the definition are unchanged.

Once a domain definition's simplicity has been approved or altered by the triage process it is added back to the pool and the next simplest combination is considered. This iterative process completely describes the control structure guiding Cyrano's *exploration* of a given vocabulary; after some period of exploration — determined by the number of new domain definitions produced, a clock time limit, or perhaps human observers — the system pauses to examine the body of definitions it has produced and to consider both altering the simplicity measures assigned to individual definitions and the possibile construction of new primitive vocabularies by *reifying* the operations of the existing vocabulary.

2.5 Reformulating Pools: Rebiasing and Reification

Cyrano's reflection begins by looking for three sorts of regularities:

- **Constant Values.** Definitions which always returns the same result.
- **Projections.** Definition which always return one of their inputs.
- **Polygenic Behaviors.** Families of otherwise unrelated definitions which yield identical results for identical inputs.

These first two sorts of definitions were identified by the triage process and barred from further combination; however, their regularity suggests underlying structure which is not explicit in the system's current representation. Cyrano examines these definitions not to use them in future combinations, but to determine — in a vague, statistical way — what sorts of combinations yielded them.

Polygenic behaviors, on the other hand, are both important for combination in future definitions and for the character of the combinations which produced them.

This analysis of 'character' is done quite primitively; a frequency analysis is performed over the occurence of consituent definitions in combinations showing interesting regularities (the three classes listed above). This analysis is then used to select out those constituent definition whose frequency of appearance is more than two standard deviations above the mean frequency.

This selected set is used to define a new generative basis for future exploration of the current domain. These definitions are chosen so as to have counterfactually 'made it easier' to invent the definitions of interest. For a variety of technical reasons, Cyrano begins from scratch with these new definitions and indeed reinvents the interesting definitions more quickly than their original invention. In a more focussed and domain-dependent version of Cyrano, other interestingness criteria might be applied to guide the same sort of rebiasing; however, the definition of polygenicity subsumes — in some sense — most other interstingness criteria since we can encode some other criteria of interest as a family of definitions with which a search for polygenic definitions can find correlations.

Rebiasing does not change the space of definitions the system *can* construct, but only how quickly those definitions will be constructed. Cyrano's other major invention mechanism — reification — constructs new objects and operations between them which would not have been constructed in the original framework.

Reificiation begins with the the partial products of curried definitions: procedures like 'add 3', 'add 4', 'subtract 5', etc. Reification transforms these procedures into objects — called 'canons' — which can be operated upon by fragments like '`terminator`' or 'postfix predecessor' to yield other procedures which may or may not correspond to canons of the same partial products. For instance, the canon 'add 4' can be processed by 'postfix by predecessor' to yield 'add 3'; however, applying 'postfix by predecessor' to 'add 0' (the identity) will yield a corresponding canon only if the domain of addition includes the negative numbers.

The translation to canons transforms fragments into operations in a new vocabulary; thus `[postfix [predecessor]]` is translated into an operation over canons of addition (it might also be translated into an operation over canons of subtraction or other operations). It also introduces a set of new domain objects corresponding to the constructed definitions which correspond to canons; thus if the system had the domain definitions `[successor]` and `[postfix [successor] [successor]]` in its original vocabulary, these would be translated into domain objects (of special interest) 'the canon of add 1' and 'the canon of add 2' in the new vocabulary.

The construction of a new vocabulary after a period of exploration combines the rebiasing of primitives based on identified regularities and the construction of wholly new objects and operations based on the reification of existing operations.

2.6 Identifying Polygenic Behaviors

In a trivial sense, all behaviors are polygenic: for instance, composing any definition with the identity produces a new definition with the same behavior but with a (trivially) different definition. Yet such trivial transformations apply to any procedure; more interesting are the transformations which change the behavior of some definitions while leaving the behavior of others unchanged. For instance, only an associative operation is behaviorally invariant under the permutation of its inputs.

In any domain with known properties, polygenic behaviors are interesting because they provide extra degrees of freedom in planning, description, or explanation. In domains whose properties are not yet known, polygenic behaviors indicate regularities which are not explicit in the representation of the domain.

In most of my experiments with Cyrano, the sole criterion of interest has been the identification of polygenic behaviors; but because this criteria is applied in the context of a whole corpus of definitions, it emerges that the system defines its own 'criteria of interest' by the space of definitions it develops over time. For instance, Cyrano early on develops the notion of 'greater than' in a variety of ways. This means that Cyrano will find interesting those definitions whose behavior is describable in terms of 'greater than'; for instance, monotonic functions or operations.

Cyrano identifies polygenic behaviors by organizing domain definitions into *cliques* of definitions with the same behavior. It begins with cliques based on the domains and ranges of domain definitions and then iteratively attempts to 'split' cliques by generating examples on which their members disagree. This splitting process is recursively applied to produce a final set of smaller cliques whose members agree on some large number of definitions. In general, cliques will form a lattice, because two definitions might agree on some cases and disagree on others; but to avoid maintaining n^2 cliques, Cyrano only identifies 'perfect' polygenicity though intermediate schemes may be possible. One such scheme would form a strict hierarchy of cliques by putting together definitions which agree whenever they are technically 'defined' (for instance, over the natural numbers, $x - y$ and $y - x$ agree on $x = y$ but are undefined anywhere they might disagree.)

Polygenic behaviors emerged as a generalization of an earlier set of interestingness criteria based on *cognitive cliches* [Chapman, 1986]. Cognitive cliches are largely domain-independent formally-specifiable regularities like 'transitive', 'symmetric' or 'continuous' which abstractly describe the behavior of relations and operations. However, I was worried that such semi-mathematical categories might be unduly influencing Cyrano's progress and furthermore realized that such cliches could be described as fixed-points for various operational transformations. Symmetric relations are the fixed point of inversion; transitive relations are the fixed point of transitive closure; and continuous functions are the fixed points of arbitrary offsets of their domain [?].

If the rebiasing process (the shifting of assigned simplicities described in the next

section) moves the system towards the production of polygenic behaviors, cliches will emerge from the commonalities among dominant cliques. For instance, one of Cyrano's combiners is a MIRROR operation which swaps another operations inputs; the MIRROR of $f(x,y)$ is $g(x,y) = f(y,x)$. The MIRROR operation often produces definitions which coincide with others, enhancing the importance of MIRROR in future vocabularies; but in an oddly circular loop, this will tend to enhance definitions yielding such symmetries, such as associative operations $f(x,y) = f(y,x)$ or mutual inverses $x \sim y \leftrightarrow y \sim x$.

2.7 Behavioral Reification

One of Cyrano's most powerful vocabulary invention mechanisms is the reification of domain level behaviors into objects at another level of description; operations or processes at one level become objects at the level 'above' it and the interaction of the lower level processes become operations and processes at the level above it. For instance, in one experiment where Cyrano was given the primitives of predecessor and successor over the natural numbers, Cyrano eventually defined addition and used reification of it to produce an 'interval arithmetic' where the objects consisted of procedures which repeatedly called successor; e.g. 'add one', 'add two', 'add three', etc. In addition to being isomomorphic to the original vocabulary ('add one' corresponded to '1'), this vocabulary turned out to be much more effective for describing (unsuprisingly) addition and (more of a suprise) multiplication.

In the current implementation, reification starts with the curried domain definitions in a vocabulary. These definitions take domain objects and return other domain level definitions; for instance, addition takes a number and returns another definition which will add that number to a second number and return the sum. The reification of an operation is the organization of these intermediate definitions into categories based on their behavior. These categories are called 'canons' of the curried definition because they provide a canonical representation of the intermediate definitions it produces; when a domain of canons is established, meta-level operations or fragments can be transformed into operations on canons. For instance, consider the fragment 'postfix by successor' applied to the canon 'add 3'; the resulting procedure is equivalent to the canon 'add 4'. In this representation, the 'canonized' version of 'postfix by successor' is equivalent to 'successor' in the earlier vocbaulary. But note that in this vocabulary, addition — which had previously had to be expressed in terms of iterative combinations of predecessor and successor — is now simply the canonized version of the higher order operation 'prefix'.

In order for a reification to succeed as a new domain, it must be relatively inexpensive to locate exactly which canon a generated definition falls into; to ensure this, Cyrano selects the reifications it constructs by criteria of *determinism* and *coherence*. Cyrano only considers reifying operations which reliably produce the same results

for the same inputs; if an operation were to produce multiple results for particular inputs, we cannot determine by a single comparision whether two definitions are in fact identical for some input.

Potential reificiations are further pruned by the *coherence* of the spaces of partial results which they define. The coherence of a space of definitions is a measure of the degree to which the behavior of individual definitions overlap. In a maximally coherent space, definitions which agree on any one particular example agree on all examples; in a minimally coherent space, definitions typically differ from other definitions on at most one example and will agree with many other definitions on everything else.

Coherent spaces support efficient distinguishability; in a maximally coherent space, one example can determine the category in which a procedure lies while in a minimally coherent space such distinction may require an arbitrary number of examples.

One extreme case of coherence is the case where each partial evaluation is distinct. These cases are caught by also assessing the reification's *divergence*: how long it takes to find a behavioral match between two procedures. The closer the divergence is to 1, the better the representation will be able to use whatever coherence it possesses.

2.8 Summary

Cyrano's operation consists of two phases: an exploration phase which explores the vocabulary's constructions in order of decreasing *simplicity* constrained by criteria of syntactic sensibility, pragmatic effectiveness, and diversity of performance. After some period of exploration, the set of generated definitions is set of definitions is refomulated. This reformulation involves both rebiasing of the simplicity metric of the original vocabulary and the definition of new spaces of objects and operations by *reifying* operations over the original vocabulary. Given these new domains and this new metric, the exploration process begins again under the same general mechanism.

Chapter 3
A Program's Progress

The Cyrano program described here is one point in a series of programs that I have developed over the past five years; this version of Cyrano (Cyrano-2) was preceded by a more complicated implementation (Cyrano-1) described in [Haase, 1986a, Haase, 1986b] and this implementation derived from Cyrano-0, an undocumented attempt at reimplementing Lenat's Eurisko program [Lenat, 1983]. It was originally hoped that Cyrano would perform in a variety of domains (like Cyrano Savinien de Bergerac, its namesake), but the inaccesibility of other domains to automatic experimentation limited the scope of this ambition. Discovery programs that have exhibited a broader scope (e.g. [Lenat, 1983] or [Langley *et al.*, 1987]) did so by human intervention. A human user served as the arbiter of 'empirical interestingness' for Lenat's Eurisko and the creators of Bacon extracted both particular experimental arrangements and observations from the notebooks of scientists. Cyrano's exploration of 'other domains' has been limited to the experimentally accessible domains of its own invention (various arithmetics and formal systems).

3.1 Issues in the Description of Programs

It is often difficult to accurately describe the operation of an AI program because it is designed as system whose moment-to-moment behavior relates to some understanding of cognitive process; however the technical content of the program's implementation must neccessariiy go 'between the gaps' where the cognitive description and underlying computational mechanisms are distinct.

The difficulty is that a clear and intuitive description would be couched in the mentalistic language of the program's specification; but a more accurate and useful description would descend to the program's mechanics. For instance, a description of a program's operations might assert 'The system then considered an operation which combined X and Y and noticed that the operation was interesting; it then considered applying transformation Z to the combination and ...'; but the questions we wish to ask are "Why did it decide to combine X and Y" or "What was so interesting about their combination?".

Even if reasons are given, they may be misleading; for instance, one might explain 'the system considered combining X and Y because they were both deterministic' but this is misleading if in fact nearly all such definitions were deterministic and all such

combinations were done. It is equally critical to consider why the system did *not* explore certain possibilities, but this is difficult to describe in the form of a connected description.

In the transcript of Cyrano's progress found below, Cyrano will generally assert when it is explicitly cutting off some avenue of inquiry; my annotations may explain why this step is either well-advised or ill-advised from a larger perspective. More subtle are the places where Cyrano's larger representational assumptions bar certain avenues of consideration; as much as possible I have attempted to point out (in the annotations on the transcript) when Cyrano 'misses' some significant or dangerous construction due to representational limitations. Of course, I cannot have located all such problems and I would not be suprised by an early 21^{st} century dissertation which uncovers unnoticed representational assumptions which variously empowered and/or crippled Cyrano.

The remainder of this chapter begins with a section describing Cyrano's starting domain (tie structures and operations upon them), a section introducing Cyrano's basic combiners (initially provided), and some notes on deciphering the transcript of its progress. After these introductions, there begins a series of sections consisting largely of program transcripts liberally annotated with my own explanations of the nature and significance of Cyrano's constructions and observations. It is unfortunate that I must play the interpreter's role, but Cyrano has no knowledge of the mathematical or common sense language in which it might describe its constructions directly.

But even I cannot claim to be a perfect or fluent interpreter; often it has taken lengthy consideration and examination of examples to determine what one of Cyrano's definitions 'really is . The reader may even discover that some construction I have not remarked upon is particulary interesting in a way I had not noticed.

In both the introductory explanations and the annotated transcripts, I will refer to events in the transcript with the syntax '(Note 1/61)'. Points in the transcript are marked with a number in the outside margin and an arrow pointing to the text; these numbers increase as one reads further into the text. The number after the slash in '(Note 1/61)' indicates the page number (e.g. Page 61); the number before the slash indicates the marginal line number (e.g. '1').

3.2 Cyrano's Starting Domain

In the version of Cyrano whose progress is described in this chapter, the program begins with three domain operations (manipulating structures called *ties*) and twelve essential combiners which operate on this domain to define a space of definitions which Cyrano explores. This section describes both Cyrano's starting domain and the built-in combiners which define the space it initially explores.

Ties are structurally identical to the 'cons cells' of pure LISP; the operations on them are illustrated in Figure 3.1. The operation `tie` takes two ties and returns

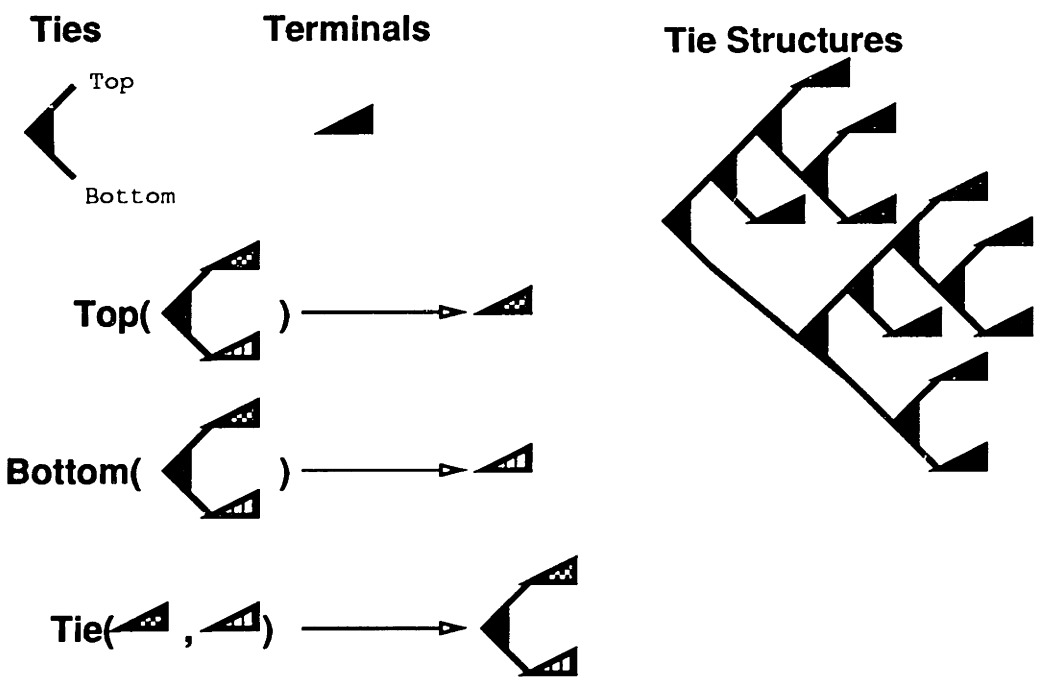


Figure 3.1: Cyrano begins with a vocabulary of objects called *ties* operated on by the primitives TIE, TOP, and BOTTOM.

a 'composite tie' which combines them; the operation top operates on a composite tie and returns the first component tie from which it was constructed; the operation bottom takes a composite tie and returns the second component tie from which it was constructed. There are no side-effect operations, so once a tie is created its top and bottom are immutable. Structures of composite ties terminate in objects called 'terminal ties' whose top and bottom are undefined.

To generate an example of these operations, Cyrano produces a random tie structure and applies the operation to generate a result. The random tie structures are produced by a procedure I have specified which slowly increases the mean depth and breadth of the tie structues generated. When this was not done, Cyrano ended up getting stuck in a subset of the natural numbers determined by the upper bound of how deep or broad generated structures were; from its point of view in this framework, there really was a 'largest number'.

I once was suprised by Cyrano's sensitivity to the properties of the particular example generation mechanisms used; once, in an attempt to speed up generation of tie structures, I began caching tie structures generated at one moment to use as parts of structures generated in the future. When Cyrano was run with this new 'just more efficient' mechanism, it suprised me by identifying and focussing upon operations and representations based on structural inclusion (which had not been a 'phenomena'

when tie structures were generated from scratch) Indeed, it formed a reified vocabulary which operated on classes of structures having a common substructure. However the caching scheme which generated these observations had its own problems of bias; generated structures clustered around particular sizes and not provide a reasonable distribution of heights and depths.

The tie-structure generating used in this scenario works by generating the 'outlines' of tie structures with minimal development of the 'innards'; this approach — admittedly contrived — means that the cost of generating tie structures of size n grows linearly with n rather than exponentially as would be the case with generating fully 'fleshed' tie structures.

3.3 Cyrano's Definition Combiners

Cyrano's built-in combination mechanisms are the way that it explores a given descriptive vocabulary. A combiner can be understood as a small computer program with empty places into which domain operations can be inserted. The resulting program can be executed as a new domain operation whose most primitive steps are those domain operations whose combination created it. For instance, the `[prefix]` combiner takes two domain operations f and g and returns a third operation which takes an input x and first applies g to x and then applies f to the resulting $g(x)$. Cyrano's operations are strict functions in the mathematical sense; for any x there is a unique result of applying the operation to x.

A central criterion in choosing Cyrano's combiners has been their repeated applicability in evolved vocabularies; it was important — given the emphasis of Cyrano on radical vocabulary invention — that Cyrano not have any 'one-shot' mechanisms which only served to bootstrap from one particular vocabulary to another. Such one-shot mechanisms are sure signs of things being 'built in' to the representation and are — ultimately — the stumbling blocks upon which the system will become stuck.

A lesser criterion has been an emphasis on definitions which create definitions which readily apply to other definitions. Though I began with a belief that combiners which merely produced 'regularity checkers' should be avoided, over time it became clear that such combiners provided important avenues for Cyrano inventing its own interestingness criteria. This is particularly clear with the `CHECK` combiner, which produces definitions which are not 'all that useful' with respect to other definitions but allows the focussing of interest on families of definitions which 'fail in the same way'. (See Section 3.3.3).

3.3.1 Simple Combinations

The `PREFIX` combiner takes two operations and uses the second to produce a 'first input' to the first. In the case of simple (non-curried) operations, this is a

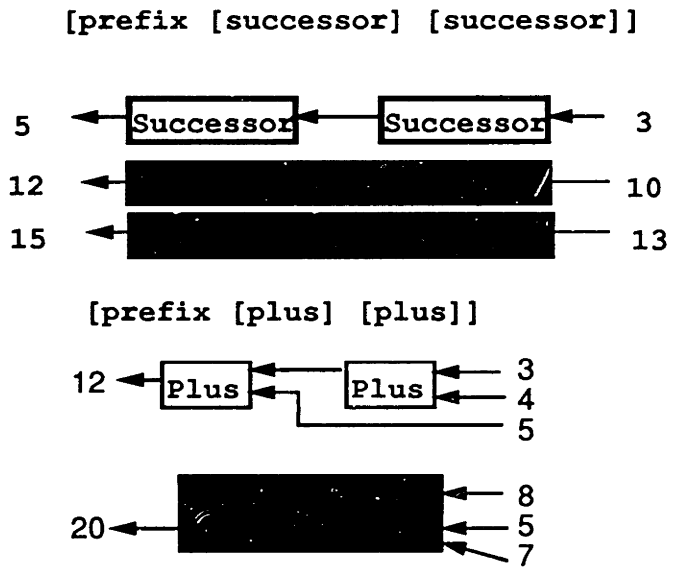


Figure 3.2: The PREFIX operation combines two operations by using one to produce the first input of the next.

simple passing on of results; for instance, [prefix [successor] [successor]] adds two to a number. In the case of more complicated operations, the second of the operations is called until it returns a value which can be passed on to the second operation; this second operation is then called with a first input of the final result of the first operation. For instance, if [plus] is a curried two argument addition operation, [prefix [plus] [plus]] is a curried three argument addition operation. (See Figure 3.2).

The POSTFIX combiner is like prefix, but with the arguments inverted; the first argument is used to produce an input to be passed to the first. A swapped version of PREFIX is important because some operations are more useful producing inputs for other procedures than for using them; in order to allow the credit assignment mechanisms of reformulation to note exactly *how* a definition was used, this separation identifies the distinction between the usefulness of an operation as a 'pre' or 'post' filter.

3.3.2 Iterative Combinations

The TERMINATOR combiner takes a single operation and returns another operation which 'follows' the first operation until it terminates. An operation A 'follows' another operation B by starting at some input x and applying B to it; if the result of this application is defined, it is used to continue following B.

In a sort of pseudo-code, [TERMINATOR B] might look like this:

```
TO terminate next from x
   IF next(x) is defined
      TERMINATE next from next(x)
   OTHERWISE
      RETURN x
```

For instance, [terminator [predecessor]] is zero over the natural numbers, while [terminator [successor]] never terminates (and is thus undefined).

The LIMIT combiner is a comparative version of TERMINATOR. It combines a *limiter* procedure and an *actor* procedure to create a two argument procedure which follows the first argument with *limiter* and the second argument with *actor*, returning whatever point *actor* has reached when *limiter* runs out.

```
TO limit by L the operation A on x and y
   IF L(x) is defined
      LIMIT by L the operation A on L(x) and A(y)
   OTHERWISE
      RETURN y
```

For instance, [limit [predecessor] [successor]] is the addition operation ([plus]) shown in Figure 3.3. [limit [predecessor] [predecessor]] is the arithmetic difference of two numbers when the second is greater than or equal to the first, but is undefined otherwise.

The MEET combiner determines where the paths computed by two operations meet; the operations combined are both followed and the paths they compute compared. The first point at which they come together is returned. MEET might be coded as something like this:

```
To MEET f and g from x and y
   LET X be a set of  x's, initially empty
   LET Y be a set of  y's, initially empty
   ADVANCE the algorithm on x and y BY
     IF f(x) is in Y
        RETURN f(x)
     IF g(y) is in X
        RETURN g(y)
     OTHERWISE
       ADD f(x) to X and
       AND g(y) to Y and
       ADVANCE the algorithm on f(x) and g(y)
```

```
To LIMIT BY predecessor THE OPERATION successor ON x AND y
   IF predecessor(x) is undefined
      RETURN y
   OTHERWISE
      LIMIT BY predecessor THE OPERATION successor ON
                  predecessor(x) AND predecessor(y)
```

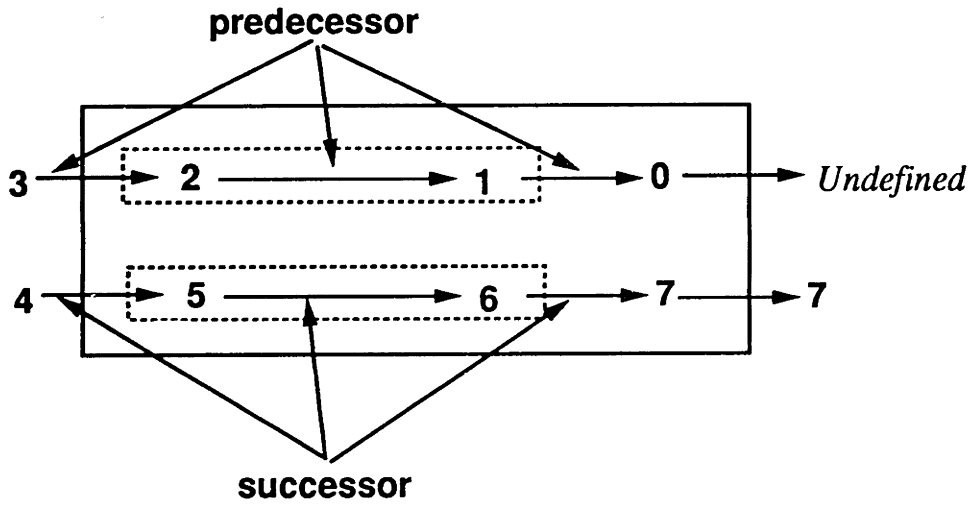


Figure 3.3: The definition [limit [predecessor] [successor]] computes the arithmetic sum of its inputs by using the repeated applications of the predecessor operation to control the repeated applications of the successor operation.

For example, [meet [predecessor] [successor]] returns the arithmetic mean ([mean]) of two numbers (if the first is smaller than the second as shown in Figure 3.4), while [meet [successor] [successor]] returns the larger of its two inputs ([max]) as shown in Figure 3.5.

3.3.3 Effectiveness Determination

A given or invented operation may be defined or undefined; for instance, the arithmetic mean calculated by [meet [successor] [predecessor]] is only defined when the first input is smaller than the second. The CHECK combiner takes a domain level operation *O* which might be undefined and returns a boolean-valued (true or false) operation which returns true only for those inputs for which *O* is defined.

For instance, [check [meet [predecessor] [successor]]] is true all numerically ordered pairs; the same predicate is defined by [check [limit [predecessor] [predecessor]]].

As sketched above and operation may be either *a priori* undefined by reaching

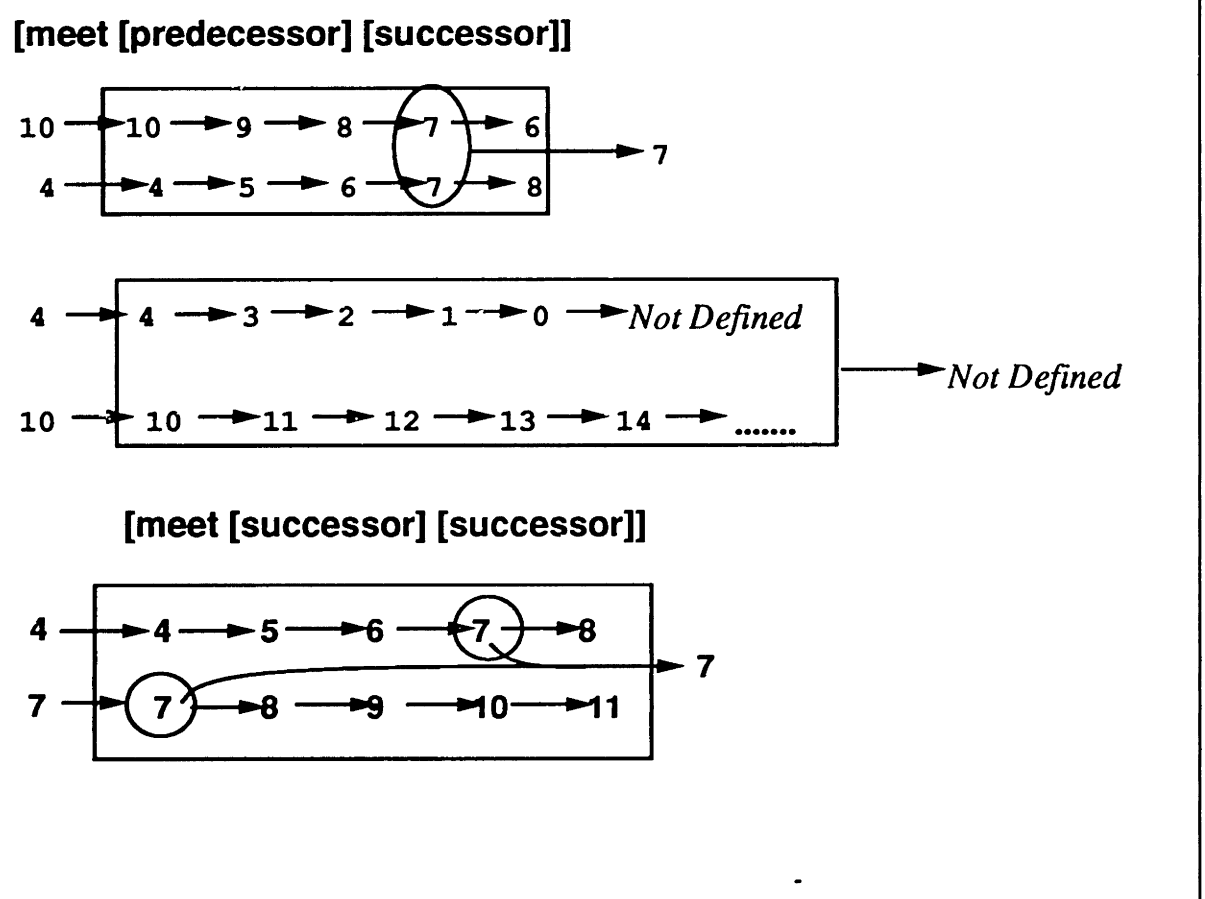


Figure 3.4: The definition [meet [predecessor] [succecessor]] computes the arithmetic mean of its inputs by repeatedly incrementing one number, decrementing the other, and returning the point at which their sequences of values cross. This is undefined when the first input is smaller than the second.

some internal state from which no further progress is possible, or it may be pragmatically undefined by not terminating before some quantum of external time is used up.

Because operations produced by check do no return 'domain level' values, they are not very useful as 'domain level' operations of themselves. Their value emerges as the codify definitional regularities in a way that allows the isolation of intersting common patterns of definition or non-definition.

3.3.4 Relational Combinations

In addition to the boolean procedures returned by WORKS? and FAILS?, Cyrano constructs other sorts of boolean valued definitions which compare results and inputs.

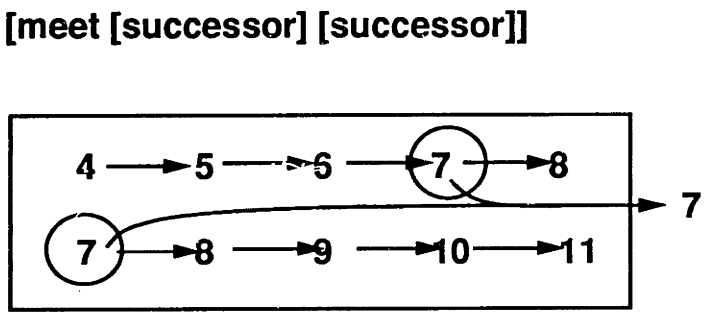


Figure 3.5: The definition [meet [successor] [succecessor]] computes the maximnum of its inputs by repeatedly incrementing two numbers until one reaches the other.

The IDENTICAL? combiner takes two domain definitions and constructs a definition which returns true if both definitions agree on a particular input; thus, [identical? [predecessor] [successor]] returns true when its first input is two more than its second input. Thus, the operation returns true for the pair (5,3) because the predecessor of 5 is the successor of 3; it is false, for instance, for the pair (5,4) because this identity does not hold.

The RELATION combiner takes a domain level definition which is not curried and returns a curried definition which is true when its second input is identical to the result of applying the uncurried definition to their first input. Thus, [relation [successor]] is a two input curried definition which returns true if its second input is one greater than its first input.

The YIELDS combiner combines a domain *object* of interest and a definition which might return that; it returns a boolean-valued version of the domain definition which is true only if the operation yields the corresponding object. For instance, [yields 0 [predecessor]] is only true for 1.

3.3.5 Permuting Combinations

Cyrano also has several combinators for merging or permuting the inputs to curried definitions.

The COALESCE combiner takes a definition which expects two inputs of the same type and defines a definition taking one input which calls the inner definition with this single input repeated. For instance, [coalesce [minus]] is always zero, while [coalesce [plus]] doubles its input (Figure 3.6).

The combiners MIRROR-2, MIRROR-3, and MIRROR-4 take curried definitions and

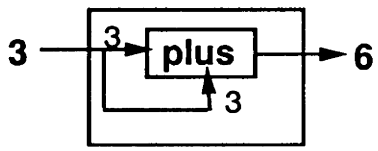


Figure 3.6: A *coalesced* operation applies a two input operation to a identical 'copies' of a single input. The coalesced version of plus adds a number to itself, doubling it.

return other curried definitions (of the same arity) which swap the first and i^{th} arguments. For instance, `[mirror-2 [plus]]` is identical to `[plus]` while `[mirror-2 [minus]]` is defined at all of the places where `[minus]` is undefined (excepting equal inputs).

3.4 Descriptive Conventions

The bulk of the remainder of this chapter is dedicated to an annotated printout of a run of Cyrano; if you have access to a version of Cyrano, you should be able to load it up, type

```
(DISCOVER! ALEPH 600 5)
```

and see a similar sort of trace. The identifier `aleph` is bound to a pool containing the tie operations and the standard combiners; the number 1200 refers to the number of seconds for which to explore between reformulations and the number 5 refers to how many reformulations to do in total.

When Cyrano runs, it provides various reports describing its internal activity; these are determined by a set of 'event traces' which determine whether particular sorts of events are recorded. In the version of Cyrano described here, there are 6 basic event types:

- **Domain Generation** The generation of domain definitions from fragments already in the pool.
- **Fragment Generation** The generation of further fragments from definitions already in the pool.
- **Assessment** The empirical assessment (usually for purposes of triage) of a domain definition.
- **Selector Intervention** Changes introduced by the selection process (triage).

- **Example Generation** Examples generated for purposes of assesment or regeneration.
- **Reformulation** Events or results having to do directly with reformulation; these events are usually summaries of processes like clique identification or reification assessment, which can be traced separately.
- **Clique Identification** The iterative reduction of a set of definitions to a set of cliques with identical behavior.
- **Reification Assessment** Analysis of potential reifications.
- **Abstraction** The generation of new vocabularies based on a particular reification.

This trace information is printed out prefixed by the time of day, a three character flag indicating the sort of event (e.g. '!!!' indicates an empirical assessment, '***' indicates domain generation, '===' indicates clique identification results, etc), and (potentially) a reference code (such as '[#@322] which refers to the SCHEME object the event refers to; e.g. the examples supporting an assessment, the domain definition generated, or a list of cliques resulting from clique division. In interactive use, these codes allow a user to look at the internal data structures Cyrano uses in generating and analyzing its results; one can imagine a more sophisticated user interface where these descriptive lines were (for instance) mouse-sensitive so as to allow the perusal of structures behind Cyrano's progress.

In normal use, Cyrano is run with example generation tracing off and everything else on; in interests of space, the transcript of this chapter also turns off tracing of fragment generation. The transcript is also annotated by my own comments set in roman type interrupting the transcript; in general, my comments will refer to the event or events which have just occurred. In the text of these comments, if I say that 'Cyrano immediately notices X' it means that the assessment and triage process (which provides a trace) picks up a particular property which I describe as X. If I say (on the other hand) that 'Cyrano eventually notices X' it means that a later reformulation process identifies some property X.

As mentioned above, the included commentary will sometimes refer to events in the transcript with expressions like '(Note 20/94)'; as you read the transcript, particular lines are marked in the outside margin with increasing numbers. The mentions in the text refer to these; for instance, (Note 1/61) is the first line of the Cyrano transcript while (Note 20/94) is where Cyrano 'realizes' (during reformulation) that addition is associative.

As described in the previous chapter, Cyrano alternates episodes of vocabulary exploration with vocabulary invention. These alternations are set at clock-time inter-

vals; in the description below, section headings separate each episode of exploration and invention.

3.5 Exploring Ties

Cyrano begins by exploring operations on tie structures; the initial pool (`aleph`) given to Cyrano consists of the combiners above and the operations `top`, `bot`, and `tie`. As given to Cyrano, the domain definitions have a greater 'simplicity' than the fragmentary combiners, so as to bias the generation of domain definitions as opposed to more abstract definitions.

```
;[12:05:51] *** Gen'd [check [top]] with simplicity 1/1441                    <== 1
;[12:05:53] *** Gen'd [check [bot]] with simplicity 1/144
;[12:05:54] *** Gen'd [check [tie]] with simplicity 1/144
;[12:05:54] While assessing the domain definition [check [tie]]
;[12:05:57]  !!! [check [tie]] appears to be constantly #T1                   <== 2
;[12:05:58] >>> Selector produced [check [tie]] with simplicity 0
```

> *The definitions above are predicates delineating the domains of the tie-structure primitives. Both* `[check [top]]` *and* `[check [bot]]` *compute the predicate* `tie?`*. Since* `tie` *can tie any two objects together, it is always defined and thus its 'check' is removed from further consideration.*

```
;[12:05:58] *** Gen'd [mirror-2 [tie]] with simplicity 1/144
;[12:06:06] *** Gen'd [terminator [top]] with simplicity 1/144
;[12:06:06] While assessing the domain definition [terminator [top]]
;[12:06:08]  !!! [terminator [top]] appears to be constantly ()
;[12:06:09] >>> Selector produced [terminator [top]] with simplicity 0
;[12:06:09] *** Gen'd [terminator [bot]] with simplicity 1/144
;[12:06:09] While assessing the domain definition [terminator [bot]]
;[12:06:11]  !!! [terminator [bot]] appears to be constantly ()
;[12:06:11] >>> Selector produced [terminator [bot]] with simplicity 0
```

> *Cyrano discovers that tie structures end — in both the top and down directions — with the terminal '().*

```
;[12:06:12] *** Gen'd [coalesce [tie]] with simplicity 1/144
;[12:06:20] *** Gen'd [prefix [tie] [top]] with simplicity 1/864
;[12:06:21] *** Gen'd [prefix [tie] [bot]] with simplicity 1/864
```

> *This operation corresponds to what we might call 'replace bottom of* x *with* y*;* `bot` *is called on a tie structure and then the remaining top is tied together with a new bottom.*

```
;[12:06:23] *** Gen'd [prefix [tie] [tie]] with simplicity 1/864
;[12:06:24] *** Gen'd [prefix [bot] [top]] with simplicity 1/864
;[12:06:25] *** Gen'd [prefix [bot] [bot]] with simplicity 1/864
;[12:06:26] *** Gen'd [prefix [bot] [tie]] with simplicity 1/864
;[12:06:26] While assessing the domain definition [prefix [bot] [tie]]
;[12:06:35]  !!! [prefix [bot] [tie]] always returns its first input
;[12:06:36] >>> Selector produced [prefix [bot] [tie]] with simplicity 0
;[12:06:36] *** Gen'd [prefix [top] [top]] with simplicity 1/864
;[12:06:37] *** Gen'd [prefix [top] [bot]] with simplicity 1/864
;[12:06:37] *** Gen'd [prefix [top] [tie]] with simplicity 1/864
;[12:06:38] While assessing the domain definition [prefix [top] [tie]]
;[12:06:48]  !!! [prefix [top] [tie]] always returns its second input
;[12:06:48] >>> Selector produced [prefix [top] [tie]] with simplicity 0
```

> *These two observations correspond to Cyrano's realization that* `top` *and* `bot` *destructure the results of* `tie`*.*

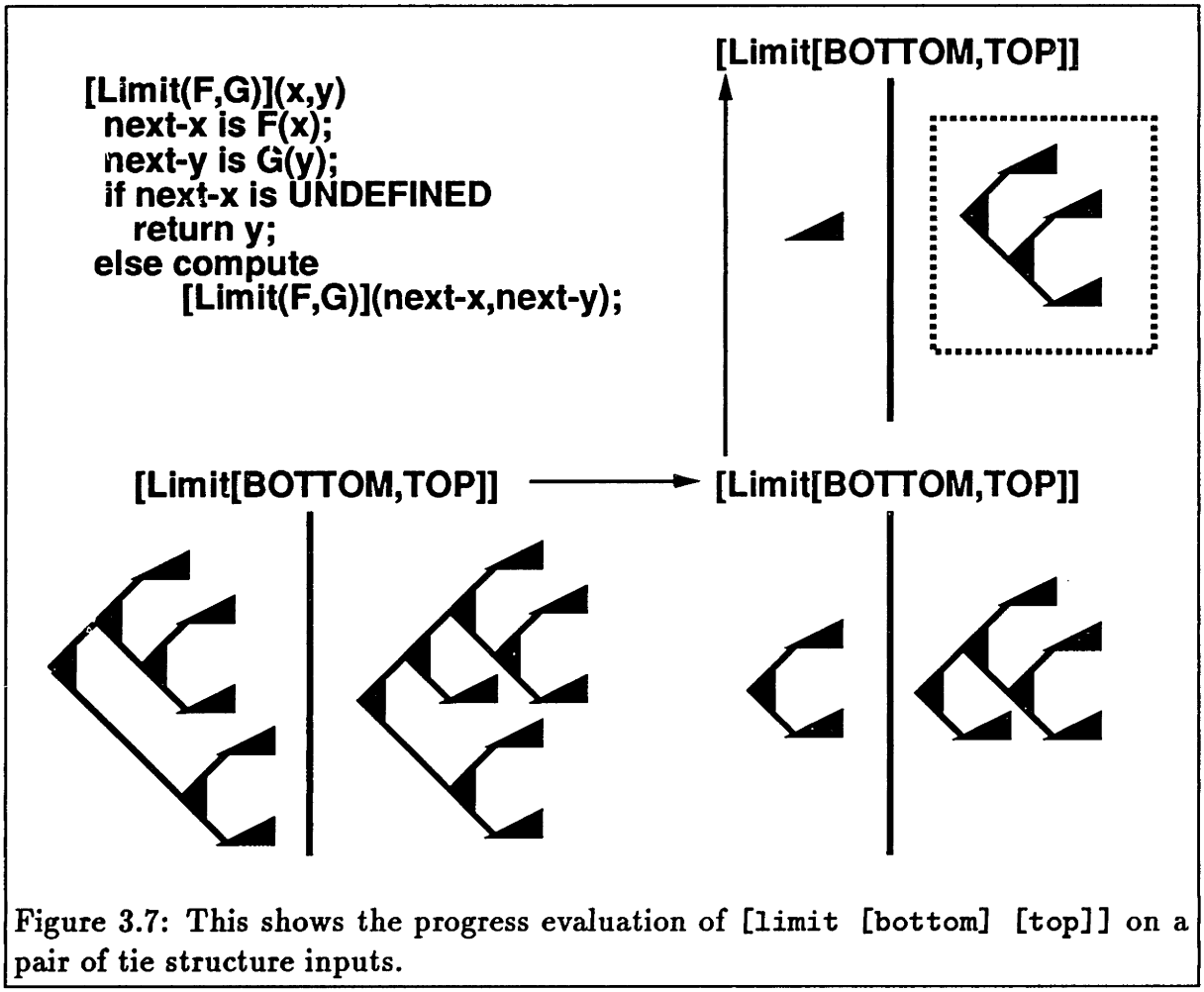


Figure 3.7: This shows the progress evaluation of `[limit [bottom] [top]]` on a pair of tie structure inputs.

```
;[12:06:49] *** Gen'd [limit [bot] [top]] with simplicity 1/864
;[12:06:49] *** Gen'd [limit [bot] [bot]] with simplicity 1/864
;[12:06:50] *** Gen'd [limit [top] [top]] with simplicity 1/864
;[12:06:52] *** Gen'd [limit [top] [bot]] with simplicity 1/864
```

These are operations which 'dismember' tie structures. Starting with two tie structures remove one half of each until one of them runs out. See Figure 3.7.

```
;[12:06:53] *** Gen'd [identical? [bot] [top]] with simplicity 1/864
;[12:06:55] *** Gen'd [identical? [bot] [bot]] with simplicity 1/864
;[12:06:56] *** Gen'd [identical? [top] [top]] with simplicity 1/864
;[12:06:56] While assessing the domain definition [identical? [top] [top]]
;[12:06:58]  !!! [identical? [top] [top]] appears to be constantly ()
;[12:06:58] >>> Selector produced [identical? [top] [top]] with simplicity 0
;[12:06:58] *** Gen'd [identical? [top] [bot]] with simplicity 1/864
;[12:06:58] While assessing the domain definition [identical? [top] [bot]]
;[12:07:01]  !!! [identical? [top] [bot]] appears to be constantly ()
;[12:07:01] >>> Selector produced [identical? [top] [bot]] with simplicity 0
```

Much to its designer's frustration, Cyrano's empirical analyses sometimes yield non-phenomena. The regularities noticed here are among the regularities Cyrano noticed that a result of bad example generation.

```
;[12:07:01] *** Gen'd [meet [bot] [top]] with simplicity 1/864
;[12:07:01] While assessing the domain definition [meet [bot] [top]]
;[12:07:05]  !!! [meet [bot] [top]] appears to be constantly ()
;[12:07:05] >>> Selector produced [meet [bot] [top]] with simplicity 0
;[12:07:06] *** Gen'd [meet [bot] [bot]] with simplicity 1/8641                ⟸ 3
;[12:07:06] *** Gen'd [meet [top] [top]] with simplicity 1/864
;[12:07:07] *** Gen'd [meet [top] [bot]] with simplicity 1/864
;[12:07:07] While assessing the domain definition [meet [top] [bot]]
;[12:07:10]  !!! [meet [top] [bot]] appears to be constantly ()
;[12:07:11] >>> Selector produced [meet [top] [bot]] with simplicity 0
```

These `meet` *operations find the common substructure of two tie structures. Note that* `'()` *is the only common sub-structure when the structures are search in separate directions. This is an artifact of the example caching mechanism. The example caching mechanism works by taking two pieces of a large structure and* `tie`*-ing them together. The top piece is gotten from the large structure by some random number of top oeprations, while the bottom piece is gotten from the large structure by some random number of bottom operations. As a result, generated tops and bottoms are drawn from distinct pools.*

```
;[12:07:11] *** Gen'd [postfix [bot] [top]] with simplicity 1/864
;[12:07:12] *** Gen'd [postfix [bot] [bot]] with simplicity 1/864
;[12:07:13] *** Gen'd [postfix [bot] [tie]] with simplicity 1/864
;[12:07:14] *** Gen'd [postfix [top] [top]] with simplicity 1/864
;[12:07:14] *** Gen'd [postfix [top] [bot]] with simplicity 1/864
;[12:07:15] *** Gen'd [postfix [top] [tie]] with simplicity 1/864
```

Various straightforward compositions of existing operations.

```
;[12:07:27] *** Gen'd [check [coalesce [tie]]] with simplicity 1/3456
;[12:07:27] While assessing the domain definition [check [coalesce [tie]]]
;[12:07:31]  !!! [check [coalesce [tie]]] appears to be constantly #T
;[12:07:31] >>> Selector produced [check [coalesce [tie]]] with simplicity 0
;[12:07:32] *** Gen'd [check [mirror-2 [tie]]] with simplicity 1/3456
;[12:07:33] While assessing the domain definition [check [mirror-2 [tie]]]
;[12:07:40]  !!! [check [mirror-2 [tie]]] appears to be constantly #T
;[12:07:40] >>> Selector produced [check [mirror-2 [tie]]] with simplicity 0
;[12:07:41] *** Gen'd [mirror-2 [mirror-2 [tie]]] with simplicity 1/3456
;[12:07:52] *** Gen'd [terminator [coalesce [tie]]] with simplicity 1/3456
;[12:07:52] While assessing the domain definition [terminator [coalesce [tie]]]
;[12:08:00]  !!! [terminator [coalesce [tie]]] never seems to terminate.
;[12:08:00] >>> Selector produced [terminator [coalesce [tie]]] with simplicity 0
```

The above definitions and their empirical examination allow Cyrano to apply tests on unary operations to the coalesced version of the tie operation. In doing so, it learns that `[coalesce [tie]]` *is always defined and that it never terminates. Some of these inferences could be made automatically; the fact that* `[check [tie]]` *is always true (above, (Note 2/61)) implies that* `[coalesce [tie]]` *is also always true, which in turn implies that* `[terminator [coalesce [tie]]]` *will never terminate.*

```
;[12:08:02] *** Gen'd [coalesce [mirror-2 [tie]]] with simplicity 1/3456
;[12:08:37] *** Gen'd [prefix [check [top]] [top]] with simplicity 1/20736
;[12:08:37] *** Gen'd [prefix [check [top]] [bot]] with simplicity 1/20736
;[12:08:38] *** Gen'd [prefix [check [top]] [tie]] with simplicity 1/20736
;[12:08:39] While assessing the domain definition [prefix [check [top]] [tie]]
4 ⟹  ;[12:08:53]  !!! [prefix [check [top]] [tie]] appears to be constantly #T1
;[12:08:54] >>> Selector produced [prefix [check [top]] [tie]] with simplicity 0
;[12:09:04] *** Gen'd [prefix [check [bot]] [top]] with simplicity 1/20736
;[12:09:05] *** Gen'd [prefix [check [bot]] [bot]] with simplicity 1/20736
;[12:09:06] *** Gen'd [prefix [check [bot]] [tie]] with simplicity 1/20736
;[12:09:06] While assessing the domain definition [prefix [check [bot]] [tie]]
;[12:09:21]  !!! [prefix [check [bot]] [tie]] appears to be constantly #T
;[12:09:21] >>> Selector produced [prefix [check [bot]] [tie]] with simplicity 0
```

Here and at (Note 4/64) above, Cyrano realizes that tie always returns ties (it had defined the definition corresponding to `TIE?` *at (Note 1/61) above).*

```
;[12:09:22] *** Gen'd [prefix [mirror-2 [tie]] [top]] with simplicity 1/20736
;[12:09:23] *** Gen'd [prefix [mirror-2 [tie]] [bot]] with simplicity 1/20736
;[12:09:25] *** Gen'd [prefix [mirror-2 [tie]] [tie]] with simplicity 1/20736
;[12:09:27] *** Gen'd [prefix [coalesce [tie]] [top]] with simplicity 1/20736
;[12:09:28] *** Gen'd [prefix [coalesce [tie]] [bot]] with simplicity 1/20736
;[12:09:29] *** Gen'd [prefix [coalesce [tie]] [tie]] with simplicity 1/20736
;[12:09:31] *** Gen'd [coalesce [postfix [top] [tie]]] with simplicity 1/20736
;[12:09:32] *** Gen'd [coalesce [postfix [bot] [tie]]] with simplicity 1/20736
;[12:09:34] *** Gen'd [coalesce [meet [top] [top]]] with simplicity 1/20736
;[12:09:34] While assessing the domain definition [coalesce [meet [top] [top]]]
;[12:09:39]  !!! [coalesce [meet [top] [top]]] always returns its first input
;[12:09:39] >>> Selector produced [coalesce [meet [top] [top]]] with simplicity 0
;[12:09:40] *** Gen'd [coalesce [meet [bot] [bot]]] with simplicity 1/20736
;[12:09:40] While assessing the domain definition [coalesce [meet [bot] [bot]]]
;[12:09:45]  !!! [coalesce [meet [bot] [bot]]] always returns its first input
;[12:09:45] >>> Selector produced [coalesce [meet [bot] [bot]]] with simplicity 0
```

Here, Cyrano realizes that every tie structure contains itself (using the definitions of containinment, defined in terms of `meet` *around (Note 3/63).*

```
;[12:09:46] *** Gen'd [coalesce [identical? [bot] [bot]]] with simplicity 1/20736
;[12:09:46] While assessing the domain definition [coalesce [identical? [bot] [bot]]]
;[12:09:51]  !!! [coalesce [identical? [bot] [bot]]] appears to be constantly #T
;[12:09:51] >>> Selector produced [coalesce [identical? [bot] [bot]]] with simplicity 0
;[12:09:52] *** Gen'd [coalesce [identical? [bot] [top]]] with simplicity 1/20736
;[12:09:54] *** Gen'd [coalesce [limit [top] [bot]]] with simplicity 1/20736
;[12:09:55] *** Gen'd [coalesce [limit [top] [top]]] with simplicity 1/20736
;[12:09:55] While assessing the domain definition [coalesce [limit [top] [top]]]
;[12:10:01]  !!! [coalesce [limit [top] [top]]] appears to be constantly ()
;[12:10:01] >>> Selector produced [coalesce [limit [top] [top]]] with simplicity 0
;[12:10:02] *** Gen'd [coalesce [limit [bot] [bot]]] with simplicity 1/20736
;[12:10:02] While assessing the domain definition [coalesce [limit [bot] [bot]]]
;[12:10:09]  !!! [coalesce [limit [bot] [bot]]] appears to be constantly ()
;[12:10:09] >>> Selector produced [coalesce [limit [bot] [bot]]] with simplicity 0
```

If we take the `limit` *operations as comparisions of heights or depths the two empirical results correspond to Cyrano's realization that identical tie structures have the same height and depth.*

```
;[12:10:10] *** Gen'd [coalesce [limit [bot] [top]]] with simplicity 1/20736
;[12:10:11] *** Gen'd [coalesce [prefix [tie] [bot]]] with simplicity 1/20736
;[12:10:12] *** Gen'd [coalesce [prefix [tie] [top]]] with simplicity 1/20736
;[12:10:13] *** Gen'd [terminator [postfix [top] [bot]]] with simplicity 1/20736
5 ⟹  ;[12:10:15] *** Gen'd [terminator [postfix [top] [top]]] with simplicity 1/207361
```

```
;[12:10:16] *** Gen'd [terminator [postfix [bot] [bot]]] with simplicity 1/20736
;[12:10:16] *** Gen'd [terminator [postfix [bot] [top]]] with simplicity 1/20736
;[12:10:18] *** Gen'd [terminator [prefix [top] [bot]]] with simplicity 1/20736
;[12:10:19] *** Gen'd [terminator [prefix [top] [top]]] with simplicity 1/20736
;[12:10:20] *** Gen'd [terminator [prefix [bot] [bot]]] with simplicity 1/20736
;[12:10:21] *** Gen'd [terminator [prefix [bot] [top]]] with simplicity 1/20736
```

> *Unlike the simple terminators of* `[top]` *or* `[bot]` *alone, the terminators of composed operations terminate (e.g. (Note 5/64)) either on the terminal tie or one tie structure away from the terminal. The latter case occurs when the tie structure is an odd number of steps high or deep.*

```
;[12:10:22] While assessing the domain definition [terminator [prefix [bot] [top]]]
;[12:10:24]  !!! [terminator [prefix [bot] [top]]] appears to be constantly ()
;[12:10:24] >>> Selector produced [terminator [prefix [bot] [top]]] with simplicity 0
;[12:10:29] *** Gen'd [limit [coalesce [tie]] [top]] with simplicity 1/20736
;[12:10:29] While assessing the domain definition [limit [coalesce [tie]] [top]]
;[12:10:59]  !!! [limit [coalesce [tie]] [top]] never seems to be defined.
;[12:11:00] >>> Selector produced [limit [coalesce [tie]] [top]] with simplicity 0
;[12:11:00] *** Gen'd [limit [coalesce [tie]] [bot]] with simplicity 1/20736
;[12:11:00] While assessing the domain definition [limit [coalesce [tie]] [bot]]
;[12:11:31]  !!! [limit [coalesce [tie]] [bot]] never seems to be defined.
;[12:11:32] >>> Selector produced [limit [coalesce [tie]] [bot]] with simplicity 0
```

> *The inference mechanisms discussed above might have helped Cyrano avoid the attempts to limit other operations by a coalesced* `tie` *operation. The same is true of the various operations below.....*

```
;[12:11:34] *** Gen'd [mirror-3 [prefix [tie] [tie]]] with simplicity 1/20736
;[12:11:43] *** Gen'd [identical? [coalesce [tie]] [top]] with simplicity 1/20736
;[12:11:43] While assessing the domain definition [identical? [coalesce [tie]] [top]]
;[12:11:49]  !!! [identical? [coalesce [tie]] [top]] appears to be constantly ()
;[12:11:49] >>> Selector produced [identical? [coalesce [tie]] [top]] with simplicity 0
;[12:11:50] *** Gen'd [identical? [coalesce [tie]] [bot]] with simplicity 1/20736
;[12:11:50] While assessing the domain definition [identical? [coalesce [tie]] [bot]]
;[12:11:56]  !!! [identical? [coalesce [tie]] [bot]] appears to be constantly ()
;[12:11:57] >>> Selector produced [identical? [coalesce [tie]] [bot]] with simplicity 0
;[12:12:03] *** Gen'd [meet [coalesce [tie]] [top]] with simplicity 1/20736
;[12:12:03] While assessing the domain definition [meet [coalesce [tie]] [top]]
;[12:12:14]  !!! [meet [coalesce [tie]] [top]] never seems to terminate.
;[12:12:14] >>> Selector produced [meet [coalesce [tie]] [top]] with simplicity 0
;[12:12:15] *** Gen'd [meet [coalesce [tie]] [bot]] with simplicity 1/20736
;[12:12:15] While assessing the domain definition [meet [coalesce [tie]] [bot]]
;[12:12:24]  !!! [meet [coalesce [tie]] [bot]] never seems to terminate.
;[12:12:24] >>> Selector produced [meet [coalesce [tie]] [bot]] with simplicity 0
;[12:12:50] *** Gen'd [postfix [coalesce [tie]] [top]] with simplicity 1/20736
;[12:12:50] While assessing the domain definition [postfix [coalesce [tie]] [top]]
;[12:12:55]  !!! [postfix [coalesce [tie]] [top]] always returns its first input
;[12:12:55] >>> Selector produced [postfix [coalesce [tie]] [top]] with simplicity 0
;[12:12:56] *** Gen'd [postfix [coalesce [tie]] [bot]] with simplicity 1/20736
;[12:12:58] While assessing the domain definition [postfix [coalesce [tie]] [bot]]
;[12:13:04]  !!! [postfix [coalesce [tie]] [bot]] always returns its first input
;[12:13:05] >>> Selector produced [postfix [coalesce [tie]] [bot]] with simplicity 0
;[12:13:06] *** Gen'd [postfix [coalesce [tie]] [tie]] with simplicity 1/20736
;[12:13:08] *** Gen'd [mirror-2 [postfix [top] [tie]]] with simplicity 1/20736
;[12:13:10] *** Gen'd [mirror-2 [postfix [bot] [tie]]] with simplicity 1/20736
;[12:13:11] *** Gen'd [mirror-2 [meet [top] [top]]] with simplicity 1/20736
;[12:13:12] *** Gen'd [mirror-2 [meet [bot] [bot]]] with simplicity 1/20736
;[12:13:14] *** Gen'd [mirror-2 [identical? [bot] [bot]]] with simplicity 1/20736
```

```
;[12:13:14] While assessing the domain definition [mirror-2 [identical? [bot] [bot]]]
;[12:13:20]  !!! [mirror-2 [identical? [bot] [bot]]] appears to be constantly ()
;[12:13:20] >>> Selector produced [mirror-2 [identical? [bot] [bot]]] with simplicity 0
;[12:13:21] *** Gen'd [mirror-2 [identical? [bot] [top]]] with simplicity 1/20736
;[12:13:21] While assessing the domain definition [mirror-2 [identical? [bot] [top]]]
;[12:13:28]  !!! [mirror-2 [identical? [bot] [top]]] appears to be constantly ()
;[12:13:28] >>> Selector produced [mirror-2 [identical? [bot] [top]]] with simplicity 0
;[12:13:29] *** Gen'd [mirror-2 [limit [top] [bot]]] with simplicity 1/20736
;[12:13:31] *** Gen'd [mirror-2 [limit [top] [top]]] with simplicity 1/20736
;[12:13:32] *** Gen'd [mirror-2 [limit [bot] [bot]]] with simplicity 1/20736
;[12:13:34] *** Gen'd [mirror-2 [limit [bot] [top]]] with simplicity 1/20736
;[12:13:37] *** Gen'd [mirror-2 [prefix [tie] [bot]]] with simplicity 1/20736
;[12:13:38] *** Gen'd [mirror-2 [prefix [tie] [top]]] with simplicity 1/20736
```

The mirror versions of various operations are not always interesting by themselves (though sometimes they are) but may occasionally focus interest on definitions which are invariant under mirroring. Below, at (Note 8/69), we will see how Cyrano notes that the operation 'common substructure' from (Note 3/63) is thus invariant under mirroring.

```
;[12:13:40] *** Gen'd [check [postfix [top] [tie]]] with simplicity 1/20736
;[12:13:40] While assessing the domain definition [check [postfix [top] [tie]]]
;[12:13:49]  !!! [check [postfix [top] [tie]]] appears to be constantly #T
;[12:13:49] >>> Selector produced [check [postfix [top] [tie]]] with simplicity 0
;[12:13:50] *** Gen'd [check [postfix [top] [bot]]] with simplicity 1/20736
6 ⟹ ;[12:13:50] *** Gen'd [check [postfix [top] [top]]] with simplicity 1/207361
;[12:13:52] *** Gen'd [check [postfix [bot] [tie]]] with simplicity 1/20736
;[12:13:53] *** Gen'd [check [postfix [bot] [bot]]] with simplicity 1/20736
;[12:13:54] *** Gen'd [check [postfix [bot] [top]]] with simplicity 1/20736
```

These correspond to various predicates on small tie structures. For instance, (Note 6/66) is a predicate satisfied by tie structures which are higher than two tie structures.

```
;[12:13:54] While assessing the domain definition [check [postfix [bot] [top]]]
;[12:13:55]  !!! [check [postfix [bot] [top]]] appears to be constantly #T
;[12:13:56] >>> Selector produced [check [postfix [bot] [top]]] with simplicity 0
;[12:13:56] *** Gen'd [check [meet [top] [top]]] with simplicity 1/20736
;[12:13:57] While assessing the domain definition [check [meet [top] [top]]]
;[12:14:07]  !!! [check [meet [top] [top]]] appears to be constantly #T
;[12:14:07] >>> Selector produced [check [meet [top] [top]]] with simplicity 0
;[12:14:08] *** Gen'd [check [meet [bot] [bot]]] with simplicity 1/20736
;[12:14:09] While assessing the domain definition [check [meet [bot] [bot]]]
;[12:14:19]  !!! [check [meet [bot] [bot]]] appears to be constantly #T
;[12:14:19] >>> Selector produced [check [meet [bot] [bot]]] with simplicity 0
;[12:14:20] *** Gen'd [check [limit [top] [bot]]] with simplicity 1/20736
;[12:14:22] *** Gen'd [check [limit [top] [top]]] with simplicity 1/20736
;[12:14:24] *** Gen'd [check [limit [bot] [bot]]] with simplicity 1/20736
;[12:14:26] *** Gen'd [check [limit [bot] [top]]] with simplicity 1/20736
```

These are predicates of the form 'higher than,' 'deeper than,' or 'higher than deeper than.'

```
;[12:14:27] *** Gen'd [check [prefix [top] [bot]]] with simplicity 1/20736
;[12:14:28] *** Gen'd [check [prefix [top] [top]]] with simplicity 1/20736
;[12:14:29] *** Gen'd [check [prefix [bot] [bot]]] with simplicity 1/20736
;[12:14:29] *** Gen'd [check [prefix [bot] [top]]] with simplicity 1/20736
;[12:14:31] *** Gen'd [check [prefix [tie] [tie]]] with simplicity 1/20736
```

These are identical to the small predicates above.

```
;[12:14:32] While assessing the domain definition [check [prefix [tie] [tie]]]
;[12:15:13]  !!! [check [prefix [tie] [tie]]] appears to be constantly #T
;[12:15:13] >>> Selector produced [check [prefix [tie] [tie]]] with simplicity 0
;[12:15:13] *** Gen'd [check [prefix [tie] [bot]]] with simplicity 1/20736
;[12:15:14] While assessing the domain definition [check [prefix [tie] [bot]]]
;[12:15:22]  !!! [check [prefix [tie] [bot]]] appears to be constantly #T
;[12:15:22] >>> Selector produced [check [prefix [tie] [bot]]] with simplicity 0
;[12:15:23] *** Gen'd [check [prefix [tie] [top]]] with simplicity 1/20736
;[12:15:23] While assessing the domain definition [check [prefix [tie] [top]]]
;[12:15:37]  !!! [check [prefix [tie] [top]]] appears to be constantly #T
;[12:15:38] >>> Selector produced [check [prefix [tie] [top]]] with simplicity 0
;[12:15:39] *** Gen'd [prefix [tie] [coalesce [tie]]] with simplicity 1/20736
;[12:15:42] *** Gen'd [prefix [tie] [mirror-2 [tie]]] with simplicity 1/20736
;[12:15:47] *** Gen'd [prefix [bot] [coalesce [tie]]] with simplicity 1/20736
;[12:15:47] While assessing the domain definition [prefix [bot] [coalesce [tie]]]
;[12:15:54]  !!! [prefix [bot] [coalesce [tie]]] always returns its first input
;[12:15:55] >>> Selector produced [prefix [bot] [coalesce [tie]]] with simplicity 0
;[12:15:57] *** Gen'd [prefix [bot] [mirror-2 [tie]]] with simplicity 1/20736
;[12:15:57] While assessing the domain definition [prefix [bot] [mirror-2 [tie]]]
;[12:16:25]  !!! [prefix [bot] [mirror-2 [tie]]] always returns its second input
;[12:16:25] >>> Selector produced [prefix [bot] [mirror-2 [tie]]] with simplicity 0
;[12:16:26] *** Gen'd [prefix [top] [coalesce [tie]]] with simplicity 1/20736
;[12:16:26] While assessing the domain definition [prefix [top] [coalesce [tie]]]
;[12:16:34]  !!! [prefix [top] [coalesce [tie]]] always returns its first input
;[12:16:34] >>> Selector produced [prefix [top] [coalesce [tie]]] with simplicity 0
;[12:16:36] *** Gen'd [prefix [top] [mirror-2 [tie]]] with simplicity 1/20736
;[12:16:36] While assessing the domain definition [prefix [top] [mirror-2 [tie]]]
;[12:17:07]  !!! [prefix [top] [mirror-2 [tie]]] always returns its first input
;[12:17:07] >>> Selector produced [prefix [top] [mirror-2 [tie]]] with simplicity 0
;[12:17:12] *** Gen'd [limit [bot] [coalesce [tie]]] with simplicity 1/20736
;[12:17:18] *** Gen'd [limit [top] [coalesce [tie]]] with simplicity 1/20736
;[12:17:24] *** Gen'd [identical? [bot] [coalesce [tie]]] with simplicity 1/20736
;[12:17:24] While assessing the domain definition [identical? [bot] [coalesce [tie]]]
;[12:17:32]  !!! [identical? [bot] [coalesce [tie]]] appears to be constantly ()
;[12:17:32] >>> Selector produced [identical? [bot] [coalesce [tie]]] with simplicity 0
;[12:17:36] *** Gen'd [identical? [top] [coalesce [tie]]] with simplicity 1/20736
;[12:17:36] While assessing the domain definition [identical? [top] [coalesce [tie]]]
;[12:17:44]  !!! [identical? [top] [coalesce [tie]]] appears to be constantly ()
;[12:17:45] >>> Selector produced [identical? [top] [coalesce [tie]]] with simplicity 0
;[12:17:48] *** Gen'd [meet [bot] [coalesce [tie]]] with simplicity 1/20736
;[12:17:48] While assessing the domain definition [meet [bot] [coalesce [tie]]]
;[12:17:58]  !!! [meet [bot] [coalesce [tie]]] never seems to terminate.
;[12:18:00] >>> Selector produced [meet [bot] [coalesce [tie]]] with simplicity 0
;[12:18:03] *** Gen'd [meet [top] [coalesce [tie]]] with simplicity 1/20736
;[12:18:03] While assessing the domain definition [meet [top] [coalesce [tie]]]
;[12:18:12]  !!! [meet [top] [coalesce [tie]]] never seems to terminate.
;[12:18:12] >>> Selector produced [meet [top] [coalesce [tie]]] with simplicity 0
;[12:18:19] *** Gen'd [postfix [bot] [coalesce [tie]]] with simplicity 1/20736
;[12:18:22] *** Gen'd [postfix [bot] [mirror-2 [tie]]] with simplicity 1/20736
;[12:18:24] *** Gen'd [postfix [bot] [check [bot]]] with simplicity 1/20736
;[12:18:26] *** Gen'd [postfix [bot] [check [top]]] with simplicity 1/20736
;[12:18:26] While assessing the domain definition [postfix [bot] [check [top]]]
;[12:18:27]  !!! [postfix [bot] [check [top]]] appears to be constantly #T
;[12:18:27] >>> Selector produced [postfix [bot] [check [top]]] with simplicity 0
;[12:18:28] *** Gen'd [postfix [top] [coalesce [tie]]] with simplicity 1/20736
;[12:18:31] *** Gen'd [postfix [top] [mirror-2 [tie]]] with simplicity 1/20736
;[12:18:33] *** Gen'd [postfix [top] [check [bot]]] with simplicity 1/20736
;[12:18:34] *** Gen'd [postfix [top] [check [top]]] with simplicity 1/20736
;[12:18:44] *** Gen'd [check [coalesce [mirror-2 [tie]]]] with simplicity 1/82944
;[12:18:44] While assessing the domain definition [check [coalesce [mirror-2 [tie]]]]
```

```
;[12:18:58]  !!! [check [coalesce [mirror-2 [tie]]]] appears to be constantly #T
;[12:18:59] >>> Selector produced [check [coalesce [mirror-2 [tie]]]] with simplicity 0
;[12:19:01] *** Gen'd [check [mirror-2 [mirror-2 [tie]]]] with simplicity 1/82944
;[12:19:01] While assessing the domain definition [check [mirror-2 [mirror-2 [tie]]]]
;[12:19:26]  !!! [check [mirror-2 [mirror-2 [tie]]]] appears to be constantly #T
;[12:19:26] >>> Selector produced [check [mirror-2 [mirror-2 [tie]]]] with simplicity 0
;[12:19:31] *** Gen'd [mirror-2 [mirror-2 [mirror-2 [tie]]]] with simplicity 1/82944
;[12:20:08] *** Gen'd [terminator [coalesce [mirror-2 [tie]]]] with simplicity 1/82944
;[12:20:08] While assessing the domain definition [terminator [coalesce [mirror-2 [tie]]]]
;[12:20:19]  !!! [terminator [coalesce [mirror-2 [tie]]]] never seems to terminate.
;[12:20:19] >>> Selector produced [terminator [coalesce [mirror-2 [tie]]]] with simplicity 0
;[12:20:27] *** Gen'd [coalesce [mirror-2 [mirror-2 [tie]]]] with simplicity 1/82944
;[12:20:52] *** Gen'd [postfix [top] [postfix [top] [tie]]] with simplicity 1/124416
;[12:20:53] *** Gen'd [postfix [top] [postfix [top] [bot]]] with simplicity 1/124416
;[12:20:54] *** Gen'd [postfix [top] [postfix [top] [top]]] with simplicity 1/124416
;[12:20:55] *** Gen'd [postfix [top] [postfix [bot] [tie]]] with simplicity 1/124416
;[12:20:56] *** Gen'd [postfix [top] [postfix [bot] [bot]]] with simplicity 1/124416
;[12:20:57] *** Gen'd [postfix [top] [postfix [bot] [top]]] with simplicity 1/124416
;[12:20:58] *** Gen'd [postfix [top] [meet [top] [top]]] with simplicity 1/124416
;[12:21:00] *** Gen'd [postfix [top] [meet [bot] [bot]]] with simplicity 1/124416
;[12:21:00] While assessing the domain definition [postfix [top] [meet [bot] [bot]]]
;[12:21:08]  !!! [postfix [top] [meet [bot] [bot]]] appears to be constantly ()
;[12:21:09] *** Gen'd [postfix [top] [identical? [bot] [bot]]] with simplicity 1/124416
;[12:21:09] While assessing the domain definition [postfix [top] [identical? [bot] [bot]]]
;[12:21:18]  !!! [postfix [top] [identical? [bot] [bot]]] appears to be constantly ()
;[12:21:19] *** Gen'd [postfix [top] [identical? [bot] [top]]] with simplicity 1/124416
;[12:21:19] While assessing the domain definition [postfix [top] [identical? [bot] [top]]]
;[12:21:28]  !!! [postfix [top] [identical? [bot] [top]]] appears to be constantly ()
;[12:21:30] *** Gen'd [postfix [top] [limit [top] [bot]]] with simplicity 1/124416
;[12:21:33] *** Gen'd [postfix [top] [limit [top] [top]]] with simplicity 1/124416
;[12:21:37] *** Gen'd [postfix [top] [limit [bot] [bot]]] with simplicity 1/124416
;[12:21:39] *** Gen'd [postfix [top] [limit [bot] [top]]] with simplicity 1/124416
```

After the system has generated some number (150 in this case) of domain definitions, it rebiases the pool of definitions by picking out polygenic behaviors, doing a frequency analysis of their constituent elements and starting up a new pool with those elements.

```
;[12:21:42] *** The pool #17{264-Pool: ALEPH} has
;[12:21:42] *** 34 constant definitions,
;[12:21:42] *** 10 identity operations, and
;[12:21:42] *** 99 other effective domain definitions.
```

These numbers don't add up to 150 because there were 17 ineffective definitions which never terminated.

```
;[12:21:42] While rebiasing the pool #17264-Pool: ALEPH
;[12:21:43]  *** Type analysis identifies five initial cliques
;[12:21:44]  ***  Size From       To                                                                  For instance
;[12:21:45]  ***  ---- ----       --                                                                  ------------
;[12:21:46]  ***  39   #16{Lists} #15{@FCN #16{Lists} --> #16{Lists}}                                  [mirror-2 [prefix [tie]
;[12:21:46]  ***  32   #16{Lists} #16{Lists}                                                           [prefix [bot] [bot]]
;[12:21:46]  ***  17   #16{Lists} #91{Booleans}                                                        [prefix [check [top]] [b
;[12:21:47]  ***  7    #16{Lists} #64{@FCN #16{Lists} --> #91{Booleans}}                               [identical? [bot] [top]]
;[12:21:47]  ***  4    #16{Lists} #50{@FCN #16{Lists} --> #15{@FCN #16{Lists} --> #16{Lists}}}         [prefix [tie] [tie]]
```

Given an initial partitioning of definitions by type, Cyrano tries to divide them further. This works by generating one example of some definition in a current clique and seeing the results of all the elements in the clique on the same inputs. This yields a new clustering of the clique. Any 'singletons' consisting of only one definition are removed and clique division continues.

```
;[12:21:48]  *** which I will try and further distinguish (in fifteen passes)
;[12:22:26]  === On first pass, generated 54 new cliques and
;[12:22:26]  ===                 removed 45 singletons, leaving 14 cliques altogether
;[12:22:42]  === On second pass, generated 11 new cliques and
;[12:22:42]  ===                 removed 10 singletons, leaving 15 cliques altogether
;[12:23:06]  === On third pass, generated 3 new cliques and
;[12:23:06]  ===                 removed 5 singletons, leaving 13 cliques altogether
;[12:23:14]  === On fourth pass, generated 1 new clique and
;[12:23:14]  ===                 removed 0 singletons, leaving 14 cliques altogether
;[12:23:22]  === On fifth pass, generated 1 new clique and
;[12:23:22]  ===                 removed 2 singletons, leaving 13 cliques altogether...
;[12:23:43]  === On ninth pass, generated 1 new clique and
;[12:23:43]  ===                 removed 2 singletons, leaving 12 cliques altogether...
;[12:23:59]  === On thirteenth pass, generated 1 new clique and
;[12:23:59]  ===                 removed 0 singletons, leaving 13 cliques altogether..
;[12:24:08]  *** After fifteen passes, finished with thirteen cliques
;[12:24:08]  ***       with a mean of 2.692 (sd=2.548; n=13) members each.
;[12:24:11]  ***  Size For instance                        Which computes
;[12:24:12]  ***  ---- ------------                       --------------
;[12:24:12]  ***  8    [check [postfix [top] [top]]]      #T <=== [Tie 18/13]1                          <== 7
;[12:24:13]  ***  3    [meet [top] [top]]                 [Tie 5/6] <=== [Tie 6/7], [Tie 14/2]1         <== 8
;[12:24:13]  ***  3    [postfix [bot] [check [bot]]]      #T <=== [Tie 10/17]1                          <== 9
;[12:24:13]  ***  3    [check [prefix [top] [bot]]]       #T <=== [Tie 24/5]1                           <== 10
;[12:24:14]  ***  2    [terminator [prefix [bot] [bot]]] [Tie 6/1] <=== [Tie 5/7]1                      <== 11
;[12:24:14]  ***  2    [prefix [top] [top]]               [Tie 6/6] <=== [Tie 8/10]1                    <== 12
;[12:24:15]  ***  2    [terminator [prefix [top] [bot]]] () <=== [Tie 4/3]1                             <== 13
;[12:24:15]  ***  2    [check [top]]                      #T <=== [Tie 7/8]
;[12:24:16]  ***  2    [meet [bot] [bot]]                 [Tie 6/2] <=== [Tie 11/9], [Tie 18/3]
;[12:24:16]  ***  2    [prefix [bot] [top]]               [Tie 5/5] <=== [Tie 8/14]
;[12:24:16]  ***  2    [prefix [top] [bot]]               [Tie 5/5] <=== [Tie 22/24]
;[12:24:17]  ***  2    [terminator [prefix [top] [top]]] [Tie 1/6] <=== [Tie 17/7]1                     <== 14
;[12:24:17]  ***  2    [postfix [bot] [bot]]              [Tie 6/15] <=== [Tie 1/17]
```

Several cliques denote simple predicates — (Note 7/69), (Note 9/69), (Note 10/69), and (Note 12/69) — which fail only on small tie structures. Note that clique analysis is not as reliable as we might wish because some sets of definitions agree on most instances but differ on a few key examples. For instance, probably not all of (Note 7/69) really compute 'higher than 2', but that is merely the leftovers which have agreed 'thus far' on particular inputs.

The terminators — (Note 11/69), (Note 13/69), and (Note 14/69) — all return nil or a terminal structure. Also note the associativity of the 'contains substructure' comparision that Cyrano notes by observing the equivalence of `[meet [top] [top]]` *and* `[mirror-2 [meet [top] [top]]]`. *This was noted above.*

```
;[12:24:17]  *** Identified two significant constants
;[12:24:18]  *** Value Size For instance
;[12:24:18]  *** ----- ---- ------------
```

```
;[12:24:18]  *** ()    18   [mirror-2 [identical? [bot] [bot]]]
;[12:24:18]  *** #T    16   [prefix [check [bot]] [tie]]
```

Note that these results would be somewhat different if I the terminals of tie structures '() were not identical to the boolean #F (false).

```
;[12:24:27]  *** Introducing new primitives based on frequency analysis
;[12:24:29]  ***  New Primitive          Frequency
;[12:24:29]  ***  -------------          ---------
;[12:24:30]  ***  [tie]                  8/285
;[12:24:30]  ***  [top]                  14/95
;[12:24:31]  ***  [bot]                  13/95
;[12:24:31]  ***  [prefix]               7/95
;[12:24:31]  ***  [postfix]              16/285
;[12:24:32]  ***  [meet]                 7/285
;[12:24:32]  ***  [terminator]           2/95
;[12:24:32]  ***  [check]                16/285
;[12:24:33]  ***  [coalesce]             2/95
;[12:24:33]  ***  [mirror-2]             4/285
;[12:24:33]  ***  [check [top]]          4/285
;[12:24:34]  ***  [check [bot]]          1/57
;[12:24:34]  ***  [postfix [top]]        8/285
;[12:24:34]  ***  [postfix [bot]]        2/95
;[12:24:35]  ***  [meet [top]]           4/285
;[12:24:35]  ***  [meet [bot]]           1/95
;[12:24:35]  ***  [coalesce [tie]]       4/285
;[12:24:35]  ***  [prefix [top]]         3/95
;[12:24:36]  ***  [prefix [bot]]         8/285
;[12:24:36]  ***  [prefix [bot] [bot]]   1/95
;[12:24:36]  ***  [prefix [top] [top]]   1/95
;[12:24:36]  ***  [prefix [top] [bot]]   1/95
;[12:24:37]  ***  [meet [bot] [bot]]     1/95
;[12:24:37]  ***  [meet [top] [top]]     4/285
;[12:24:37]  ***  [postfix [bot] [bot]] 1/95
;[12:24:37]  ***  [postfix [top] [top]] 1/95
```

Cyrano analyzes the constituents of all the definitions which show interesting regularities: polygenic behaviors (large clique participants), constant valued definitions, and projections (variants of the identity operation). Based on the fragmentary definitions which appear most commonly among this set, it picks out the definitions above and constructs a new pool from them; to this pool, it also adds the constant terms identified above and the fastest implementations of each behavioral clique it has identified.
When the new pool has been constructed, exploration begins again....

```
;[12:24:47] *** Gen'd [meet [bot] [bot] ()] with simplicity 41/23400
;[12:24:48] While assessing the domain definition [meet [bot] [bot] ()]
;[12:24:50]  !!! [meet [bot] [bot] ()] appears to be constantly ()
;[12:24:50] >>> Selector produced [meet [bot] [bot] ()] with simplicity 0
```

Noticing this regularity is equivalent to saying "the common substructure of any tie structure with the terminal tie structure is the terminal tie structure."

```
;[12:24:52] ??? The combination of [postfix [top] [top]] with (),
;[12:24:53] ??? which looked legal, turns out to be undefined.
```

Cyrano tells the observer if combinations which seemed reasonable to its typing system actually yielded undefined values. Since Cyrano is not told the difference between terminal ties and non-terminal ties, it has no way of knowing that the above combination cannot yield a result.

```
;[12:24:54] ??? The combination of [postfix [bot] [bot]] with (),
;[12:24:55] ??? which looked legal, turns out to be undefined.
;[12:24:57] ??? The combination of [prefix [top] [bot]] with (),
;[12:24:57] ??? which looked legal, turns out to be undefined.
;[12:24:59] *** Gen'd [meet [top] [top] ()] with simplicity 41/23400
;[12:25:00] While assessing the domain definition [meet [top] [top] ()]
;[12:25:02]  !!! [meet [top] [top] ()] appears to be constantly ()
;[12:25:03] >>> Selector produced [meet [top] [top] ()] with simplicity 0
;[12:25:15] ??? The combination of [prefix [check [top]] [bot]] with (),
;[12:25:15] ??? which looked legal, turns out to be undefined.
;[12:25:17] ??? The combination of [prefix [check [bot]] [top]] with (),
;[12:25:18] ??? which looked legal, turns out to be undefined.
;[12:25:25] ??? The combination of [postfix [top] [bot]] with (),
;[12:25:25] ??? which looked legal, turns out to be undefined.
;[12:25:40] *** Gen'd [postfix [bot] [meet [top] [top]]] with simplicity 41/40560
;[12:25:40] While assessing the domain definition [postfix [bot] [meet [top] [top]]]
;[12:25:51]  !!! [postfix [bot] [meet [top] [top]]] appears to be constantly ()
;[12:25:52] >>> Selector produced [postfix [bot] [meet [top] [top]]] with simplicity 0
```

Because tops and bottoms are generated from different sources (as described above), the only common substructure found between one tie structure and the bottom of another is the terminal tie structure.

```
;[12:25:53] *** Gen'd [postfix [bot] [prefix [top] [bot]]] with simplicity 41/40560
;[12:25:54] *** Gen'd [postfix [bot] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:25:55] *** Gen'd [postfix [bot] [postfix [top] [top]]] with simplicity 41/40560
;[12:25:56] *** Gen'd [postfix [bot] [meet [bot] [bot]]] with simplicity 41/40560
;[12:25:57] *** Gen'd [postfix [bot] [check [bot]]] with simplicity 41/40560
;[12:25:57] *** Gen'd [coalesce [meet [top] [top]]] with simplicity 41/40560
;[12:25:57] While assessing the domain definition [coalesce [meet [top] [top]]]
;[12:26:08]  !!! [coalesce [meet [top] [top]]] always returns its first input
;[12:26:08] >>> Selector produced [coalesce [meet [top] [top]]] with simplicity 0
;[12:26:08] *** Gen'd [coalesce [meet [bot] [bot]]] with simplicity 41/40560
;[12:26:09] While assessing the domain definition [coalesce [meet [bot] [bot]]]
;[12:26:18]  !!! [coalesce [meet [bot] [bot]]] always returns its first input
;[12:26:19] >>> Selector produced [coalesce [meet [bot] [bot]]] with simplicity 0
;[12:26:20] *** Gen'd [prefix [top] [meet [top] [top]]] with simplicity 41/40560
;[12:26:24] *** Gen'd [prefix [top] [prefix [top] [bot]]] with simplicity 41/40560
;[12:26:25] *** Gen'd [prefix [top] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:26:27] *** Gen'd [prefix [top] [postfix [top] [top]]] with simplicity 41/40560
;[12:26:28] *** Gen'd [prefix [top] [meet [bot] [bot]]] with simplicity 41/40560
;[12:26:35] *** Gen'd [meet [bot] [prefix [top] [bot]]] with simplicity 41/40560
;[12:26:35] While assessing the domain definition [meet [bot] [prefix [top] [bot]]]
;[12:26:39]  !!! [meet [bot] [prefix [top] [bot]]] appears to be constantly ()
;[12:26:40] *** Gen'd [meet [bot] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:26:41] *** Gen'd [meet [bot] [postfix [top] [top]]] with simplicity 41/40560
;[12:26:41] While assessing the domain definition [meet [bot] [postfix [top] [top]]]
;[12:26:45]  !!! [meet [bot] [postfix [top] [top]]] appears to be constantly ()
;[12:26:47] *** Gen'd [postfix [top] [meet [top] [top]]] with simplicity 41/40560
;[12:26:49] *** Gen'd [postfix [top] [prefix [top] [bot]]] with simplicity 41/40560
;[12:26:49] *** Gen'd [postfix [top] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:26:50] *** Gen'd [postfix [top] [postfix [top] [top]]] with simplicity 41/40560
;[12:26:50] *** Gen'd [postfix [top] [meet [bot] [bot]]] with simplicity 41/40560
;[12:26:51] While assessing the domain definition [postfix [top] [meet [bot] [bot]]]
;[12:27:02]  !!! [postfix [top] [meet [bot] [bot]]] appears to be constantly ()
```

```
;[12:27:02] *** Gen'd [postfix [top] [check [bot]]] with simplicity 41/40560
;[12:27:03] *** Gen'd [check [meet [top] [top]]] with simplicity 41/40560
;[12:27:04] While assessing the domain definition [check [meet [top] [top]]]
;[12:27:17]  !!! [check [meet [top] [top]]] appears to be constantly #T
;[12:27:18] >>> Selector produced [check [meet [top] [top]]] with simplicity 0
;[12:27:18] *** Gen'd [check [prefix [top] [bot]]] with simplicity 41/40560
;[12:27:18] While assessing the domain definition [check [prefix [top] [bot]]]
;[12:27:19]  !!! [check [prefix [top] [bot]]] appears to be constantly #T
;[12:27:19] >>> Selector produced [check [prefix [top] [bot]]] with simplicity 0
;[12:27:19] *** Gen'd [check [postfix [bot] [bot]]] with simplicity 41/40560
;[12:27:20] *** Gen'd [check [postfix [top] [top]]] with simplicity 41/40560
;[12:27:20] *** Gen'd [check [meet [bot] [bot]]] with simplicity 41/40560
;[12:27:21] While assessing the domain definition [check [meet [bot] [bot]]]
;[12:27:32]  !!! [check [meet [bot] [bot]]] appears to be constantly #T
;[12:27:33] >>> Selector produced [check [meet [bot] [bot]]] with simplicity 0
;[12:27:34] *** Gen'd [meet [top] [prefix [top] [bot]]] with simplicity 41/40560
;[12:27:34] While assessing the domain definition [meet [top] [prefix [top] [bot]]]
;[12:27:37]  !!! [meet [top] [prefix [top] [bot]]] appears to be constantly ()
;[12:27:37] *** Gen'd [meet [top] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:27:38] While assessing the domain definition [meet [top] [postfix [bot] [bot]]]
;[12:27:41]  !!! [meet [top] [postfix [bot] [bot]]] appears to be constantly ()
;[12:27:42] *** Gen'd [meet [top] [postfix [top] [top]]] with simplicity 41/40560
;[12:27:43] *** Gen'd [mirror-2 [meet [top] [top]]] with simplicity 41/40560
;[12:27:44] *** Gen'd [mirror-2 [meet [bot] [bot]]] with simplicity 41/40560
;[12:27:45] *** Gen'd [prefix [bot] [meet [top] [top]]] with simplicity 41/40560
;[12:27:49] *** Gen'd [prefix [bot] [prefix [top] [bot]]] with simplicity 41/40560
;[12:27:50] *** Gen'd [prefix [bot] [postfix [bot] [bot]]] with simplicity 41/40560
;[12:27:51] *** Gen'd [prefix [bot] [postfix [top] [top]]] with simplicity 41/40560
;[12:27:52] *** Gen'd [prefix [bot] [meet [bot] [bot]]] with simplicity 41/40560
;[12:27:56] ??? The combination of [top] with (),
;[12:27:56] ??? which looked legal, turns out to be undefined.
;[12:27:57] *** Gen'd [prefix [bot] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:27:57] While assessing the domain definition [prefix [bot] [terminator [prefix [top] [bot]]]]
;[12:28:00]  !!! [prefix [bot] [terminator [prefix [top] [bot]]]] appears to be constantly ()
;[12:28:01] *** Gen'd [prefix [bot] [postfix [top] [bot]]] with simplicity 1/1560
;[12:28:02] *** Gen'd [prefix [bot] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:28:02] While assessing the domain definition [prefix [bot] [terminator [postfix [bot] [bot]]]]
;[12:28:04]  !!! [prefix [bot] [terminator [postfix [bot] [bot]]]] appears to be constantly ()
;[12:28:05] *** Gen'd [prefix [bot] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:28:09] *** Gen'd [meet [top] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:28:09] While assessing the domain definition [meet [top] [terminator [prefix [top] [bot]]]]
;[12:28:15]  !!! [meet [top] [terminator [prefix [top] [bot]]]] appears to be constantly ()
;[12:28:16] *** Gen'd [meet [top] [postfix [top] [bot]]] with simplicity 1/1560
;[12:28:16] While assessing the domain definition [meet [top] [postfix [top] [bot]]]
;[12:28:20]  !!! [meet [top] [postfix [top] [bot]]] appears to be constantly ()
;[12:28:21] *** Gen'd [meet [top] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:28:21] While assessing the domain definition [meet [top] [terminator [postfix [bot] [bot]]]]
;[12:28:38]  !!! [meet [top] [terminator [postfix [bot] [bot]]]] appears to be constantly ()
;[12:28:40] *** Gen'd [meet [top] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:28:43] *** Gen'd [check [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:28:44] While assessing the domain definition [check [terminator [prefix [top] [bot]]]]
;[12:28:45]  !!! [check [terminator [prefix [top] [bot]]]] appears to be constantly #T
;[12:28:46] >>> Selector produced [check [terminator [prefix [top] [bot]]]] with simplicity 0
;[12:28:46] *** Gen'd [check [postfix [top] [bot]]] with simplicity 1/1560
;[12:28:47] *** Gen'd [check [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:28:47] While assessing the domain definition [check [terminator [postfix [bot] [bot]]]]
;[12:28:49]  !!! [check [terminator [postfix [bot] [bot]]]] appears to be constantly #T
;[12:28:49] >>> Selector produced [check [terminator [postfix [bot] [bot]]]] with simplicity 0
;[12:28:50] *** Gen'd [check [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:28:50] While assessing the domain definition [check [terminator [postfix [top] [top]]]]
;[12:28:52]  !!! [check [terminator [postfix [top] [top]]]] appears to be constantly #T
;[12:28:52] >>> Selector produced [check [terminator [postfix [top] [top]]]] with simplicity 0
```

```
;[12:28:54] *** Gen'd (()) with simplicity 1/1560
```

> *Once domain objects are introduced into the pool, the domain operations in the pool will produce new domain objects with corresponding simplicity.*

```
;[12:28:55] *** Gen'd [postfix [top] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:28:56] *** Gen'd [postfix [top] [postfix [top] [bot]]] with simplicity 1/1560
;[12:28:57] *** Gen'd [postfix [top] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:28:59] *** Gen'd [postfix [top] [check [prefix [bot] [bot]]]] with simplicity 1/1560
;[12:29:00] *** Gen'd [postfix [top] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:29:01] *** Gen'd [postfix [top] [prefix [check [bot]] [top]]] with simplicity 1/1560
;[12:29:02] *** Gen'd [postfix [top] [prefix [check [top]] [bot]]] with simplicity 1/1560
;[12:29:02] While assessing the domain definition [postfix [top] [prefix [check [top]] [bot]]]
;[12:29:03]  !!! [postfix [top] [prefix [check [top]] [bot]]] appears to be constantly #T
;[12:29:09] ??? The combination of [prefix [top] [top]] with (),
;[12:29:09] ??? which looked legal, turns out to be undefined.
;[12:29:10] *** Gen'd [terminator [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:29:10] While assessing the domain definition [terminator [terminator [prefix [top] [bot]]]]
;[12:29:18]  !!! [terminator [terminator [prefix [top] [bot]]]] never seems to terminate.
;[12:29:18] >>> Selector produced [terminator [terminator [prefix [top] [bot]]]] with simplicity 0
;[12:29:18] *** Gen'd [terminator [postfix [top] [bot]]] with simplicity 1/1560
;[12:29:20] *** Gen'd [terminator [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:29:21] While assessing the domain definition [terminator [terminator [postfix [bot] [bot]]]]
;[12:29:28]  !!! [terminator [terminator [postfix [bot] [bot]]]] never seems to terminate.
;[12:29:28] >>> Selector produced [terminator [terminator [postfix [bot] [bot]]]] with simplicity 0
;[12:29:29] *** Gen'd [terminator [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:29:30] While assessing the domain definition [terminator [terminator [postfix [top] [top]]]]
;[12:29:37]  !!! [terminator [terminator [postfix [top] [top]]]] never seems to terminate.
;[12:29:38] >>> Selector produced [terminator [terminator [postfix [top] [top]]]] with simplicity 0
;[12:29:39] *** Gen'd [meet [bot] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:29:39] While assessing the domain definition [meet [bot] [terminator [prefix [top] [bot]]]]
;[12:29:45]  !!! [meet [bot] [terminator [prefix [top] [bot]]]] appears to be constantly ()
;[12:29:46] *** Gen'd [meet [bot] [postfix [top] [bot]]] with simplicity 1/1560
;[12:29:46] While assessing the domain definition [meet [bot] [postfix [top] [bot]]]
;[12:29:49]  !!! [meet [bot] [postfix [top] [bot]]] appears to be constantly ()
;[12:29:50] *** Gen'd [meet [bot] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:29:53] *** Gen'd [meet [bot] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:29:54] While assessing the domain definition [meet [bot] [terminator [postfix [top] [top]]]]
;[12:30:13]  !!! [meet [bot] [terminator [postfix [top] [top]]]] appears to be constantly ()
;[12:30:16] ??? The combination of [bot] with (),
;[12:30:16] ??? which looked legal, turns out to be undefined.
;[12:30:18] ??? The combination of [prefix [bot] [bot]] with (),
;[12:30:18] ??? which looked legal, turns out to be undefined.
;[12:30:19] *** Gen'd [prefix [top] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:30:21] *** Gen'd [prefix [top] [postfix [top] [bot]]] with simplicity 1/1560
;[12:30:22] *** Gen'd [prefix [top] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:30:24] *** Gen'd [prefix [top] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:30:24] While assessing the domain definition [prefix [top] [terminator [postfix [top] [top]]]]
;[12:30:26]  !!! [prefix [top] [terminator [postfix [top] [top]]]] appears to be constantly ()
;[12:30:28] *** Gen'd [tie ()] with simplicity 1/1560
;[12:30:31] *** Gen'd [postfix [bot] [terminator [prefix [top] [bot]]]] with simplicity 1/1560
;[12:30:31] While assessing the domain definition [postfix [bot] [terminator [prefix [top] [bot]]]]
;[12:30:33]  !!! [postfix [bot] [terminator [prefix [top] [bot]]]] appears to be constantly ()
;[12:30:34] *** Gen'd [postfix [bot] [postfix [top] [bot]]] with simplicity 1/1560
;[12:30:35] *** Gen'd [postfix [bot] [terminator [postfix [bot] [bot]]]] with simplicity 1/1560
;[12:30:36] *** Gen'd [postfix [bot] [check [prefix [bot] [bot]]]] with simplicity 1/1560
;[12:30:38] *** Gen'd [postfix [bot] [terminator [postfix [top] [top]]]] with simplicity 1/1560
;[12:30:38] While assessing the domain definition [postfix [bot] [terminator [postfix [top] [top]]]]
;[12:30:41]  !!! [postfix [bot] [terminator [postfix [top] [top]]]] appears to be constantly ()
;[12:30:41] *** Gen'd [postfix [bot] [prefix [check [bot]] [top]]] with simplicity 1/1560
;[12:30:42] While assessing the domain definition [postfix [bot] [prefix [check [bot]] [top]]]
;[12:30:43]  !!! [postfix [bot] [prefix [check [bot]] [top]]] appears to be constantly #T
```

```
;[12:30:43] *** Gen'd [postfix [bot] [prefix [check [top]] [bot]]] with simplicity 1/1560
;[12:30:44] While assessing the domain definition [postfix [bot] [prefix [check [top]] [bot]]]
;[12:30:48]  !!! [postfix [bot] [prefix [check [top]] [bot]]] appears to be constantly #T
;[12:31:17] *** Gen'd [postfix [bot] [tie]] with simplicity 1/2704
;[12:31:18] *** Gen'd [postfix [bot] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:31:19] *** Gen'd [postfix [bot] [check [top]]] with simplicity 1/2704
;[12:31:19] >>> Selector produced [postfix [bot] [check [top]]] with simplicity 0
;[12:31:20] *** Gen'd [postfix [bot] [prefix [top] [top]]] with simplicity 1/2704
;[12:31:21] *** Gen'd [postfix [bot] [coalesce [tie]]] with simplicity 1/2704
;[12:31:22] *** Gen'd [postfix [bot] [top]] with simplicity 1/2704
;[12:31:23] *** Gen'd [prefix [top] [tie]] with simplicity 1/2704
;[12:31:23] While assessing the domain definition [prefix [top] [tie]]
;[12:31:48]  !!! [prefix [top] [tie]] always returns its second input
;[12:31:48] >>> Selector produced [prefix [top] [tie]] with simplicity 0
;[12:31:49] *** Gen'd [prefix [top] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:31:50] *** Gen'd [prefix [top] [prefix [top] [top]]] with simplicity 1/2704
;[12:31:50] *** Gen'd [prefix [top] [coalesce [tie]]] with simplicity 1/2704
;[12:31:51] While assessing the domain definition [prefix [top] [coalesce [tie]]]
;[12:31:59]  !!! [prefix [top] [coalesce [tie]]] always returns its first input
;[12:31:59] >>> Selector produced [prefix [top] [coalesce [tie]]] with simplicity 0
;[12:32:02] *** Gen'd [meet [bot] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:32:03] *** Gen'd [meet [bot] [prefix [top] [top]]] with simplicity 1/2704
;[12:32:04] While assessing the domain definition [meet [bot] [prefix [top] [top]]]
;[12:32:07]  !!! [meet [bot] [prefix [top] [top]]] appears to be constantly ()
;[12:32:07] *** Gen'd [meet [bot] [coalesce [tie]]] with simplicity 1/2704
;[12:32:07] >>> Selector produced [meet [bot] [coalesce [tie]]] with simplicity 0
;[12:32:08] *** Gen'd [meet [bot] [top]] with simplicity 1/2704
;[12:32:08] While assessing the domain definition [meet [bot] [top]]
;[12:32:11]  !!! [meet [bot] [top]] appears to be constantly ()
;[12:32:11] >>> Selector produced [meet [bot] [top]] with simplicity 0
;[12:32:12] *** Gen'd [terminator [prefix [bot] [bot]]] with simplicity 1/2704
;[12:32:13] *** Gen'd [terminator [bot]] with simplicity 1/2704
;[12:32:13] While assessing the domain definition [terminator [bot]]
;[12:32:15]  !!! [terminator [bot]] appears to be constantly ()
;[12:32:15] >>> Selector produced [terminator [bot]] with simplicity 0
;[12:32:15] *** Gen'd [terminator [prefix [top] [top]]] with simplicity 1/2704
;[12:32:17] *** Gen'd [terminator [coalesce [tie]]] with simplicity 1/2704
;[12:32:17] >>> Selector produced [terminator [coalesce [tie]]] with simplicity 0
;[12:32:18] *** Gen'd [terminator [top]] with simplicity 1/2704
;[12:32:18] While assessing the domain definition [terminator [top]]
;[12:32:20]  !!! [terminator [top]] appears to be constantly ()
;[12:32:20] >>> Selector produced [terminator [top]] with simplicity 0
;[12:32:24] *** Gen'd [postfix [top] [tie]] with simplicity 1/2704
;[12:32:25] *** Gen'd [postfix [top] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:32:26] *** Gen'd [postfix [top] [check [top]]] with simplicity 1/2704
;[12:32:27] *** Gen'd [postfix [top] [prefix [top] [top]]] with simplicity 1/2704
;[12:32:28] *** Gen'd [postfix [top] [coalesce [tie]]] with simplicity 1/2704
;[12:32:30] *** Gen'd [check [tie]] with simplicity 1/2704
;[12:32:30] While assessing the domain definition [check [tie]]
;[12:32:41]  !!! [check [tie]] appears to be constantly #T
;[12:32:41] >>> Selector produced [check [tie]] with simplicity 0
;[12:32:42] *** Gen'd [check [prefix [top] [top]]] with simplicity 1/2704
;[12:32:43] *** Gen'd [check [coalesce [tie]]] with simplicity 1/2704
;[12:32:43] While assessing the domain definition [check [coalesce [tie]]]
;[12:32:51]  !!! [check [coalesce [tie]]] appears to be constantly #T
;[12:32:51] >>> Selector produced [check [coalesce [tie]]] with simplicity 0
;[12:32:53] *** Gen'd [meet [top] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:32:53] While assessing the domain definition [meet [top] [prefix [bot] [bot]]]
;[12:32:56]  !!! [meet [top] [prefix [bot] [bot]]] appears to be constantly ()
;[12:32:57] *** Gen'd [meet [top] [bot]] with simplicity 1/2704
;[12:32:57] While assessing the domain definition [meet [top] [bot]]
;[12:33:00]  !!! [meet [top] [bot]] appears to be constantly ()
```

```
;[12:33:00] >>> Selector produced [meet [top] [bot]] with simplicity 0
;[12:33:01] *** Gen'd [meet [top] [prefix [top] [top]]] with simplicity 1/2704
;[12:33:02] *** Gen'd [meet [top] [coalesce [tie]]] with simplicity 1/2704
;[12:33:02] >>> Selector produced [meet [top] [coalesce [tie]]] with simplicity 0
;[12:33:02] *** Gen'd [mirror-2 [tie]] with simplicity 1/2704
;[12:33:04] *** Gen'd [prefix [bot] [tie]] with simplicity 1/2704
;[12:33:04] While assessing the domain definition [prefix [bot] [tie]]
;[12:33:30]  !!! [prefix [bot] [tie]] always returns its first input
;[12:33:30] >>> Selector produced [prefix [bot] [tie]] with simplicity 0
;[12:33:31] *** Gen'd [prefix [bot] [prefix [bot] [bot]]] with simplicity 1/2704
;[12:33:32] *** Gen'd [prefix [bot] [prefix [top] [top]]] with simplicity 1/2704
;[12:33:32] *** Gen'd [prefix [bot] [coalesce [tie]]] with simplicity 1/2704
;[12:33:33] While assessing the domain definition [prefix [bot] [coalesce [tie]]]
;[12:33:41]  !!! [prefix [bot] [coalesce [tie]]] always returns its first input
;[12:33:41] >>> Selector produced [prefix [bot] [coalesce [tie]]] with simplicity 0
;[12:33:42] *** Gen'd [prefix [bot] [top]] with simplicity 1/2704
;[12:33:57] *** Gen'd [meet [postfix [top] [top]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:33:58] While assessing the domain definition [meet [postfix [top] [top]] [prefix [top] [bot]]]
;[12:34:00]  !!! [meet [postfix [top] [top]] [prefix [top] [bot]]] appears to be constantly ()
;[12:34:01] *** Gen'd [meet [postfix [top] [top]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:34:01] While assessing the domain definition [meet [postfix [top] [top]] [postfix [bot] [bot]]]
;[12:34:04]  !!! [meet [postfix [top] [top]] [postfix [bot] [bot]]] appears to be constantly ()
;[12:34:05] *** Gen'd [meet [postfix [top] [top]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:34:08] *** Gen'd [meet [postfix [bot] [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:34:08] While assessing the domain definition [meet [postfix [bot] [bot]] [prefix [top] [bot]]]
;[12:37:46]  !!! [meet [postfix [bot] [bot]] [prefix [top] [bot]]] appears to be constantly ()
;[12:37:47] *** Gen'd [meet [postfix [bot] [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:37:49] *** Gen'd [meet [postfix [bot] [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:37:49] While assessing the domain definition [meet [postfix [bot] [bot]] [postfix [top] [top]]]
;[12:37:52]  !!! [meet [postfix [bot] [bot]] [postfix [top] [top]]] appears to be constantly ()
;[12:37:54] *** Gen'd [meet [prefix [top] [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:37:56] *** Gen'd [meet [prefix [top] [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:37:57] While assessing the domain definition [meet [prefix [top] [bot]] [postfix [bot] [bot]]]
;[12:38:00]  !!! [meet [prefix [top] [bot]] [postfix [bot] [bot]]] appears to be constantly ()
;[12:38:01] *** Gen'd [meet [prefix [top] [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:38:01] While assessing the domain definition [meet [prefix [top] [bot]] [postfix [top] [top]]]
;[12:38:05]  !!! [meet [prefix [top] [bot]] [postfix [top] [top]]] appears to be constantly ()
;[12:38:16] *** Gen'd [prefix [check [bot]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:38:19] *** Gen'd [prefix [check [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:38:20] While assessing the domain definition [prefix [check [bot]] [prefix [top] [bot]]]
;[12:38:22]  !!! [prefix [check [bot]] [prefix [top] [bot]]] appears to be constantly #T
;[12:38:23] *** Gen'd [prefix [check [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:38:25] *** Gen'd [prefix [check [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:38:27] *** Gen'd [prefix [check [bot]] [meet [bot] [bot]]] with simplicity 1681/31636800
;[12:38:34] *** Gen'd [prefix [meet [bot] [bot]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:38:34] While assessing the domain definition [prefix [meet [bot] [bot]] [meet [top] [top]]]
;[12:38:43]  !!! [prefix [meet [bot] [bot]] [meet [top] [top]]] never seems to terminate.
;[12:38:44] >>> Selector produced [prefix [meet [bot] [bot]] [meet [top] [top]]] with simplicity 0
;[12:38:45] *** Gen'd [prefix [meet [bot] [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:38:45] While assessing the domain definition [prefix [meet [bot] [bot]] [prefix [top] [bot]]]
;[12:38:54]  !!! [prefix [meet [bot] [bot]] [prefix [top] [bot]]] appears to be constantly ()
;[12:38:55] *** Gen'd [prefix [meet [bot] [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:38:57] *** Gen'd [prefix [meet [bot] [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:38:58] While assessing the domain definition [prefix [meet [bot] [bot]] [postfix [top] [top]]]
;[12:39:08]  !!! [prefix [meet [bot] [bot]] [postfix [top] [top]]] appears to be constantly ()
;[12:39:09] *** Gen'd [prefix [meet [bot] [bot]] [meet [bot] [bot]]] with simplicity 1681/31636800
;[12:39:09] While assessing the domain definition [prefix [meet [bot] [bot]] [meet [bot] [bot]]]
;[12:39:18]  !!! [prefix [meet [bot] [bot]] [meet [bot] [bot]]] never seems to terminate.
;[12:39:19] >>> Selector produced [prefix [meet [bot] [bot]] [meet [bot] [bot]]] with simplicity 0
;[12:39:20] *** Gen'd [prefix [postfix [top] [top]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:39:23] *** Gen'd [prefix [postfix [top] [top]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:39:25] *** Gen'd [prefix [postfix [top] [top]] [postfix [bot] [bot]]] with simplicity 1681/31636800
```

```
;[12:39:26] *** Gen'd [prefix [postfix [top] [top]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:39:27] *** Gen'd [prefix [postfix [top] [top]] [meet [bot] [bot]]] with simplicity 1681/31636800
;[12:39:31] *** Gen'd [prefix [postfix [bot] [bot]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:39:34] *** Gen'd [prefix [postfix [bot] [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:39:35] *** Gen'd [prefix [postfix [bot] [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:39:37] *** Gen'd [prefix [postfix [bot] [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:39:38] *** Gen'd [prefix [postfix [bot] [bot]] [meet [bot] [bot]]] with simplicity 1681/31636800
;[12:39:42] *** Gen'd [prefix [prefix [top] [bot]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:39:45] *** Gen'd [prefix [prefix [top] [bot]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:39:46] *** Gen'd [prefix [prefix [top] [bot]] [postfix [bot] [bot]]] with simplicity 1681/31636800
;[12:39:47] *** Gen'd [prefix [prefix [top] [bot]] [postfix [top] [top]]] with simplicity 1681/31636800
;[12:39:49] *** Gen'd [prefix [prefix [top] [bot]] [meet [bot] [bot]]] with simplicity 1681/31636800
;[12:39:54] *** Gen'd [prefix [meet [top] [top]] [meet [top] [top]]] with simplicity 1681/31636800
;[12:39:54] While assessing the domain definition [prefix [meet [top] [top]] [meet [top] [top]]]
;[12:40:03]  !!! [prefix [meet [top] [top]] [meet [top] [top]]] never seems to terminate.
;[12:40:03] >>> Selector produced [prefix [meet [top] [top]] [meet [top] [top]]] with simplicity 0
;[12:40:04] *** Gen'd [prefix [meet [top] [top]] [prefix [top] [bot]]] with simplicity 1681/31636800
;[12:40:04] While assessing the domain definition [prefix [meet [top] [top]] [prefix [top] [bot]]]
;[12:40:14]  !!! [prefix [meet [top] [top]] [prefix [top] [bot]]] appears to be constantly ()
```

3.6 Inventing Numbers

After generating a space of definitions in both its initial vocabulary and a rebiased version of this vocabulary, Cyrano examines the definitions in both these spaces for potential reifications. Like rebiasing, this examiniation begins with a clique analysis; the purpose of this rebiasing, however, is not to identify definitions of particular interest but to avoid consider the reifications of identical operations with different definitions.

```
;[12:40:16] *** Type analysis identifies three initial cliques
;[12:40:17] ***  Size From       To                                                    For instance
;[12:40:17] ***  ---- ----       --                                                    ------------
;[12:40:18] ***  91   #16Lists #15@FCN #16Lists --> #16Lists                          [meet [top] [terminator [prefix [t
;[12:40:18] ***  11   #16Lists #64@FCN #16Lists --> #91Booleans                       [postfix [top] [identical? [bot] [
;[12:40:18] ***  4    #16Lists #50@FCN #16Lists --> #15@FCN #16Lists --> #16Lists [prefix [tie] [tie]]
```

> *There are only three initial cliques because Cyrano only does clique analysis on the operations which might make useful reifications; i.e. operations which return procedures as opposed to domain objects.*

```
;[12:40:19] *** which I will try and further distinguish (in fifteen passes)
;[12:41:25] === On first pass, generated 45 new cliques and
;[12:41:25] ===                removed 38 singletons, leaving 10 cliques altogether
;[12:42:02] === On second pass, generated 11 new cliques and
;[12:42:02] ===                removed 12 singletons, leaving 9 cliques altogether
;[12:42:31] === On third pass, generated 8 new cliques and
;[12:42:31] ===                removed 7 singletons, leaving 10 cliques altogether
;[12:43:02] === On fourth pass, generated 6 new cliques and
;[12:43:02] ===                removed 6 singletons, leaving 10 cliques altogether
;[12:43:31] === On fifth pass, generated 2 new cliques and
;[12:43:31] ===                removed 2 singletons, leaving 10 cliques altogether..
;[12:44:47] === On eighth pass, generated 1 new clique and
;[12:44:47] ===                removed 2 singletons, leaving 9 cliques altogether.
;[12:45:34] === On tenth pass, generated 1 new clique and
;[12:45:34] ===                removed 0 singletons, leaving 10 cliques altogether.
;[12:46:41] === On twelfth pass, generated 1 new clique and
;[12:46:41] ===                removed 0 singletons, leaving 11 cliques altogether...
;[12:48:09] *** After fifteen passes, finished with eleven cliques
;[12:48:09] ***       with a mean of 3.545 (sd=3.355; n=11) members each.
;[12:48:14] ***  Size For instance                                          Which computes
;[12:48:14] ***  ---- ------------                                         --------------
;[12:48:15] ***  6    [postfix [top] [meet [top] [top]]]                   () <=== [Tie 16/3], ()
;[12:48:15] ***  5    [meet [prefix [top] [bot]] [postfix [bot] [bot]]]    () <=== [Tie 14/26], [Tie 8/24]
;[12:48:16] ***  5    [postfix [bot] [meet [bot] [bot]]]                   [Tie 6/3] <=== [Tie 19/4], [Tie 10/30]
;[12:48:16] ***  4    [identical? [bot] [top]]                             () <=== [Tie 9/6], [Tie 3/29]
;[12:48:16] ***  4    [meet [bot] [postfix [top] [top]]]                   () <=== (), ()
;[12:48:17] ***  4    [meet [bot] [terminator [prefix [top] [bot]]]]       () <=== [Tie 23/4], [Tie 14/16]
;[12:48:17] ***  3    [postfix [top] [meet [bot] [bot]]]                   () <=== [Tie 11/20], [Tie 21/7]
;[12:48:17] ***  2    [meet [bot] [prefix [bot] [bot]]]                    [Tie 6/9] <=== [Tie 10/17], [Tie 13/10]
;[12:48:18] ***  2    [meet [top] [postfix [top] [bot]]]                   () <=== [Tie 14/7], [Tie 16/5]
;[12:48:18] ***  2    [meet [top] [prefix [top] [top]]]                    [Tie 5/6] <=== [Tie 17/20], [Tie 7/24]
;[12:48:18] ***  2    [prefix [meet [bot] [bot]] [prefix [top] [bot]]]     () <=== [Tie 2/3], [Tie 30/2]
```

In consider the potential reification of an operation, Cyrano considers its convergence and divergence. The convergence of a representation is computed statistically in the following manner:

1. *Generate a random partial evaluation p of O and generate an example e of this partial evaluation at the domain level. (e.g. if O = Addition, generate 'add 3 to x' and the example '5 → 8')*
2. *Keep generating partial evaluations of O until you find one q which behaves identicall on the example e.*
3. *Try generating random examples of p and see how many are also examples of q.*
4. *Keep an average of this quantity for some number of different p.*

Potential reifications with low convergence are bad choices for reifications because it is unlikely that a quickly found canon (standard reification) is indeed correct or unique. In addition to computing the coherence of a potential reification, Cyrano computes a divergence based the average number of reifications that have to be generated before finding the match q. In some sense, the divergence measures how long it takes to find a match, while the coherence measures how good the match is.
Cyrano considers the reifications of all the effective (terminating) definitions which return domain definitions as results (e.g. take more than one domain input and yield a domain output). Potential reifications are classified as either 'useless' if their divergence is too high, 'marginal' if their divergence is acceptable but their coherence too low, and 'promising' if both have reasonable values.

```
;[12:53:43] @@@ There are fifty operations which might form useful reifications.
;[12:53:44] While considering the refications of fifty operations
;[12:54:15]  @@@ (Marginal)  [meet [bot] [postfix [bot] [bot]]]; Coherence=0.702; Divergence=0.842
;[12:54:38]  @@@ (Marginal)  [meet [top] [postfix [top] [top]]]; Coherence=0.670; Divergence=0.781
;[12:54:58]  @@@ (Useless)   [meet [bot] [bot]]; Coherence=0.681; Divergence=0.737
;[12:55:15]  @@@ (Marginal)  [meet [prefix [top] [bot]] [postfix [bot] [bot]]]; Coherence=1.0; Divergence=1.0
;[12:55:36]  @@@ (Marginal)  [meet [prefix [top] [bot]] [postfix [top] [top]]]; Coherence=1.0; Divergence=1.0
;[12:56:02]  @@@ (Marginal)  [meet [bot] [prefix [top] [bot]]]; Coherence=1.0; Divergence=1.0
;[12:56:42]  @@@ (Marginal)  [postfix [top] [meet [bot] [bot]]]; Coherence=1.0; Divergence=1.0
;[12:57:27   @@@ (Marginal)  [prefix [meet [bot] [bot]] [prefix [top] [bot]]]; Coherence=1.0; Divergence=1.0
;[12:57:55]  @@@ (Marginal)  [meet [bot] [postfix [top] [bot]]]; Coherence=1.0; Divergence=1.0
;[12:58:16]  @@@ (Marginal)  [meet [top] [top]]; Coherence=0.678; Divergence=0.842
;[12:58:35]  @@@ (Marginal)  [identical? [bot] [top]]; Coherence=1.0; Divergence=1.0
;[12:59:31]  @@@ (Promising) [postfix [top] [limit [top] [bot]]]; Coherence=1.0; Divergence=0.650
;[13:00:32]  @@@ (Promising) [postfix [top] [limit [top] [top]]]; Coherence=1.0; Divergence=0.475
;[13:01:17]  @@@ (Promising) [postfix [top] [limit [bot] [bot]]]; Coherence=1.0; Divergence=0.885
;[13:01:53]  @@@ (Marginal)  [postfix [top] [limit [bot] [top]]]; Coherence=1.0; Divergence=0.972
;[13:02:03]  @@@ (Useless)   [mirror-2 [limit [top] [bot]]]; Coherence=0.0; Divergence=0.0
;[13:02:15]  @@@ (Useless)   [mirror-2 [limit [top] [top]]]; Coherence=0.0; Divergence=0.0
;[13:02:26]  @@@ (Useless)   [mirror-2 [limit [bot] [bot]]]; Coherence=0.0; Divergence=0.0
;[13:02:37]  @@@ (Useless)   [mirror-2 [limit [bot] [top]]]; Coherence=0.0; Divergence=0.0
;[13:03:21]  @@@ (Marginal)  [check [postfix [bot] [tie]]]; Coherence=1.0; Divergence=1.0
;[13:04:29]  @@@ (Marginal)  [check [limit [top] [bot]]]; Coherence=0.797; Divergence=0.961
;[13:05:19]  @@@ (Promising) [check [limit [top] [top]]]; Coherence=0.780; Divergence=0.895
```

```
;[13:06:04]  @@@ (Marginal)  [check [limit [bot] [bot]]]; Coherence=0.665; Divergence=1.0
;[13:07:01]  @@@ (Marginal)  [check [limit [bot] [top]]]; Coherence=0.768; Divergence=0.915
;[13:07:16]  @@@ (Useless)   [limit [bot] [coalesce [tie]]]; Coherence=0.0; Divergence=0.0
;[13:07:30]  @@@ (Useless)   [limit [top] [coalesce [tie]]]; Coherence=0.0; Divergence=0.0
;[13:07:37]  @@@ (Promising) [limit [bot] [top]]; Coherence=1.0; Divergence=0.350
;[13:07:47]  @@@ (Promising) [limit [bot] [bot]]; Coherence=1.0; Divergence=0.533
;[13:07:55]  @@@ (Promising) [limit [top] [top]]; Coherence=1.0; Divergence=0.600
;[13:08:04]  @@@ (Promising) [limit [top] [bot]]; Coherence=1.0; Divergence=0.550
;[13:08:21]  @@@ (Promising) [meet [postfix [top] [top]] [prefix [top] [bot]]]; Coherence=1.0; Divergence=0.830
;[13:08:35]  @@@ (Marginal)  [meet [postfix [top] [top]] [postfix [top] [top]]]; Coherence=0.589; Divergence=0.914
;[13:08:53]  @@@ (Marginal)  [meet [postfix [bot] [bot]] [prefix [top] [bot]]]; Coherence=1.0; Divergence=0.925
;[13:09:07]  @@@ (Promising) [meet [postfix [bot] [bot]] [postfix [bot] [bot]]]; Coherence=0.850; Divergence=0.712
;[13:09:16]  @@@ (Marginal)  [meet [postfix [bot] [bot]] [postfix [top] [top]]]; Coherence=1.0; Divergence=0.920
;[13:09:41]  @@@ (Marginal)  [meet [prefix [top] [bot]] [prefix [top] [bot]]]; Coherence=0.949; Divergence=1.0
;[13:11:14]  @@@ (Marginal)  [prefix [check [bot]] [meet [top] [top]]]; Coherence=1.0; Divergence=0.994
;[13:17:28]  @@@ (Marginal)  [prefix [check [bot]] [me  [bot] [bot]]]; Coherence=0.952; Divergence=0.990
;[13:18:43]  @@@ (Promising) [prefix [meet [bot] [bot]] [postfix [bot] [bot]]]; Coherence=0.788; Divergence=0.730
;[13:21:38]  @@@ (Promising) [prefix [postfix [top] [top]] [meet [top] [top]]]; Coherence=0.793; Divergence=0.800
;[13:23:25]  @@@ (Marginal)  [prefix [postfix [top] [top]] [meet [bot] [bot]]]; Coherence=0.705; Divergence=0.815
;[13:27:22]  @@@ (Promising) [prefix [postfix [bot] [bot]] [meet [top] [top]]]; Coherence=0.814; Divergence=0.640
;[13:29:21]  @@@ (Marginal)  [prefix [postfix [bot] [bot]] [meet [bot] [bot]]]; Coherence=0.753; Divergence=0.957
;[13:32:09]  @@@ (Marginal)  [prefix [prefix [top] [bot]] [meet [top] [top]]]; Coherence=0.682; Divergence=0.860
;[13:34:41]  @@@ (Marginal)  [prefix [prefix [top] [bot]] [meet [bot] [bot]]]; Coherence=0.722; Divergence=0.850
;[13:35:31]  @@@ (Marginal)  [meet [top] [terminator [postfix [top] [top]]]]; Coherence=1.0; Divergence=0.980
;[13:36:16]  @@@ (Marginal)  [meet [bot] [terminator [postfix [bot] [bot]]]]; Coherence=1.0; Divergence=1.0
;[13:38:48]  @@@ (Marginal)  [prefix [top] [meet [top] [top]]]; Coherence=0.629; Divergence=0.841
;[13:41:52]  @@@ (Marginal)  [prefix [top] [meet [bot] [bot]]]; Coherence=0.554; Divergence=0.914
;[13:45:18]  @@@ (Marginal)  [prefix [bot] [meet [top] [top]]]; Coherence=0.584; Divergence=0.886
;[13:45:18] @@@ Found thirteen (first pass) candidates for reification:
;[13:45:20] @@@ Reification Of                                    Coherence Divergence
;[13:45:21] @@@ --------------                                    --------- ----------
;[13:45:21] @@@ [prefix [postfix [bot] [bot]] [meet [top] [top]]] 0.814     0.640
;[13:45:21] @@@ [prefix [postfix [top] [top]] [meet [top] [top]]] 0.793     0.800
;[13:45:22] @@@ [prefix [meet [bot] [bot]] [postfix [bot] [bot]]] 0.788     0.730
;[13:45:22] @@@ [meet [postfix [bot] [bot]] [postfix [bot] [bot]]] 0.850     0.712
;[13:45:22] @@@ [meet [postfix [top] [top]] [prefix [top] [bot]]] 1.0       0.830
;[13:45:22] @@@ [limit [top] [bot]]                               1.0       0.550
;[13:45:23] @@@ [limit [top] [top]]                               1.0       0.600
;[13:45:23] @@@ [limit [bot] [bot]]                               1.0       0.533
;[13:45:23] @@@ [limit [bot] [top]]                               1.0       0.350
;[13:45:24] @@@ [check [limit [top] [top]]]                       0.780     0.895
;[13:45:24] @@@ [postfix [top] [limit [bot] [bot]]]               1.0       0.885
;[13:45:24] @@@ [postfix [top] [limit [top] [top]]]               1.0       0.475
;[13:45:24] @@@ [postfix [top] [limit [top] [bot]]]               1.0       0.650
```

Once a first round of considerations is over, Cyrano applies its criteria again to all the definitions judged promising. Occasionally, the vagaries of Cyrano's example generation mechanisms will cause misclassifications, such as (Note 15/80) below. The operation `[limit [top] [top]]` *would make a fine reification, but the assessment mechanism is never able to compute useful coherence information for it, leaving it with an inaccurrate and pessimal assessment.*

```
;[13:45:24] While considering the reifications of thirteen operations
;[13:49:37]  @@@ (Marginal)  [prefix [postfix [bot] [bot]] [meet [top] [top]]]; Coherence=0.593; Divergence=0.920
;[13:52:59]  @@@ (Promising) [prefix [postfix [top] [top]] [meet [top] [top]]]; Coherence=0.822; Divergence=0.777
;[13:54:25]  @@@ (Marginal)  [prefix [meet [bot] [bot]] [postfix [bot] [bot]]]; Coherence=0.630; Divergence=0.826
;[13:54:33]  @@@ (Marginal)  [meet [postfix [bot] [bot]] [postfix [bot] [bot]]]; Coherence=0.600; Divergence=0.866
;[13:54:52]  @@@ (Marginal)  [meet [postfix [top] [top]] [prefix [top] [bot]]]; Coherence=1.0; Divergence=0.937
;[13:55:01]  @@@ (Promising) [limit [top] [bot]]; Coherence=1.0; Divergence=0.366
```

```
15 ⟹    ;[13:55:08]  @@@ (Useless)   [limit [top] [top]]; Coherence=0.0; Divergence=0.01
        ;[13:55:15]  @@@ (Promising) [limit [bot] [bot]]; Coherence=1.0; Divergence=0.500
        ;[13:55:24]  @@@ (Promising) [limit [bot] [top]]; Coherence=1.0; Divergence=0.800
        ;[13:56:24]  @@@ (Marginal)  [check [limit [top] [top]]]; Coherence=0.765; Divergence=0.925
        ;[13:57:10]  @@@ (Marginal)  [postfix [top] [limit [bot] [bot]]]; Coherence=1.0; Divergence=0.915
        ;[13:58:34]  @@@ (Promising) [postfix [top] [limit [top] [top]]]; Coherence=1.0; Divergence=0.825
        ;[14:00:09]  @@@ (Promising) [postfix [top] [limit [top] [bot]]]; Coherence=1.0; Divergence=0.400
        ;[14:00:09] @@@ Found six (second pass) candidates for reification:
        ;[14:00:10] @@@ Reification Of                                    Coherence Divergence
        ;[14:00:11] @@@ --------------                                    --------- ----------
        ;[14:00:11] @@@ [postfix [top] [limit [top] [bot]]]               1.0       0.400
        ;[14:00:11] @@@ [postfix [top] [limit [top] [top]]]               1.0       0.825
        ;[14:00:12] @@@ [limit [bot] [top]]                               1.0       0.800
        ;[14:00:12] @@@ [limit [bot] [bot]]                               1.0       0.500
        ;[14:00:12] @@@ [limit [top] [bot]]                               1.0       0.366
        ;[14:00:13] @@@ [prefix [postfix [top] [top]] [meet [top] [top]]] 0.822     0.777
```

When Cyrano has selected a set of reasonable operations to reify, it considers whether any of them might in fact be identical. It does this by computing the same coherence it computed before, but this time between reifications p of one operation and potential matching reifications o of another reification.

```
;[14:00:13] While looking for identities between six reifications
;[14:02:09]  @@@ Coherence of 0 and divergence of 0 between reifications of
;[14:02:09]  @@@       [postfix [top] [limit [top] [bot]]] and
;[14:02:09]  @@@       [postfix [top] [limit [top] [top]]]
;[14:02:54]  @@@ Coherence of 0 and divergence of 0 between reifications of
;[14:02:54]  @@@       [postfix [top] [limit [top] [bot]]] and
;[14:02:55]  @@@       [limit [bot] [top]]
;[14:03:29]  @@@ Coherence of 1 and divergence of 11/20 between reifications of
;[14:03:29]  @@@       [postfix [top] [limit [top] [bot]]] and
;[14:03:30]  @@@       [limit [bot] [bot]]
```

Cyrano uncovers a match here between operations which both remove n bottom elements from their second input; the number n for the first operation is simply one minus the height of the first input (which produced the partial evaluation); the number n for the second operation is simply the depth of the first input.

```
;[14:03:51]  @@@ Coherence of 0 and divergence of 0 between reifications of
;[14:03:51]  @@@       [postfix [top] [limit [top] [bot]]] and
;[14:03:51]  @@@       [limit [top] [bot]]
;[14:08:05]  @@@ Coherence of 0 and divergence of 0 between reifications of
;[14:08:05]  @@@       [postfix [top] [limit [top] [bot]]] and
;[14:08:05]  @@@       [prefix [postfix [top] [top]] [meet [top] [top]]]
;[14:11:25]  @@@ Coherence of 0 and divergence of 0 between reifications of
;[14:11:26]  @@@       [prefix [postfix [top] [top]] [meet [top] [top]]] and
;[14:11:26]  @@@       [limit [top] [bot]]
;[14:13:11]  @@@ Coherence of 8833/19760 and divergence of 5/8 between reifications of
;[14:13:11]  @@@       [prefix [postfix [top] [top]] [meet [top] [top]]] and
;[14:13:11]  @@@       [limit [bot] [top]]
;[14:16:03]  @@@ Coherence of 29/140 and divergence of 3/4 between reifications of
;[14:16:03]  @@@       [prefix [postfix [top] [top]] [meet [top] [top]]] and
;[14:16:03]  @@@       [postfix [top] [limit [top] [top]]]
;[14:16:55]  @@@ Coherence of 1 and divergence of 1/2 between reifications of
;[14:16:55]  @@@       [postfix [top] [limit [top] [top]]] and
;[14:16:55]  @@@       [limit [bot] [top]]
;[14:17:50]  @@@ Coherence of 0 and divergence of 0 between reifications of
```

```
;[14:17:50]  @@@       [postfix [top] [limit [top] [top]]] and
;[14:17:51]  @@@       [limit [top] [bot]]
;[14:18:04]  *** Defining new representation
;[14:18:04]  ***  [Canonicalizer [One of [Curries of [postfix [top] [limit [top] [bot]]]]
;[14:18:04]  ***                          [Curries of [limit [bot] [bot]]]]]
;[14:18:04]  *** from the reficiations of
;[14:18:04]  ***     [postfix [top] [limit [top] [bot]]]
;[14:18:05]  ***     [limit [bot] [bot]]
;[14:18:11]  *** Defining new representation
;[14:18:11]  ***   [Canonicalizer [Curries of [prefix [postfix [top] [top]] [meet [top] [top]]]]]
;[14:18:11]  ***  from the reficiations of
;[14:18:11]  ***     [prefix [postfix [top] [top]] [meet [top] [top]]]
;[14:18:16]  *** Defining new representation
;[14:18:16]  ***  [Canonicalizer [One of [Curries of [postfix [top] [limit [top] [top]]]]
;[14:18:16]  ***                          [Curries of [limit [bot] [top]]]]]
;[14:18:16]  ***  from the reficiations of
;[14:18:17]  ***     [postfix [top] [limit [top] [top]]]
;[14:18:17]  ***     [limit [bot] [top]]
;[14:26:09]  *** Defining new representation [Canonicalizer [Curries of [limit [top] [bot]]]]
;[14:26:09]  ***  from the reficiations of
;[14:26:11]  ***     [limit [top] [bot]]
;;; There are four potential abstractions to explore
          #549[Canonicalizer [One of [Curries of [postfix [top] [limit [top] [bot]]]]
                                     [Curries of [limit [bot] [bot]]]]]
          #550[Canonicalizer [Curries of [prefix [postfix [top] [top]] [meet [top] [top]]]]]
          #553[Canonicalizer [One of [Curries of [postfix [top] [limit [top] [top]]]]
                                                 [Curries of [limit [bot] [top]]]]]
          #554[Canonicalizer [Curries of [limit [top] [bot]]]]
;;;  at random I choose #549[Canonicalizer [One of [Curries of [postfix [top] [limit [top] [bot]]]]
;;;                                                [Curries of [limit [bot] [bot]]]]]
;;; Would you like to rename this abstraction? yes
;;; What would you like to name it? Depth
```

> *Cyrano chooses a reification at random and generates a new vocabulary around it. This generation begins by generating instances of the canons of the reification. Then Cyrano looks at each definition in the initial and rebiased pools from which the reification emerged and looks for both potential domain objects and operations. In the transcript, I have given the canons the corresponding cardinal names 'one', 'two', etc to clarify the presentation. Note that neither this renaming nor the renaming of the abstraction above ('Depth') cause any change in the courses of action taken by Cyrano.*

```
;[15:27:07] While generating a new vocabulary around [Depth]
;[15:27:08]  ^^^ Generating instances of the new domain.............................
;[15:29:57]  ^^^ Generated thirty-three examples of #555Depths
```

> *Cyrano now looks through the combined pools for generated definitions which correspond to particular canons; it finds four, corresponding to the simplest compositions and the first four counting numbers.*

```
;[15:29:57]  While abstracting objects into the domain specified by [Depth]
;[15:30:07]   ^^^ Generated new object #558Canon Four from [prefix [postfix [bot] [bot]] [postfix [bot] [bot]]]
;[15:30:09]   ^^^ Generated new object #559Canon Three from [postfix [bot] [prefix [bot] [bot]]]
;[15:30:12]   ^^^ Generated new object #561Canon One from [bot]
;[15:30:12]   ^^^ Generated new object #563Canon Two from [prefix [bot] [bot]]
```

Once Cyrano has generated both a space of examples and isolated a few examples to focus on, it looks for operations by examining the fragmentary definitions and seeing if their domain (what they can combine with) and their range (what such combinations produce) might possible correspond to the canons of the new domain. If they do, Cyrano constructs an 'abstracted' version of the operation which takes a canon as input, applies the fragmentary combiner to its representative procedure and then returns a canon matching the produced definition. If there is no such canon, the value of the abstracted operation is undefined. These abstracted operations are further pruned by the same criteria for triage: are they never defined, are they constant, or are they simply the identity. The abstracted operations which pass this criteria are then run through clique analysis to pick out definitions which happen to be the same in the new domain.

```
;[15:27:07] While generating a new vocabulary around [Depth]
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:30:23]   While assessing [ABSTRACT [Depth] [postfix [coalesce [mirror-2 [tie]]]]]
;[15:33:01]    !!! [ABSTRACT [Depth] [postfix [coalesce [mirror-2 [tie]]]]] never seems to be defined.
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:33:02]   ~~~ Rejecting [ABSTRACT [Depth] [postfix [coalesce [mirror-2 [tie]]]]].
```

Cyrano rejects this abstraction, which would have been equivalent to predecessor, for efficency reasons; it is never able to generate any examples of it to assess.

```
;[15:33:04]   While assessing [ABSTRACT [Depth] [prefix [coalesce [mirror-2 [tie]]]]]
;[15:34:25]    !!! [ABSTRACT [Depth] [prefix [coalesce [mirror-2 [tie]]]]] never seems to be defined.
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:34:25]   ~~~ Rejecting [ABSTRACT [Depth] [prefix [coalesce [mirror-2 [tie]]]]].
;[15:34:28]   While assessing [ABSTRACT [Depth] [prefix [postfix [bot] [top]]]]
;[15:35:22]    !!! [ABSTRACT [Depth] [prefix [postfix [bot] [top]]]] never seems to be defined.
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:35:22]   ~~~ Rejecting [ABSTRACT [Depth] [prefix [postfix [bot] [top]]]].
;[15:35:26]   While assessing [ABSTRACT [Depth] [prefix [prefix [bot] [top]]]]
;[15:36:19]    !!! [ABSTRACT [Depth] [prefix [prefix [bot] [top]]]] never seems to be defined.
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:36:20]   ~~~ Rejecting [ABSTRACT [Depth] [prefix [prefix [bot] [top]]]].
;[15:36:28]   While assessing [ABSTRACT [Depth] [postfix [postfix [bot] [top]]]]
;[15:37:31]    !!! [ABSTRACT [Depth] [postfix [postfix [bot] [top]]]] never seems to be defined.
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:37:31]   ~~~ Rejecting [ABSTRACT [Depth] [postfix [postfix [bot] [top]]]].
;[15:37:33]   While assessing [ABSTRACT [Depth] [postfix [prefix [bot] [top]]]]
;[15:38:16]    !!! [ABSTRACT [Depth] [postfix [prefix [bot] [top]]]] never seems to be defined.
```

None of the above definitions make useful abstractions because they compute operations which are orthogonal to the distinctions captured by the 'Depth' reification.

```
;[15:30:21]  While abstracting operations into the domain specified by #549[Depth]
;[15:38:16]   ~~~ Rejecting [ABSTRACT [Depth] [postfix [prefix [bot] [top]]]].
;[15:38:28]   ~~~ Generated [ABSTRACT [Depth] [postfix [prefix [bot] [bot]]]] from [postfix [prefix [bot] [bot]]]
;[15:38:30]   ~~~ with domain #555Depths
;[15:38:30]   ~~~ and  range  #555Depths
```

Composing a canon with two `bot` *operations produces a canon which simple removes two more bottom elements. This is equivalent to the operation 'add 2' in the new domain.*

```
;[15:38:33]  While assessing [ABSTRACT [Depth] [postfix [prefix [top] [top]]]]
;[15:39:24]   !!! [ABSTRACT [Depth] [postfix [prefix [top] [top]]]] never seems to be defined.
```

Since `top` *operations are orthogonal to the depth reification, this i never defined.*

```
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:39:25]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [prefix [top] [top]]]].
;[15:39:40]  ~~~ Generated [ABSTRACT [Depth] [postfix [coalesce [tie]]]] from [postfix [coalesce [tie]]]
;[15:39:41]  ~~~ with domain #555Depths
;[15:39:41]  ~~~ and  range  #555Depths
```

In the natural numbers framework, this abstraction corresponds to predecessor. It ties an object to itself before passing it to another operation; in the case of the depth reification, this makes a tie structure one level deeper before actually removing anything from it. Note that there is no canon corresponding to the application of this operation to the identity; the identity operation corresponds to zero in this space.

```
;[15:39:46]  ~~~ Generated [ABSTRACT [Depth] [prefix [prefix [bot] [bot]]]] from [prefix [prefix [bot] <bot>]]1
;[15:39:47]  ~~~ with domain #555Depths
;[15:39:47]  ~~~ and  range  #555Depths
;[15:39:49]  While assessing [ABSTRACT [Depth] [prefix [prefix [top] [top]]]]
;[15:40:57]   !!! [ABSTRACT [Depth] [prefix [prefix [top] [top]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:40:58]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [prefix [top] [top]]]].
;[15:41:00]  While assessing [ABSTRACT [Depth] [prefix [coalesce [tie]]]]
;[15:42:13]   !!! [ABSTRACT [Depth] [prefix [coalesce [tie]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:42:13]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [coalesce [tie]]]].
```

Note that while `[postfix [coalesce [tie]]]` *abstraction has a definition in the space of depth canons, the* `[prefix [coalesce [tie]]]` *abstraction has no corresponding definition, since no sequence of* `bot` *operations will ever create the new tie structure which the prefixed call to* `[coalesce [tie]]` *does.*

```
;[15:42:15]  While assessing [ABSTRACT [Depth] [prefix [terminator [prefix [top] [bot]]]]]
;[15:42:59]   !!! [ABSTRACT [Depth] [prefix [terminator [prefix [top] [bot]]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:42:59]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [terminator [prefix [top] [bot]]]]].
;[15:43:00]  While assessing [ABSTRACT [Depth] [prefix [postfix [top] [bot]]]]
;[15:44:06]   !!! [ABSTRACT [Depth] [prefix [postfix [top] [bot]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:44:06]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [postfix [top] [bot]]]].
;[15:44:08]  While assessing [ABSTRACT [Depth] [prefix [terminator [postfix [bot] [bot]]]]]
;[15:45:14]   !!! [ABSTRACT [Depth] [prefix [terminator [postfix [bot] [bot]]]]] appears to be constantly #590Canon Fo
;[15:45:14]   !!! [ABSTRACT [Depth] [prefix [terminator [postfix [bot] [bot]]]]] always returns its first input
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:45:15]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [terminator [postfix [bot] [bot]]]]].
```

Again, Cyrano's example generation mechanisms cause problems; the??? canon 'forty-two' is the highest number Cyrano has seen, so it takes it as the terminator of plus2.

```
;[15:45:17]  While assessing [ABSTRACT [Depth] [prefix [terminator [postfix [top] [top]]]]]
;[15:46:15]   !!! [ABSTRACT [Depth] [prefix [terminator [postfix [top] [top]]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:46:15]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [terminator [postfix [top] [top]]]]].
;[15:46:16]  While assessing [ABSTRACT [Depth] [postfix [terminator [prefix [top] [bot]]]]]
;[15:47:13]   !!! [ABSTRACT [Depth] [postfix [terminator [prefix [top] [bot]]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:47:14]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [terminator [prefix [top] [bot]]]]].
;[15:47:15]  While assessing [ABSTRACT [Depth] [postfix [postfix [top] [bot]]]]
;[15:48:02]   !!! [ABSTRACT [Depth] [postfix [postfix [top] [bot]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:48:03]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [postfix [top] [bot]]]].
;[15:48:04]  While assessing [ABSTRACT [Depth] [postfix [terminator [postfix [bot] [bot]]]]]
;[15:49:14]   !!! [ABSTRACT [Depth] [postfix [terminator [postfix [bot] [bot]]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:49:15]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [terminator [postfix [bot] [bot]]]]].
;[15:49:16]  While assessing [ABSTRACT [Depth] [postfix [terminator [postfix [top] [top]]]]]
;[15:50:23]   !!! [ABSTRACT [Depth] [postfix [terminator [postfix [top] [top]]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:50:24]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [terminator [postfix [top] [top]]]]].
;[15:50:25]  While assessing [ABSTRACT [Depth] [postfix [prefix [top] [bot]]]]
;[15:51:26]   !!! [ABSTRACT [Depth] [postfix [prefix [top] [bot]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:51:27]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [prefix [top] [bot]]]].
;[15:51:33]  ~~~ Generated [ABSTRACT [Depth] [postfix [postfix [bot] [bot]]]] from [postfix [postfix [bot] [bot]]]
;[15:51:33]  ~~~ with domain #555Depths
;[15:51:33]  ~~~ and  range  #555Depths
```

This is another version of the add2 *operation constructed above (Note 16/83); Cyrano will notice their identity when it does clique analysis below. It also defines this yet again in (Note 17/84)*

```
;[15:51:35]  While assessing [ABSTRACT [Depth] [postfix [postfix [top] [top]]]]
;[15:52:53]   !!! [ABSTRACT [Depth] [postfix [postfix [top] [top]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:52:54]  ~~~ Rejecting [ABSTRACT [Depth] [postfix [postfix [top] [top]]]].
;[15:52:57]  While assessing [ABSTRACT [Depth] [prefix [prefix [top] [bot]]]]
;[15:53:51]   !!! [ABSTRACT [Depth] [prefix [prefix [top] [bot]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:53:51]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [prefix [top] [bot]]]].
17 ⟹ ;[15:54:01]  ~~~ Generated [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]] from [prefix [postfix [bot] [bot]]]1
;[15:54:01]  ~~~ with domain #555Depths
;[15:54:02]  ~~~ and  range  #555Depths
;[15:54:03]  While assessing [ABSTRACT [Depth] [prefix [postfix [top] [top]]]]
;[15:55:09]   !!! [ABSTRACT [Depth] [prefix [postfix [top] [top]]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[15:55:10]  ~~~ Rejecting [ABSTRACT [Depth] [prefix [postfix [top] [top]]]].
;[15:55:19]  ~~~ Generated [ABSTRACT [Depth] [prefix [bot]]] from [prefix [bot]]
;[15:55:19]  ~~~ with domain #555Depths
;[15:55:19]  ~~~ and  range  #555Depths
```

This is the predecessor operation; it moves from a canon like 'remove 2' to a canon 'remove 3'. It is alternatively defined as [postfix [bot]] *(Note 18/85).*

```
;[15:55:21]  While assessing [ABSTRACT [Depth] [postfix [top]]]
;[16:04:02]   !!! [ABSTRACT [Depth] [postfix [top]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[16:04:10]  ^^^ Rejecting [ABSTRACT [Depth] [postfix [top]]].
;[16:04:37]  ^^^ Generated [ABSTRACT [Depth] [prefix]] from [prefix]
;[16:04:38]  ^^^ with domain #555Depths
;[16:04:38]  ^^^ and  range  #6180FCH #555Depths --> #555Depths
```

This is the operation 'add' in the domain of natural numbers because it takes two depth canons and composes their operation. The [postfix] *combiner produces the same abstracted definition due to the composability of depth canons (Note 19/85).*

```
;[16:04:39]  While assessing [ABSTRACT [Depth] [prefix [top]]]
;[16:05:45]   !!! [ABSTRACT [Depth] [prefix [top]]] never seems to be defined.
;[15:30:21] While abstracting operations into the domain specified by #549[Depth]
;[16:05:45]  ^^^ Rejecting [ABSTRACT [Depth] [prefix [top]]].
;[16:05:51]  ^^^ Generated [ABSTRACT [Depth] [postfix [bot]]] from [postfix [bot]]1          <== 18
;[16:05:51]  ^^^ with domain #555Depths
;[16:05:51]  ^^^ and  range  #555Depths
;[16:06:16]  ^^^ Generated [ABSTRACT [Depth] [postfix]] from [postfix]1                     <== 19
;[16:06:16]  ^^^ with domain #555Depths
;[16:06:16]  ^^^ and  range  #6180FCH #555Depths --> #555Depths
```

Once the set of operations in the new vocabulary has been generated, Cyrano tries to pare them down by isolating polygenic behaviors. In doing so, it identifies three: 'add 2', 'add 1', and 'add'.

```
;[16:06:16]  ^^^ Finding polygenic behaviors among nine operations
;[16:06:37]  === On first pass, generated 2 new cliques and removed 1 singleton, leaving 3 cliques altogether.....
;[16:09:24]  ^^^ Analyzed clique structure for new primitives
;[16:09:25]  ^^^  Size For instance
;[16:09:26]  ^^^  ---- ------------
;[16:09:26]  ^^^  4    [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]]
;[16:09:27]  ^^^  2    [ABSTRACT [Depth] [postfix [bot]]]
;[16:09:27]  ^^^  2    [ABSTRACT [Depth] [prefix]]
```

Cyrano finally produces a new vocabulary and assigns simplicity measures to definiiions according to their occurence; the resulting objects and operations are summarized below...

```
;[15:27:07] While generating a new vocabulary around [Depth]
;[16:09:27]  ^^^ Produced a new vocabulary over #555Depths
;[16:09:28]  ^^^ with four initial object and nine initial operations
;[16:09:28]  ^^^ The four objects:
;[16:09:28]  ^^^ #563Canon Two
;[16:09:28]  ^^^ #561Canon One
;[16:09:28]  ^^^ #559Canon Three
;[16:09:28]  ^^^ #558Canon Four
;[16:09:30]  ^^^  Operation
;[16:09:31]  ^^^  ---------
;[16:09:31]  ^^^  [ABSTRACT [Depth] [postfix [coalesce [tie]]]]
;[16:09:31]  ^^^  [ABSTRACT [Depth] [postfix]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [postfix]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [prefix [postfix [bot] [bot]]]]
;[16:09:32]  ^^^  [ABSTRACT [Depth] [postfix [bot]]]
;[16:09:33]  ^^^  [ABSTRACT [Depth] [postfix [bot]]]
```

3.7 Exploring Arithmetic

Unfortunately, operating in the reified vocabulary introduced in the previous chapter quickly runs into technical problems having to do with the overhead incurred when every operation involves many other operations in comparing generated procedures with known canons in the new representation.

In order to make the examples of this document clearer, I pick up the story after having given Cyrano versions of the operations it invented earlier which operate directly on numbers. Proceeding from this starting place, it explores arithmetic operations and new vocabularies spun off from those operations. In order to avoid the misunderstanding of Lenat — wherein the factorization routine he provided was more valuable than that which AM actually invented (see Section 5.2.2; Page 132) — I should tell the reader that the domain provided here is in fact more complete — in some ways — than that devised by Cyrano. In particular, since Cyrano's domain depends on finite structures from the example generator, its arithmetic is limited to operations whose magnitude is limited by those structures. However, the arithmetic vocabulary which I give Cyrano is more complete and does not have the holes of Cyrano's invented representation.

So Cyrano begins, as was defined above, with the natural numbers one through four, the operations of addition, successor, 'add 2', and predecessor. From these, in combination with the standard combiners introduced at the beginning of this chapter, it begins to explore the domain of arithmetic.

```
;[22:15:53] *** Gen'd 5 with simplicity 1/400
;[22:15:54] *** Gen'd 6 with simplicity 1/400
```

Applying basic operations to objects of special interest yields new objects of interest. In some sense, these numbers are interesting because they are the simple application of arithmetic primitives to small numbers (where the small numbers are interesting because they correspond to simple compositions of successor operations, as abstracted into the domain of numbers.

```
;[22:15:54] *** Gen'd [check [add2]] with simplicity 1/400
;[22:15:54] While assessing [check [add2]]
;[22:15:55]  !!! [check [add2]] appears to be constantly #T
;[22:15:55] >>> Selector produced [check [add2]] with simplicity 0
;[22:15:56] *** Gen'd [check [c+]] with simplicity 1/400
;[22:15:56] While assessing [check [c+]]
;[22:15:58]  !!! [check [c+]] appears to be constantly #T
;[22:15:58] >>> Selector produced [check [c+]] with simplicity 0
;[22:15:58] *** Gen'd [check [predecessor]] with simplicity 1/400
;[22:15:58] While assessing [check [predecessor]]
;[22:15:59]  !!! [check [predecessor]] appears to be constantly #T
;[22:15:59] >>> Selector produced [check [predecessor]] with simplicity 0
;[22:15:59] *** Gen'd [check [successor]] with simplicity 1/400
;[22:16:00] While assessing [check [successor]]
;[22:16:00]  !!! [check [successor]] appears to be constantly #T
;[22:16:00] >>> Selector produced [check [successor]] with simplicity 0
```

The above definitions are all always defined. When Cyrano proceeds with its invented vocabulary, this structure is not nearly so clear because sometimes there isn't a successor for some number defined as a canon yet.

```
;[22:16:04] *** Gen'd [c+ 3] with simplicity 1/400
;[22:16:04] *** Gen'd [c+ 1] with simplicity 1/400
;[22:16:05] *** Gen'd [c+ 4] with simplicity 1/400
;[22:16:05] *** Gen'd [add2] with simplicity 1/400
```

These are all curried versions of plus.

```
;[22:16:10] *** Gen'd 0 with simplicity 1/400
;[22:16:11] *** Gen'd [terminator [add2]] with simplicity 1/400
;[22:16:11] While assessing [terminator [add2]]
;[22:16:18]  !!! [terminator [add2]] never seems to terminate.
;[22:16:18] >>> Selector produced [terminator [add2]] with simplicity 0
;[22:16:19] *** Gen'd [terminator [predecessor]] with simplicity 1/400
;[22:16:19] While assessing [terminator [predecessor]]
;[22:16:20]  !!! [terminator [predecessor]] appears to be constantly 0
;[22:16:21] >>> Selector produced [terminator [predecessor]] with simplicity 0
;[22:16:21] *** Gen'd [terminator [successor]] with simplicity 1/400
;[22:16:21] While assessing [terminator [successor]]
;[22:16:28]  !!! [terminator [successor]] never seems to terminate.
;[22:16:28] >>> Selector produced [terminator [successor]] with simplicity 0
;[22:16:30] *** Gen'd [coalesce [c+]] with simplicity 1/400
;[22:16:36] *** Gen'd [mirror-2 [c+]] with simplicity 1/400
;[22:16:37] *** Gen'd [mirror-2 [mirror-2 [c+]]] with simplicity 1/8000
;[22:16:39] *** Gen'd [mirror-2 [c+] 3] with simplicity 1/8000
;[22:16:39] *** Gen'd [mirror-2 [c+] 1] with simplicity 1/8000
;[22:16:40] *** Gen'd [mirror-2 [c+] 4] with simplicity 1/8000
;[22:16:40] *** Gen'd [mirror-2 [c+] 2] with simplicity 1/8000
```

Since addition is associative, all of these will have identical behaviors to definitions constructed above.

```
;[22:16:43] *** Gen'd [limit [successor] [add2]] with simplicity 1/8000
;[22:16:43] While assessing [limit [successor] [add2]]
;[22:16:50]  !!! [limit [successor] [add2]] never seems to terminate.
;[22:16:50] >>> Selector produced [limit [successor] [add2]] with simplicity 0
;[22:16:51] *** Gen'd [limit [successor] [predecessor]] with simplicity 1/8000
;[22:16:51] While assessing [limit [successor] [predecessor]]
;[22:16:53]  !!! [limit [successor] [predecessor]] never seems to be defined.
;[22:16:53] >>> Selector produced [limit [successor] [predecessor]] with simplicity 0
;[22:16:53] *** Gen'd [limit [successor] [successor]] with simplicity 1/8000
;[22:16:53] While assessing [limit [successor] [successor]]
;[22:17:00]  !!! [limit [successor] [successor]] never seems to terminate.
;[22:17:00] >>> Selector produced [limit [successor] [successor]] with simplicity 0
```

The above definitions all fail because one cannot use successor to limit another operation.

```
;[22:17:01] *** Gen'd [limit [predecessor] [add2]] with simplicity 1/8000
```

This computes $f(x, y) = 2x + y$.

```
;[22:17:02] *** Gen'd [limit [predecessor] [predecessor]] with simplicity 1/8000
```

This is the subtraction operation; note that it is undefined if the first input is larger than the second.

```
;[22:17:02] *** Gen'd [limit [predecessor] [successor]] with simplicity 1/8000
```

This is a redefinition of addition in terms of predecessor and successor.

```
;[22:17:03] *** Gen'd [limit [add2] [add2]] with simplicity 1/8000
;[22:17:03] While assessing [limit [add2] [add2]]
;[22:17:10]  !!! [limit [add2] [add2]] never seems to terminate.
;[22:17:10] >>> Selector produced [limit [add2] [add2]] with simplicity 0
;[22:17:11] *** Gen'd [limit [add2] [predecessor]] with simplicity 1/8000
;[22:17:11] While assessing [limit [add2] [predecessor]]
;[22:17:13]  !!! [limit [add2] [predecessor]] never seems to be defined.
;[22:17:13] >>> Selector produced [limit [add2] [predecessor]] with simplicity 0
;[22:17:14] *** Gen'd [limit [add2] [successor]] with simplicity 1/8000
;[22:17:14] While assessing [limit [add2] [successor]]
;[22:17:21]  !!! [limit [add2] [successor]] never seems to terminate.
;[22:17:21] >>> Selector produced [limit [add2] [successor]] with simplicity 0
```

These all fail because add2 *can't be used to limit other operations either (since it is never undefined).*

```
;[22:17:24] *** Gen'd [identical? [successor] [add2]] with simplicity 1/8000
;[22:17:25] *** Gen'd [identical? [successor] [predecessor]] with simplicity 1/8000
;[22:17:25] While assessing [identical? [successor] [predecessor]]
;[22:17:26]  !!! [identical? [successor] [predecessor]] appears to be constantly ()
;[22:17:26] >>> Selector produced [identical? [successor] [predecessor]] with simplicity 0
;[22:17:26] *** Gen'd [identical? [successor] [successor]] with simplicity 1/8000
;[22:17:27] *** Gen'd [identical? [predecessor] [add2]] with simplicity 1/8000
```

This is a sampling error. The definition computes the relation $x = y + 2$ yet the generation mechanisms must not have found any solutions (by random search) when the definition was being assessed. The same sampling error shows up in the definitions below.

```
;[22:17:28] *** Gen'd [identical? [predecessor] [predecessor]] with simplicity 1/8000
;[22:17:28] *** Gen'd [identical? [predecessor] [successor]] with simplicity 1/8000
;[22:17:29] *** Gen'd [identical? [add2] [add2]] with simplicity 1/8000
;[22:17:29] While assessing [identical? [add2] [add2]]
;[22:17:30]  !!! [identical? [add2] [add2]] appears to be constantly ()
;[22:17:30] >>> Selector produced [identical? [add2] [add2]] with simplicity 0
;[22:17:31] *** Gen'd [identical? [add2] [predecessor]] with simplicity 1/8000
;[22:17:31] While assessing [identical? [add2] [predecessor]]
;[22:17:32]  !!! [identical? [add2] [predecessor]] appears to be constantly ()
;[22:17:32] >>> Selector produced [identical? [add2] [predecessor]] with simplicity 0
;[22:17:33] *** Gen'd [identical? [add2] [successor]] with simplicity 1/8000
;[22:17:33] While assessing [identical? [add2] [successor]]
;[22:17:33]  !!! [identical? [add2] [successor]] appears to be constantly ()
;[22:17:34] >>> Selector produced [identical? [add2] [successor]] with simplicity 0
;[22:17:35] *** Gen'd 7 with simplicity 1/8000
;[22:17:35] *** Gen'd [coalesce [mirror-2 [c+]]] with simplicity 1/8000
;[22:17:38] *** Gen'd 8 with simplicity 1/8000
;[22:17:41] *** Gen'd [prefix [successor] [add2]] with simplicity 1/8000
;[22:17:42] *** Gen'd [prefix [successor] [c+]] with simplicity 1/8000
;[22:17:43] *** Gen'd [prefix [successor] [predecessor]] with simplicity 1/8000
;[22:17:43] While assessing [prefix [successor] [predecessor]]
;[22:17:44]  !!! [prefix [successor] [predecessor]] always returns its first input
;[22:17:44] >>> Selector produced [prefix [successor] [predecessor]] with simplicity 0
```

Here Cyrano 'notes' that predecessor and successor are inverses of one and other.

```
;[22:17:45] *** Gen'd [prefix [successor] [successor]] with simplicity 1/8000
;[22:17:45] *** Gen'd [prefix [predecessor] [add2]] with simplicity 1/8000
;[22:17:46] *** Gen'd [prefix [predecessor] [c+]] with simplicity 1/8000
;[22:17:47] *** Gen'd [prefix [predecessor] [predecessor]] with simplicity 1/8000
;[22:17:48] *** Gen'd [prefix [predecessor] [successor]] with simplicity 1/8000
;[22:17:48] While assessing [prefix [predecessor] [successor]]
;[22:17:49]  !!! [prefix [predecessor] [successor]] always returns its first input
;[22:17:49] >>> Selector produced [prefix [predecessor] [successor]] with simplicity 0
;[22:17:49] *** Gen'd [prefix [c+] [add2]] with simplicity 1/8000
;[22:17:51] *** Gen'd [prefix [c+] [c+]] with simplicity 1/8000
;[22:17:53] *** Gen'd [prefix [c+] [predecessor]] with simplicity 1/8000
;[22:17:54] *** Gen'd [prefix [c+] [successor]] with simplicity 1/8000
;[22:17:54] *** Gen'd [prefix [add2] [add2]] with simplicity 1/8000
;[22:17:55] *** Gen'd [prefix [add2] [c+]] with simplicity 1/8000
;[22:17:56] *** Gen'd [prefix [add2] [predecessor]] with simplicity 1/8000
;[22:17:57] *** Gen'd [prefix [add2] [successor]] with simplicity 1/8000
```

The above operations all increment the inputs to addition in one way or another.

```
;[22:17:58] *** Gen'd [terminator [coalesce [c+]]] with simplicity 1/8000
;[22:17:58] While assessing [terminator [coalesce [c+]]]
;[22:18:05]  !!! [terminator [coalesce [c+]]] never seems to terminate.
;[22:18:05] >>> Selector produced [terminator [coalesce [c+]]] with simplicity 0
;[22:18:06] *** Gen'd [terminator [add2]] with simplicity 1/8000
;[22:18:06] While assessing [terminator [add2]]
;[22:18:13]  !!! [terminator [add2]] never seems to terminate.
;[22:18:13] >>> Selector produced [terminator [add2]] with simplicity 0
;[22:18:13] *** Gen'd [terminator [c+ 4]] with simplicity 1/8000
;[22:18:14] While assessing [terminator [c+ 4]]
;[22:18:21]  !!! [terminator [c+ 4]] never seems to terminate.
;[22:18:21] >>> Selector produced [terminator [c+ 4]] with simplicity 0
;[22:18:21] *** Gen'd [terminator [c+ 1]] with simplicity 1/8000
;[22:18:21] While assessing [terminator [c+ 1]]
;[22:18:28]  !!! [terminator [c+ 1]] never seems to terminate.
;[22:18:29] >>> Selector produced [terminator [c+ 1]] with simplicity 0
;[22:18:29] *** Gen'd [terminator [c+ 3]] with simplicity 1/8000
;[22:18:29] While assessing [terminator [c+ 3]]
;[22:18:36]  !!! [terminator [c+ 3]] never seems to terminate.
;[22:18:36] >>> Selector produced [terminator [c+ 3]] with simplicity 0
```

None of the 'add n' operations can form reasonable terminators.

```
;[22:18:37] ??? The combination of [predecessor] with 0,
;[22:18:38] ??? which looked legal, turns out to be undefined.
;[22:18:43] *** Gen'd [meet [successor] [add2]] with simplicity 1/8000
```

This operation is undefined if its first input is larger than its second and of a different parity; otherwise, it computes the maximum of its arguments.

```
;[22:18:44] *** Gen'd [meet [successor] [predecessor]] with simplicity 1/8000
```

When the first input is less than or equal to the second, this computes the arithmetic mean of its two inputs by marching along the number line in opposite directions.

```
;[22:18:45] *** Gen'd [meet [successor] [successor]] with simplicity 1/8000
```

This computes the maximum of two numbers.

```
;[22:18:46] *** Gen'd [meet [predecessor] [add2]] with simplicity 1/8000
```

While undefined if its first input is smaller than its second, otherwise this computes the mean of its first input with twice its second input.

```
;[22:18:47] *** Gen'd [meet [predecessor] [predecessor]] with simplicity 1/8000
```

This computes the minimum of two numbers.

```
;[22:18:48] *** Gen'd [meet [predecessor] [successor]] with simplicity 1/8000
```

When the second input is less than or equal to the first, this computes the arithmetic mean of its two inputs by marching along the number line in opposite directions.

```
;[22:18:49] *** Gen'd [meet [add2] [add2]] with simplicity 1/8000
```

This computes the maximum of two numbers if they have the same parity.

```
;[22:18:52] *** Gen'd [meet [add2] [predecessor]] with simplicity 1/8000
;[22:18:58] *** Gen'd [meet [add2] [successor]] with simplicity 1/8000
;[22:18:59] *** Gen'd [yields 2 [add2]] with simplicity 1/8000
;[22:19:00] While assessing [yields 2 [add2]]
;[22:19:00]  !!! [yields 2 [add2]] appears to be constantly ()
;[22:19:00] >>> Selector produced [yields 2 [add2]] with simplicity 0
```

Another sampling error. This is the test $x = 0$, yet example generation never genreates a zero to test it on. The same thing happens to $x = 3$ (`[yields 2 [predecessor]]`) below.

```
;[22:19:01] *** Gen'd [yields 2 [c+]] with simplicity 1/8000
;[22:19:02] *** Gen'd [yields 2 [predecessor]] with simplicity 1/8000
;[22:19:02] While assessing [yields 2 [predecessor]]
;[22:19:03]  !!! [yields 2 [predecessor]] appears to be constantly ()
;[22:19:03] >>> Selector produced [yields 2 [predecessor]] with simplicity 0
;[22:19:04] *** Gen'd [yields 2 [successor]] with simplicity 1/8000
;[22:19:06] *** Gen'd [yields 4 [add2]] with simplicity 1/8000
;[22:19:07] While assessing [yields 4 [add2]]
;[22:19:07]  !!! [yields 4 [add2]] appears to be constantly ()
;[22:19:07] >>> Selector produced [yields 4 [add2]] with simplicity 0
;[22:19:08] *** Gen'd [yields 4 [c+]] with simplicity 1/8000
;[22:19:09] *** Gen'd [yields 4 [predecessor]] with simplicity 1/8000
;[22:19:10] *** Gen'd [yields 4 [successor]] with simplicity 1/8000
;[22:19:11] *** Gen'd [yields 1 [add2]] with simplicity 1/8000
;[22:19:11] While assessing [yields 1 [add2]]
;[22:19:12]  !!! [yields 1 [add2]] appears to be constantly ()
;[22:19:12] >>> Selector produced [yields 1 [add2]] with simplicity 0
;[22:19:12] *** Gen'd [yields 1 [c+]] with simplicity 1/8000
;[22:19:14] *** Gen'd [yields 1 [predecessor]] with simplicity 1/8000
;[22:19:14] *** Gen'd [yields 1 [successor]] with simplicity 1/8000
;[22:19:15] *** Gen'd [yields 3 [add2]] with simplicity 1/8000
;[22:19:16] *** Gen'd [yields 3 [c+]] with simplicity 1/8000
;[22:19:17] *** Gen'd [yields 3 [predecessor]] with simplicity 1/8000
;[22:19:18] *** Gen'd [yields 3 [successor]] with simplicity 1/8000
;[22:19:20] *** Gen'd [c+ 0] with simplicity 1/8000
;[22:19:21] While assessing [c+ 0]
;[22:19:21]  !!! [c+ 0] always returns its first input
;[22:19:21] >>> Selector produced [c+ 0] with simplicity 0
```

Noticing that zero is the identity for addition.

```
;[22:19:22] *** Gen'd [c+ 6] with simplicity 1/8000
;[22:19:23] *** Gen'd [c+ 5] with simplicity 1/8000
;[22:19:32] *** Gen'd [postfix [successor] [add2]] with simplicity 1/8000
;[22:19:33] *** Gen'd [postfix [successor] [c+]] with simplicity 1/8000
;[22:19:34] *** Gen'd [postfix [successor] [predecessor]] with simplicity 1/8000
;[22:19:34] While assessing [postfix [successor] [predecessor]]
;[22:19:34]  !!! [postfix [successor] [predecessor]] always returns its first input
;[22:19:34] >>> Selector produced [postfix [successor] [predecessor]] with simplicity 0
;[22:19:35] *** Gen'd [postfix [successor] [successor]] with simplicity 1/8000
;[22:19:36] *** Gen'd [postfix [predecessor] [add2]] with simplicity 1/8000
;[22:19:37] *** Gen'd [postfix [predecessor] [c+]] with simplicity 1/8000
;[22:19:38] *** Gen'd [postfix [predecessor] [predecessor]] with simplicity 1/8000
;[22:19:38] *** Gen'd [postfix [predecessor] [successor]] with simplicity 1/8000
;[22:19:39] While assessing [postfix [predecessor] [successor]]
;[22:19:39]  !!! [postfix [predecessor] [successor]] always returns its first input
;[22:19:39] >>> Selector produced [postfix [predecessor] [successor]] with simplicity 0
;[22:19:41] *** Gen'd [postfix [add2] [add2]] with simplicity 1/8000
;[22:19:42] *** Gen'd [postfix [add2] [c+]] with simplicity 1/8000
;[22:19:43] *** Gen'd [postfix [add2] [predecessor]] with simplicity 1/8000
;[22:19:44] *** Gen'd [postfix [add2] [successor]] with simplicity 1/8000
;[22:19:45] *** Gen'd [check [mirror-2 [c+]]] with simplicity 1/8000
;[22:19:45] While assessing [check [mirror-2 [c+]]]
;[22:19:52]  !!! [check [mirror-2 [c+]]] appears to be constantly #T
;[22:19:52] >>> Selector produced [check [mirror-2 [c+]]] with simplicity 0
;[22:19:53] *** Gen'd [check [coalesce [c+]]] with simplicity 1/8000
;[22:19:53] While assessing [check [coalesce [c+]]]
;[22:19:56]  !!! [check [coalesce [c+]]] appears to be constantly #T
;[22:19:56] >>> Selector produced [check [coalesce [c+]]] with simplicity 0
;[22:19:57] *** Gen'd [check [add2]] with simplicity 1/8000
;[22:19:58] While assessing [check [add2]]
;[22:19:58]  !!! [check [add2]] appears to be constantly #T
;[22:19:58] >>> Selector produced [check [add2]] with simplicity 0
;[22:19:59] *** Gen'd [check [c+ 4]] with simplicity 1/8000
;[22:19:59] While assessing [check [c+ 4]]
;[22:19:59]  !!! [check [c+ 4]] appears to be constantly #T
;[22:19:59] >>> Selector produced [check [c+ 4]] with simplicity 0
;[22:20:00] *** Gen'd [check [c+ 1]] with simplicity 1/8000
;[22:20:00] While assessing [check [c+ 1]]
;[22:20:00]  !!! [check [c+ 1]] appears to be constantly #T
;[22:20:01] >>> Selector produced [check [c+ 1]] with simplicity 0
;[22:20:01] *** Gen'd [check [c+ 3]] with simplicity 1/8000
;[22:20:01] While assessing [check [c+ 3]]
;[22:20:02]  !!! [check [c+ 3]] appears to be constantly #T
;[22:20:02] >>> Selector produced [check [c+ 3]] with simplicity 0
```

The above definitions all confirm the reliability of addtion.

```
;[22:20:09] *** Gen'd 10 with simplicity 1/160000
;[22:20:09] *** Gen'd 9 with simplicity 1/160000
;[22:20:11] *** Gen'd [check [postfix [add2] [successor]]] with simplicity 1/160000
;[22:20:11] While assessing [check [postfix [add2] [successor]]]
;[22:20:12]  !!! [check [postfix [add2] [successor]]] appears to be constantly #T
;[22:20:12] >>> Selector produced [check [postfix [add2] [successor]]] with simplicity 0
;[22:20:12] *** Gen'd [check [postfix [add2] [predecessor]]] with simplicity 1/160000
;[22:20:13] While assessing [check [postfix [add2] [predecessor]]]
;[22:20:13]  !!! [check [postfix [add2] [predecessor]]] appears to be constantly #T
;[22:20:13] >>> Selector produced [check [postfix [add2] [predecessor]]] with simplicity 0
;[22:20:14] *** Gen'd [check [postfix [add2] [c+]]] with simplicity 1/160000
;[22:20:14] While assessing [check [postfix [add2] [c+]]]
;[22:20:19]  !!! [check [postfix [add2] [c+]]] appears to be constantly #T
;[22:20:19] >>> Selector produced [check [postfix [add2] [c+]]] with simplicity 0
;[22:20:19] *** Gen'd [check [postfix [add2] [add2]]] with simplicity 1/160000
```

```
;[22:20:19] While assessing [check [postfix [add2] [add2]]]
;[22:20:20]  !!! [check [postfix [add2] [add2]]] appears to be constantly #T
;[22:20:20] >>> Selector produced [check [postfix [add2] [add2]]] with simplicity 0
;[22:20:21] *** Gen'd [check [postfix [predecessor] [predecessor]]] with simplicity 1/160000
```

This is the test $x > 1$.

```
;[22:20:22] *** Gen'd [check [postfix [predecessor] [c+]]] with simplicity 1/160000
;[22:20:22] While assessing [check [postfix [predecessor] [c+]]]
;[22:20:26]  !!! [check [postfix [predecessor] [c+]]] appears to be constantly #T
;[22:20:26] >>> Selector produced [check [postfix [predecessor] [c+]]] with simplicity 0
;[22:20:27] *** Gen'd [check [postfix [predecessor] [add2]]] with simplicity 1/160000
;[22:20:27] While assessing [check [postfix [predecessor] [add2]]]
;[22:20:28]  !!! [check [postfix [predecessor] [add2]]] appears to be constantly #T
;[22:20:28] >>> Selector produced [check [postfix [predecessor] [add2]]] with simplicity 0
;[22:20:28] *** Gen'd [check [postfix [successor] [successor]]] with simplicity 1/160000
;[22:20:28] While assessing [check [postfix [successor] [successor]]]
;[22:20:29]  !!! [check [postfix [successor] [successor]]] appears to be constantly #T
;[22:20:29] >>> Selector produced [check [postfix [successor] [successor]]] with simplicity 0
;[22:20:30] *** Gen'd [check [postfix [successor] [c+]]] with simplicity 1/160000
;[22:20:30] While assessing [check [postfix [successor] [c+]]]
;[22:20:35]  !!! [check [postfix [successor] [c+]]] appears to be constantly #T
;[22:20:35] >>> Selector produced [check [postfix [successor] [c+]]] with simplicity 0
;[22:20:35] *** Gen'd [check [postfix [successor] [add2]]] with simplicity 1/160000
;[22:20:35] While assessing [check [postfix [successor] [add2]]]
;[22:20:36]  !!! [check [postfix [successor] [add2]]] appears to be constantly #T
;[22:20:36] >>> Selector produced [check [postfix [successor] [add2]]] with simplicity 0
;[22:20:37] *** Gen'd [check [c+ 5]] with simplicity 1/160000
;[22:20:37] While assessing [check [c+ 5]]
;[22:20:38]  !!! [check [c+ 5]] appears to be constantly #T
;[22:20:38] >>> Selector produced [check [c+ 5]] with simplicity 0
;[22:20:38] *** Gen'd [check [c+ 6]] with simplicity 1/160000
;[22:20:38] While assessing [check [c+ 6]]
;[22:20:39]  !!! [check [c+ 6]] appears to be constantly #T
;[22:20:39] >>> Selector produced [check [c+ 6]] with simplicity 0
;[22:20:41] *** Gen'd [check [meet [add2] [successor]]] with simplicity 1/160000
;[22:20:41] While assessing [check [meet [add2] [successor]]]
;[22:20:46]  !!! [check [meet [add2] [successor]]] appears to be constantly #T
;[22:20:46] >>> Selector produced [check [meet [add2] [successor]]] with simplicity 0
;[22:20:47] *** Gen'd [check [meet [add2] [predecessor]]] with simplicity 1/160000
;[22:20:49] *** Gen'd [check [meet [add2] [add2]]] with simplicity 1/160000
;[22:20:52] *** Gen'd [check [meet [predecessor] [successor]]] with simplicity 1/160000
;[22:20:54] *** Gen'd [check [meet [predecessor] [predecessor]]] with simplicity 1/160000
;[22:20:54] While assessing [check [meet [predecessor] [predecessor]]]
;[22:21:00]  !!! [check [meet [predecessor] [predecessor]]] appears to be constantly #T
;[22:21:00] >>> Selector produced [check [meet [predecessor] [predecessor]]] with simplicity 0
;[22:21:01] *** Gen'd [check [meet [predecessor] [add2]]] with simplicity 1/160000
;[22:21:03] *** Gen'd [check [meet [successor] [successor]]] with simplicity 1/160000.
;[22:21:03] While assessing [check [meet [successor] [successor]]]
;[22:21:08]  !!! [check [meet [successor] [successor]]] appears to be constantly #T
;[22:21:09] >>> Selector produced [check [meet [successor] [successor]]] with simplicity 0
;[22:21:09] *** Gen'd [check [meet [successor] [predecessor]]] with simplicity 1/160000
;[22:21:11] *** Gen'd [check [meet [successor] [add2]]] with simplicity 1/160000
;[22:21:11] While assessing [check [meet [successor] [add2]]]
;[22:21:17]  !!! [check [meet [successor] [add2]]] appears to be constantly #T
;[22:21:17] >>> Selector produced [check [meet [successor] [add2]]] with simplicity 0
;[22:21:18] *** Gen'd [check [prefix [add2] [successor]]] with simplicity 1/160000
;[22:21:18] While assessing [check [prefix [add2] [successor]]]
;[22:21:19]  !!! [check [prefix [add2] [successor]]] appears to be constantly #T
;[22:21:19] >>> Selector produced [check [prefix [add2] [successor]]] with simplicity 0
;[22:21:19] *** Gen'd [check [prefix [add2] [predecessor]]] with simplicity 1/160000
;[22:21:20] While assessing [check [prefix [add2] [predecessor]]]
;[22:21:20]  !!! [check [prefix [add2] [predecessor]]] appears to be constantly #T
```

```
;[22:21:20] >>> Selector produced [check [prefix [add2] [predecessor]]] with simplicity 0
;[22:21:21] *** Gen'd [check [prefix [add2] [c+]]] with simplicity 1/160000
;[22:21:21] While assessing [check [prefix [add2] [c+]]]
;[22:21:34]  !!! [check [prefix [add2] [c+]]] appears to be constantly #T
;[22:21:35] >>> Selector produced [check [prefix [add2] [c+]]] with simplicity 0
;[22:21:35] *** Gen'd [check [prefix [add2] [add2]]] with simplicity 1/160000
;[22:21:35] While assessing [check [prefix [add2] [add2]]]
;[22:21:36]  !!! [check [prefix [add2] [add2]]] appears to be constantly #T
;[22:21:36] >>> Selector produced [check [prefix [add2] [add2]]] with simplicity 0
;[22:21:37] *** Gen'd [check [prefix [c+] [successor]]] with simplicity 1/160000
;[22:21:37] While assessing [check [prefix [c+] [successor]]]
;[22:21:42]  !!! [check [prefix [c+] [successor]]] appears to be constantly #T
;[22:21:42] >>> Selector produced [check [prefix [c+] [successor]]] with simplicity 0
;[22:21:43] *** Gen'd [check [prefix [c+] [predecessor]]] with simplicity 1/160000
;[22:21:43] While assessing [check [prefix [c+] [predecessor]]]
;[22:21:48]  !!! [check [prefix [c+] [predecessor]]] appears to be constantly #T
;[22:21:48] >>> Selector produced [check [prefix [c+] [predecessor]]] with simplicity 0
;[22:21:50] *** Gen'd [check [prefix [c+] [c+]]] with simplicity 1/160000
;[22:21:50] While assessing [check [prefix [c+] [c+]]]
;[22:22:13]  !!! [check [prefix [c+] [c+]]] appears to be constantly #T
;[22:22:14] >>> Selector produced [check [prefix [c+] [c+]]] with simplicity 0
;[22:22:14] *** Gen'd [check [prefix [c+] [add2]]] with simplicity 1/160000
;[22:22:14] While assessing [check [prefix [c+] [add2]]]
;[22:22:25]  !!! [check [prefix [c+] [add2]]] appears to be constantly #T
;[22:22:25] >>> Selector produced [check [prefix [c+] [add2]]] with simplicity 0
;[22:22:25] *** Gen'd [check [prefix [predecessor] [predecessor]]] with simplicity 1/160000
;[22:22:26] *** Gen'd [check [prefix [predecessor] [c+]]] with simplicity 1/160000
;[22:22:26] While assessing [check [prefix [predecessor] [c+]]]
;[22:22:45]  !!! [check [prefix [predecessor] [c+]]] appears to be constantly #T
;[22:22:45] >>> Selector produced [check [prefix [predecessor] [c+]]] with simplicity 0
;[22:22:46] *** Gen'd [check [prefix [predecessor] [add2]]] with simplicity 1/160000
;[22:22:46] While assessing [check [prefix [predecessor] [add2]]]
;[22:22:47]  !!! [check [prefix [predecessor] [add2]]] appears to be constantly #T
;[22:22:47] >>> Selector produced [check [prefix [predecessor] [add2]]] with simplicity 0
;[22:22:47] *** Gen'd [check [prefix [successor] [successor]]] with simplicity 1/160000
;[22:22:47] While assessing [check [prefix [successor] [successor]]]
;[22:22:48]  !!! [check [prefix [successor] [successor]]] appears to be constantly #T
;[22:22:48] >>> Selector produced [check [prefix [successor] [successor]]] with simplicity 0
;[22:22:54] *** Gen'd [check [prefix [successor] [c+]]] with simplicity 1/160000
;[22:22:54] While assessing [check [prefix [successor] [c+]]]
;[22:23:12]  !!! [check [prefix [successor] [c+]]] appears to be constantly #T
;[22:23:12] >>> Selector produced [check [prefix [successor] [c+]]] with simplicity 0
;[22:23:12] *** Gen'd [check [prefix [successor] [add2]]] with simplicity 1/160000
;[22:23:13] While assessing [check [prefix [successor] [add2]]]
;[22:23:13]  !!! [check [prefix [successor] [add2]]] appears to be constantly #T
;[22:23:14] >>> Selector produced [check [prefix [successor] [add2]]] with simplicity 0
;[22:23:15] *** Gen'd [check [coalesce [mirror-2 [c+]]]] with simplicity 1/160000
;[22:23:15] While assessing [check [coalesce [mirror-2 [c+]]]]
;[22:23:23]  !!! [check [coalesce [mirror-2 [c+]]]] appears to be constantly #T
;[22:23:23] >>> Selector produced [check [coalesce [mirror-2 [c+]]]] with simplicity 0
;[22:23:25] *** Gen'd [check [limit [predecessor] [successor]]] with simplicity 1/160000
;[22:23:26] While assessing [check [limit [predecessor] [successor]]]
;[22:23:33]  !!! [check [limit [predecessor] [successor]]] appears to be constantly #T
;[22:23:33] >>> Selector produced [check [limit [predecessor] [successor]]] with simplicity 0
;[22:23:33] *** Gen'd [check [limit [predecessor] [predecessor]]] with simplicity 1/160000
;[22:23:35] *** Gen'd [check [limit [predecessor] [add2]]] with simplicity 1/160000
;[22:23:35] While assessing [check [limit [predecessor] [add2]]]
;[22:23:42]  !!! [check [limit [predecessor] [add2]]] appears to be constantly #T
;[22:23:43] >>> Selector produced [check [limit [predecessor] [add2]]] with simplicity 0
;[22:23:44] *** Gen'd [check [mirror-2 [c+] 2]] with simplicity 1/160000
;[22:23:44] While assessing [check [mirror-2 [c+] 2]]
;[22:23:48]  !!! [check [mirror-2 [c+] 2]] appears to be constantly #T
```

```
;[22:23:48] >>> Selector produced [check [mirror-2 [c+] 2]] with simplicity 0
;[22:23:49] ••• Gen'd [check [mirror-2 [c+] 4]] with simplicity 1/160000
;[22:23:49] While assessing [check [mirror-2 [c+] 4]]
;[22:23:53]  !!! [check [mirror-2 [c+] 4]] appears to be constantly #T
;[22:23:54] >>> Selector produced [check [mirror-2 [c+] 4]] with simplicity 0
;[22:23:54] ••• Gen'd [check [mirror-2 [c+] 1]] with simplicity 1/160000
;[22:23:54] While assessing [check [mirror-2 [c+] 1]]
;[22:23:59]  !!! [check [mirror-2 [c+] 1]] appears to be constantly #T
;[22:23:59] >>> Selector produced [check [mirror-2 [c+] 1]] with simplicity 0
;[22:23:59] ••• Gen'd [check [mirror-2 [c+] 3]] with simplicity 1/160000
;[22:24:00] While assessing [check [mirror-2 [c+] 3]]
;[22:24:04]  !!! [check [mirror-2 [c+] 3]] appears to be constantly #T
;[22:24:04] >>> Selector produced [check [mirror-2 [c+] 3]] with simplicity 0
;[22:24:05] ••• Gen'd [check [mirror-2 [mirror-2 [c+]]]] with simplicity 1/160000
;[22:24:05] While assessing [check [mirror-2 [mirror-2 [c+]]]]
;[22:24:20]  !!! [check [mirror-2 [mirror-2 [c+]]]] appears to be constantly #T
;[22:24:20] >>> Selector produced [check [mirror-2 [mirror-2 [c+]]]] with simplicity 0
;[22:24:21] ••• Gen'd [postfix [add2] [mirror-2 [c+]]] with simplicity 1/160000
;[22:24:23] ••• Gen'd [postfix [add2] [coalesce [c+]]] with simplicity 1/160000
;[22:24:25] ••• Gen'd [postfix [add2] [add2]] with simplicity 1/160000
;[22:24:28] While rebiasing the pool #42229-Pool: XBET
;[22:24:28]  ••• Type analysis identifies five initial cliques
;[22:24:29]  ••• Hash Code Size From         To
;[22:24:30]  ••• --------- ---- ----         --
;[22:24:31]  ••• #284      28   #13Numbers #13Numbers
;[22:24:31]  ••• #285      24   #13Numbers #146FCE #13Numbers --> #13Numbers
;[22:24:32]  ••• #286      14   #13Numbers #147Booleans
;[22:24:32]  ••• #287      11   #13Numbers #910FCE #13Numbers --> #147Booleans
;[22:24:32]  ••• #288      1    #13Numbers #1190FCE #13Numbers --> #146FCE #13Numbers --> #13Numbers
;[22:24:33]  ••• which I will try and further distinguish (in fifteen passes)
;[22:24:50]  === On first pass, generated 20 new cliques and
;[22:24:50]  ===                removed 11 singletons, leaving 13 cliques altogether
;[22:25:07]  === On second pass, generated 7 new cliques and
;[22:25:07]  ===                removed 3 singletons, leaving 17 cliques altogether
;[22:25:22]  === On third pass, generated 2 new cliques and
;[22:25:22]  ===                removed 2 singletons, leaving 17 cliques altogether
;[22:25:39]  === On fourth pass, generated 2 new cliques and
;[22:25:39]  ===                removed 2 singletons, leaving 17 cliques altogether
;[22:25:55]  === On fifth pass, generated 1 new clique and
;[22:25:55]  ===                removed 0 singletons, leaving 18 cliques altogether
;[22:26:10]  === On sixth pass, generated 2 new cliques and
;[22:26:10]  ===                removed 2 singletons, leaving 18 cliques altogether....
;[22:27:21]  === On eleventh pass, generated 1 new clique and
;[22:27:21]  ===                removed 2 singletons, leaving 17 cliques altogether....
;[22:28:23]  ••• After fifteen passes, finished with seventeen cliques
;[22:28:23]  •••       with a mean of 3.235 (sd=3.119; n=17) members each.
;[22:28:28]  ••• Hash Code Size For instance                                  Which computes
;[22:28:28]  ••• --------- ---- ------------                                  --------------
;[22:28:28]  ••• #297      6    [prefix [predecessor] [add2]]                 15 <=== 14
;[22:28:28]  ••• #298      6    [prefix [successor] [add2]]                   17 <=== 14
;[22:28:29]  ••• #299      5    [yields 4 [predecessor]]                      () <=== 6
;[22:28:29]  ••• #300      5    [prefix [add2] [add2]]                        19 <=== 15
;[22:28:29]  ••• #301      4    [postfix [add2] [c+]]                         16 <=== 3, 11
;[22:28:30]  ••• #302      4    [add2]                                        7 <=== 5
;[22:28:30]  ••• #303      3    [prefix [successor] [c+]]                     13 <=== 0, 12
;[22:28:30]  ••• #304      3    [check [meet [add2] [predecessor]]]           #T <=== 4, 0
20 ⟹ ;[22:28:30]  ••• #305      3    [mirror-2 [mirror-2 [c+]]]                    12 <=== 11, 11
;[22:28:31]  ••• #306      2    [prefix [c+] [predecessor]]                   23 <=== 13, 11
;[22:28:31]  ••• #307      2    [check [prefix [predecessor] [predecessor]]] #T <=== 15
;[22:28:31]  ••• #308      2    [prefix [predecessor] [predecessor]]          14 <=== 16
;[22:28:31]  ••• #309      2    [identical? [predecessor] [predecessor]]      () <=== 8, 0
;[22:28:32]  ••• #310      2    [meet [predecessor] [successor]]              10 <=== 6, 13
```

```
;[22:28:32]  *** #311      2    [yields 3 [successor]]                    () <=== 3
;[22:28:32]  *** #312      2    [coalesce [c+]]                           6 <=== 3
;[22:28:32]  *** #313      2    [check [meet [predecessor] [successor]]]  () <=== 7, 5
;[22:28:33]  *** Identified three significant constants
;[22:28:33]  *** Value Size For instance
;[22:28:34]  *** ----- ---- ------------
;[22:28:34]  *** #T    46   [check [meet [successor] [add2]]]
;[22:28:34]  *** ()    8    [identical? [add2] [predecessor]]
;[22:28:34]  *** 0     1    [terminator [predecessor]]
;[22:28:42]  *** Introducing new primitives based on frequency analysis
;[22:28:44]  *** Hash Code New Primitive          Frequency
;[22:28:44]  *** --------- -------------          ---------
;[22:28:44]  *** #22       [mirror-2]             8/417
;[22:28:44]  *** #28       [yields]               7/417
;[22:28:44]  *** #33       [check]                7/417
;[22:28:45]  *** #30       [meet]                 2/139
;[22:28:45]  *** #35       [postfix]              5/139
;[22:28:45]  *** #25       [prefix]               5/139
;[22:28:45]  *** #29       [c+]                   9/139
;[22:28:45]  *** #19       [successor]            37/417
;[22:28:46]  *** #43       [add2]                 8/139
;[22:28:46]  *** #24       [predecessor]          12/139
;[22:28:46]  *** #31       1                      41/417
;[22:28:46]  *** #21       2                      4/139
;[22:28:47]  *** #36       3                      8/417
;[22:28:47]  *** #26       4                      4/417
;[22:28:47]  *** #316      [postfix [add2]]       2/139
;[22:28:47]  *** #317      [postfix [predecessor]] 5/417
;[22:28:48]  *** #318      [postfix [successor]]  4/417
;[22:28:48]  *** #54       [c+ 3]                 5/417
;[22:28:48]  *** #55       [c+ 1]                 3/139
;[22:28:48]  *** #319      [meet [predecessor]]   4/417
;[22:28:48]  *** #60       0                      10/139
;[22:28:49]  *** #320      [prefix [add2]]        4/417
;[22:28:49]  *** #321      [prefix [predecessor]] 4/417
;[22:28:49]  *** #322      [prefix [successor]]   4/417
;[22:28:49]  *** #70       [mirror-2 [c+]]        8/417
;[22:28:56] *** Gen'd [check [limit [predecessor] [predecessor] 0]] with simplicity 9/8000
;[22:28:56] While assessing [check [limit [predecessor] [predecessor] 0]]
;[22:28:57]  !!! [check [limit [predecessor] [predecessor] 0]] appears to be constantly #T
;[22:28:57] >>> Selector produced [check [limit [predecessor] [predecessor] 0]] with simplicity 0
;[22:28:58] *** Gen'd [meet [predecessor] [successor] 0] with simplicity 9/8000
;[22:28:58] While assessing [meet [predecessor] [successor] 0]
;[22:29:28]  !!! [meet [predecessor] [successor] 0] appears to be constantly 0
;[22:29:28]  !!! [meet [predecessor] [successor] 0] always returns its first input
;[22:29:28] >>> Selector produced [meet [predecessor] [successor] 0] with simplicity 0
;[22:29:30] *** Gen'd [check [meet [predecessor] [add2] 0]] with simplicity 9/8000
;[22:29:38] *** Gen'd [c+ 1] with simplicity 9/8000
;[22:29:38] *** Gen'd [identical? [successor] [successor] 0] with simplicity 9/8000
;[22:29:40] *** Gen'd [add2] with simplicity 9/8000
;[22:29:41] ??? The combination of [postfix [predecessor] [predecessor]] with 0,
;[22:29:41] ??? which looked legal, turns out to be undefined.
;[22:29:42] ??? The combination of [prefix [c+] [predecessor]] with 0,
;[22:29:43] ??? which looked legal, turns out to be undefined.
;[22:29:44] *** Gen'd [limit [predecessor] [successor] 0] with simplicity 9/8000
;[22:29:44] While assessing [limit [predecessor] [successor] 0]
;[22:29:45]  !!! [limit [predecessor] [successor] 0] always returns its first input
;[22:29:45] >>> Selector produced [limit [predecessor] [successor] 0] with simplicity 0
;[22:29:46] *** Gen'd 6 with simplicity 9/10000
;[22:29:46] *** Gen'd 5 with simplicity 9/10000
;[22:29:52] *** Gen'd 7 with simplicity 9/10000
;[22:30:04] *** Gen'd [prefix [add2] [c+ 3]] with simplicity 9/10000
```

```
;[22:30:07] *** Gen'd [prefix [add2] [c+ 1]] with simplicity 9/10000
;[22:30:08] *** Gen'd [check [c+ 3]] with simplicity 9/10000
;[22:41:00] While assessing [check [c+ 3]]
;[22:41:01]  !!! [check [c+ 3]] appears to be constantly #T
;[22:41:01] >>> Selector produced [check [c+ 3]] with simplicity 0
;[22:41:01] *** Gen'd [check [c+ 1]] with simplicity 9/10000
;[22:41:01] While assessing [check [c+ 1]]
;[22:41:02]  !!! [check [c+ 1]] appears to be constantly #T
;[22:41:02] >>> Selector produced [check [c+ 1]] with simplicity 0
;[22:41:02] *** Gen'd [c+ 0] with simplicity 9/10000
;[22:41:02] While assessing [c+ 0]
;[22:41:03]  !!! [c+ 0] always returns its first input
;[22:41:03] >>> Selector produced [c+ 0] with simplicity 0
;[22:41:04] *** Gen'd [postfix [add2] [c+ 3]] with simplicity 9/10000
;[22:41:04] *** Gen'd [postfix [add2] [c+ 1]] with simplicity 9/10000
;[22:41:05] *** Gen'd [prefix [successor] [c+ 3]] with simplicity 9/10000
;[22:41:06] *** Gen'd [prefix [successor] [c+ 1]] with simplicity 9/10000
;[22:41:07] *** Gen'd [postfix [successor] [c+ 3]] with simplicity 9/10000
;[22:41:07] *** Gen'd [postfix [successor] [c+ 1]] with simplicity 9/10000
;[22:41:08] *** Gen'd [prefix [predecessor] [c+ 3]] with simplicity 9/10000
;[22:41:09] *** Gen'd [prefix [predecessor] [c+ 1]] with simplicity 9/10000
;[22:41:09] While assessing [prefix [predecessor] [c+ 1]]
;[22:41:09]  !!! [prefix [predecessor] [c+ 1]] always returns its first input
;[22:41:10] >>> Selector produced [prefix [predecessor] [c+ 1]] with simplicity 0
;[22:41:10] *** Gen'd [postfix [predecessor] [c+ 3]] with simplicity 9/10000
;[22:41:11] *** Gen'd [postfix [predecessor] [c+ 1]] with simplicity 9/10000
;[22:41:11] While assessing [postfix [predecessor] [c+ 1]]
;[22:41:12]  !!! [postfix [predecessor] [c+ 1]] always returns its first input
;[22:41:12] >>> Selector produced [postfix [predecessor] [c+ 1]] with simplicity 0
;[22:41:13] *** Gen'd [mirror-2 [c+] 0] with simplicity 9/10000
;[22:41:13] While assessing [mirror-2 [c+] 0]
;[22:41:19]  !!! [mirror-2 [c+] 0] always returns its first input
;[22:41:19] >>> Selector produced [mirror-2 [c+] 0] with simplicity 0
;[22:41:20] *** Gen'd [meet [predecessor] [c+ 3]] with simplicity 9/10000
;[22:41:22] *** Gen'd [meet [predecessor] [c+ 1]] with simplicity 9/10000
;[22:41:24] ??? The combination of [predecessor] with 0,
;[22:41:24] ??? which looked legal, turns out to be undefined.
;[22:42:00] *** Gen'd [check [limit [predecessor] [predecessor] 1]] with simplicity 1/2000
;[22:42:01] *** Gen'd [check [limit [predecessor] [predecessor] 3]] with simplicity 1/2000
;[22:42:03] *** Gen'd [check [limit [predecessor] [predecessor] 2]] with simplicity 1/2000
;[22:42:04] *** Gen'd [check [limit [predecessor] [predecessor] 4]] with simplicity 1/2000
;[22:42:06] *** Gen'd [meet [predecessor] [successor] 1] with simplicity 1/2000
;[22:42:06] While assessing [meet [predecessor] [successor] 1]
;[22:42:37]  !!! [meet [predecessor] [successor] 1] appears to be constantly 1
;[22:42:37]  !!! [meet [predecessor] [successor] 1] always returns its first input
;[22:42:38] >>> Selector produced [meet [predecessor] [successor] 1] with simplicity 0
;[22:42:39] *** Gen'd [meet [predecessor] [successor] 3] with simplicity 1/2000
;[22:42:51] *** Gen'd [meet [predecessor] [successor] 2] with simplicity 1/2000
;[22:43:00] *** Gen'd [meet [predecessor] [successor] 4] with simplicity 1/2000
;[22:43:10] *** Gen'd 8 with simplicity 1/2000
;[22:43:11] *** Gen'd [check [meet [predecessor] [add2] 1]] with simplicity 1/2000
;[22:43:15] *** Gen'd [check [meet [predecessor] [add2] 3]] with simplicity 1/2000
;[22:43:20] *** Gen'd [check [meet [predecessor] [add2] 2]] with simplicity 1/2000
;[22:43:34] *** Gen'd [check [meet [predecessor] [add2] 4]] with simplicity 1/2000
;[22:46:34] *** Gen'd [add2] with simplicity 1/2000
;[22:46:35] *** Gen'd [c+ 4] with simplicity 1/2000
;[22:46:36] *** Gen'd [c+ 3] with simplicity 1/2000
;[22:46:37] *** Gen'd [c+ 5] with simplicity 1/2000
;[22:46:39] *** Gen'd [identical? [successor] [successor] 1] with simplicity 1/2000
;[22:46:40] *** Gen'd [identical? [successor] [successor] 3] with simplicity 1/2000
;[22:46:40] While assessing [identical? [successor] [successor] 3]
;[22:46:41]  !!! [identical? [successor] [successor] 3] appears to be constantly ()
```

```
;[22:46:41] >>> Selector produced [identical? [successor] [successor] 3] with simplicity 0
;[22:46:43] *** Gen'd [identical? [successor] [successor] 2] with simplicity 1/2000
;[22:46:44] *** Gen'd [identical? [successor] [successor] 4] with simplicity 1/2000
;[22:46:44] While assessing [identical? [successor] [successor] 4]
;[22:46:45]  !!! [identical? [successor] [successor] 4] appears to be constantly ()
;[22:46:45] >>> Selector produced [identical? [successor] [successor] 4] with simplicity 0
;[22:46:47] *** Gen'd [c+ 3] with simplicity 1/2000
;[22:46:48] *** Gen'd [c+ 5] with simplicity 1/2000
;[22:46:49] *** Gen'd [c+ 4] with simplicity 1/2000
;[22:46:50] *** Gen'd [c+ 6] with simplicity 1/2000
;[22:46:51] ??? The combination of [postfix [predecessor] [predecessor]] with 1,
;[22:46:51] ??? which looked legal, turns out to be undefined.
;[22:47:00] *** Gen'd [c+ 0] with simplicity 1/2000
;[22:47:01] While assessing [c+ 0]
;[22:47:01]  !!! [c+ 0] always returns its first input
;[22:47:01] >>> Selector produced [c+ 0] with simplicity 0
;[22:47:02] *** Gen'd [add2] with simplicity 1/2000
;[22:47:04] *** Gen'd [c+ 1] with simplicity 1/2000
;[22:47:04] *** Gen'd [c+ 3] with simplicity 1/2000
;[22:47:07] *** Gen'd [limit [predecessor] [successor] 1] with simplicity 1/2000
;[22:47:08] *** Gen'd [limit [predecessor] [successor] 3] with simplicity 1/2000
;[22:47:10] *** Gen'd [limit [predecessor] [successor] 2] with simplicity 1/2000
;[22:47:10] *** Gen'd [limit [predecessor] [successor] 4] with simplicity 1/2000
;[22:47:23] *** Gen'd [meet [predecessor] [coalesce [c+]]] with simplicity 1/2000
;[22:47:25] *** Gen'd [meet [predecessor] [add2]] with simplicity 1/2000
;[22:47:27] *** Gen'd [meet [predecessor] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:47:28] *** Gen'd [meet [predecessor] [c+ 4]] with simplicity 1/2000
;[22:47:44] *** Gen'd [postfix [predecessor] [check [limit [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:47:49] *** Gen'd [postfix [predecessor] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:47:52] *** Gen'd [postfix [predecessor] [yields 3 [successor]]] with simplicity 1/2000
;[22:47:53] *** Gen'd [postfix [predecessor] [coalesce [c+]]] with simplicity 1/2000
;[22:47:54] *** Gen'd [postfix [predecessor] [check [meet [predecessor] [add2]]]] with simplicity 1/2000
;[22:48:01] *** Gen'd [postfix [predecessor] [add2]] with simplicity 1/2000
;[22:48:02] *** Gen'd [postfix [predecessor] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:48:03] *** Gen'd [postfix [predecessor] [identical? [successor] [successor]]] with simplicity 1/2000
;[22:48:03] While assessing [postfix [predecessor] [identical? [successor] [successor]]]
;[22:48:10]  !!! [postfix [predecessor] [identical? [successor] [successor]]] appears to be constantly ()
;[22:48:10] *** Gen'd [postfix [predecessor] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:48:12] *** Gen'd [postfix [predecessor] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:48:12] *** Gen'd [postfix [predecessor] [yields 4 [predecessor]]] with simplicity 1/2000
;[22:48:13] *** Gen'd [postfix [predecessor] [check [postfix [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:48:14] *** Gen'd [postfix [predecessor] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:48:15] *** Gen'd [postfix [predecessor] [c+ 4]] with simplicity 1/2000
;[22:48:16] *** Gen'd [postfix [predecessor] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:48:21] *** Gen'd [prefix [predecessor] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:48:24] *** Gen'd [prefix [predecessor] [coalesce [c+]]] with simplicity 1/2000
;[22:48:25] *** Gen'd [prefix [predecessor] [add2]] with simplicity 1/2000
;[22:48:27] *** Gen'd [prefix [predecessor] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:48:30] *** Gen'd [prefix [predecessor] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:48:32] *** Gen'd [prefix [predecessor] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:48:34] *** Gen'd [prefix [predecessor] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:48:36] *** Gen'd [prefix [predecessor] [c+ 4]] with simplicity 1/2000
;[22:48:37] *** Gen'd [prefix [predecessor] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:48:40] *** Gen'd [mirror-2 [check [limit [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:48:42] *** Gen'd [mirror-2 [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:48:45] *** Gen'd [mirror-2 [check [meet [predecessor] [add2]]]] with simplicity 1/2000
;[22:48:47] *** Gen'd [mirror-2 [prefix [c+] [successor]]] with simplicity 1/2000
;[22:48:48] *** Gen'd [mirror-2 [identical? [successor] [successor]]] with simplicity 1/2000
;[22:48:50] *** Gen'd [mirror-2 [postfix [add2] [c+]]] with simplicity 1/2000
;[22:48:51] *** Gen'd [mirror-2 [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:48:53] *** Gen'd [mirror-2 [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:48:54] *** Gen'd [postfix [successor] [check [limit [predecessor] [predecessor]]]] with simplicity 1/2000
```

```
;[22:48:57] *** Gen'd [postfix [successor] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:49:03] *** Gen'd [postfix [successor] [yields 3 [successor]]] with simplicity 1/2000
;[22:49:04] *** Gen'd [postfix [successor] [coalesce [c+]]] with simplicity 1/2000
;[22:49:05] *** Gen'd [postfix [successor] [check [meet [predecessor] [add2]]]] with simplicity 1/2000
;[22:49:08] *** Gen'd [postfix [successor] [add2]] with simplicity 1/2000
;[22:49:09] *** Gen'd [postfix [successor] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:49:10] *** Gen'd [postfix [successor] [identical? [successor] [successor]]] with simplicity 1/2000
;[22:49:12] *** Gen'd [postfix [successor] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:49:13] *** Gen'd [postfix [successor] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:49:14] *** Gen'd [postfix [successor] [yields 4 [predecessor]]] with simplicity 1/2000
;[22:49:15] *** Gen'd [postfix [successor] [check [postfix [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:49:16] *** Gen'd [postfix [successor] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:49:18] *** Gen'd [postfix [successor] [c+ 4]] with simplicity 1/2000
;[22:49:18] *** Gen'd [postfix [successor] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:49:32] *** Gen'd [prefix [successor] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:49:35] *** Gen'd [prefix [successor] [coalesce [c+]]] with simplicity 1/2000
;[22:49:37] *** Gen'd [prefix [successor] [add2]] with simplicity 1/2000
;[22:49:38] *** Gen'd [prefix [successor] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:49:41] *** Gen'd [prefix [successor] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:49:43] *** Gen'd [prefix [successor] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:49:45] *** Gen'd [prefix [successor] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:49:47] *** Gen'd [prefix [successor] [c+ 4]] with simplicity 1/2000
;[22:49:49] *** Gen'd [prefix [successor] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:49:51] *** Gen'd [postfix [add2] [check [limit [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:49:54] *** Gen'd [postfix [add2] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:49:55] *** Gen'd [postfix [add2] [yields 3 [successor]]] with simplicity 1/2000
;[22:49:56] *** Gen'd [postfix [add2] [coalesce [c+]]] with simplicity 1/2000
;[22:49:58] *** Gen'd [postfix [add2] [check [meet [predecessor] [add2]]]] with simplicity 1/2000
;[22:50:00] *** Gen'd [postfix [add2] [add2]] with simplicity 1/2000
;[22:50:01] *** Gen'd [postfix [add2] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:50:03] *** Gen'd [postfix [add2] [identical? [successor] [successor]]] with simplicity 1/2000
;[22:50:10] *** Gen'd [postfix [add2] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:50:11] *** Gen'd [postfix [add2] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:50:11] While assessing [postfix [add2] [postfix [predecessor] [predecessor]]]
;[22:50:12]  !!! [postfix [add2] [postfix [predecessor] [predecessor]]] always returns its first input
;[22:50:13] >>> Selector produced [postfix [add2] [postfix [predecessor] [predecessor]]] with simplicity 0
;[22:50:13] *** Gen'd [postfix [add2] [yields 4 [predecessor]]] with simplicity 1/2000
;[22:50:14] *** Gen'd [postfix [add2] [check [postfix [predecessor] [predecessor]]]] with simplicity 1/2000
;[22:50:15] While assessing [postfix [add2] [check [postfix [predecessor] [predecessor]]]]
;[22:50:16]  !!! [postfix [add2] [check [postfix [predecessor] [predecessor]]]] appears to be constantly #T
;[22:50:16] >>> Selector produced [postfix [add2] [check [postfix [predecessor] [predecessor]]]] with simplicity 0
;[22:50:17] *** Gen'd [postfix [add2] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:50:19] *** Gen'd [postfix [add2] [c+ 4]] with simplicity 1/2000
;[22:50:20] *** Gen'd [postfix [add2] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:50:23] *** Gen'd [check [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:50:30] *** Gen'd [check [coalesce [c+]]] with simplicity 1/2000
;[22:50:30] While assessing [check [coalesce [c+]]]
;[22:50:37]  !!! [check [coalesce [c+]]] appears to be constantly #T
;[22:50:37] >>> Selector produced [check [coalesce [c+]]] with simplicity 0
;[22:50:38] *** Gen'd [check [add2]] with simplicity 1/2000
;[22:50:38] While assessing [check [add2]]
;[22:50:39]  !!! [check [add2]] appears to be constantly #T
;[22:50:39] >>> Selector produced [check [add2]] with simplicity 0
;[22:50:39] *** Gen'd [check [prefix [c+] [successor]]] with simplicity 1/2000
;[22:50:40] While assessing [check [prefix [c+] [successor]]]
;[22:50:48]  !!! [check [prefix [c+] [successor]]] appears to be constantly #T
;[22:50:49] >>> Selector produced [check [prefix [c+] [successor]]] with simplicity 0
;[22:50:49] *** Gen'd [check [postfix [add2] [c+]]] with simplicity 1/2000
;[22:50:49] While assessing [check [postfix [add2] [c+]]]
;[22:50:59]  !!! [check [postfix [add2] [c+]]] appears to be constantly #T
;[22:50:59] >>> Selector produced [check [postfix [add2] [c+]]] with simplicity 0
;[22:50:59] *** Gen'd [check [prefix [c+] [predecessor]]] with simplicity 1/2000
```

```
;[22:51:00] While assessing [check [prefix [c+] [predecessor]]]
;[22:51:09]  !!! [check [prefix [c+] [predecessor]]] appears to be constantly ST
;[22:51:09] >>> Selector produced [check [prefix [c+] [predecessor]]] with simplicity 0
;[22:51:09] *** Gen'd [check [c+ 4]] with simplicity 1/2000
;[22:51:09] While assessing [check [c+ 4]]
;[22:51:10]  !!! [check [c+ 4]] appears to be constantly ST
;[22:51:10] >>> Selector produced [check [c+ 4]] with simplicity 0
;[22:51:11] *** Gen'd [check [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:51:11] While assessing [check [limit [predecessor] [successor]]]
;[22:51:21]  !!! [check [limit [predecessor] [successor]]] appears to be constantly ST
;[22:51:21] >>> Selector produced [check [limit [predecessor] [successor]]] with simplicity 0
;[22:51:23] *** Gen'd [prefix [add2] [meet [predecessor] [successor]]] with simplicity 1/2000
;[22:51:30] *** Gen'd [prefix [add2] [coalesce [c+]]] with simplicity 1/2000
;[22:51:32] *** Gen'd [prefix [add2] [add2]] with simplicity 1/2000
;[22:51:33] *** Gen'd [prefix [add2] [prefix [c+] [successor]]] with simplicity 1/2000
;[22:51:36] *** Gen'd [prefix [add2] [postfix [add2] [c+]]] with simplicity 1/2000
;[22:51:38] *** Gen'd [prefix [add2] [postfix [predecessor] [predecessor]]] with simplicity 1/2000
;[22:51:39] While assessing [prefix [add2] [postfix [predecessor] [predecessor]]]
;[22:51:39]  !!! [prefix [add2] [postfix [predecessor] [predecessor]]] always returns its first input
;[22:51:40] >>> Selector produced [prefix [add2] [postfix [predecessor] [predecessor]]] with simplicity 0
;[22:51:41] *** Gen'd [prefix [add2] [prefix [c+] [predecessor]]] with simplicity 1/2000
;[22:51:44] *** Gen'd [prefix [add2] [c+ 4]] with simplicity 1/2000
;[22:51:45] *** Gen'd [prefix [add2] [limit [predecessor] [successor]]] with simplicity 1/2000
;[22:51:47] *** Gen'd [prefix [add2] [c+]] with simplicity 1/2500
;[22:51:49] *** Gen'd [prefix [add2] [add2]] with simplicity 1/2500
```

3.8 Technical Problems with Cyrano

It was originally hoped that this chapter would include a connected run of Cyrano going from tie structures to modular numbers; unfortunately, resource issues forced the presentation of fragmented scenarios and this frank discussion about the limits of the present implementation.

Cyrano has the capacity — in theory — to traverse the chain of representations leading from tie structures to modular numbers. It does not easily perform this traversal for two particular reasons: the cost of modelling reified vocabularies and the combinatorics of vocabulary invention.

Both are relatively deep phenomena which have no 'quick fix' but will be the subject of future technical work. But I expect that the things which are slowing Cyrano down are some of the same things that keep human scientists and mathematicians from making the quick moves that Cyrano does. Indeed, we should perhaps be worried if Cyrano could invent so much of mathematics in the time that it has.

The expense of using reified vocabularies is simply that when the objects being manipulated and identified are procedures, the manipulation and identification of them involves the calling of procedures at a lower level multiple times. The space and time overhead — in Cyrano's implementation — of these operations cause significant performance penalties. However, I expect that with substantial tuning and a modest extension of resources, those problems could be alleviated.

The combinatorics of vocabulary formation are a problem which is harder to deal with. In particular, if one vocabulary might spawn a large number of reified vocabularies, the problem of exploring all of them is substantial, especially as the space of definitions grows exponentially with the number of vocabularies. However, this problem might be partially resolved by large scale parallelism which allows different processes to follow up on the exploration of different vocabularies.

This is definitely work in progress and I expect that a later version of this report will have a full blown trace and a more detailed discussion of the past and future performance problems of the system.

3.9 Things Cyrano Didn't Do (And Perhaps Should Have)

The above sections sketched several of Cyrano's conceptions in arithmetic; they and a variety of others (arithmetic means, maximum and minimum operations, to name a few) are described in detail in Chapter 3. But for any system that solves problems or invents things which have the capacity to suprise us, an important question is 'what should it have come up with which it didn't?'. The answers to these questions will reveal the representational limits of the system and give important pointers to new research directions.

The asking of such questions and their answers in implementation are open to a

certain criticism of 'tuning'; it is the case that I would sometimes note Cyrano 'almost getting something' and use this to guide where I put my programming effort. For instance, an early version of Cyrano suprised me once by 'almost getting' modular numbers; it failed because its facilities for computing in abstracted vocabularies were too awkward to provide an efficient implementation of the modular abstraction of the numeric abstraction of operations on list structures. However, the potential revealed by 'almost getting' modular numbers led me to put extra energy into the abstraction mechanisms which enabled such steps.

One of the things Cyrano did not discover was the generic abstraction N mod r; though it constructed several modular fields (including numbers (mod 2) (the evens and odds) where it conjectured some of Euclid's arithmetic axioms), it never abstracted this operation. The reason was that the key element in this formation step was reification, which was separated from the process of exploration and rebiasing. This separation was crucial because reification introduces new standards of simplicity and thus could not compete as it should in the old arena. This problem suggests a next step of having layers of discovery systems, where reification is an action on the vocabulary below, but higher levels can notice structure among the actions of reification.

Another deficit of Cyrano's was that it failed to generate any of the numerical systems which are larger than the natural numbers. For instance, it never defines the negative numbers, although it does reify a version of subtraction into a language whose objects are differences. One possible way to add a mechanism that would allow such expansions is to look for sets of reified representations which do not conflict on where the assign particular behaviors. For instance, reified subtraction and addition only share the identity operation and there is one 'object' in each corresponding to this object. Having identified such sets, the union of them may potentially form a larger vocabulary with interesting properties.

In contrast to Douglas Lenat's seminal AM program, Cyrano invents neither the prime numbers nor the related conjectures about unique factorization. At first, this was done because I could think of no principled way to invert multiplication (AM's algorithm for inverting operations had an error which made its definition of prime numbers technically incorrect; it might have misidentified some composite numbers as prime); but eventually, I had to be just as unprincipled to implement reification (which requires the implicit computation of an inverse). But AM also had a great deal of specific knowledge about sets and set operations which were used in its construction of the primes; in its current incarnation, Cyrano doesn't have such knowledge and thus cannot discover those same regularities.

Part of Cyrano's simplicity and effectiveness may come from the very areas of its ignorance; a uniform representation of sets might require a more complicated control structure for restricting formulations. There is generally a tradeoff in any representational system between simplicity and power on the one hand and ultimate

coverage on the other; this applies to the design of Cyrano as an AI program, but it also applies to the processes of discovery itself. If we are seeking simple and effective systems of description (for invention or prediction) in a large space, we can only seek them in a space which is less simple and effective than the systems themselves; this places pragmatic limits on the inventive processes that must function in these neccessarily less productive spaces. An effective representation owes its effectiveness as much to the things it 'does not see' as to the things it 'does see', but the criticism and improvement of the representation must lose this effectiveness to avoid this same blindness.

Chapter 4
Vocabulary Invention
Limits and Possibilities

"For a variety of reasons, perhaps best understood by psychoanalysis, when we talk or write about scientific discovery, we tend to dwell lovingly on great events – Galileo and uniform accelaeration, Newton and universal gravitation, Einstein and relativity. We insist that a theory of discovery postulate processes sufficiently powerful to produce these events. It is right to so insist, but we must not forget how rare such events are, and we must not postulate processes so powerful that they predict a discovery of first magnitude as a daily matter.

On the contrary, for each such event there is an inverstment of thousands of man-years of investigation by hundreds of talented and hard-working scientists. This particular slot machine produces many stiff arms for every jackpot. At the same time that we explain how Schrödinger, in 1926, came to quantum mechanics, we must explain why Planck, Bohr, Einstein, de Broglie, and other men of comparable ability struggled for the preceding twenty years without *completing this discovery. Scientific discovery is a rare event; a theory to explain it must predict innumerable failures for every success."*

Herbert A. Simon,
"Scientific Discovery and The Psychology of Problem Solving",
[Simon, 1966]

This chapter discusses the motivations and consequences of emphasizing vocabulary invention in the discovery process. It is widely accepted in Artificial Intelligence that a program's representation has has enormous consequences for its performance. A good representation for an AI program makes simple and perspicuous the statement of solutions to the problems which the program addresses. A representation where generated descriptions are likely to satisfy some particular predicate is said to be *dense* with respect to that predicate. It has often been argued that — given the right representation — any AI problem can be characterized as a search problem; this is because the right representation ensures that the earliest (and hence most likely) steps of a potentially combinatorial search are likely to yield desired results.

Given the importance of problem representation, it is vitally important to consider the processes by which representations are invented or transformed; but here

we have a curious problem because just as a representation makes certain solutions more accessible, it also makes certain transformations more accessible than others. I refer to representations with potential for transformation as *vocabularies*; the term 'vocabulary' is meant to capture the intuition that our systems of description not only 're-present' the world but have deep implications for the questions which one asks or the modifications which one considers.

Informally, a vocabulary is not merely a formal specification of a set of statements, but involves both standards of preference and standards of reference. The standards of preference are some explanation which makes certain constructions more likely to be considered than others; if the representation is a sort of recursive grammar, these standards may simply be standards of descriptive parsimony. The standards of reference are more complicated, involving the way in which constructions in the representation *refer* to objects in a putative model or experiments, actions, and perceptions in the world.

The problem of vocabulary invention is different from the problems of concept formation or invention as traditionally studied in Machine Learning; in forming concepts from empirical data, it is generally assumed that the formed concepts will have a relatively simple expression in terms of a known set of basic primitives. In vocabulary invention, the problem is as much to identify an appropriate set of primitives as to determine those combinations deemed valuable. In some ways it is somewhat misleading to discuss the 'problem' of vocabulary invention since there is so much latitude in what counts as a 'solution'.

A program designed for the continuing invention and application of new vocabularies has a different structure from one seeking to identify regularities and structure in a vocabulary which is initially fixed. Its structure is like the structure of an inductive demonstration in mathematics; its starting representation must have particular properties to which its mechanisms can be applied, but the product of those mechanisms must (at least potentially) have the same properties. In particular, both the starting vocabulary and the generated vocabularies must be *dense* in that simple formulations lead to significant results. (Figure 4.1).

Since discovery processes must labor under maintaining this density requirement, we might ask about the consequences of this for the scope of the representations they may invent and the limits to the eventual progress of vocabulary invention from a given basis. This chapter addresses these questions by introducing a distinction between the *generative* and *descriptive* efficacy of vocabularies and pointing out that a vocabulary can become more effective at generating results while become less effective at describing a domain in general and its own coverage in particular. This 'useful blindness' consequently means that the space of vocabulary invention must be space much less effective than the space defined by any particularly effective vocabulary.

The invention and use of vocabularies is prone to an *entrenchment* which artificially diminishes both the likelihood of realizing the vocabulary's inadequacy and the

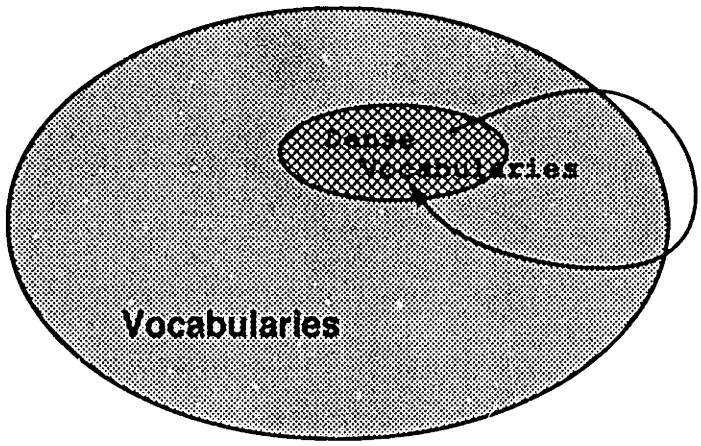


Figure 4.1: Discovery processes must produce and act upon vocabularies which are *dense*; simple constructions in such vocabularies yield significant results.

apparent feasability of valid or better alternative vocabularies. If a system is searching for new vocabularies, this diminished feasability will effectively bar the consideration of valid alternatives. In the invention and application of vocabularies, there is a constant tension between effectiveness and flexibility; if we accquire a vocabulary for generating certain sorts of results in a domain, we at the same time rule out (practically) whole families of results which may be equally valid. But this does not mean that it is unwise to ever form or use vocabularies for fear of entrenchment; indeed, without forming vocabularies — systems of biased generation — we can never get anywhere at all. There is merely an essential tension between the isolation of powerful and simple vocabularies and the flexibility neccessary to uncover vocabularies which are simpler *and* more powerful.

4.1 The Importance of Representation

Naming has power. The organization of our world into systems of description allows us to multiply our manipulations of those systems into manipulations of the world which we would otherwise never attain. By making important things saliently and compactly expressed, naming serves as an aid to both our imagination and our memory. I vividly recall the amazement I felt as a freshman physics student — struggling with a problem involving a hilltop and a cannon — when a classmate began his problem solving by saying "We'll just rotate things 45° and have gravity going down this way...". When so transformed, the problem's answer quickly yielded to methods I already possessed and I realized the profound way in which the description of a problem abets or frustrates its solution.

But naming also has power over us. I was already applying quite a powerful

framework — analytic geometry — to my problem when my classmate relaxed one of my assumptions and saved the day; had I not been quite so confident and facile in that framework, it is possible that I would have sought perspective and not continued to struggle until my classmate's unbidden arrival. And the power of a representation is not merely our affective adherence to it; a representation gives us power at the cost of making awkward the expression of alternatives. This very awkardness may blind the user of the representation to its restrictions.

In Artificial Intelligence, the importance of representation is universally acknowledged; Winston's textbook puts it well: "... The much greater perspicuity and the inherent thinking advantages of powerful representations enable progress that would be impossibly difficult with anything less adequate..."[Winston, 1984, Page 22]. While in some sense, nearly all representations are ultimately equivalent (any representation powerful enough to describe a computer can describe a computer manipulating a different representation), the difference is that a good representation make certain things easy to say; a property less commonly identified is that equally important are the things it makes *difficult* to say.

An effective representation is often a mark of the 'quality' of AI programs operating in particular domains; but in discovery systems, power and parsimony of representation often leads critics to label a program like Lenat's AM as a 'wind-up toy' or to describe a program like BACON as 'trivial'. But if we consider the focus of a discovery program to be the way it invents representations and not what it does with its given representations, the weight of such criticisms diminishes.

AI's own successes have hinged so much on representation that its critics have often assailed the dependence on these representations as the Achilles heel of AI; systems have been limited to 'microworlds', according to these skeptics[Dreyfus, 1979], because when representations break down (as they invariably do) a system is stuck without options for change. Indeed, recent analyses of the successes and failures of expert systems have highlighted this same brittleness of representation; but rather than taking this as an argument against representational approaches, they take it to be a question about how systems can 'step back' from their representations to a place where a brittle representation can be extended or patched by either appeal to general principles [Davis, 1979, Laird *et al.*, 1987] or analogy to broad world knowledge[Lenat and Shepard, 1990].

The position that the the invention of vocabularies and the formulation of problems is an important but in-principle mechanizable problem is by no means new in Artificial Intelligence. But rather than trying to invent representations which simplify the solution of particular problems, Cyrano focusses (like programs such as AM [Lenat, 1976]) on inventing new spaces of description in which new sorts of problems can be invented. And like the bulk of researchers in Artificial Intelligence, I believe that such mechanisms are in-principle mechanizable.

However, I am not confident of an easy solution. Below I describe how nearly any

representation can become more and more effective while unwittingly losing important flexibility and coverage. The fact that a representation both limits and empowers us often makes it impossible — based on the representation's performance — to accurately judge the representation's correctness.

The space of representations is a space of radically weaker constraints than exploration within any 'domain' space; because we cannot commit to any powerful representation without fear for losing our ability to assess it, we must generate and consider representations in a space less effective than that of any individual representation we might consider. Thus, the space of representations — if searched by criteria of parsimony and accuracy — is riddled with local maxima of increasing conciseness and effectiveness which in fact move away from an optimal representation. While we may have heuristics for improvement or adaptation, they are at best only principles whose true mettle will only be revealed when they fail.

4.2 Generative and Descriptive Efficacy

The power and limitations of representations arise not from their abstract and 'disconnected' description of the world, but from how they are used to shape questions and consider alternatives. In the beginning of this chapter, I introduced the term 'vocabulary' to capture these generative connotations of representations which are used. A vocabulary is not merely a set of terms, connectives, and rules for describing some class of claims, but is also the set of biases and criteria that guide the formation of such claims. A trained scientist or an educated layperson in one field may be able to read papers and follow arguments in a quite different field; but this does not neccessarily imply the ability to generate effective hypotheses or rule out classes of experiments in the way an expert in the field might. The slaveboy in Plato's *Meno* was able to follow the logical structure — when prompted and given foci of description — of a claim in a proof system with which he had no familiarity; but had he been merely asked to prove the geometric properties without guidance, he would have floundered and gotten nowhere.

When I speak to a scientist in another field, she may describe her research in a way that I can understand; I will know why some of the questions she is asking are interesting, why she must apply certain controls, why a certain result might indicate that another result was inaccurate, or how certain results might suggest certain follow-up experiments. Indeed, I can even generate other suggestions for experiments or questions with which she would agree. But on the other hand, some of the experiments I might suggest would be impossible to perform or hinge on distinctions not considered interesting in the field; and there are some questions I would not ever consider because they involve issues or 'open questions' with which I am not familiar or because they involves assumptions of 'uninterstingness' I inherit from my own field. While I might easily be able to follow my friend's experiments and arguments, I could not reasonably

sit down in her lab and do her experiments and advise her students!

This illustrates a distinction between *descriptive* and *generative* efficacy in a vocabulary. A theory is not merely a way of making predictions about the world, it is also a way of *making other theories* and from this arises the distinction between descriptive and generative efficacy of a theorie's vocabulary. This distinction is made particularly clear in Cyrano, where the system begins with a universal language whose 'descriptive adequacy' embraces all that the system might eventually discover. However, the system's 'generative adequacy' is determined by the criteria of simplicity it is either given or has evolved over time. At any point, I can give an arbitrary definition to Cyrano as a highly valued primitive; from this 'revelation,' Cyrano would dutifuly generate extensions and variations and seek to identify coherence among these consequences and the definitions it already had. But it will not find any such coherence without the auxiliary concepts from which the definition had been constructed; to use the definition and realize its sigfnicance, it must develop the generative adequacy connected with it.

The power of a vocabulary — characterized in this way — lies in its generative adequacy. A good vocabulary generates a profusion of effective alternatives and options for exploration. But this fertility is paid for in the coin of flexibility; the assumptions and biases which enable my friend to ferret out effective directions for her research can not be uniformly translated to *my* field and applied to *my* questions. For her and my own biases are as much about what things *not to see* as things to see and the translation of such restrictions is unlikely to be effective. Such cross fertilization may be possible, but usually only at the cost of diminishing the apparent 'simplicity' of the biases and results by 'letting in' things that would otherwise be excluded.

The focus on generative efficacy is especially important given the universality of representational mechanisms like Turing Machines. A Turing Machine can 'simulate' any other Turing Machine; if our representation is Turing-equivalent (which is hard to avoid), it is extensionally equivalent to all other such representations. Thus the criteria of extensional equivalence loses its utility and leaves us with a criteria of descriptive or generative parsimony; how well does the representation's parsimony correlate with the actual domain we are seeking to represent? It is often understood that vocabularies allow and disallow certain things; but in fact, they generally disallow them only by criteria of parsimony and not in-principle distinctions.

To pick an absurd example as a limiting case, we could propose that Aristotlean Mechanics were 'literally true' and that we were merely being fed impressions — or even simulated by — a Turing Machine which simulated Newtonian or relativistic mechanics. By such an immense contrivance, we can 'explain' the phenomena of one particular sort of mechanics in the framework of a very different sort of mechanics. However the complexity of this contrivance reveals that the criteria of any theory is not merely its formal agreement with observation but implicitly involves at least some

standards of feasability and simplicity.

On a more realistic plane, this in-principle equivalence of representations or theories is similar to the ability of a person to learn to think in another representation just as they may learn to speak and comprehend another language. An active user of Aristotlean mechanics could understand Newtonian mechanics, but only by learning what amounts to a new language and not by translating into the language she already knows. By a similar translation, an active user of Newtonian mechanics can — by the same effort of learning another language — understand how someone might embrace and believe Aristolean mechanics.

This distinction between translation and 'learning a new language' is explored in detail in [Kuhn, 1983] as an account of the incommensurability of scientific theories. Thus I expect that having a vocabulary makes it possible (though perhaps difficult) to 'learn' any other vocabulary; and — as with most cases of learning — the learner is not really quite the same when she finishes.

4.3 Density and Convergence

This section sketches an attempted formalization of the distinction between generative or descriptive efficacy introduced in the previous section. This formalization allows the description of a phenomena called *entrenchment* where a vocabulary's generative efficacy masks its descriptive inadequacy and thus diminishes its ultimate flexibility.

I introduce an account of representation and then introduce two evaluations of represenations with respect to the world they are taken as representing: the *density* of the representation, and the *convergence* of the representation. Density is a property I have referred to above; a dense representation is likely to yield significant results. Convergence is a combination of density with a constraint on *coverage*; it is the degree to which measures of simplicity or complexity in a representation correspond to simplicity or complexity in the 'world'.

Like any formalization, the efforts below gloss important distinctions and some of the definitions may be contentious. Yet the tradeoff between density and convergence applies to any account of representation which accurately captures the way our representations influence the questions we ask. I expect that future work will generalize parts of the model represented here.

In discussing representations, we must have an account of the thing we are representing. I represent the world as a set of objects $\mathcal{W}$ with a probability distribution $\mathcal{T} : \mathcal{W} \rightarrow [0,1]$ where $T(w)$ is the probability with which the object w appears in random samples of $\mathcal{W}$. Members of $\mathcal{W}$ may be construed as encountered objects, events, pairings of events, etc. In what may be the most contentious point of this formalization, I assume that these members of $\mathcal{W}$ are independent. As a thumbnail justification of this assumption, I argue that one can transform any space of events

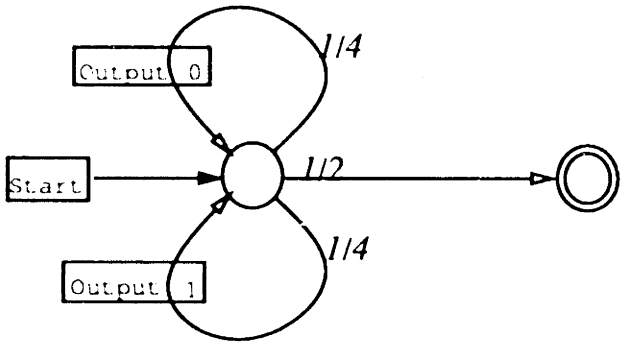


Figure 4.2: This probabilistic finite state machine produces the space of binary strings $\{0,1\}^*$ with the probability of producing a particular string s is $2^{-|s|}$ where $|s|$ is the length of s.

$\mathcal{W}_1$ where some events are dependent into a space $\mathcal{W}_2$ consisting of pairs of events and thus remove much of the dependence into explicit probabilities assigned to pairs.

Given a world described as above, I introduce a representation as consisting of a space of representation objects $\mathcal{R}$, a representation function $M : \mathcal{W} \rightarrow \mathcal{R}$ from world objects into representations, and a generative bias $Q : \mathcal{R} \rightarrow [0,1]$ which indicates the probability that a user of the representation will consider a particular element of $\mathcal{R}$. One possible representation might be the space of binary strings $\{0,1\}^*$. One might imagine these strings being produced by a Markov process modelled by the state graph of Figure 4.2. In this case the bias function Q assigns higher probabilities to shorter strings; $Q(s) = 2^{-|s|}$ where $|s|$ is the size (in bits) of the string s.

The representation function M is a strict many-to-one function; representations are ambiguous but not redundant. In Section 4.6, I will note that maintaining multiple redundant representations is one way to reduce the danger of representational entrenchment; but in order to describe the phenomenon to start with, I define representation in this manner.

The properties of density and convergence are both properties of representations with respect to a world it is representing. Briefly put, the density of a representation is the probability that the reference of randomly generated representation will be 'out there' in the world; the convergence of a representation, on the other hand, is the number of guesses required to identify the representational token corresponding to a given object in the world. The two measures derive from different models of the way a representation interacts with the world. Representational density is based on a 'one-shot guess' model; representational convergence is based on the 'twenty questions' model where a presumably unchanging sample is tested by a series of questions.

Suppose we have a skeletal world $\mathcal{W}$ consisting of the roman letters 'A' through

'E' with associated probabilities of appearance based on English text. We represent this world with a space $\mathcal{R}$ of the Greek letters 'α' through 'ϵ' using a simple identity of ordering. The density of this vocabulary is revealed in the following game: player A selects a random roman letter, which player B guesses; once player B has guessed, she has either won or lost and player A selects a new letter. The density of the representation is the number of times B wins by guessing correctly. The convergence, on the other hand, is revealed in the following game: again, player A selects a roman letter at random; however, this time, player B guesses repeatedly (while not keeping track of previous guesses) until she correctly identifies the letter. The convergence of the representation is the expected number of questions which A has to ask.

We can compute the density of a representation by a straightforward application of probability theory:

$$Density = \sum_{r \in \mathcal{R}} Q(r) \cdot \sum_{w \in M^{-1}(r)} \mathcal{T}(w)$$

where $M^{-1}(r)$ indicates the set $\{w \in \mathcal{W} : M(w) = r\}$. Assuming that we are only considering one r at a time, this probability is the sum — over each $r \in \mathcal{R}$ — of the probability that we consider that particular r times the probability that one of the $w \in \mathcal{W}$ represented by r is actually present. Since M is a strict function, we can express the nested sum above as a single sum over $\mathcal{W}$:

$$Density = \sum_{w \in \mathcal{W}} Q(M(w)) \cdot \mathcal{T}(w) \tag{4.1}$$

An obvious consequence of this characterization is that we can increase the density of a representation by focussing on the more common phemonena and either disregarding or seldom attending to the less common phenomena. As an example, suppose we are using the alphabetic $\mathcal{W}$ and $\mathcal{R}$ introduced above; our best strategy for guessing would be to always guess 'E', since in English 'E' is the most common letter, the density of our representation would be:

$$Q(\text{'A'}) \cdot \mathcal{T}(\text{'A'}) + \ldots + Q(\text{'E'}) \cdot \mathcal{T}(\text{'E'}) + \ldots + Q(\text{'Z'}) \cdot \mathcal{T}(\text{'Z'})$$

and the most 'profitable' place to put our guesses is in $Q(\text{"E"})$. However, such a scheme would not be very good for generating descriptions of particular English strings or even for generating sequences of letters which might look like English text.

This is where the base distinction between generative and descriptive efficacy emerges. A representation may be dense with significant results but this density may only be at the cost of its effectiveness at covering all of the things it putatively describes. To describe this loss of coverage, we move to another measure of representational quality: the *convergence* of the representation.

Like density, convergence is defined with respect to the world one is representing, but with a different model. In characterizing convergence, we assume that the world

'sits still' for us; we have an object in w and we wish to identify it as quickly as possible by asking a series of questions in r. In our model, we assume that we do not keep track of which questions we have asked already; this will have a definite effect for small representations, but for larger representations the effect is minimal. The average number of questions that a representation user must ask to identify an object is the *convergence* of the representation, computed as follows:

$$Convergence = \sum_{w \in \mathcal{W}} \mathcal{T}(w) \cdot \sum_{i}^{\infty} (1 - Q(M(w)))^i$$

as the sum of the probabilities of a particular $w \in \mathcal{W}$ occurring times the average number of guesses that must be made in $\mathcal{R}$ before hitting that particular w. Collapsing the geometric progression of the inner sum, we get

$$Convergence = \sum_{w \in \mathcal{W}} \frac{\mathcal{T}(w)}{Q(M(w))} \tag{4.2}$$

which displays an interesting symmetry with Equation 4.1. But while increasing density means a representation which is better at getting results, increasing convergence — how many questions one must ask to identify an object — means the representation is getting worse at identifying an arbitrary object.

Convergence illustrates the negative effects of 'artificially' enchancing the density. Assigning small (or zero) probabilities to particular representational strings at the expense of others will drive up the convergence. Thus, when a representation is made artificially denser, the convergence will rise. The minimal (and thus best) convergence for a representation is attained by making:

$$\mathcal{T}(w) = \sum_{r \in M^{-1}(w)} Q(r)$$

Note that if we used a representation of binary strings for members of $\mathcal{W}$ (i.e. $rep \subseteq \{0,1\}^*$) and minimized its convergence, the corresponding representation would be optimal and its density would correspond to the entropy of $\mathcal{W}$ under $\mathcal{T}$. This gives us a rule of thumb for assessing a representation whose density we have computed; if the representation is far denser than the entropy we expect that the universe has, we have probably made the representation artificially dense. However, the problems of identifying the actual entropy or convergence are difficult when we must do it through the window of our own representations.

4.4 Assessing Representations

In comparing the equations for density and convergence, we see that the density can be directly approximated (by counting the ratio of 'yes' answers that a representation yields) while convergence requires some direct access to the world to be

adequately approximated. This section considers how the approximations we may make of the convergence are effected by increasing a representation's density (effectiveness). It demonstrates that as the density of a representation increases beyond the entropy of the world it is representing, the quality of the approximation to the convergence decreases rapidly.

The straightfoward simplification of the density equation (Equation 4.1) simply moved an inner sum outward. The density of a representation is most effected by how its most common tokens work, so a sum over the representation will effectively approximate the actual density of the representation. The convergence, on the other hand, is most effected by the representations which are least common; as a result, summing over the representation is unlikely to work. Indeed, if we sum over the representation and use that to generate samples of $\mathcal{W}$ to compute the divergence from, we end up computing:

$$\sum_{r \in \mathcal{R}} Q(r) \cdot \sum_{w \in M^{-1}(r)} \frac{\mathcal{T}(w)}{Q(r)}$$

or simply

$$\sum_{w \in \mathcal{W}} \mathcal{T}(w)$$

which is minimally effected by the least common elements of $\mathcal{W}$. Thus, the actual convergence of a representation is unavailable to the representation user because the use of the representation masks any assessment of its efficacy. How then, can we detect a representation which grows overly dense?

One hint comes from the 'optimal representation' discussed above; if the density of our representation is much better than the entropy we expect in the world, then we are probably focussing on a subset of the world whose entropy is manageable. The entropy of the world is:

$$\sum_{w \in \mathcal{W}} \mathcal{T}(w) \cdot \log \mathcal{T}(w)$$

but as above, we cannot sum over $\mathcal{W}$ directly, and so we really compute

$$\sum_{r \in \mathcal{R}} Q(r) \cdot \sum_{w \in M^{-1}(r)} \mathcal{T}(w) \cdot \log \mathcal{T}(w) = \sum_{w \in \mathcal{W}} Q(M(w)) \cdot \mathcal{T}(w) \cdot \log \mathcal{T}(w)$$

which is a better approximation than the correspondence approximation, but in a sufficiently entrenched representation (where, for instance, some $w \in \mathcal{W}$ are never considered) we can even have an illusion about the entropy of the world when seen through the veil of our representations.

The central point of these descriptions is that a representation must be assessed from the standpoint of a weaker representation; a representation judging itself may easily become entrenched in particular biases which affect its judgement as well as its performance.

4.5 An Example from Cyrano: Rebiasing

The above examination of entrenchment began as an effort to understand a set of experiments where Cyrano became stuck in particular representational ruts. This section describes those experiments as an example of how Cyrano in particular can *become* entrenched.

[Solomonoff, b] describes an appealing approach to reformulating vocabularies for systems which generate computer programs. In this scheme, a system generates a series of definitions in order of decreasing 'probability' where each atomic symbol is given a probability and the probability of a sequence of symbols is the product of their individual probabilities. At some point, a subset of the generated definitions is selected and analyzed statistically; the elementary symbols are given new probabilities based on their frequency in the selected set of examples. In addition, frequencies are computed for symbol pairs; any extrema in this frequency analysis are used to define new primitive symbols with probabilities equivalent to their associated frequency. [This may not be completely correct.]

In the first experiments with Solomonoff style generation in Cyrano, this rebiasing algorithm was used. The generating language included one operation `CONTROL` which took a *controller* and a *controlee* procedure as inputs and returned a new procedure `instantiate-controller` of two or more arguments; `instantiate-controller` procedure calls the *controller* on the first input to produce a procedure which is called on the *controlee* to produce a final procedure to be applied to the remaining arguments. The idea is that the controller uses a domain input to control the actions of another domain procedure.[1]

For instance, one common controller procedure used one operation to count down the occurences of another operation; e.g. `ITERATE-BY-DECREMENT` was a controller which took an input i and returned a procedure which accepted a procedure p and called p recursively i times. (`ITERATE-BY-DECREMENT` had been produced by the `ITERATOR` operator.) But until operations like `ITERATE-BY-DECREMENT` had been defined, `CONTROLLER` could not be successfully used. When running these experiments with Cyrano, if reformulation occurred too early, the frequency of `CONTROL` was brought to zero because the prerequistes for its use had not yet been generated.

If we look at rebiasing as attempting to generate a language corresponding to a desired corpus of examples, the problem with the simple frequency-based rebiasing approach is that it assumes that the frequency of useful symbols or substrings is uniform across strings of all lengths. Unfortunately, there are whole families of grammars and systems without this constraint. Consider what might be called *hierarchical languages* where a particular symbol k in a language A are replaced by the strings of a language B. If A and B are regular languages, such a language could be generated by taking a finite state machine generating the language A and replacing the

[1] In the current version of Cyrano, this has become (with some limitation) the `limit` operation.

transitions which generate k with a copy of the FSM which generates the language B. Such languages describe a wide variety of designed systems: engineered devices, large computer programs, social structures. We would like our systems to evolve such structure for their own good; such hierarchries assure a fair degree of flexibility by virtue of changes isolated to particular levels.

If we are trying to generate a hierarchical grammar, we cannot know its real entropy unless we make assumptions about the depth of the hierarchy. [Is this true?] Thus the problem is that — by some metric — it is never safe to reformulate in so radical a fashion that you throw away your primitives.

4.6 Multiple Representations: Heuristics for Avoiding Entrenchment

Are we stuck? It seems that the move to a denser and more effective representation is always potentially dangerous because it may marginalize things which might be important in the future. Indeed, we cannot be sure that our current accounts of the world — scientific, personal, or cultural — are not entrenched in a way which makes our appearance of progress an illusion. On the other hand, without a concise representation, we cannot productively generate important predictions or definitions.

The problem of developing an ultimately correct representation, given some assumptions or assertions about the entropy of the universe, may be susceptible to formal analysis. By starting from a highly redundant representation, an assumption about the world's entropy, one might move cautiously to less redundant representations, eventually converging on a representation which is increasingly accurate yet provably not entrenched. But even so, it is clear that such a mechanism was not used by any of the brilliant individuals or effective communities who developed and eventually repudiated the visions and models of history; thus, addressing these issues heuristically is of both technical and historic interest, as we seek to understand how the structures of science and evolution both work today and have worked in the past.

Here are a handful of ideas about how to avoid representational entrenchment in a system:

- **Representational Plurality.** Having a variety of representational systems means that the tendency of one of them to become entrenched will be diminished by the need to correlate it with others.

- **Isolation of Adaption.** If the mechanisms for evolving a representation have limited scope, developments (good or bad) in one will not quickly spread to others.

- **Hierarchies of Density.** If a dense representation is maintained in combination with a less dense one, the less dense representation may notice and recover when its denser companion starts to get stuck.

- **Historical Criticism.** By keeping track of the way our representations evolve, we can identify phenomena which might be unjustifiably marginalized.

- **Publishing Results.** The attempt to translate results to other vocabularies — for instance, the understanding of other scientists — forces a move to a less dense and more flexible vocabulary.

- **Focussing on Anomaly.** Entrenchment occurs as we develop theories which focus on those phenomena we can explain and marginalize those we do not. Focussing on anomaly causes us to try and develop representations and explanations for exactly the phenomena which are difficult to describe.

All of these measures focus on the use and coordination of multiple representations; they are at best heuristic, since it is possible for two representational schemes to both be entrenched in the same way. But multiple-representation approaches (which are equivalent to having M not be a strict function into $\mathcal{R}$ but rather map into subsets of $\mathcal{R}$) makes the representation less likely to be entrenched (because it both lowers the density that can be acheived) and more sensitive to error because demanding that there be a distinction between several $M(w)$ strings increases the information content demanded of the representation. In particular, it asserts that two distinct representations are not 'really' redundant.

We can implement one version of representational plurality by using similar representations and isolating their adaptions and evolution; when the correlations of the two (initially identical) representations begin to diverge, it may indicate that one or the other may be becoming entrenched on 'non-phenomena' that are products of its representation and not of the space it is representing.

Another approach to representational plurality is to maintain representations at several levels of density; if a more dense representation becomes entrenched in its own assumptions, the less dense representations will provide important flexibility when the entrenched vocabulary begins to break down.

A particular form of this hierarchical approach involves the use of 'historical criticism' to recover the less dense (or dense in a different fashion) representations which preceded the current representation. The historical context is one which includes both the current representation and the representations which preceded it; as a result it is necessarily less dense than either.

But from the point of view of any individual representation, such recovery may be a 'step backward', tossing out hard won elegance or effectiveness that will not survive under a reformulation of what counts as evidence or justification.

In each of these, we presume that information and results are somehow being coordinated between the different representations; one important way of avoiding entrenchment is to attempt the translation of one's ideas into other vocabularies. This act of translation automatically moves into a less dense space, providing we

have not mistakenly reduced the other vocabulary to a still denser subset of our own. Thus the act of 'publication' is not only important for the circulation of results, but is vitally important for avoiding entrenchment in one's own biases.

One mechanism for avoiding entrenchment which is used in the human communities of mathematics and science — but not in Cyrano — is the focus on anomaly. By focussing on phenomena which we know our vocabulary does not describe very well, we are directly addressing the tendency to entrenchment within our representations. The history of science is filled with the development of methods and technologies for exploring the areas we do not know in a search for confirmation or disconfirmation of what we believe to be true.

Chapter 5
AM
An Exegesis

This chapter offers a detailed description of Lenat's AM program in order to focus on some important issues which have generally been glossed over in the literature. In particular, I believe that the understanding of 'discovery as vocabulary formation' will be clarified by focussing on aspects of AM which are usually misunderstood. In particular, I understand AM's great innovations as those which led it to characterize 'new domains' as opposed to the enhancement of performance in a given domain. Cyrano picks up on the former of these innovations, while most critics and follow-ups of AM (including Lenat's own Eurisko) have dwelled on the latter.

I begin with a detailed exegesis of the 'high points' in AM's progress; I then address the places where AM 'formed new vocabulary' in a deep way and also where AM subsequently fumbled in this process. I then discuss the development of the Eurisko program from AM and the development of Cyrano based on the results of both AM and the development of Eurisko.

5.1 The AM Program

The abstract of [Lenat, 1976] neatly summarizes the performance of AM:

> *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 behaviour, 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. Rpeatedly, the program selects from the agenda the task having the best suporting 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).*

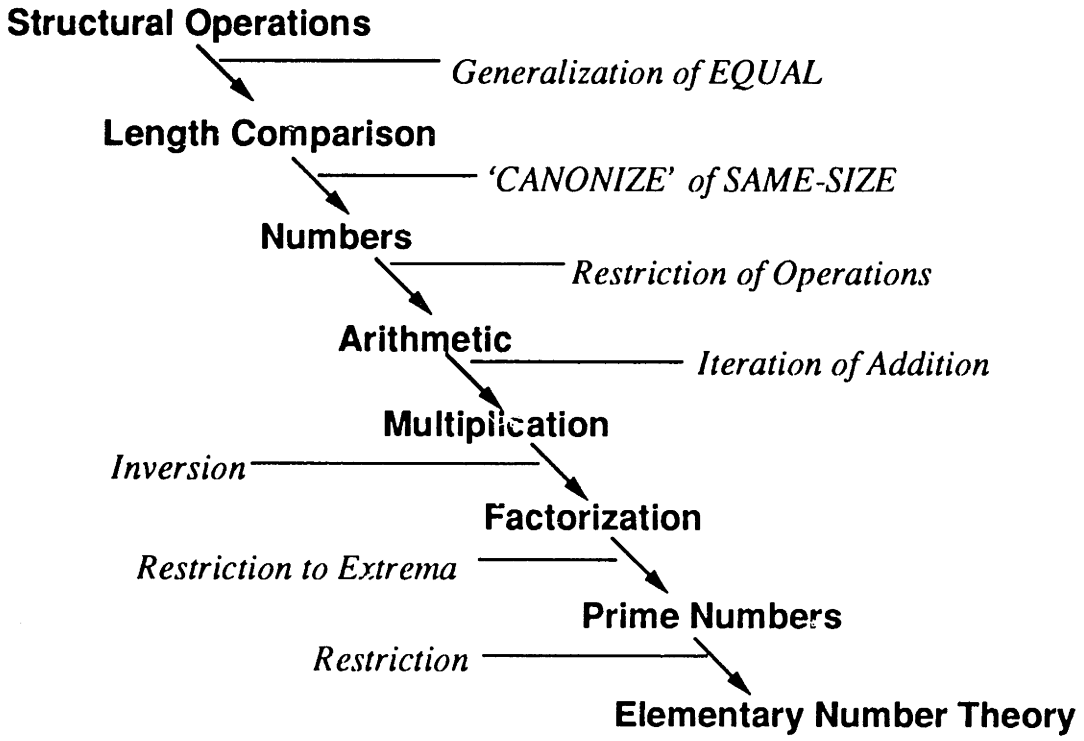


Figure 5.1: In the space of a few hours, the AM program went from simple operations on structures to significant conjectures about number theory.

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

Given an initial vocabulary of set theoretic concepts and operators, the AM program — through the application of simple AI techniques — proceeds to define and develop concepts including 'numbers', 'multiplication', 'factoring', and 'prime numbers'. Based on the empirical properties of these concepts, AM makes conjectures such as unique factorization (there is one prime factorization for every number) and Goldbach's conjecture (every even number can be expressed as the sum of two primes).

AM's achievements provoked both technical and episitemic questions. How did AM make all of these discoveries? And exactly what are the outputs of the program that one is calling 'number' or 'primes'? In this section, I address these questions by describing a 'good run' of the AM program and highlighting some of AM's particular discoveries; these will be the same discoveries that I laud or criticize in my analysis of the program. A graphic summary of AM's progression can be found in Figure 5.1. The running description is also annotated by boxed comments (in italics) which clarify some aspects of AM's operation.

5.1.1 AM's Starting Place

AM begins with a domain of data structures including sets (unordered collections with no repeated elements), bags (unordered collections with repeated elements; the mathematician's multi-set), ordered sets (ordered collections with no repeated elements) and lists (ordered collections with repeated elements). Operations among and between these data structures are possible, including union, difference, and comparision operations.

AM is also given a collection of operations on operations; for instance, the operation-operation COALESCE takes an operation with two inputs (for instance, the operation SET-INTERSECTION) and defines a new operation (SET-INTERSECTION-SELF) which takes a single input and applies SET-INTERSECTION to this input twice.[1] In the case of SET-INTERSECTION, the new operation is simply the identity; in other cases, it has more interesting behavior. Operation-operations are applicable to operation-operations; we could COALESCE the COMPOSE operation-operation to produce an operation COMPOSE-WITH-SELF.

Each of AM's operations and structure-classes are described by a *concept* consisting of a name and a set of properties. The values of these properties are either random LISP objects or the names of other concepts. The language used to encode AM's concepts is a simple frame language much like FRL [Goldstein, 1976] or UNITS [Stefik, 1979].

All of these concepts are assigned a numeric *worth* which increases or decreases as the concept is found 'more interesting' or 'less interesting' for one reason or another. These worths are used to give *priorities* to *tasks*. AM's tasks apply a small class of operations (FILL-IN, CHECK, etc) on slots of objects; a single task might be (for instance) 'FILL-IN the GENERALIZATIONS of OBJECT-EQUAL' or 'CHECK the EXAMPLES of BAGS'.

AM's agenda mechanism simply selects the task with the highest priority and executes it; the complexity in this strategy lies in numerous mechanisms for computing and modifying worths. AM can be described as searching in a space of concept definitions for concepts of high interest; AM's tasks are steps in this search space, filling out new components of somewhat interesting concepts or searching for reasons that an existing concept should be more interesting.

The execution of a task brings to bear a variety of methods specialized to particular sorts of tasks. For instance, a sophisticated automatic programming module is brought to bear on tasks which FILL-IN the GENERALIZATIONS of predicates or comparisons. Heuristics applicable to a task may also suggest other tasks; e.g. the heuristic for CHECKing EXAMPLES of a definition might suggest, based on their empirical rarity, that a task be created to FILL-IN the GENERALIZATIONS of the definition.

[1] Precisely, the coalesced version $g(x)$ of some operation $f(x, y)$ would be defined $g(x) = f(x, x)$.

Much of AM's sophistication lay in the implementation of these heuristic methods; furthermore, their individual implementation — while consolidated in English language rule-summaries in [Lenat, 1976] — was often divided across several procedures or modules in the actual implementation.

5.1.2 Inventing Cardinality

AM began by generating examples of the operations and classes it was given; in this process, it examines the abstract definitions of sets and decides that some simple cases of sets are quite interesting. In particular, it notes that the empty set, the singleton set, the doubleton set, and the tripleton set are of some interest due to their extreme 'simplicity.'

AM decides on these particular definitions by examining the inductive definition of a set: a set is either the empty set or a set with one element added. The first few finite unwindings of this recursive definition correspond to the simple sets which (in Lenat's terms) are extreme examples of sets because they 'barely' pass the criterion of set-ness.

After some time, AM notices that the comparison operation for objects in general — OBJECT-EQUAL — is rarely satisfied; few pairs of objects pass its determination and AM decides to attempt to generalize it. This generalization is shown in Figure 5.2; by removing one clause (the boxed clause) from the doubly recursive definition of OBJECT-EQUAL, AM produces a comparison which compares the length of two structures and ignores the identity of individual elements. AM names this new implementation OBJECT-EQUAL2, but it is quickly renamed SAME-SIZE by Lenat.

In terms of AM's tasks and representations, the invention of SAME-SIZE began with AM executing a task to FILL-IN the EXAMPLES of OBJECT-EQUAL. After having generated a few examples, a task of CHECKing these EXAMPLES of OBJECT-EQUAL noticed their empirical rarity; this rarity suggested creating a task to FILL-IN the GENERALIZATIONS of OBJECT-EQUAL. This final task — applying AM's programming expertise to transforming OBJECT-EQUAL's definition — produced the definition of SAME-SIZE.

In defining SAME-SIZE, AM learns to count by successively removing objects from two structures; it accquires this by generalizing a more specific activity (comparing two objects completely). The relation characterized by this comparison is cardinality;

X and Y are EQUAL if either:
X and Y are the same primitive element
OR
X and Y are both empty structures
OR
The following are all true:
X and Y are both non-empty structures
AND
the first elements of X and Y are EQUAL
AND
with these first elements removed,
the remaining structures are EQUAL

X and Y are SAME-SIZE if either:
X and Y are the same primitive element
OR
X and Y are both empty structures
OR
The following are all true:
X and Y are both non-empty structures
AND
with these first elements removed,
the remaining structures are SAME-SIZE

Figure 5.2: AM produced the predicate SAME-SIZE (equivalent to the cardinal equivalence of structures) by removing the italicized clause from the definition of OBJECT-EQUAL.

if two sets have the same number of members, they are equivalent in SAME-SIZE's terms.

Though Lenat suggests at one point that AM's primitives are akin to those described by Piaget as characteristic of the concrete operational theory, AM's construction of number is quite different from that of a human child. In a human child, the notion of exact identity of sets comes after the notion of identity of size, rather than vice versa. [I think this is true.]

5.1.3 Inventing Numbers

The definition SAME-SIZE turns out to be satisfied much more often than OBJECT-EQUAL and AM defines a *canonical representation* based on it. AM's CANONIZE operation takes a relation r and synthesizes a function f such that $r(x,y) \longleftrightarrow f(x) \equiv f(y)$; i.e. the function f maps objects related by r into the same object in some new representational space.

For SAME-SIZE, AM's invented function transforms any structure into a 'bag of Ts' where 'T' is a standardized symbol; for instance, a set, bag, or list with five arbitrary elements is transformed into a bag containing five copies of the symbol T. The new concept BAGS-OF-Ts is introduced as a specializations of BAGS; this new definition is — in some sense — equivalent to the notion of 'number', since there is exactly one BAG-OF-Ts for every number. Lenat renames the concept BAGS-OF-Ts TO NUMBERS.

> *This step might be considered the high point of AM's progress because it invents a completely new representation and (in transforming operations to this new representation) enters a new domain: arithmetic. The CANONIZE heuristic, unfortunately, had many problems; these will be focussed on in the analysis of Section 5.2.1. Note that the way in which Cyrano invents numbers — the reifications of invented tie structure comparisions — introduces a new class of objects, rather than specializing an existing one. Curiously, however, when Cyrano was given examples of tie structures which share significant structure, it constructs a representation which uses those substructures to identify categories of interest, just as AM (as explicitly programmed) invented such distinguishing tokens. (This was a suprise to me).*

AM restricts many of its definitions to this newly defined class of NUMBERS (BAGS-OF-TS): BAG-UNION becomes NUMBER-UNION, BAG-INTERSECT becomes NUMBER-INTERSECT, BAG-DIFFERENCE becomes NUMBER-DIFFERENCE, and so forth. In the domain of arithmetic, these operations are ADDITION, MINIMUM, and SUBTRACTION. Lenat awards AM's synthesized concepts with the names ADD2, MIN, SUB2.

Ritchie and Hannah [Ritchie and Hanna, 1984] point out that in the published transcripts, AM is implicitly steered when the concepts it invents are renamed by the user. One of AM's heuristics is 'If the user recently mentioned X, X is more interesting'; in AM, renaming constituted mention. While this is a valid criticism, Lenat reports (personal communication) that running AM without renaming led to all the same discoveries, albeit more slowly than when concepts were renamed. The syntactic space of definitions explored by AM is sufficiently small that such 'advice' makes its behaviour more decipherable while not enabling any fundamentally different steps.

AM generates examples of these arithmetic operations in action and notices that SUBTRACT often (about half the time) produces the empty list (or zero in arithmetic parlance). Based on this regularity, AM defines the pairs whose 'difference' is zero, which Lenat calls LEQ (for 'less than or equal to').

Since AM's represented numbers in unary (five was a bag of five Ts), it had no concept of negative numbers. Subtraction, as derived from the implementation of BAG-DIFFERENCE, simpy returned the empty bag when the first bag was a subbag of the second bag; subtracting a larger number from a smaller number (in arithmetic parlance) simply produced zero.

5.1.4 Extending Arithmetic

AM possesses several operators for structural combination and iteration; one of these, called Parallel-Join2, is used to define multiplication. In particular, the operation Parallel-Join2 takes an operation on pairs as input (call it O) and produces another operation on pairs (call it P) which repeatedly applies O to combine each element of P's first argument (a composite structure) with P's second argument. It then appends the results of these individual operations together.

When Parallel-Join2 is applied to the operation Projection2 (a two argument operation which ignores its first argument and returns its second), the result is an operation which appends together one instance of its second argument for each element of its first argument; applied to a list of length 5 and a list of length 3, the new operation returns 5 copies of the 3-long list appended together: a list of $3 \times 5 = 15$ elements. Lenat calls this operation TIMES.

The presence of Parallel-Join2 is not quite as ad-hoc as it might seem; it probably owes its heritage to the InterLISP utility function MAPAPPEND2. The function MAPAPPEND2 applies its second argument (a function of two arguments) repeatedly to its third argument and to each element of its first argument; it returns the appended result of these applications. Parallel-Join2 could be considered a 'curry' (in the functional calculus sense) of MAPAPPEND2; such a mechanism was useful in InterLISP because the lack of lexical scoping made it otherwise difficult to have outside parameters accessible to a mapped function.

Later on, AM comes up with two other ways of defining multiplication which, by AM's standards, makes multiplication very interesting. Quickly summarized, its other paths to multiplication were:

- **Iteration**
  Given two inputs x and y, repeatedly accumulate x for each element of y.

- **Collect and Append**
  This was a two stage discovery which began with the construction of a function like Parallel-Join2 which combined each element of an input x with an input y; but instead of appending the results as in Parallel-Join2, a list of lists is returned. The second stage of this discovery is to decide to compose APPEND with this newly invented function to produce a definition identical to that generated by Parallel-Join2.

AM applies the COALESCE operation (which I described above) to addition and multiplication to get DOUBLE and SQUARE repsectively. The numbers produced by DOUBLE are isolated as a category by AM and Lenat renames this 'Evens'; the numbers produced by SQUARE are also isolated and renamed 'PerfectSquares'.

These various mechanims are reduced to the `limit` *combiner in Cyrano. One curious twist in Cyrano (as compared to AM), is that the state elements of AM's definitions of multiplication are avoided in Cyrano because Cyrano describes multiplication in a vocabulary where the objects are not numbers but addition operations and their iterated composition yields multiplication.*

5.1.5 Factorization

Looking at examples of its arithmetic operations, AM notices that both ADD and TIMES are insensitive to argument order; based on this conclusion, AM generalizes

these operations to BAGs of NUMBERs. At this point, AM also notes an informal analogy between addition and subtraction, although this connection is never seriously used.

AM now considers the inverses of addition and multiplication. In the case of addition, this is the set of all pairs whose sum is a particular number; in the case of multiplication, it is the set of all pairs whose product is particular number. Its algorithms for these inverses are inefficient blind searches which keep trying combinations of addends or multiplicands.

There is nothing particularly interesting about these inverses, but the growing interest in the TIMES operation (prompted by its 4 different specifications) leads to the application of the operator COMPOSE to the UNION operation and the inverse of TIMES; the result is an operation which Lenat calls DIVISORS. Unfortunately, due to the inefficency of AM's blind search for divisors, AM asks for (and Lenat provides) a more efficent implementation.

> *Looking at published descriptions and the available sources for AM, it appears that the blind search for divisors enumerates* known *examples of concepts (like Numbers) in an effort to find combinations which produce the appropriate products or sums. If this is the case, the definition of DIVISORS produced by AM is not completely correct; it is not guaranteed to produce ALL the divisors of a number. A number might exist with factors which had not yet been generated; in fact, as far as the AM program 'knows,' it might be possible that a very large number could be the factor of a smaller number. This might not have made any final difference because the 'efficent' implementation donated by Lenat was probably 'accidentally' correct. Given this incompleteness in AM's derived definition of DIVISORS, its eventual definition of prime numbers is also — while correct in spirit — incorrect in detail; using it's own definition, AM might have misidentified some composite numbers as prime.*

At this point, AM's earlier explorations of sets interact with the newly defined DIVISORS; AM had earlier found the empty set, singletons, doubletons, and tripletons interesting. By combining these two different definitions, AM specifies the classes of numbers with no divisors, numbers with one divisor, numbers with two divisors, and numbers with three divisors. These are those numbers whose divisors are respectively the empty set, singletons, doubletons, and tripletons.

5.1.6 Prime Numbers

AM finds some simple patterns among these definitions: the set of numbers with zero divisors is empty; the set of numbers with one divisor is itself a singleton (this is found uninteresting); the set of numbers with two divisors (primes without one) is somewhat interesting because there are a reasonable number of examples.

The excitment begins as AM notices that all numbers with three divisors are also perfect squares;[2] this makes them interesting enough to consider what their square roots are (AM had earlier invented SQUARE-ROOT by inverting SQUARE). It turns out that the square roots of these numbers are all numbers with two divisors. This regularity boosts the interestingness of this class (now renamed 'Primes' by Lenat) to a point where detailed examiniations of the properties of primes are undertaken.

5.1.7 Properties of Primes

As with the invention of numbers, AM restricts various operations to primes and examines their behaviour. This leads to several notable discoveries. For instance, the operation PRIME-FACTORS is invented by restricting the output of 'inverse of TIMES' to bags of primes. This operation reveals itself to be one-to-one; each number has a single bag of prime factors. This regularity — translated from its operational definition — corresponds to the unique factorization of integers.

Restricting the inverse of ADD to to pairs of primes (as addends) and EvenNumbers (defined via the DOUBLING operation) as sums, AM notes that there is always at least one pair of primes for every even number; this corresponds to Goldbach's conjecture that every even number is expressible as the sum of two primes.

Another of AM's discoveries in this area was the concept of *maximally divisible numbers*; these are numbers none of whose predecessors have more factors. AM did some simple experiments with this concept but found nothing of interest. This was partially due to the lack of more powerful vocabulary with regard to numbers in general. Lenat at first though that this concept was genuinely new — and did some new mathematics based on it — but a search revealed that the Indian mathematical prodigy Ramanjuan had explored exactly the same concept.

5.1.8 AM's Malaise

After a flurry of discovery around prime numbers, the AM program experienced a somewhat sudden malaise; after conjecturing a variety of interesting things in this new domain, the AM program moved back to experiments with basic set theory. These

[2]To see this, consider an integer n with three divisors. One must be 1 and one must be n itself. The remaining divisor p must produce n by repeated multiplication so that $n = p^i (i > 1)$. If $i \neq 2$, p^2 must also be a factor of n. Since n has only three divisors, this cannot be the case so $n = p^2$ making n a perfect square.

didn't get very far and soon AM's measure of its own concepts' worths produced rapidly declining task priorities. As AM itself noted in its lasts moments:

Warning: No task on the agenda has priority over 200!

5.2 AM's Significant Inventions

It was a careful examination of AM and the attempt to re-implement it in the Cyrano-0 program that led to the understanding of discovery as vocabulary formation that is the focus of this dissertation. In looking at AM's progress, we see many places where the program moved to new domains of inquiry by forming new vocabularies: from operations on bags to operations on numbers; from simple combinations of operations to more complicated iterative combinations; from operations on number to operations on their prime factorizations.

AM's essential expertise was the manipulation of list-like data structures representing abstract objects like sets, multisets, ordered sets, etc. Its power came from the breadth of its heuristics for manipulating, experimenting with, and inventing operations on such structures. The design of LISP make the expression of useful operations on lists very simple; in the sense introduced in previous chapters, LISP constitutes a *dense* vocabulary for the specification of set operations. This is the same point made by [Lenat and Brown, 1983] where the underlying similarities between LISP and Mathematics are used to explain the success of the AM program.

[Minsky, 1987] has suggested that the connection between LISP and mathematics actually exists between 'nearly everything' and mathematics; in particular, that in most universal vocabularies, mathematics is 'easy to express'. Indeed, mathematics accquires its power from its wide-ranging application to diverse domains; mathematical results in one domain (even the 'toy domains' in which most mathematical intuitions are uncovered) may have profound implications for more complex domains which can be described mathematically. In [Lenat and Brown, 1983], Lenat and Seely-Brown argue that LISP evolved from mathematics as an explanation of how easily AM rediscovered mathematics; but it may be more accurate to say that both LISP and mathematics evolved in the direction of universality where they converged.

But AM's substrate of LISP and set operations also limited the AM program in fundamental ways; AM's eventual vocabulary, though implemented in terms of list structures and operations, had a significantly different 'semantics' that that for which AM was designed. The following sections describe two of these problems: the dependence of the `CANONIZE` operation on particular data formats and the dependence of the `INVERT` operation on the statistical character of the domains of the operations it is inverting.

5.2.1 The Canonize Problem

In previous chapters, we introduced the notion of a 'cycle of discovery' for describing the progress of systems which devise definitions that can be used as elements in further definitions. This understanding of discovery processes demands that each component of the process be brought to the touchstone of 'closure': does a component produce outputs which can be used by other components and can a component adapt to new vocabularies introduced by other components? In Cyrano, these criteria were addressed by two design constraints: every component of the system had to produce definitions for later use; and each component had to be used more than once in different contexts.

These constraints — and the 'cycle' model that justifies them — emerged in part from an examiniation of AM's `CANONIZE` heuristic. The `CANONIZE` heuristic accomplished what may be AM's most impressive feat: the invention of the natural numbers based on the notion of cardinality defined by `SAME-SIZE` (a generalization of `OBJECT-EQUAL`). The CANONIZE operation takes two related two-place predicates (one is a generalization of the other over the same domain) and produces an automorphism of their domain which preserves the algebraic structure they define over it. Precisely, given $p : A \times A \Longrightarrow \{T, F\}$ and a generalization $r : A \times A \Longrightarrow \{T, F\}; p(x,y) \rightarrow r(x,y)$, CANONIZE finds a function $f : A \Longrightarrow A$ such that $r(x,y) \longleftrightarrow p(f(x), f(y))$. This function f generates a "canonical representation" of A which preserves the equivalence partition defined over A by r. In generating f, CANONIZE recognizes the algebraic structure of A under r and exploits it, but the partition of A is never explicitly and accessibly declared.

AM used CANONIZE to define the canonicalization of bags (multisets) under the SAME-SIZE relation (cardinality) relative to LIST-EQUAL. Given a synthesized notion of SAME-SIZE (a generalization of LIST-EQUAL), AM tried to find a mapping of lists into lists such that lists of the same size would be mapped into lists that were equal. The successful result of this attempt was a mapping (f) which took every element of a list and replaced it with the single symbol `T`. (See Figure 5.3.) `BAGS-OF-Ts`, the range of this mapping (representing the equivalence partitions of SAME-SIZE), was interesting because of its generation from SAME-SIZE and was eventually renamed `Numbers` by Lenat. This one discovery, depicted in Figure 5.3, was the basis of AM's forays into number theory, where all of its more significant discoveries were made.

Despite its immense contribution to AM's progress, the implementation of CANONIZE provides a compelling example of how AM sometimes failed to close the cycle of discovery. It is significant that CANONIZE was only used once by AM: for the representational leap from list structures to numbers. CANONIZE had what might be called an 'output' problem and an 'input' problem. The output problem lay in the canonical representation it invented for SAME-SIZE, bags of Ts; this was not

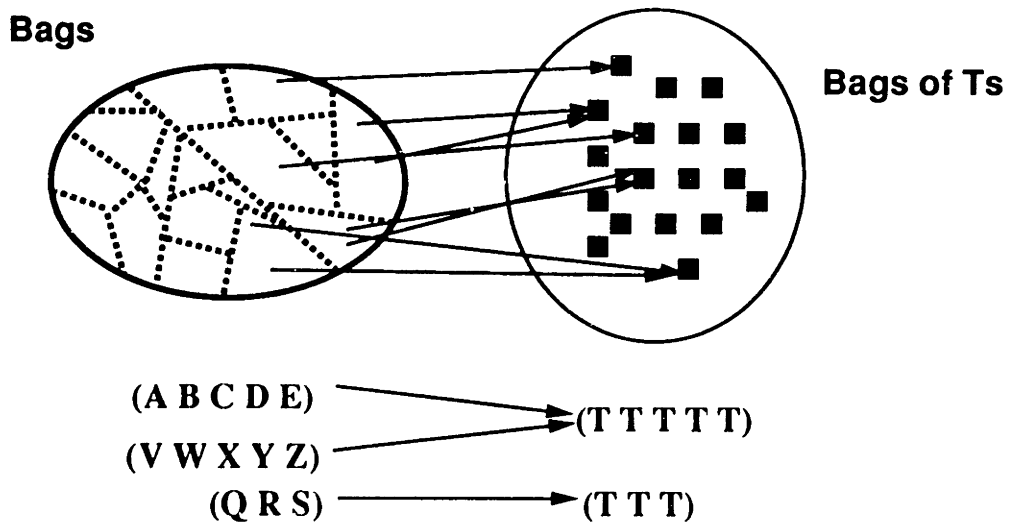


Figure 5.3: AM's CANONIZE heuristic found a partition preserving mapping from the domain of SAME-SIZE to the domain of OBJECT-EQUAL. This mapping transformed each element of SAME-SIZE lists into the unique symbol T, producing OBJECT-EQUAL lists.

strictly a new representation, but rather the specialization of an old one. It might be fair to say that AM did not explore operations on numbers but rather operations on a particular subset of BAGS. Unfortunately, it was difficult to specialize this class much further!

The input problem lay in how AM generated the canonicalizations it did; its mechanisms were intimately tied to operations on simple, relatively flat, list-like structures. AM constructed a canonicalization for a given comparison by experimentally applying the comparison to randomly mutated structures; the mutations to which the comparison was insensitive suggested canonicalizing functions. One of these mutations consisted of changing elements at random; if this did not affect the comparison, the canonicalization of changing every element to some particular element was suggested. However, once AM's representation had moved beyond simple lists and structures, its implementation of CANONIZE was no longer effective. It is revealing that AM only used CANONIZE once, for the revolutionary leap from list-like representations to numbers. With its first application, CANONIZE instantly obsoleted itself.

It can be argued that by installing a test for element invariance and a corresponding canonicalization method, the representation of numbers 'discovered' by AM was actually built in; however, this would be unfair, since Lenat simply enumerated all the simple structural mutations and one of the happened to correspond to a reasonable representation for numbers.

5.2.2 Indeterminate Inversion

Another highlight of AM's progress (and the one most focussed upon in discussions) was the program's invention of the notion of prime numbers. AM defined prime numbers by identifying the number of factors of an integer and defining those integers with a very large or a very small number of factors. This led to the definition of 'all primes but 1' (numbers with exactly two factors) in terms of which AM eventually proposed conjectures including the unique factorization of integers.

Unfortunately, AM's definition of primes depended on its definition of DIVISORS-OF which was at best an approximation to the mathematical 'divisors of' procedure. The DIVISORS-OF procedure invented by AM worked by an exhaustive search of a number's possible factorizations number and returned all the factorizations it found; but this search was limited by the 'numbers' AM had already seen so that AM might miss a factorization whose elements had not yet been generated in the course of its other experiments. Furthermore, AM did not have any guarantee that its exhaustive search could stop at some point; in particular, it never appeared to note that TIMES was a monotonically increasing function of its inputs, limiting the space in which factors could be searched for. Indeed, as far as AM knew, some very large number might be a factor of '3'; the reason AM never considered such a claim was the concrete limits of its exhaustive search but those very limits weakened the correctness of its definition.

Thus, AM might have mistakenly recognized a composite number (like 6) as prime because it had never seen the number '3' or '2'. In practice, AM never ran into this problem because Lenat (at AM's request) provided a more efficient implementation of factorization which happened ('accidentally') to also be correct.

This is not to say that AM did not really invent prime numbers, but rather that it invented an empirically adequate version of prime numbers to serve as the basis for futher discoveries. The problem is that AM's native 'inversion' algorithm made assumptions about the distribution of phenomena which were unwarranted and which might eventually be violated in invented representations.

The same search mechanism are embedded in Cyrano's reification mechanisms; a reification assumes certain uniformities among the particular evaluations of a reified operation. The test for coherence applied by Cyrano to reified operations is an attempt to verify that these uniformities hold, but they are not invariants. This element of AM' and Cyrano's representations point out that all representations embody certain empirical assumptions; if the assumptions break down, the representations may fail to be 'good' representations providing effective distinguishability and generalization.

The broad ranging searches conducted by AM are by no means absent from the history of mathematics; quests to square the circle or find the exact rational root of 2 were serious endeavors terminated by a formulation of the problem space which called

their solutions 'out of bounds' while formulating whole new descriptive vocabularies.

5.3 From AM to Eurisko

The Eurisko program was Lenat's own attempt to repair the deficits of AM. His own analysis of AM's malaise began with the same initial hypothesis as my own: AM dwindled because its representations outgrew its heuristics. The answer to this, embodied in the Eurisko program, was to apply the concept formation heuristics of AM to themselves. With this addition, AM would improve its own heuristics to keep pace with its representations.

In the process of developing Eurisko, it became clear that to reasonably modify its own heuristics, they would have to be reorganized. Many of AM's heuristics were large fragments of LISP code; to effectively transform these definitions, it would be neccessary (Lenat realized) to divide the heuristics into smaller pieces and perform mutations based on those small pieces.

Unfortunately, these innovations failed to resolve AM's crisis; applied to AM's domain, Eurisko got little further than AM. Lenat suggests that this was that elementary mathematics was 'mined out'. My own diagnosis is that — despite the ability to devise new heuristics — Eurisko could still not survive the leap from lists to numbers; too many representational assumptions were built into its operations in AM's domain.

In particular, I argue that without careful attention to the starting vocabualary and the way in which that vocabulary is changed, we cannot 'just mutate' an existing representation into a more appropriate one. The AM-like vocabulary of Eurisko suffered from the same 'closure violations' of AM; no simple mutations could relieve its fundamental representational assumptions.

Eurisko's acheivements in other domains, however, were quite impressive; the program decisively demonstrated the effectiveness of AM's general techniques applied to discovery in other domains. In devising 3-dimensional VLSI circuits, Eurisko came up with a variety of interesting new designs; in designing massive space fleets for a 'role-playing' competition, Eurisko's fleets twice won the national competition; in synthesizing and studying simple LISP definitions, Eurisko constructed many simple list manipulation functions and unexpectedly identified (and patched) bugs with the pre-release version of InterLISP in which it was implemented. In these and Eurisko's other applications, the results which emerged were the result of interaction between the program and a user; in the space fleet design task, Lenat divides the credit 40% Eurisko/60% Lenat while emphasizing that neither could have done it unassisted.

But in none of these domains did Eurisko come up with radically new representations; indeed, the meta-heuristics listed in the literature were operations which specialized or customized existing heuristics based on their empirical behavior in a target domain. One way of looking at Eurisko is that it is given a domain and a

simple 'expert system' for working in that domain. This expert system consists of basic distinctions and simple operations which are applied by a collection of weak (in the sense of 'very general') control mechanisms. However, Eurisko is a self-analyzing and optimizing system which quickly transforms these distinctions and operations into a system which grows increasingly fluent and effective. Lenat has suggested that programs like Eurisko work best in new barely-explored domains; given my analysis, it would seem that this is the case because well-worn domains have already been customized and streamlined by armies of human experts.

While Cyrano would certainly benefit from the mechanisms Eurisko uses, it focusses on a fundamentally different problem: the synthesis of new domains and representations. In any given domain, it is neccessary that Cyrano accquire fluency and effectiveness; but the important aspect of Cyrano is that it constructs new domains and representational schemes based on regularities in previous domains and representations. Much of the work that remains to be done on Cyrano is just the sort of self-analysis at which Eurisko excelled; but in view of the analysis of AM presented earlier, this self-analysis a neccessary but not sufficient requirement for an effective discovery program.

5.4 From AM to Cyrano

The first implementation of Cyrano, Cyrano-0, was an attempt at more or less directly reimplementing Lenat's AM and Eurisko programs. Experiences with Cyrano-0 led to the perspective of the cycle of discovery and the subsequent design of the current version of Cyrano. This section points out the major differences between AM and Cyrano; some of these come from isolable deficits in AM, while others emerge from more general methodological criticisms.

Generality. Cyrano attempts to free itself from the domain specificity of AM; this domain specificity, as we saw with CANONIZE, ultimately crippled AM's ability to comprehend its own creations. Cyrano attempts to avoid this problem by characterizing its domains solely in terms of categories and operations; new concepts are formed by combinations of those operations and categories, rather than by internal mutations. In the scenario, we saw how this generality allows Cyrano to survive the representational leap from pair-structures to numbers; in fact, it survives well enough to make another leap into arithmetic over modular fields.

Representation. CYRANO's representation is implemented by a module with a precise contract. TYPICAL is the 'inference' component of Cyrano, making a small set of tractable inferences about definitions provided by Cyrano. AM, on the other hand, had a rich set of slots and concepts with varying semantics. The shift in representation has two goals: closing the cycle of discovery by providing a combinational framework for concept definition; and making the overall structure of the program less opaque to users and critics.

Control. Both AM and its successor Eurisko had sophisticated agenda mechanisms based on numeric worths attached to concepts. As implemented, **CYRANO**'s control structure for exploration is radically simpler; definitions are considered in strict order of simplicity and this search is pruned only by superficial empirical analysis eliminating non-terminating, constant, or identity operations. Cyrano's invention mechanisms, however, are more *ad hoc* but I expect that design work may make them more modular. Note that a clearly defined control structure for invention does not mean that the space of inventions has suddenly been made dense, but merely that the steps of invention or rebiasing are being explicitly made.

User Presentation. In his work with AM and Eurisko, Lenat labored under the constraint that the program 'appear' intelligent; this led to many of AM's conventions about focussing and task organization. In the design of Cyrano, I have made a point to ignore such issues.

Explainablity. Instead of being constrained to 'appear intelligent,' the design of Cyrano has been constrained by a demand to 'appear'. In particular, the modularization of the program is intended to support the effective description of new concepts and the explanation of their genesis. An original intention in the design of Cyrano was a simple discourse interface through which Cyrano and an observer (teacher/colleague) would communicate; the structure of Cyrano has been constrained by a desire for the explicability that such an interface would require.

Chapter 6
Discovery as Problem Solving
A Critique

This chapter considers and criticizes the view of discovery as a simple problem solving process qualitatively identical to the problem solving processes of everyday reasoning. I argue that while the formulation of results in any particular vocabulary is effectively described as 'problem solving', that the formation of such vocabularies is not always so effectively described. This is not to say that such processes are either inaccessible to study or inherently 'non-computational', but merely that most of problem solving's stronger methods do not apply in the space where new vocabularies are being constructed. In particular, I argue that the character of everyday problem solving is that it operates in spaces which are dense with solutions (in senses discussed previously) and there is not a space of dense vocabularies which is itself dense.

I argue for this position in two steps. In Section 6.1, I argue from several historical examples that vocabulary invention has been a central component of scientific progress and that often such inventions can involve radical changes in the ways that terms in a theory *refer* to the objects they describe. Since we inherit these patterns of reference, the discoveries made at the time seem obvious applications of simple mechamisms while the real story — the invention and evolution of vocabularies — disappears. Given the import and centrality of such inventions, I proceed to argue in Section 6.2 that the identification or invention of systems of reference is radically different from the search for formulations within an established system of references. In addition, this latter search is highly sensitive to the problems of entrenchment described in Chapter 4.

These arguments might be held in opposition to the arguments of [Langley *et al.*, 1987], where the authors argue eloquently for view of discovery as an incremental process where small changes in existing theories yield new and better theories. They propose a 'normative' theory of discovery which can objectively compare mechanisms for proposing hypotheses, constructing terms, or defining experiments. The exemplars of this theory are the mechanisms by which their programs — BACON, GLAUBER, STAHL, etc — reproduce versions of several classic results in the history science. These experiments lucidly demonstrate how — given a sufficient statement of a 'scientific problem' — simple methods suffice to produce the resulting theoretical results. Furthermore, by reference to the historical record, they show that such a 'sufficient statement' was apparently available to and used by the scientists who produced these results.

But the simplicity of these methods is the very argument that convinces this author that the bulk of the work lies in the formation of these 'sufficient statements of the problem' and that the real problem — both historically and for our current programs — is the identification of models and terms in which simple laws may be expressed. The particular 'normative theory' proposed by [Langley *et al.*, 1987] describes discovery as a process of inceremental transformations and extensions of existing theories. While the mechanisms they propose are undoubtedly a key element in discovery processes, their account of vocabulary invention — revealed in both their implemented programs and their accounts of historic discoveries — miss the crucial transformations of reference that are often involved in vocabulary invention or evolution.

In looking at historic results in science, it is important to recognize that the contribution of a given result is as much the vocabulary or framework in which the result is 'true' as the result itself. While it is easier to isolate credit for the individual result than for the framework in which it is expressed, focussing on the result ignores the connecting threads that enable its expression. When one scientist builds upon the work of another, she is as much working in the framework established by earlier workers as she is building upon any particular assertion or result. A scientific community seeks to define a framework within which reliable predictions can be made and plausible explanations constructed; to accurately describe the progress of scientific discovery we must look not at the results as we see them today, but at the evolution of explanations, models, and vocabularies which made a given result seem plausible and important at the time.

In retrospect, discovery seems to proceed naturally and easily from given data and straightforward experiments to technical results; but that is because we are the inheritors of exactly the tools and models which make those technical results perspicious. But the problem of devising models, patterns of reference, and the attachment of experimental methods to theoretical entities is not nearly as well formed as the exploration of a set of entities and patterns of reference which are already established.

6.1 Invention in History

In this section, I sketch two episodes from the history of science and discuss the vocabularies or frameworks underlying them. In describing Ohm's discovery of the relation between voltage, resistance, and current and Planck's formulation of his law of black body radiation, I argue that a claim of description or reproduction must embrace the application or transformation of frameworks and vocabularies along with the generation of particular technical conclusions within a given vocabulary. In order to dispel any impressions of tremendous scholarship on my part, let me first say that my discussions of these events come entirely from second sources; I am not citing from these sources to press any particular view of vocabulary invention

or application, but only to illustrate that there is often more than meets the eye to seemingly straightforward events. Unlike [Langley *et al.*, 1987], I do not attempt to provide particular accounts of processes involved in the invention or application of vocabularies which I describe other than to suggest that this application and invention involves the way in which the theories and representations *refer*. I believe that such accounts are possible and usefully expressible in computational terms, but I would not deign — without serious study of original sources — to propose any particular mechanisms.

One problem with examining the historical record for evidences of vocabulary invention is that we in the future have inherited — usually with little reflection — the vocabularies that resulted from scientific revolutions of the past. Such revolutions are successful if they so transform the vocabulary of subsequent ages that it becomes difficult to realize 'what all the fuss was about.'

In a brief example, it is often explained that the astronomer Copernicus realized that 'the planets orbited the Sun rather than the Earth'. Yet as pointed out in [Kuhn, 1989] such a sentence makes no sense in the pre-Copernican framework because 'planet' meant a different thing before the 'Copernican Revolution'; in particular, the Sun was a planet and the Earth was not. The shift involved in the Copernican revolution was primarily this shift of reference; indeed, the framework in which this shift was applied was still very much the framework of Ptolemaic astronomers until later workers simplified the model still further, *given* Copernicus' innovation.

A more contemporary example, which has not yet been so absorbed into our language as to confound its explanation, is the redefinition of observation and the undefinition of simultaneity involved in relativistic mechanics. The equations describing relativistic mechanics are not merely equations relating terms or definitions which were known before; instead, they introduce a new account of observation where (since, for instance, simultaneity has been banished) certain things are not expressible and others seem senseless. It may be that some future generation will explain how Einstein's revolution was simply a shift in describing empirical data, rather than the representational revolution it really was. [1]

The examples below are retellings of two stories from [Langley *et al.*, 1987] which bring into focus the enabling vocabularies which allowed or supported major technical discoveries. In the case of Ohm's law, I argue that one fundamental contribution was a shift in vocabulary which explained a way to conflate two concepts — electrical tension and electrical current — which had distinct referents at the time. In Planck's black-body radiation low, I argue that the explanation underlying his law was as much his contribution as the specific form of the law itself; and the development of this explanation both preceded and succeeded the 'moment of discovery' described in [Langley *et al.*, 1987]. Equally importantly, the 'derivation in progress' had significant

[1] Though, in the case of relativity, I am skeptical of such a change until the velocities involved in everyday interactions make relativistic effects relevant.

influences on the formal technical steps taken by Planck.

My aim in re-describing these examples is not to argue that the mechanisms proposed in [Langley *et al.*, 1987] are insufficient to the phenomena they describe, but rather to suggest that their description of the phenomena is incomplete and that there remain important aspects of these particular events which are neither described by the authors or adequately covered by the incremental mechanisms they propose. The authors express similar sentiments in the discussions surrounding the descriptions of their programs, but their examples occasionally seem to slide over crucial (and perhaps primary) transformations of vocabulary. Furthermore, the details of these inventions will be used to argue — in Section 6.2 — that the invention or transformation of vocabularies has a substantially different character than exploration of any given vocabulary.

6.1.1 Resistance to Ohm's Law

In the same way that the shift engendered by the Copernican revolution makes his assertion 'the planets orbit the sun' seem obvious, a shift in the vocabulary for describing electrical systems enabled BACON-1 to rediscover Ohm's Law based on Ohm's data [Langley *et al.*, 1987, Pages 84-85]. BACON-1 is given the data from Ohm's experiments where the length (L) of a conducting wire between two terminals of a battern is varied while the effect of this variance upon a galvonometer needle (I) is measured. From this data, BACON derives the regularity:

$$I = \frac{b}{L - a}$$

where b and a are (in modern terms) the voltage and internal resistance of the battery. In a paradigm example of Baconian science, analysis of empirical results yields a scientific law.

However, the historical context of Ohm's discoveries, as described in [Schagrin, 1962] tells a somewhat larger story. The description and quotations here are taken from [Schagrin, 1962] to reveal the larger context of his discoveries. In particular, Ohm's results were not immediately accepted by his colleagues because he had conflated an important distinction between 'electrical tension' and 'electrical current' which was current at the time.

Electrical theory at the time had evolved from a study of electrostatics and the 'voltage' of the source measured by Ohm was in fact the 'electrical tension' indicated at one terminal by a gold-leaf electrometer while the other terminal was grounded. However, when the two terminals were connected, the tension disappeared and current flowed, yielding magnetic or electrochemical effects. Schagrin summarized this distinction in [Schagrin, 1962] by offering a quote of Ampere's where the distinction

between electrical tension and electrical current is described:[2]

> *"... there is no more tension electricity, light bodies are no longer sensibly attracted, and the ordinary electroscope serves no longer to indicate what takes place in the object; however the electromotive action continues to act; ... but ... when one interrupts the cirtuict the needle returns to its orginial position, the tensions reestablish themselves, and light bodies are freshly attracted, all of this proves quite well that the tensions cause neither the decomposition of water nor the change in direction of the magnetic needle...."*

These quantities of electrical tension and electrical current were demonstrably independent; for instance, placing electrical cells in series yielded much higher measurements of electrical tension while producing no substantial change in the measured electrical current. On the other hand, making batteries with larger plates increased the measured electrical current (called the 'quantity' at the time) while having no effect on the electrical tension (called the 'intensity') measured by an electrometer:

> *"... however much we may increase the quantity of electricity by employing very large plates..., we cannot obtain from such an instrument any of those effects which require for their production a certain intensity, as well as quantity of electricity. In order to obtain these latter effects, we must employ the compound battery... The former of these instruments, accordingly, will be capable of producing such effects as depend upon mere quantity, without regard to intensity; such as the evolution of heat, the ignition and deflagration of the metals, and electromagnetic phenomenon. The compound apparatus, on the other hand, will afford the more ordinaary electric appearances (such as the spark and the phenomena of attraction and repulsion)..."*

The distinction between current and tension was an important one for the experimenters of Ohm's day. For instance, in describing an experiment of Gay-Lussac and Th`enard, Ampere distinguished their separate effects

> *"Now it is quite evident that the tension electricity of the extremities of the wires that are immersed in the liquid will not be increased...; nor is it probable that it be decreased to the degree that this liquid becomes a better conductor; what is increased in this case is the current electricity...."*

While Ohm's data regarding the linear relation of conductor length and current measurement were generally, accepted, his critics could not accept the equation involving what we today call voltage because it ignored this important distinction; as the commentator Pohl remarks:

[2]Quoted in [Schagrin, 1962] from [Ampère, 1820].

> *"... where the author had dealt with the observation of th reactions of a completely closed circuit, it succeeded in showing for him an apparaent confirmation of his results ... We say apparent; since, in the first place, it was an error when a quantitative proporitionality of the magnetic effect of the closed circuit took the place of the electrtical reactions of an opened one, ... as the magnetic needle suddenly came to represent the function of the electroscope...."*

As described by Schagrin, Ohm was able to explain away the observations that distinguished beween current and tension electricity, yet the distinction had fruitfully yielded results and its abandonment took time, leading to the negative reception of Ohm's results. Yet our inheritance is that framework of explanation and again we see how a successful revolution hides its roots.

When looked at through modern eyes, Ohm's law seems a clear example of inferring an elegant descriptive law from obvious observational parameters. Yet the definition of Ohm's law is as much the characterization of those parameters — which we inherit — as it is the qualitative regularity relating them. The contribution of Ohm was as much his explanation supporting a shift of reference — merging the phenomena of tension and current — as his induction of his law from the data he was given.

6.1.2 Black Body Radiation

In the opening arguments of [Langley *et al.*, 1987], an example from the early history of quantum physics is cited to support the view of discovery as an incremental process of refinement of previous theories. The example involves Planck's law of black-body radiation, which asserts that the equilibrium radiation of frequency v in a perfectly reflecting sphere is expressible as:

$$I = \frac{A}{\epsilon^{\frac{kv}{T}} - 1}$$

where I is the intensity of radiation, T is the temperature of the cavity and A and k are constants. As the story goes, Planck derived this equation by an overnight modification of the well-known Wien distribution

$$I = \frac{A}{\epsilon^{\frac{kv}{T}}}$$

upon hearing that new distribution data for lower frequencies was inversely linear rather than inversely exponential in the frequency. As noted by the authors of [Langley *et al.*, 1987], Planck was intimately familiar with Wien's law at the time and had just completed a derivation of the empirical law from classical principles. This

derivation applied the arguments of thermodynamics — developed for describing collections of molecules — and applied them to collections of hypothetical 'resonators'. When the lower frequency data became available, Planck modified the equation for Wien's distribution and quickly guessed where the error in his original derivation lay; there was an internal contradition wherein he assumed that the sum of the energy levels of the resonators was constrained by U yet their individual energies were independent. Planck then spent 'a few weeks of the most strenuous work of my life' modifying the derivation of the Wien distribution to explain the new law.

It is tempting to explain Planck's discovery as a pardigmatic example of Baconian science; the moment of discovery was when Planck realized that a different — yet still quite simple — equation explained both new and existing data. This is the picture painted in [Langley *et al.*, 1987], where they offer Planck's own description of his discovery of the new equation by mere formal manipulations; from Planck's description, they go on to describe how the discovery might also be made by simply searching for a function which interpolated between the Wien law and a nearly linear law describing the lower frequency data. In this hypothetical version of the story, once the equation had been uncovered by formal manipulation leading to the interpolation, one proceeds to seek a derivation which yields the new empirical equation.

In offering Planck's description of his own derivation of his law, [Langley *et al.*, 1987, Page 52] paints the following picture:[3]

> *"Now, Planck goes on to say in his October 19 paper (Kangro 1970, p. 36)*
>
> > I finally started to construct completely arbitrary expressions for the entropy which although they are more complicated than Wien's expression still seem to satisfy just as completely **all requirements of the thermodynamic and electromagnetic theory.**
> >
> > I was especially attracted by one of the expressions thus constructed which is nearly as simple as Wien's expression and which would deserve to be investigated since Wien's expression is not sufficient to cover all observations. We get this expression by putting
> >
> > $$\frac{d^2S}{dU^2} = \frac{\alpha}{U(\beta + U)}$$
> >
> > It is by far the simplest of all expressions which lead to S as a logarithmic function of U... **(an assumption suggested by the probability equation.)**[4]

[3]The citation in this quote is to [Kangro, 1972].

[4]Additional text inserted from the quote in [Kuhn, 1978, Page 97] and [Kangro, 1976, Page 209].

> *If we accepts this account as an actual description of the process that Planck used to solve his problem, then we see that he searched among an undefined class of functions until he found a 'simple' one that had the desired limiting properties. We do not have any very good measure of how hard or how easy a search this is, except that Planck completed it in a few hours.*
>
> *There is an alternative path to the goal that makes use of the intensity laws without calculating the derivatives of the entropies. It consists in looking directly for a 'simple' interpolating function ...."*

[Langley *et al.*, 1987] goes on to discuss this alternative path and indeed describes an informal experiment where professional physicists and mathematicians are able to construct Planck's distribution — not realizing its significance — by a simple interpolation between Wien's distribution and a linear law describing the anomalous data which led Planck to his own equation. But this transformation leaves behind Planck's constraint that the chosen formulation 'satisfy just as completely all the requirements of the thermodynamic and electromagnetic theory' as well as Planck's explanation that the 'logarithmic function of U' was 'an assumption suggested by the probability equation' ([Kangro, 1976, Page 209] and [Kuhn, 1978, Page 97]) based on the framework of his orginal derivation as well as the canons of explanation of the time.

While one way of reading history was that these were *post hoc* considerations to the equation invented by Planck, I would suggest that Planck's consideration of alternatives and the reason he was 'especially attracted' to the formulation he proposed were that the modifications he made to the theory were consonant with his physical intuitions — not yet coalesced into a new derivation — about the nature of the processes they described. Indeed, if we look at division of labor, we see the strenuous weeks of seeking a new derivation as a sign of the importance which the explanation — as opposed to the equation — had for him.

I would never assert that the mathematical skills and heuristics applied by Planck were not crucial to both his hypothesis of the equation and his derivation of the mechanism; but these skills and heuristics were embedded in a framework of models and explanations which Planck's strenuous weeks of work modified and extended (though not in radical ways). Here we return again to the way in which scientific results are as much the vocabularies they provide for description as the technical results expressed in these vocabularies. In Planck's case, the vocabularies and Planck's innovation to them were the application of thermodynamic explanations (in his earlier derivation of of Wien's distribution) and his application of Boltzmann's accounts for entropy (in his derivation of the distribution function which bears his name).

6.2 Is Vocabulary Formation a Well-Formed Problem?

In the preceding discussion, I discussed several discoveries which looked — to modern eyes — like simple induction from immediate observables; yet by touching on descriptions of their historical context, I argued that the content of these discoveries was as much their characterization of objects or observables as it was the assertion about relations between them. Given this distinction, the crucial element of discovery is the formulation or transformation of vocabularies rather than particular results within the vocabulary (of course, the result is what makes people listen). In this section, I make an argument that by their very nature, these processes are less accessible to 'problem solving' than activity within any given vocabulary.

In previous sections, I described how various shifts and uses of vocabulary and framework underlie discoveries which — to our modern eyes — seem to follow directly from given data. In these transformations, the basis for observation, reference, and explanation played a significant enabling or confirming role in the specification of a formal technical result.

In Ohm's conflation of the experimentally distinguishable phenomena of electrical tension and electrical current, an elegant formulation was made possible by collapsing two sorts of phenomena into one. Note that there could be no observable correlation of the phenomena of tension and current without a theory which bridges the distinct situations of the open and closed circuits; like the classic problem of identifying the 'morning star' and the 'evening star', identifying the two phenomena requires a theory which posits unity of reference while explaining multiplicity of manifestation.

In the case of Planck's black body radiation law, I suggest — with [Kuhn, 1978] — that there is no vocabulary formation going on in the formation of the law, but that the vocabulary formation preceded and succeeded the hypothesis of the law by the construction of derivations from physical models based on thermodynamic systems. In this case — where we have descriptions of the participants as to their methods — we note that the aspects of the Wien distribution law which Planck examined were informed by his model of the process which produced the law; a model much on his mind from his recent (flawed) derivation of the Wien distribution.

In all of these cases, the shift or application of a theory involves the particular ways in which the theory *refers*. A theory's empirical assertions are not just formally manipulated in its development or transformation but involve the constant consideration of the way they refer to the experiments we conduct and the models we seek to verify. I would not, like some, assert that these reference relations are beyond the reach of the systems we build, but I would argue that the problem of inventing useful systems of reference and description is significantly harder than the production of interesting or useful results in any given system.

The space of systems of reference is much broader than the space imposed by any particular system of reference. Indeed, a system of reference includes social,

cultural, and technological elements which place them partially outside the scope of any individual's activity. A theory of discovery for humans or machines must include some account of how this space is explored.

In the Cyrano program, the space of reference systems is explored by the construction of computer programs; these programs 'refer' to the behaviors they produce. Cyrano employs reification to yield programs which refer not to individual behaviors, but to families of behaviors computed by particular definitions. This reficiations are intriniscally limited by the primitives of operations on tie structures; Cyrano will make no theories of quasars or quarks without a radically different (and larger) set of primitives.

Yet for a human scientist engaged in a community of problems, the range of primitives is much larger and the space of systems of reference includes all the experiments she might perform, the apparati she might construct, or the technological instruments her culture might produce. But at the same time, this space of potential systems has its own criteria of likelihood; some systems are more likely to be proposed than others. This bias, while it makes discovery possible, brings with it the danger of entrenchment; there will be shifts of representation which are hard to see, leading to revolutionary shifts of vocabulary which are not strictly translatable — because the standards of reference have shifted — between one and other.

Chapter 7
So What? Critique & Responses

This chapter seeks to clarify and defend the approaches and conclusions offered in previous chapters by presenting and addressing a variety of serious critiques which might be applied to Cyrano's design and analysis. In some sense, the arguments below make an effort — admittedly a biased one — to do for Cyrano what Chapter 5 did for AM and what Chapter 6 did for the systems presented in [Langley *et al.*, 1987].

The criticisms being replied to in this chapter are not soley applicable to Cyrano; some apply (and have been applied by the author in preceding chapters) to other efforts in discovery, machine learning, or the overall enterprise of artificial intelligence. My replies to these criticisms will not be 'knock-down' arguments since all of the criticism raises serious concerns which cannot be easily 'knocked down'; I merely discuss the issues which these criticisms raise and attempt to argue that they are not 'knock-down' arguments either.

Among the important criticisms that might be addressed to Cyrano are:

- **Cyrano Didn't Discover Anything New.** Cyrano reinvented certain arithmetical systems, yet did not conjecture or propose any results new to Humankind. If the true test of an experiment in discovery is the novelty of its results, Cyrano failed to produce results which were new and significant.

- **Cyrano is a Wind-up Toy.** Cyrano begins with a particular representation and produces particular discoveries; however, its status as a *designed* artifact brings into question whether it deserves any 'credit' for its acheivements. In particular, did Cyrano discover arithmetic over the modular fields, or did *I* (as Cyrano's designer/user) simply discover an interesting way to define the modular fields in terms of structural operations on tie structures?

- **Cyrano is an Isolated Discoverer.** Discovery — be it by scientists or children — is an activity which draws on and is embedded in a social community. There is no component of Cyrano which obviously corresponds to the phenomena of communication, explanation, or criticism which are crucial elements of invention and evolution in communities.

- **Math isn't Science.** The definitions and systems which Cyrano constructs are mathematical in nature yet the mechanisms are those of physical sciences. This distinction raises the question of what sort of discovery (if any sort) Cyrano is actually performing.

- **Cyrano Doesn't Have Goals.** Cyrano can be understood as searching for systems of description, yet searches in Artifical Intelligence are usually highly constrained by a system's goals. Cyrano has no explicit goals and this in some ways diminishes the 'sensibility' of its performance.

- **Cyrano Has Weird Primitives.** The primitives with which Cyrano begins are very abstract and mathematical. The common-sense language from which we construct theories or the experiences of action and reaction that form a child's understanding seem to generate the abstract from the particular, while Cyrano seems to go in the other direction.

All of these criticisms have kernels of truth which identify both current problems and future directions for the research beginning here.

7.1 Reevaluating Rediscovery

How do we evaluate a discovery system? Since the very word 'discovery' connotes the revelation of the unexpected, it is difficult to imagine prearranged standards for judging the results of a discovery system. Unlike computer programs which solve problems, analyze data, or induce concepts, a discovery program succeeds if it provides answers to questions we regard as interesting yet would not have thought to ask.

A relatively objective version of this criteria would be to ask the question 'Has this program made any significant discoveries new to Humankind?' For Cyrano, the answer to this question is clearly 'no,' yet such newness is a great deal to ask of any program; indeed, it would also be a great deal to ask of a human being. In this regard, the criteria instantiates the superhuman/human fallacy ; in order to be sure that the system is not 'tricking us' through a clever arrangement of special case hacks, we demand that it solve problems with a generality which eludes its creators and (indeed) most human beings.[1]

How might we weaken this criteria of newness to something more useful as a measure of our programs' success? An obvious way of retaining the spirit while weakening the letter would be to narrow the community by which we define 'new' and 'known'. For instance, we might

(i) ask whether the system yields results that are 'new' to it by not being easily reducible to the constructions with which it started; or

(ii) we might ask whether the system — given the problems and data of a particular technical community in history — produces the same results which were valued as new and interesting by that community; or

[1]This denies the reasonable possibility that human beings are in fact a clever arrangement of special case hacks.

(iii) we might ask whether the system produces results new to its designers.

Criteria (i) is passed by any reasonably sophisticated system which acheives 'constructive closure'; if it uses the results it constructs, eventually it will be constructing claims and definitions in a space quite distant from that in which it began. However, there are trivial spaces where the distance and opacity of derivation corrupts any attribution of newness. It is easy to make systems so blind that the simplest combinations will 'suprise' them.

Criteria (ii) is one which we might apply to systems like BACON [Langley *et al.*, 1987] which are presented with the experimental arrangements, parameters, and observations of a scientific community at a certain point in time and — by the application of straightforward heuristics and analytic methods — yield results similar to those produced by the workers within that community.

The problem with this criteria was touched on in the previous chapter; it is difficult without very careful thought, reading between the lines, and the abandonment of our contemporary perspective, to adequately determine what a past community of workers did or did not know and could or could not see. When a scientist is deeply successful in influencing scientific thought, it becomes difficult to see where the innovation in their experiments or approach lies. Particularly when the system is given an initial representation, we must ask how much of the historical scientists insight lay in the isolation of important factors and exclusion of influences implicit in the representation of the problem which we inherit.

Finally, criteria (iii) (that of suprising the designers) is double edged in almost the same way that 'suprising itself' is; it indicates either a genuninely creative program or a designer's lack of insight into her own creation. When Cyrano produces a result which suprises me, it is partially due to my own lack of understanding or imagination regarding Cyrano's operation. When Cyrano defined arithmetic means or modular equality, one might say that I should have 'forseen' these consequences given the original combiners and primitives. When AM developed the notion of number, a critic might complain that Lenat should have known that his implementation of `CANONIZE` implicitly contained a representation of cardinality in its test for element invariance.

Yet for any criteria, we need to ask this same question about what is 'built-in' and to what degree we can ascribe the system's discoveries to what is given it initially. The next section discusses this problem in greater detail.

7.2 Wind up Toys

One criticism often levied against discovery systems like Cyrano is that they are 'wind-up toys' carefully designed and tuned to yield a particular series of discoveries. This supposed particularlity brings into doubt the generality of the system's mechanisms and architecture. These critics point out — quite accurately — that the

system's success is as much due to its initial representation as to whatever mechanisms or heuristics it deploys in manipulating this representation.

Yet every AI system is a wind-up toy in this sense; we design representations which are likely to quickly yield significant results. Indeed, this is a property of every 'designed artifact' be it computer program, scientific theory, or mathematical framework. It is the hallmark of designed representations that they quickly yield results of interest. This very density of representation is what a designer seeks.

The core of the 'wind-up' criticism is not the density and effectiveness of representations, but the generality of the mechanisms being used. This is a crucial criticism of representations in general and particularly important in discovery where the system's own progress yields new domains of operation to which existing mechanisms must be applicable. We must ask of each component of our discovery systems: is this component broadly applicable across vocabularies or is it particular to some particular discovery or series of discoveries? If the latter description holds, I argue, we have found something which is 'built in'. In designing Cyrano, I have tried to be conscious of how many times and in how many ways different components are applied in its operation. Early versions of Cyrano focussed on defnitions whose behavior satisfied certain 'cognitive cliches' corresponding to mathematical notions such as transitivity, symmetry, etc. Eventually, I grew concerned that those regularities were too specific to mathematics and devised the criteria of polygenicity as a more generally focussing criteria which — it turned out — subsumed the cliche-based representation.

Yet this very activity of generalization is a sort of 'tuning' which shifts the system towards the phenomena which the designer notices. When Cyrano first defined modular equality, it failed to construct the corresponding vocabulary because of weaknesses and particularities in the implementation of its abstraction mechanisms (what later became reification). This 'just missed' opportunity led me to develop those abstractions mechanisms to a greater degree of generality and applicability.

There is no question that Cyrano is a designed artifact and that much of its behavior was anticipated before its occurence; indeed, as it has ran many times in the process of its development, none of its results are truly 'new' since any single discovery described here has been 'rediscovered' each time the program has been restarted. This continual rediscovery points to a difficulty with the way I have spoken of Cyrano's discovery; I may claim that Cyrano has discovered something when it has merely defined something and noted regularities of interest about it. One might suggest that the true 'moment of discovery' was my remarking 'what an interesting way to compute an arithmetic mean' or 'wow, it found $=_{\text{mod}3}$!'.

To what degree are Cyrano's discoveries genuninely Cyrano's and to what degree are they *mine* utilizing the tool of Cyrano's exhaustive imagination? The role of translation is not a superficial one and the task of translation is sometimes a difficult one. There have been times when Cyrano has devised definitions which I found interesting yet could not concisely describe; after ruminating upon the definition for

a while, I realized that there was in fact a common sense name — like 'fixed point' — for the definition Cyrano had constructed.

The neccessity of translation and interpretation mitigates the 'wind-up toy' criticism which this section addresses but leads to confusion about the assignment of credit for Cyrano's discoveries. This confusion about credit is not particular to the pair of Cyrano and myself (its designer); indeed, in the 'real' sciences, it is somtimes difficult to assign credit when one scientist notices results in another scientist's experiments, or when one scientist develops a method which enables another scientist to ferret out an important result. The fact that this distinction may be central to the assessment of Cyrano indicates another deficit of Cyrano. Cyrano is an isolated discoverer, proceeding in its private world by constructing definitions and systems of description without connection to an outer community of standards and conventions into which its inventions might fit.

7.3 The Scientific Community

Our modern mythology gives us the isolated scientist as the 'pioneer' of our intellectual culture, mapping out new continents of problems and programmes which later workers colonize and domesticate. Yet a historically accurate view of science, while not denying the brilliance of certain individuals, tells us a story of communities of workers and of common questions, methods and standards which are considered, clarified, codified, questioned, and abandoned in common.

At first glance, programs like Cyrano seem the idealized example of the isolated discoverer, closeted with descriptions and methods and apparati and producing concise and powerful tools for characterizing and manipulating their world. Yet the criticisms of the previous sections cut through this illusion and draw out the fact that Cyrano (and programs like it) are the result of an interactive process in which the designer plays a significant role.

This perspective, while valid in important ways, does not diminish the importance of experiments like Cyrano but rather suggests that these systems are even better models of the processes of discovery than we might have imagined! While they do participate in a community of sorts (with their designer(s),) their great deficit is their discursive dumbness which brings us to look briefly at the role which *explanation* plays in discovery.

Discovery systems do not generally *explain* their discoveries but present them as consequences which their designers then interpret and explain with reference to existing systems of description. Every explanation is with respect to a set of standards and primitives; but a discovery system which is constructing new standards and primitives cannot produce explanations of its own unless it has access to the manifold existing 'systems of explanation' available to the human scientist. And even in these cases, the human scientist is not explaining *with respect to* an existing framework but

by analogy to an existing framework.

A crucial element of radical discovery — by humans or programs — is the construction of new frameworks for explanation and the corresponding transmittal of those frameworks to others. This was the phenomenon alluded to in the previous chapter where the *physical models* accompanying qualitative accounts were offered as an important element of any technical result. Indeed, if we are to take these frameworks of explanation as primary, the 'paradigm examples' of a field are not so much *constituitive* of the field as *descriptive* of its framework. But the transmittal of a framework to another is impossible without some understanding of the frameworks in which the other is situated, and this is a problem that may only be resolved by either broad ranging databases [Lenat and Shepard, 1990] and a deeper understanding of essential human ontologies.

What is the potential for discoveries without this transmittal? Could we not build a discovery system which explores its own vocabularies without concern for transmitting them to others? Science Fiction often introduces the intelligent supercomputer explores the world in just this way, quickly leaving behind the panel of scientists interpreting its output[Jones, 1966]. What might such a system acheive if left to its own devices?

One grave problem with such an approach is the potential for entrenchment. While the mechanisms of Section 4.6 — plurality, isolation, hierarchy, historical criticism — are important for avoiding entrenchment, they may be understood to suffice only by *creating* a community within the machine. There is a deep connection between the importance of community and the need for less-dense frameworks for the avoidance of entrenchment. Thus an 'isolated discoverer' using mechanisms like these to avoid entrenchment would itself grow into a community with divisions, distinctions, and boundaries like those of the human community.

The importance of a community is that it provides a context of standards and criteria which is more general (while technically weaker) than those individual perspectives and processes it contains. Cyrano does a miserable job of being a member of the human community — it has to have itself explained or examined by human interpreters -- yet within Cyrano there consist a plurality of processes and perspectives in the various vocabularies it develops and the emphases it organizes. But within Cyrano, what are the critical criteria of this larger context?

7.4 Proof and Experiment

Cyrano is in a rather odd situation in the way it addresses its domain. What it discovers are mathematical systems, but its methods of discovery are more like those of physics: experiment, regularity identification, the transformation of phenomena into theoretical objects. The oddity of this position is especially revealed when we consider how Cyrano's judges (or fails to judge) its own results.

In physics, there are standards of experimental agreement, repeatability, and falsifiability to which accounts or frameworks can be brought for judgement. In mathematics, there are standards and systems of proof, contradiction, and counterexample by which systems of objects or particular claims are judged. Cyrano constructs mathematical systems (or systems which I describe as mathematical) yet proves little about them; these systems are codifications of the behaviors of observational primitives (the operations on tie structures) which Cyrano has experimentally confirmed, but Cyrano does not have expectations for its experiments as a physicist does.

Mathematics and physics have deep differences in what counts as results, progress, or refutation. One aspect of these differences is that (broadly speaking) when something goes wrong in mathematics (a counterexample or contradiction), some construction is removed from the system; when something goes wrong in physics (a failed experiment, or an unaccounted-for case), some construction is added to the system. Looking for this same distinction in Cyrano reveals that Cyrano has no real account of responses to anomaly.

Cyrano's critical faculties are momentary ones which are applied only at the genesis of definitions. The triage process in vocabulary exploration or the identification of coherent reifications in vocabulary invention are both checks that some generated definition or vocabulary will be accessible to experiment and extension. But once a definition or vocabulary has been accepted, there is no later expectation about what the definitions or vocabularies will eventually become. Science and mathematics are both involved in a search for anomaly; but in Cyrano, anomaly is neither sought nor explicitly identified. Indeed, it only reveals itself when the offspring of some definition or vocabulary fails to be productive.

Consider, for instance, Cyrano's construction of the natural numbers based on the reifications of repeated `BOTTOM` operations. The fact that this representation was useful depended on the determinism and composability of the BOTTOM operation; these were the assumptions which Cyrano made when it took the apparent coherence of `[limit [bottom] [bottom]]` as suggesting a reification. If either of these assumptions were somehow violated and the `BOTTOM` operation began behaving oddly in some contexts, the result would be that the identification of numbers — the isolation of a correct *canon* for a given sequence of `bottom` operations — would become impossible or somehow non-deterministic; the resulting vocabularies would begin to yield definitions of which examples could not be found or which never revealed any interesting structure.

Anomaly has an effect in Cyrano but is not an active consideration for the program. The value of seeking out anomaly is that it provides an important critical tool for assessing and pruning the space of definitions. The quest for anomaly is also a palliative for entrenchment. The identification and explanation of anomaly is — in essence — a process which steps out of one framework of representational assumptions into a weaker yet more flexible framework.

Were Cyrano to seek out anomaly, it would also have to contain some mechanism for dealing with anomalies once found. In general, this will involve a choice between conflicting alternatives and the basis for making such choices brings us to another crucial question: What does Cyrano want?

7.5 What Does Cyrano Want?

Goals are useful for dealing with conflicts about what to do. Since Cyrano never really has any such conflicts, it does not have a real and immediate need for goals. However, the absence of conflicts is an artifact of the simplicity of its world and the absence of real world constraints; furthermore, the goals it has — the identification of systems of interconnected definitions built upon a base language — have been encoded implicitly in its design.

The above discussion of how mathematics differed from physics hinged on distinguishing the goals of the two disciplines: physics aims at a complete description of the physical world; mathematics aims at families of powerful and consistent systems of description. In attempting to place Cyrano in one or another camp, we discover that Cyrano has no real goals (in this sense) and so is difficult to classify under either mathematics or physics.

However, another perspective on Cyrano might assert that its goal is *inventing* new goals. Cyrano's criteria of interest — the way it picks out families of definitions it wants to extend — is the identification of polygenic behaviors. However, once this bias is asserted, the system has a more specific goal: 'generate definitions like these.' Since this starts from the polygenic behaviors Cyrano first notices, we might redescribe Cyrano's initial goal as 'invent goals easily expressible in terms of these primitives'.

This definition of goal hinges on the subtle way in which representations implicitly provide goals and if we seek to develop new representations we are also seeking to develop new or transformed goals. It is sometimes uncomfortable to consider the degree to which our representations embody particular goals for we wish to believe that our goals are explicit and chosen and may (if called for) be abandoned. Yet the density of a representation — what it makes easy to say or difficult to say — implicitly imposes goals as surely as if they were objects in an ordered priority queue.

The answer to the question 'What does Cyrano want?' will not point to any particular set of tokens in its implementation, but will rather point to Cyrano's representation and reply 'this is what Cyrano can easily say' or 'this is what Cyrano cannot easily say'....

7.6 What are Primitives Anyway?

This series of questions brings us to examine Cyrano's initial representation. When Cyrano begins with the operations of predecessor and successor, it is pretty clear to all involved that Cyrano is going to be inventing definitions and finding regularities in arithmetic. By the same argument, when Cyrano begins with operations on tie structures, we might expect that Cyrano is going to invent definitions and find regularities among tie structures. While this occurs to a limited extent — Cyrano notices that tie structures end in terminals and that `top` and `bottom` destructure the results of `tie` — little else of interest is uncovered; operations on tie structures do not yield the interesting polygenicity which arithmetic operations (once defined) do.

Yet Cyrano's base vocabulary and combinational framework have their own emphases which cannot be denied. In particular, they focus on simple and iterated compositions of strictly functional operations. Cyrano illustrates — in some sense — that arithmetic will emerge from any system which supports such composability and strict functionality. This is akin to Minsky's observation [Minsky, 1987] that the space of the simplest Turing Machines is largely populated with arithmetic operations of one sort or another.

Given that Cyrano's implicit goals and representational assumptions contain these emphases, we may wish to ask whether these primitives are in accord with the sorts of basic elements applied by human mathematicians. I expect that they are not and that a revolution in our understanding of representations will be necessary before our computer programs can take their place beside us in the community of modern science. Scientific and mathematical explanations (as opposed to laws or proofs) are usually rich in metaphor and model in a manner which depends on the most basic ways in which we organize our understanding. And as I discussed in the previous chapter, the true contribution of a piece of work is often the explanation or framework which yielded a result and not the result itself.

An open and important question is hence the character of those 'primitives' by which we understand and accept (or fail to accept) an explanation. My suspicion (still undeveloped) is that a large component of this character lies in their provision of a guide to *attention.* In a program like Cyrano which manipulates and combines tokens, a dense representation can be acheived by making worthy descriptions short; in a mind which works in the framework of an established inner model, this same density is achieved by placing significant results along the paths in the model which the mind's attention naturally follows. Thus the arrangement of elements in a mental model is vitally important to the effectiveness of explanations with respect to the model.

Until we have an adequate account of these patterns of attention and the way they work on our internal images, it is difficult to imagine an intelligent system which could yield theoretical results without need of interpretation by a skilled translator.

This is not to say that the system together with translator would not yield results which neither could have yielded alone, but merely that we cannot deny the difficulty of connecting our program's explorations to the patterns of explanation from which our intellectual culture is constructed.

Chapter 8
Conclusions & Connections

In the preceding pages, I have expanded on and argued for a view of discovery as the interleaving of two distinct processes: the exploration of particular representational vocabularies and the invention of new representational vocabularies from existing ones. Cyrano, a discovery system built around this model, was presented and several episodes from its operation described. These included the invention of vocabularies like natural numbers and numbers modulo 3 as well as the definition of interesting operations within these vocabularies.

The discussion of Cyrano led to a discussion of the issues of vocabulary invention in general and the introduction of an account of representational *entrenchment* wherein a representation becomes more effective at the cost of both generality and the ability to assess improvements in its effectiveness or scope. The problem of entrenchment can be mitigated by a variety of mechanisms all of which involve manipulations in a space which is less effective than those in which the system actually does its exploration and problem solving. The need to separate exploration and invention arises from this essential tension between the flexibility to adequately judge one's performance and the ability to perform effectively in particular domains.

The issues of vocabulary invention are followed by a detailed exegesis of the AM program and a comparision of this program (and its descendant, Eurisko) with Cyrano. The discussion of AM is followed by a discussion of the BACON program which criticizes the position offered in [Langley *et al.*, 1987] that discovery is a species of 'problem solving'. It suggests that 'problem solving' — as used in a precise technical sense — generally occurs in spaces which are *dense* with solutions; vocabualry invention, on the other hand, is attempting to develop systems which are dense but the space of vocabularies — because of the danger of entrenchment — must be a space of radically weaker constraint than any particular effective vocabulary. By reference to some of the discoveries described in [Langley *et al.*, 1987], it is pointed out that an important element of the discoveries which presumably occured in one vocabulary actually involved significant transformations of vocabulary which are invisible to modern eyes without attention to the historical context of the discoveries.

After discussing the AM and BACON programs, I went on to suggest and address some criticisms of Cyrano. The criticisms so presented serve as a partial for the future work section appearing at the end of this chapter.

8.1 Related Work

In the body of this thesis, Chapter 5 and Chapter 6 presented detailed examinations of two seminal works in automated discovery to which this work owes both an important debt and offers some important criticisms. This section is an attempt to briefly connect the work with other significant work in related fields.

There has been much work in automated discovery of physical laws following the systems presented in [Langley *et al.*, 1987]. In particular, the application of qualitative physics to provide physical models underlying qualitative laws addresses — to some degree — the criticisms I made in Chapter 6.

In discovery of mathematics, it is important to mention Wei-min Shen's reimplementation of the AM program; starting from a formal language akin to Backus' FP[Backus, 1978], his program reproduced all of AM's discoveries as well as inventing other unexpected definitions (like division). The program, unlike AM, Eurisko, and Cyrano, had no inherent interstingness criteria but exhaustively searched the space of interesting definitions.

8.2 Future Work

The criticisms of Section 7 point to several directions for extending the research described here.

A first effort might be directed at introducing goals into Cyrano by having it propose theories, seek anomalies, and attempt to identify theories describing the anomalies. In addition, providing a domain with its own goals — unlike tie structures or arithmetic operations — might provide a powerful source of constraint on the system's exploration.

Another direction of interest is the *explanation* of invented systems. For instance, if we were to hook Cyrano's mechanisms up to a large scale knowledge base of the sort described in [Lenat and Shepard, 1990], we might find existing explanations whose structure mirrors that of those which Cyrano invents.

Another effort which merges these two directions is the application of Cyrano to different domains with both more intrinsic structure and clearer goals for performance and mechanisms for evaluation.

Appendix A
References

[Abelson and Sussman, 1985] Harold Abelson and Gerald J. Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Ampère, 1820] A. M. Ampère. De l'action exercée sur un courant électrique par un autre courant... *Ann. Chem. Phys.*, (15), 1820. Reprinted in *Mémoire su l'electromagnétisme et l'électrodynamique* (Gauthier-Villars et Cie, Paris, 1921).

[Backus, 1978] John Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8), 1978.

[Chapman, 1986] David Chapman. Cognitive Cliches. Working Paper 286, MIT Artificial Intelligence Laboratory, 1986.

[Davis, 1979] Randall K. Davis. Meta-rules: Reasoning about control. memo 576, MIT Artificial Intelligence Laboratory, 1979.

[Dreyfus, 1979] Hubert Dreyfus. *What Computers Can't Do (2^{nd} edition).* Harper and Row, 1979.

[Goldstein, 1976] Ira Goldstein. Frl: A frame representation language. Memo 333, MIT Artificial Intelligence Laboratory, 1976.

[Greiner, 1980] Russel Greiner. RLL-1: A Representation Langauge Language. Working Paper 80-9, Stanford Heuristic Programming Project, October 1980.

[Haase, 1986a] Kenneth W. Haase. Discovery Systems. In *Advances in Artificial Intelligence.* European Conference on Artificial Intelligence, North-Holland, 1986.

[Haase, 1986b] Kenneth W. Haase. Discovery Systems. Memo 898, MIT Artificial Intelligence Laboratory, 1986.

[Haase, 1987] Kenneth W. Haase. Typical: A knowledge representation system for automated discovery and inference. Technical Report 922, MIT Artificial Intelligence Laboratory, 1987.

[Haase, 1990] Ken Haase. Scheme utilities. Ai memo 1075, Artificial Intelligence Laboratory, MIT, 1990. (In preparation).

[Jones, 1966] D.F Jones. *Colossus.* 1966.

[Kangro, 1972] Hans Kangro. *Planck's Original Papers in Quantum Physics.* Wiley, 1972.

[Kangro, 1976] Hans Kangro. *Early History of Planck's Radiation Law.* Taylor & Francis Ltd, 1976.

[Kuhn, 1961] Thomas S. Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, 1961.

[Kuhn, 1978] Thomas S. Kuhn. *Black-body theory and the Quantum Discontinuity.* University of Chicago Press, 1978.

[Kuhn, 1983] Thomas S. Kuhn. Commensurability, comparability, communicability. In *Proceedings of the Philosophy of Science Association.* Philosophy of Science Association, 1983.

[Kuhn, 1987] Thomas S. Kuhn. What are scientific revolutions? In Lorenz Kruger, Lorraine J. Daston, and Michael Heidelberger, editors, *The Probabilistic Revolution*, pages 7–22. The MIT Press, Cambridge, Massachussetts, 1987.

[Kuhn, 1989] Thomas S. Kuhn. Possible worlds in the history of science. In Allén Sture, editor, *Possible Worlds in Humanities, Arts, and Sciences: Proceedings of the Nobel Symposium 65*, pages 9–32. Walter de Gruyter, Berlin, 1989.

[Laird *et al.*, 1987] John Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1987.

[Langley *et al.*, 1987] Pat Langley, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press, 1987.

[Lenat and Brown, 1983] Douglas B. Lenat and Jon S. Brown. Why AM and Eurisko Appear to Work. *Artificial Intelligence*, 23, 1983.

[Lenat and Shepard, 1990] Douglas B. Lenat and Mary Shepard. *CYC: Representing Encyclopedic Knowledge.* Digital Press, 1990.

[Lenat, 1976] Douglas B. Lenat. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* PhD thesis, Stanford University, 1976.

[Lenat, 1983] Douglas B. Lenat. Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21, 1983.

[Minsky, 1954] Marvin Minsky. *Neural-Analog Nets and the Brain Model Problem.* PhD thesis, Princeton University, 1954.

[Minsky, 1986] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1986.

[Minsky, 1987] Marvin Minsky. Why aliens will speak english. In *CETI.* Conference on CETI, North-Holland, 1987.

[Poincaré, 1929] H. Poincaré. *The Foundations of Science, Science and Hypothesis, The Value of Science, Science and Method.* The Science Press, 1929.

[Polya, 1962] G. Polya. *Mathematical Discovery.* John Wiley and Sons, 1962.

[Rees and Clinger, 1986] Jonathan Rees and William Clinger. The revised^3 report on the algorithmic language scheme. Memo 848a, Artificial Intelligence Laboratory, MIT, 1986.

[Ritchie and Hanna, 1984] G.D. Ritchie and F.K. Hanna. AM: A Case Study in AI Methodology. *Artificial Intelligence*, (23), 1984.

[Schagrin, 1962] Morton L. Schagrin. Resistance to ohm's law. *American Journal of Physics*, 1962.

[Simon, 1966] Herbert A. Simon. Scientific Discovery and The Psychology of Problem Solving. In R. Colodny, editor, *Mind and Cosmos.* University of Pittsburgh Press, 1966. Also in the collection "Models of Discovery" [Simon, 1977].

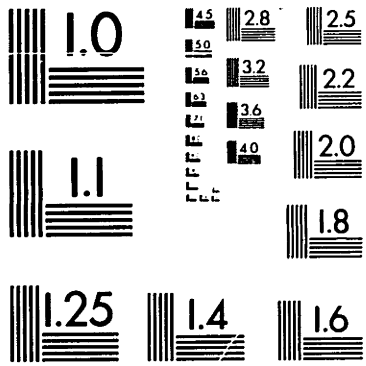
[Simon, 1977] Herbert A. Simon. *Models of Discovery*, volume 54 of *Boston studies in the philosophy of science.* D. Reidel Publishing Company, P.O. Box 17, Dodrecht, NL and Lincon Building, 160 Old Derby Street, Hingham, MA 02043, USA, 1977.

[Solomonoff, a] R. J. Solomonoff. Complexity-based induction systems: Comparisons and convergence theorems. *IEEE Transactions on Information Theory*, IT-24(4).

[Solomonoff, b] Ray Solomonoff. A system for incremental learning based on algorithmic probability.

[Stefik, 1979] M. Stefik. An Examination of a Frame Structured Representation System. In *Proceedings of the Sixth IJCAI.* IJCAI, 1979.

[Winston, 1984] Patrick H. Winston. *Artificial Intellience.* Addison-Wesley, 1984.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

24 : 1