

A Computational Theory of Early Mathematical Cognition

by

Albert Goldfain

June 1, 2008

Dissertation Committee:

William J. Rapaport (Major Professor)

Stuart C. Shapiro

Douglas H. Clements

A dissertation
submitted to the Faculty of the Graduate School
of State University of New York at Buffalo
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Department of Computer Science and Engineering

Copyright by
Albert Goldfain
2008

Contents

1	Introduction	1
1.1	Mathematical Cognition	1
1.2	Motivation	2
1.3	Two “Foundations” Programs	2
1.4	The “Burning” Questions	4
1.5	Computational Math Cognition	4
1.5.1	Understanding	5
1.5.2	SNePS for Mathematical Cognition	5
1.5.3	Focus	6
1.6	Significance	6
1.7	Methodology	7
1.8	Outline	8
2	Previous Work on Mathematical Cognition	9
2.1	Research from the Branches of Cognitive Science	9
2.1.1	Psychology	9
2.1.1.1	Piaget	10
2.1.1.2	Gelman and Gallistel	11
2.1.1.3	Wynn	13
2.1.1.4	Butterworth	14
2.1.2	Neuroscience	15
2.1.2.1	Dehaene	16
2.1.3	Philosophy	17
2.1.3.1	Shapiro	17
2.1.3.2	Wittgenstein	18
2.1.3.3	Glaserfeld	19
2.1.4	Linguistics	20
2.1.4.1	Lakoff and Núñez	20
2.1.4.2	Wiese	23
2.1.5	Education	24
2.1.5.1	Schoenfeld	24
2.1.5.2	Sfard	25
2.1.6	Anthropology	26

2.1.6.1	Sapir and Whorf	27
2.1.6.2	Hutchins	27
2.2	Research in Mathematics	28
2.2.0.3	Polya	28
2.2.0.4	Mac Lane	29
2.3	Research in Artificial Intelligence	30
2.3.1	Automated Theorem Proving	30
2.3.2	Production-System Simulations	31
2.3.2.1	Klahr	31
2.3.2.2	Fletcher and Dellarosa	31
2.3.3	Heuristic Driven Models	32
2.3.3.1	Lenat	32
2.3.3.2	Ohlsson and Rees	33
2.3.4	Connectionist Models	34
2.3.5	Tutoring Systems and Error Models	35
2.3.5.1	Nwana	36
2.3.5.2	Brown and VanLehn	36
2.3.6	Word-Problem Solving Systems	37
2.3.6.1	Bobrow	37
3	Understanding and Explanation: Towards a Theory of Mathematical Cognition	39
3.1	Claims	39
3.1.1	Multiple Realizability	39
3.1.2	Representability	40
3.1.3	Performability	42
3.1.4	Empirical Testability	42
3.2	SNePS	43
3.2.1	Overview of SNePS	43
3.2.1.1	GLAIR	43
3.2.1.2	SNIP	44
3.2.1.3	SNeRE	45
3.2.1.4	SNeBR	46
3.2.2	SNePS for Mathematical Cognition	46
3.3	Understanding	47
3.3.1	Modes of Understanding	47
3.3.2	Understanding in SNePS	49
3.3.3	Characterizations of Understanding	50
3.3.3.1	Logical Characterizations	50
3.3.3.2	Educational Characterizations	52
3.3.3.3	Computational Characterizations	55
3.4	From Knowledge to Understanding	55
3.4.1	Formalizations of JTB	56
3.4.1.1	Belief	56

3.4.1.2	Truth	57
3.4.1.3	Justification	57
3.4.2	JTB in SNePS	58
3.4.3	First-person knowledge	59
3.4.4	Understanding as a Gradient of Features	60
3.5	Exhaustive Explanation	60
3.5.1	A Turing Test for Mathematical Understanding	61
3.5.2	The Endpoint of an Explanation	62
3.5.3	Multiple Justifications	64
3.5.4	Behavior and Procedural Understanding	65
4	Abstract Internal Arithmetic	68
4.1	Representations for Abstract Internal Arithmetic	68
4.1.1	A Case-Frame Dictionary for Mathematical Cognition	69
4.1.1.1	Counting Case-Frames	70
4.1.1.2	Arithmetic Acts	72
4.1.1.3	Evaluation and Result Case-Frames	73
4.1.1.4	Results as Procepts	75
4.2	A Taxonomy of Arithmetic Routines	76
4.3	Semantic Arithmetic	77
4.3.1	Counting	77
4.3.1.1	The Greater-Than Relation	79
4.3.2	Count-Addition	81
4.3.3	Count-Subtraction	82
4.3.4	Iterated-Addition Multiplication	83
4.3.5	Iterated-Subtraction Division	85
4.3.6	Inversion Routines	87
4.4	Cognitive Arithmetic Shortcuts	88
4.5	UVBR and Cognitive Plausibility	90
4.6	Syntactic Arithmetic	91
4.6.1	Multidigit Representations	92
4.6.2	Extended Arithmetic	92
4.7	Questions in SNePS	93
4.8	Why Questions and Procedural Decomposition	95
4.9	Conceptual Definitions	97
4.10	Questions in Natural Language	98
4.11	A Case Study: Greatest Common Divisor	99
4.11.1	Human Protocols	100
4.11.2	A Commonsense Natural-Language GCD Algorithm	101
4.11.3	An Idealized Dialogue	101
4.11.4	Implementation	102
4.11.5	Cassie's Exhaustive Explanation of GCD	104

5	Embodied External Arithmetic	109
5.1	Quantities	110
5.1.1	Representations of Quantity	110
5.1.1.1	Category Systems	110
5.1.1.2	First-Order Logic	113
5.1.1.3	Natural Language	114
5.1.2	Quantities for Embodied Arithmetic	115
5.1.3	Natural Numbers in Quantities (n)	115
5.1.4	Sortals (s)	116
5.1.5	Relations Between Numbers and Sortals (r)	117
5.1.5.1	Collections	118
5.1.5.2	Constructions	118
5.1.5.3	Measures	120
5.1.6	Grounding Quantities in Perception (p)	121
5.1.6.1	Challenges	122
5.1.6.2	Grounded Quantities in SNePS	123
5.2	Embodied Enumeration and Its Applications	125
5.2.1	Models of Enumeration	126
5.2.2	Implementation	127
5.2.3	KL	128
5.2.4	PML	132
5.2.4.1	PMLa	132
5.2.4.1.1	BuildEnumFrame	132
5.2.4.1.2	Perceive	133
5.2.4.1.3	DistinguishByColor	133
5.2.4.1.4	DistinguishByShape	133
5.2.4.1.5	AddToCollection	133
5.2.4.1.6	AttendToNewObject	134
5.2.4.1.7	CheckIfClear	134
5.2.4.2	PMLb	134
5.2.4.2.1	threshold-classify	134
5.2.4.2.2	clock-wise-chain-code	134
5.2.4.2.3	object-gamma	136
5.2.4.2.4	wipe-object	136
5.2.4.3	PMLc	137
5.2.5	Sample Run	137
5.2.6	Representing Experience with a Sortal	140
5.3	Assignment Numeracy and its Applications	140
5.3.1	Self-Action Enumeration	141
5.3.1.1	Implementation	142
5.3.1.2	Sample Run	143
5.3.2	Ordinal Assignment to Self-Action	145

5.3.3	The Jail-Sentence Problem	147
5.3.4	The Breakfast-Always-Served Problem	148
5.4	A Model of Count-Addition Strategy Change	150
5.4.1	Strategies	150
5.4.2	Computational Models	151
5.4.3	Abstraction of Counting Targets	152
5.4.4	Commutativity, Comparison, and Cardinalities	152
5.4.5	<i>CA</i> to <i>CO</i>	154
5.4.6	<i>CO</i> to <i>COF</i> >: Metacounting	154
5.4.7	<i>CO</i> to <i>COF</i> >: Duration Estimation	154
5.5	Summary	155
6	A Unified View of Arithmetic	156
6.1	Multiple Realization	157
6.2	The Abstraction of Arithmetic: A Functional Analysis	158
6.2.1	Agent	158
6.2.2	Embodied Experience	160
6.2.3	From Embodied Experience to Abstract Facts	163
6.2.4	<i>BRIDGE</i> in SNePS	165
6.3	The Ontogeny of Sortals	166
6.4	Summary	168
7	Conclusion and Further Work	169
7.1	Summary	169
7.2	Significance	171
7.3	Attacks on Multiple Realizability	172
7.4	Extensions and Further Work	176
7.4.1	Integers, Rationals, and Reals	176
7.4.2	Geometry	178
7.4.3	Scaling Up	178
7.4.4	Syntactic Semantics	179
7.4.5	Quantity Disposition	179
7.4.6	Multi-agent Systems	181
7.4.7	Cognitive Robotics	182
7.5	Final Thoughts	182
A	SNePS Implementation	183

List of Figures

3.1	Two paths back to first principles.	63
3.2	Ways of understanding division.	64
3.3	Possible interactions between an interrogator and acting agent.	66
4.1	Cassie’s belief that $2 + 3 = 5$	76
4.2	Numeron-numerlog association. m_1, m_2 , and m_3 establish the finite initial segment of the natural-number progression up to three. m_4, m_5, m_6 , and m_7 associate this progression with the numerals 1, 2, 3, and 4. m_8, m_9, m_{10} , and m_{11} associate this progression with the number names.	79
4.3	Path based inference of $5 > 1$. The <i>lesser</i> arc of m_5 is implied by path p_1 (from m_1 to n_1) and the <i>greater</i> arc of m_5 is implied by path p_2 (from m_1 to n_5	80
4.4	Representing an extended layer (EL), and its interaction with the knowledge layer (KL).	94
4.5	Procedural Decomposition	96
5.1	Two grounding relations for the quantity 3 apples: (1) Perceived and (2) Visualized	124
5.2	Two models of enumeration	126
5.3	(a) Original 2D image (b) Ideal extracted object pixels	135
5.4	Chain-code representation of shape	135
5.5	The delta values computed from an object’s top-left object-pixel and its chain code.	136
6.1	A functional description of GLAIR.	160
6.2	The generalization of “ $3 + 2$ ”.	164
6.3	The act CountAdd ignores the (grayed out) embodied representation.	165
6.4	Linear discrimination of apple exemplars A from uncategorized exemplars B	167
7.1	Classes of objections to multiple realizability.	173
7.2	The procept “a count of x ”.	180

List of Tables

5.1	Aristotelian Categories.	111
5.2	Kantian Categories.	111
5.3	Husserl's Object Category-System	112
5.4	Category System of Hoffman and Rosenkrantz.	112
7.1	Quantity Dispositions.	181

Acknowledgments

Throughout the process of writing this dissertation I have enjoyed the support of many great minds and patient ears. I owe the largest debt to my major professor (and good friend) Bill Rapaport. Bill helped steer my idea from its sketchy infancy to something I am proud of. He consistently provided precise guidance, but also allowed me to discover the implications of my ideas (and allowed them to be *my* ideas). Bill has an uncanny skill for suggesting just-the-right-reading at just-the-right moment, and is able to take on the guise of philosopher, computer scientist, and cognitive scientist with equal expertise. Against the stereotype, I never dreaded our weekly one-on-one meeting (in fact, it is one of the things I will miss most).

I want to thank Stu Shapiro for his incredible dedication to both SNePS and the remarkable research group that has developed and maintained it for several decades. Stu inspired and encouraged the “AI by agent-building” approach that my work employed. His experience as a researcher was of great help in the “under-charted” waters of math cognition.

Since joining my dissertation committee, Doug Clements has been a great source of inspiration. He notes the urgency of providing a better mathematical education for children, and he has noted the importance of applying research-based methods towards this end. His comments were very insightful and he offered a unique perspective on my work. Remarkably, this was done while juggling his own papers, books, pilot studies, and service on a presidential childhood mathematics panel.

I want to especially thank Len Talmy for introducing me to the depth, complexity, and beauty of cognitive science. For a large part of my graduate career, I served as a TA for Adrienne Decker. I want to thank her for allowing me to develop my teaching skills and for providing a wealth of advice along the way.

I owe very much to the past, present, and future versions of SNeRG: Carl Alphonse, Josephine Anstey, Jonathan Bona, Deb Burhans, Trupti Devdas Nayak, Paul Heider, Michael Kandefer, David Pierce, Mike Prentice, Fran Johnson, Scott Settembre, and Patrice Seyed. I also received great interdisciplinary motivation from some superb linguists: Shane Axtel, Paula Chesley, Yana Petrova, Shakthi Poornima, Fabian Rodriguez, and Andy Wetta (and another special one who I acknowledge below), and from debating high-powered philosophers: John Corcoran, Randall Dipert, Mike McGlone, Steve Haladay, Amanda Hicks, Adam Taylor, and Bill Duncan.

I want to thank Peter Scott and Hung Ngo for being great mentors to me in the early stages of my graduate studies. I may not have gone on to get a PhD without their early encouragement. Also, I want to acknowledge the support of several co-workers at Welch Allyn (my perpetual “summer job” in the “real world”), especially: Ron Blaszak, Matt Bobowski, Song Chung, Chad Craw, Jim

Dellostritto, Frank Lomascolo, and David Rynkiewicz. In my opinion, these guys dream as big as anyone in academia.

This is also for all of those friends who saw to it that I finished my work and had some semblance of a “life” during my graduate career: Chris Bierl, Chris Clader (who I thank for reading an earlier version of this work), Jill Foley, Mike Gaudet, Rich Giomundo, Uday Kodukula, Hugh McNeill, and Jason Uronis.

Finally, this work is dedicated to my family. They brought me to this country and helped me flourish here. To my mother, for her love of language; to my father, for his love of science; to my brother, for his love of music and nature; to my grandparents, aunt, uncle, and cousin, for showing genuine interest in my progress; and to Kirsta, for keeping me sane, motivated, and loved throughout this journey.

For my family

Abstract

The primary focus of this dissertation is a computational characterization of developmentally early mathematical cognition. Mathematical cognition research involves an interdisciplinary investigation into the representations and mechanisms underlying mathematical ability. Significant contributions in this emerging field have come from research in psychology, education, linguistics, neuroscience, anthropology, and philosophy. Artificial intelligence techniques have also been applied to the subject, but often in the service of modeling a very restricted set of phenomena. This work attempts to provide a broader computational theory from the perspective of symbolic artificial intelligence. Such a theory should serve two purposes: (1) It should provide cognitively plausible mechanisms of the human activities associated with mathematical understanding, and (2) it should serve as a suitable model on which to base the computational implementation of math-capable cognitive agents.

In this work, a theory is developed by synthesizing those ideas from the cognitive sciences that are applicable to a formal computational model. Significant attention is given to the developmentally early mechanisms (e.g., counting, quantity representation, and numeracy). The resulting model is validated by a cognitive-agent implementation using the SNePS knowledge representation, reasoning, and acting system, and the GLAIR architecture. The implementation addresses two aspects of early arithmetic reasoning: abstract internal arithmetic, which is characterized by mental acts performed over abstract representations, and embodied external arithmetic, which is characterized by the perception and manipulation of physical objects.

Questions of whether or not a computer can “understand” something in general, and can “understand mathematics” in particular, rely on the vague and ambiguous term “understanding”. The theory provides a precise characterization of mathematical understanding, along with an empirical method for probing an agent’s mathematical understanding called ‘exhaustive explanation’. This method can be applied to either a human or computational agent.

As a case study, the agent is given the task of providing an exhaustive explanation in the task of finding the greatest common divisor of two natural numbers. This task is intended to show the depth of the theory. To illustrate the breadth of the theory, the same agent is given a series of embodied tasks, several of which involve integrating quantitative reasoning with commonsense non-quantitative reasoning.

Chapter 1

Introduction

In this dissertation, I develop a computational theory of developmentally early mathematical cognition. This chapter serves as a brief introduction to the field of mathematical cognition (sometimes called the cognitive science of mathematics or mathematical idea analysis (Lakoff and Núñez, 2000)) and motivates the need for computational theories. An outline of my specific research focus and my approach to developing a computational theory is given here.

1.1 Mathematical Cognition

Mathematical cognition¹ is an interdisciplinary investigation of the representations and mechanisms underlying mathematical ability. The domain of mathematics provides human beings with a robust set of tools for general-purpose reasoning, as well as a syntactic language through which real-world quantitative phenomena can be expressed. This makes it a potentially fruitful domain for cognitive science research. Currently, mathematical cognition is an emerging sub-field of cognitive science and has seen contributions from psychology, linguistics, anthropology, philosophy, and education. For the most part, mathematical cognition has focused on the cognitive abilities of humans (with the notable exception of animal-studies research on the mathematical abilities of non-human primates, birds, and dogs; see Chapter 2 for details). The primary focus has been on examining the earliest mathematical representations and mechanisms (i.e., number concepts, counting, mental arithmetic) rather than the workings of professional mathematicians.

The field of artificial intelligence (*qua* sub-field of cognitive science) is also in a position to make a significant contribution to mathematical cognition research. Computational modeling and simulation have become standard techniques in cognitive science, and computationalism pervades many theories of how the human mind works (Rapaport, 1993).

¹I will use “mathematical cognition” to refer to both the ability and the field of study. The intended sense should be clear from context.

1.2 Motivation

This work is both motivated by, and a response to, the theory of embodied mathematical cognition put forward by Lakoff and Núñez (2000). Their work has come to be viewed as one of the most influential (and controversial) in the field of mathematical cognition. One of its major claims is that mathematics is *not* purely syntactic, disembodied, and objective, but rather a highly semantic, embodied, and subjective endeavor. This is primarily a rejection of the Platonist conception of mathematics: the objects and operations of mathematics are not in an ideal Platonic realm, but rather are constructed by human beings.

I am very sympathetic to this view. I believe that meaning in early mathematics flows from our experiences as agents situated and acting in the world, our interactions with objects in the world, and our abstractions from these experiences. This is especially the case in the early conceptualization of numbers, counting, and arithmetic operations. I believe that any theory of mathematical cognition (computational or otherwise) cannot treat mathematics as *just* symbol manipulation or disembodied calculation.

However, in their argument for rejecting Platonic realism, Lakoff and Núñez conclude that mathematical understanding can only be realized in a *human* cognitive agent along with its particular embodiment, representations, and inference mechanisms:

Mathematics as we know it is human mathematics, a product of the human mind . . . a product of the neural capacities of our brains, the nature of our bodies, our evolution, our environment, and our long social and cultural history. (Lakoff and Núñez, 2000)

Thus, the implementation of mathematical cognition in a computational medium cannot result in a system that understands mathematics. In many ways, this is an argument against strong AI, using mathematical reasoning as a counterexample.

I disagree with this position, and I will argue for the multiple realizability of mathematical cognition. That is, a suitable theory of math cognition can yield a computational agent that understands mathematics. The defense of this position rests on two points: (1) The embodied activities that provide a semantics for arithmetic are performable by computational agents and (2) abstraction from these activities yields representations that can be reasoned with by such agents (while still remaining associated with the originating embodied activities).

1.3 Two “Foundations” Programs

Mathematics is among the historically oldest and developmentally earliest of human activities, yet taking a cognitive science approach towards the foundations of mathematics has not been very common. It is worth identifying two separate foundations programs in recent mathematical history. The *mathematical* foundations program took place at the turn of the twentieth century and is most often associated with Frege, Hilbert, Russell, Whitehead, Gödel and others. This was a movement initiated by mathematicians for the purpose of making mathematics more rigorous. This effort largely rested on re-phrasing the various branches of mathematics in the language of formal logic and set theory. Interestingly, this movement emphasized the exclusion of psychology:

It may, of course, serve some purpose to investigate the ideas and changes of ideas which occur during the course of mathematical thinking; but psychology should not imagine that it can contribute anything whatever to the foundation of arithmetic (Frege, 1884/1974).²

The exclusion of psychology from the mathematical foundations program is not surprising. Cognitive science did not yet exist as a unified science, and the psychological methods of the time were not characterized by the rigor and precision that the mathematicians were seeking.

Throughout the twentieth century, researchers such as Piaget set the epistemic groundwork for a new sort of foundations program (Piaget, 1965; Beth and Piaget, 1966). At the turn of the twenty-first century, a *cognitive* foundations program emerged:

The central question we ask is this: How can cognitive science bring systematic rigor to the realm of human mathematical ideas, which lies outside the rigor of mathematics itself? Our job is to help make precise what mathematics itself cannot: the nature of mathematical ideas (Lakoff and Núñez, 2000)

I will not go as far as to claim that mathematical idea analysis is necessarily outside the scope of mathematics. However, I do believe that just attempting to provide a computational account of math cognition can shed some light on the cognitive semantics of mathematics (i.e., that such an account *is* a kind of mathematical idea analysis).

Another trend of this cognitive foundations camp is to separate mathematics from the logical foundations of mathematics for the purposes of educating students:

[M]athematics could find itself thus reduced to logic, which is certainly not a view that it would be worthwhile conveying to students (Sierpiska, 1994)[p.70]

I do not think that either mathematical foundations program needs to segregate itself from the other. In fact, they are both useful perspectives with different goals and “tools of the trade”. Rather than banish psychology or logic, I will develop a foundation that borrows from both foundations programs and is augmented with computational techniques. The prevalence of computationalism in cognitive science is actually evidence for the compatibility of psychology and logic since understanding the brain as a computer requires both disciplines.

The mathematical foundations program cannot tell the whole story of mathematical cognition because it insists on minimal ontology and a policy of reductionism (i.e., everything is sets and logic). The cognitive foundations program is incomplete and dispersed across disciplines. Perhaps importing aspects of both foundations programs into a computational theory will help to overcome these difficulties. Brian Butterworth, in a preface to the very first volume of the journal *Mathematical Cognition*, seems to call on philosophers and logicians for an integration of the foundations programs:

One issue that has only recently arisen as a problem for empirical research is the best way to characterise and individuate mathematical concepts, and what it means to understand them. Here philosophers and logicians can be of real help. (Butterworth, 1995)

²Frege was reacting primarily to the psychologically motivated theories of John Stuart Mill (1843/2002)

The question of what it means to understand a mathematical concept will occupy the majority of this work, and I believe, like Butterworth, that it is an empirical one.

1.4 The “Burning” Questions

While any field of inquiry can be characterized by its achievements and accomplishments, it is also sometimes useful to characterize it by its open questions. This is especially useful for the young sub-disciplines of cognitive science. Some of the “burning questions” in mathematical cognition are:

1. An Innateness Question: *How much mathematical ability is innate?* Do we have some mathematical concepts “hard-wired” into our brains at birth? Or, perhaps, as Locke would claim, we are born with a “blank slate” and must construct such concepts from our earliest experiences. If some numerical abilities are innate, then how do we look for these abilities in infants and non-human animals in a scientifically rigorous way? These are specific instances of the more general nature-vs-nurture questions in cognitive science.
2. An Embodiment Question: *How do our bodies impact the way we learn and understand mathematics?* Does having ten fingers lead us to prefer a decimal notation? How is mathematical ability “implemented” across individuals, cultures, and (perhaps) species?
3. A Dissociability Question: *To what degree can mathematical cognition be dissociated from other cognitive systems?* In particular, can mathematical cognition be dissociated (i.e., vary in a neurologically independent way) from language cognition?
4. The Relational Question: A family of questions of the form “What is the relationship between X and mathematical cognition?” where X can be perception, action, metacognition, logical cognition etc.

More specific questions can be raised when examining mathematical cognition from a particular sub-discipline of cognitive science. For example, a neuroscientist may be interested in which parts of the brain are active during numeric reasoning, or determining how mathematical reasoning is impacted by a stroke. From the perspective of AI, we are interested in questions of representation and implementation.

1.5 Computational Math Cognition

I believe that a large part of early mathematical cognition can be expressed in a computational theory and hence, can be implemented in a computational cognitive agent. The extent of such an implementable part is an empirical question and, even if this portion turns out to be smaller than expected, we will gain some understanding in the attempt.

Any computational theory that defends multiple realizability of a human activity must be based on an abstraction from that activity and an implementation of this abstraction in some computational medium (Rapaport, 1999; Rapaport, 2005). The abstraction, by its very nature, must ignore

some details of the original implementation (i.e., the implementation of mathematical cognition in human beings) in order to remain general. I will base my abstraction of human mathematical cognition on the relevant cognitive science research, looking for those features of human mathematical ability that are suitable for a computational implementation. The re-implementation of the abstraction in a computational medium will fill in the details removed by the abstraction, but perhaps in different ways and with implementation side effects. As such, there are two places where a computational theory can be praised or criticized: the abstraction and the computational implementation.

Various mathematical abilities are worthy candidates for abstraction and various computational paradigms might be considered suitable for the implementation medium. This work will focus on the representations and abilities required for early mathematical understanding and an implementation of these in a symbolic AI medium.

1.5.1 Understanding

It is worth asking why a computational cognitive theory should concern itself with foundations at all (mathematical or cognitive). The answer lies in the nature of mathematical understanding. Following Rapaport (1988,1995) I believe that understanding is recursive. We always understand something in terms of other, antecedently understood things. Because understanding is possible, this recursion must have a basis, and it is worth looking at the foundations programs for such a basis.

Despite claiming an ultimate reliance on the embodied human mind, Lakoff and Núñez (2000) claim that mathematical procedures *qua* algorithms can be detached from the human embodiment completely:

... one of the best things about mathematical calculation—extended from simple arithmetic to higher forms of mathematics—is that the algorithm, being freed from meaning and understanding, can be implemented in a physical machine called a computer, a machine that can calculate everything perfectly without understanding anything at all (Lakoff and Núñez, 2000)[p.86]

The question then arises: Can a computational agent come to understand human mathematics? More narrowly, can a computational agent come to understand human arithmetic? Importantly, we can treat these questions as empirical ones once we have a criterion for mathematical understanding in hand. Throughout this work we will be concerned with the conditions for mathematical understanding in its various guises.

1.5.2 SNePS for Mathematical Cognition

The SNePS knowledge representation, reasoning, and acting system (Shapiro and Rapaport, 1987; Shapiro and Rapaport, 1995; Shapiro, 2000; Shapiro and SNePS Implementation Group, 2008) is an ideal target for implementing such a computational theory of mathematical cognition. SNePS is a general model of cognition that does not make presuppositions about the domain in which the

knowledge engineer is working. This strength is reinforced by the applications in which SNePS agents have been deployed.

SNePS falls within the tradition of symbolic artificial intelligence. This work is not intended to exclude other approaches from (e.g., connectionist, production-system models). Indeed, the project of mathematical cognition is ambitious enough to require input from all areas in AI. However, I will defer a justification of why a symbolic approach was used until the conclusion (see §7.3). At that point, there will be an agent implementation on the table (i.e., something whose relative merits and flaws can be discussed).

This research is the first application of SNePS to a theory of mathematical cognition. Agents developed in SNePS are often provided with an *ad hoc* way of doing the task-specific math they need. This dissertation strives towards providing a more general method of providing mathematical reasoning to SNePS agents.

1.5.3 Focus

Early mathematical cognition is a rich enough topic to involve almost every aspect of early reasoning. Without narrowing this research, a mathematical theory of early math cognition could become what Shapiro (1992) calls an “AI-complete” problem, i.e., a problem whose solution would require the solution of all major research problems in AI.

This research can be viewed from several perspectives, some of which will receive more attention than others. I hope this dissertation will serve as each of the following:

- A reply to the position of Lakoff and Núñez that the human mind, along with its particular embodiment, representations, and inference mechanisms, is privileged with respect to mathematical understanding.
- A computational model of human mathematical cognition.
- A philosophical investigation of the necessary and sufficient conditions for mathematical understanding.
- A definition of mathematical understanding broad enough to apply to both human and computational agents, together with a set of empirical tests for features of mathematical understanding.
- A computational approach to the cognitive semantics of mathematics.
- A specification of the representations and inference rules necessary for mathematical cognition in SNePS.

1.6 Significance

This research can be viewed from a variety of perspectives. It can be taken as a work of computational psychology or computational philosophy, which are two important aspects of AI research

(Shapiro, 1992; Rapaport, 1993). As a work of computational psychology, it can be seen as being that part of current psychological theory and data that is readily applied to a computational implementation. It also may serve as a simulation of human behavior and a platform for predicting human outcomes in various quantitative tasks. As a work of computational philosophy, it should be seen as a response to Lakoff and Núñez (2000), as well as a philosophical investigation of fundamental questions from a computational perspective. This is significant because answering a question “What is a number?” must be precise to be useful in an implementation.

This work may also be significant to education researchers. The agent’s understanding will very much depend on the order of acquired concepts and abilities (the agent progresses through something like Piaget’s stages of development). This work can provide evidence as to why some students find it hard to understand (and explain) certain concepts.

A good deal of effort concerns computational semantics. This work can also be seen as a computational account of how such an abstract realm of human concepts and acts ends up meaning so much and finding such widespread application.

Finally, as the prevalence of the word “computational” above may suggest, I believe this is a significant project for computer science. It investigates how to build a better computational agent, equipped with plausible representations and powerful general rules of inference. AI researchers have taken up the challenges presented by language (in the AI subfield of computational linguistics) and have found a wealth of research potential pursuing it. I believe that a similar effort should be put towards computational mathematical cognition and that, with such an effort, a wealth of new cognitive science projects may arise. This work is significant because it pursues this end.

I want to stress that this work outlines a preliminary theory. I certainly have not solved all of the problems in mathematical cognition, nor have I given the final answers to the burning questions. However, I do believe that developing this computational theory makes the important issues much clearer.

1.7 Methodology

I will offer an in-depth investigation of a particular application requiring abstract arithmetic: finding the greatest common divisor (GCD) of two natural numbers. Clearly, the concept of GCD is clearly acquired and deployed “after” anything which might be called *early* mathematical reasoning. However, GCD is useful in demonstrating the theory’s ability to scale beyond counting and arithmetic. Moreover, I will use a commonsense algorithm that can be decomposed into the early (and often used) early arithmetic routines.

After examining the GCD problem, I will investigate the theory’s applicability to a breadth of embodied (external) applications. In so doing, I will examine how suitable the representations and inference mechanisms are across applications.

I will strive to make the agent implementation of the theory cognitively plausible. Wherever this is not possible, I will analyze the obstacles to plausibility.

The agent implementation is described from a bottom-up perspective. Counting is described first, and other abstract arithmetic abilities are built up from this starting point. Counting will then be shown to occupy a sort of “middle ground” in an agent’s path to numeracy. Indeed, the path

leading to count-based abstract arithmetic includes embodied experiences.

As mentioned above, my implementation will primarily use the techniques of symbolic AI (as opposed to connectionist techniques) and will focus on numeric reasoning (as opposed to geometric reasoning). Among other reasons, this helps focus the work. However, this is in no way intended to diminish the importance of investigating the different aspects of mathematical cognition using different computational formalisms.

1.8 Outline

In Chapter 2, I will review the relevant literature for this research. I will survey the contributions from the cognitive sciences outside of AI and attempt to extract features that are usable in a computational theory of math cognition. I will also survey the existing models of mathematical reasoning within the AI literature.

In Chapter 3, I will elaborate the varieties of mathematical understanding and their conditions. The totality of an agent's understanding will be treated as a gradient of features. An attempt to describe necessary and sufficient conditions for several features of computational math understanding will be made. These conditions are derived by extending the classic conditions for knowledge from epistemology. Also in Chapter 3, I will describe the SNePS knowledge-representation, reasoning, and acting system which will serve as the underlying representation, logic, and agent-implementation platform for my theory.

In Chapter 4, I will discuss a computational theory of abstract (internal) arithmetic and will provide an extended case study involving greatest common divisors.

In Chapter 5, I will discuss a computational theory of embodied (external) arithmetic and some non-mathematical applications of numeracy.

In Chapter 6, I will unify and generalize the perspectives offered in chapters 4 and 5 by demonstrating how the external-internal gap is bridged.

Chapter 2

Previous Work on Mathematical Cognition

In this chapter I survey the relevant literature in mathematical cognition. First, I briefly review the significant contributions from the branches of cognitive science: psychology, neuroscience, philosophy, linguistics, education, and anthropology. I also group with these some contributions from professional mathematicians who have considered the cognitive aspects of their discipline. I then review in detail the existing work from artificial intelligence (and its subfields). I will defer a review of the mathematical-understanding literature until the next chapter, where that theme will be discussed in more detail.

This chapter is by no means an exhaustive list of relevant readings, but instead a broad survey of the major influential works in the field.¹ My specific contribution is in the area of knowledge-representation and reasoning, using the symbolic AI approach, so I will necessarily restrict myself to those research threads informing this work. I hope that, after reading this chapter, the reader will feel that there is a definite niche for symbolic computational theories of math cognition.

2.1 Research from the Branches of Cognitive Science

Cognitive science is a unification of disparate disciplines. This multidisciplinary aspect is significant, because it forces the cognitive science researcher to consider a handful of methodologies and to select from a variety of vocabularies when laying out a theory. In this section, I consider contributions from the sub-disciplines of cognitive science.

2.1.1 Psychology

The field of psychology has provided the longest and broadest sustained effort in the investigation of mathematical cognition. Psychologists from Piaget onward set a “bottom-up” agenda for math cognition by focusing on the earliest abilities of counting and arithmetic in children rather than on the thought processes of expert mathematicians.

Psychological investigations of mathematical ability tend to focus on the following questions:

¹Another survey of the field of mathematical cognition is Gallistel (2005) and an anthology of important readings can be found in Campbell (2004).

- **Innateness:** Are mathematical representations and abilities innate? Psychologists consider an ability to be innate if its presence can be empirically detected in human infants or non-human primates. Thus a bulk of the psychologist's task is to develop experiments that will demonstrate certain abilities in these non-verbal subjects.
- **Universality:** Is mathematical reasoning distributed universally across different reasoning tasks, across different cultures, or across different species?
- **Development:** What are the stages of numeracy? How do representations change during learning and across tasks?
- **Abnormal Psychology:** What can pathologies such as number-blindness or dyscalculia tell us about mathematical ability and its relation to other cognitive systems?

Many psychological issues stemming from these questions have been examined theoretically and experimentally.

2.1.1.1 Piaget

Jean Piaget (1965) studied young children's numerical abilities within his larger framework involving the stages of cognitive development. Mathematical competence, in his view, develops stage by stage alongside spatial and temporal reasoning abilities. In particular, he looked at abilities such as conservation of discrete and continuous quantity, provoked and spontaneous one-to-one correspondence, ordination, and cardination.

Piaget saw a parallel development of numerical and logical ability:

Our hypothesis is that the construction of number goes hand-in-hand with the development of logic, and that a cardination-numerical period corresponds to a pre-logical level (p. viii)

With this in mind Piaget was also troubled by the banishment of the psychological from the mathematical foundations program (cf. Chapter 1):

[T]he connections established in the field of experimental psychology needed to be verified in the field of logistics. In studying the literature on the subject, we were surprised to find to what extent the usual point of view was 'realist' rather than 'operational' ... [t]his fact accounts for the connections, many of them artificial, established by Russell, which forcibly separated logistic investigation from psychological analysis, whereas each should be a support for the other in the same way as mathematics and experimental physics (p. ix)

Of particular importance to my theory is Piaget's method. His is a "procedure of free conversation with the child, conversation which is governed by the questions put, but which is compelled to follow the direction indicated by the child's spontaneous answers" (p. vii). Piaget makes the tacit assumption that children can verbalize their mental procedures and representations and that

this verbalization is a window into their understanding. Also, the fact that a conversation is compelled to follow a course dictated by responses proves important to my conception of a method of exhaustive explanation during a mathematical Turing test (see Chapters 3 and 4).

An important consequence of a staged-theory of cognitive development for computational modeling is that we can treat a given computational agent at a given time as being a “snapshot” of a particular developmental stage. We can then attempt to model the mechanisms involved in the transition to the next stage or else ask what competencies the agent can exercise while occupying the stage. As an example, we might attempt to model the transition of a child in stages of counting. In an early stage, the child requires physical objects to be wholly present in order to count, whereas, in a later stage, they can “count in their head”.

Piaget (1955) also addressed the importance of sensori-motor experience in the very earliest stage of cognitive development. The ability to initiate actions from an embodiment (however limited) is foundational to any further development.

2.1.1.2 Gelman and Gallistel

An influential and in-depth study on preschool mathematical cognition is given by Rochel Gelman and Charles R. Gallistel (1978) . Their count-centric approach is evident from the outset:

[I]t appears that the child’s arithmetic reasoning is intimately related to the representations of numerosity that are obtained by counting. The domain of numerosities about which the child reasons arithmetically seems to expand as the child becomes able to count larger and larger numerosities. Although it remains possible that children younger than age 2 can recognize differences in numerosity without counting, representations of numerosity obtained by direct perception do not appear to play a significant role in arithmetic reasoning (p. 72)

The authors treat subitization, i.e., the automatic recognition of the cardinality of a collection, as an organizing principle that facilitates counting, not as a replacement for counting. As evidence for this, Gelman and Gallistel cite situations in which small subitizable collections are shown to young children for very short intervals, and, even when naming the correct cardinality, the children complain that they were not given enough time to count the items. I attempt to integrate knowledge-level counting and perceptual-level subitization in the task of embodied enumeration described in Chapter 5.

Gelman and Gallistel point out that the counting model for adults is agreed upon and well analyzed:

It involves the coordinated use of several components: noticing the items in an array one after another; pairing each noticed item with a number name; using a conventional list of number names in the conventional order; and recognizing that the last name used represents the numerosity of the array [p. 73]

However, the child’s counting model is significantly different from the adult’s. In particular, children know how to count before they know the count words of their particular language. The authors

call the pre-verbal mental tags used in counting *numérons*. Numerons must be unique and applied in a fixed order (p. 76). Numerons are contrasted with *numerlogs*, which are the actual counting words of a language. I maintain the numeron-numerlog distinction in my computational theory.

The most cited aspect of Gelman and Gallistel (1978) is the proposed “how-to-count principles” employed in the child’s counting model:

1. **The One-One Principle:** Use one and only one numeron for each item in the array to be counted. This involves two processes: (1) partitioning the array into counted and uncounted sets, and (2) “summoning” distinct numerons for items as needed. The coordination of these processes is helped by pointing to each item as it is tagged (p. 77).
2. **The Stable-Order Principle:** The numerons used to tag items in the array must be “arranged in a stable — that is repeatable — order” (p. 79).
3. **The Cardinal Principle:** The ability to designate the last numeron used as the numerosity (cardinality) of the array (p. 80).
4. **The Abstraction Principle:** “The preceding principles can be applied to *any* array or collection of entities” (p. 80).
5. **The Order-Invariance Principle:** “The order of enumeration is irrelevant . . . the order in which items are tagged, and hence which item receives which tag, is irrelevant” (p. 82).

When discussing the stable-order principle, Gelman and Gallistel make the following point:

It is well known that the human mind, unlike the computer, has great difficulty in forming long, stably recallable lists of arbitrary names...having no generating rules underlying the sequence (p. 79).

Thus, to make a computational theory of early numerical processing plausible, the number of numerons will be kept to 10.

When discussing the abstraction principle, Gelman and Gallistel point out:

[F]or the purposes of counting, perceptual entities may be categorized in the most general way, as *things*. Even *things* may be too limiting a specification of what children and adults regard as countable. We have observed children counting the *spaces* between items in an array. From the standpoint of the adult, this extremely general category of what is countable may seem highly abstract. The view that the ability to classify physical events as things is indeed very abstract and is implicit in theories of cognitive development that postulate an elaborate hierarchy of subcategories as the basis for categorization skills (p. 81).

I refer to this abstracted class of countable things as “sortals” (see Chapter 5).

2.1.1.3 Wynn

Karen Wynn is best known for her remarkable preferential looking time experiments described in Wynn (1992,1995), designed to show the innateness of numerical representation by testing infants. After undergoing a process of habituation, infants tend to stare longer at results that are arithmetically unexpected, i.e., at a situation that violates an arithmetic fact. In Wynn's experiment, two puppets are shown and then covered by a screen. The infant then sees one puppet removed and the screen is lifted. The infant stares for a shorter time when the result is the expected result (one puppet), but stares longer (indicating surprise) when there are zero or two puppets when the screen is lifted. Some empiricist-leaning psychologists have pointed out some problems with Wynn's experiment (Carey, 2002), but it is generally agreed that some aspects of numeracy must be innate or else develop in infancy. What this result indicates is that it is perfectly plausible for a computational theory of early math cognition to assume some hard-wired facts or pre-programmed primitives. It is, of course, debatable as to *which* abilities can be plausibly hard-wired into a computational system, but the goal should be to limit these to those features introduced by the computational formalism itself (e.g., the fact that I use SNePS implies that my agent inherits the "innate" ability to perform the sorts of inferences SNePS agents are capable of) and to otherwise keep this set as small as possible.

Wynn (1996) examines an infant's ability to individuate and enumerate actions. The action in question is the number of hops of a puppet. Her findings show that "infants can individuate and enumerate actions in a sequence, indicating that their enumeration mechanism is quite general in the kinds of entities over which it will operate" (p. 164). This sequential action enumeration is a more general case of a notion of self-action enumeration I present in Chapter 5. Action enumeration provides a unique challenge that is not present in object enumeration. Human infants have been shown to subitize small numbers of objects. However, a sequence of two or three actions cannot be subitized because the actions must unfold in time. Thus, actions are an altogether different kind of entity to be counted:

If infants' numerical discrimination ability results from a pattern-recognition process, it should not apply to kinds of entities other than objects or depictions of objects . . . Alternatively, the underlying mechanism may be inherently unrestricted in the kinds of entities it can count (p. 164).

In subsequent chapters, we shall see that even actions can be the target of counting for a computational agent. My computational agent is able to count its own actions because those actions are reified as nodes in the agent's semantic network, which is a representation of its mind.

In SNePS, agent actions are explicitly individuated from neighboring actions. Each distinct act is either primitive, or given in terms of a well-defined action that can be decomposed into individuated primitives. Self-action enumeration is significant as a computational theory in light of Wynn's commentary on human action enumeration:

. . . virtually nothing is known about how people individuate actions. The individuation of actions is likely to be complex because actions are not definable purely in terms of objective properties. Frege's observation that number is not an inherent property of

portions of the world applies as much to actions as it does to physical matter. Just as a given physical aggregate may be an instance of many different numbers . . . so may a given portion of an activity (one dance may equally be an instance of 437 steps, one speech may be an instance of 72 utterances) (p. 165).

Sharon and Wynn (1998) extend Wynn (1996) by demonstrating how infants can “parse a stream of motions into distinct actions” (p. 357). The restrictions on the enumeration ability is that “action boundaries are specified by motionlessness . . . [infants] are not able to parse heterogeneous actions from a continuous, nonpatterned stream of motion” (p. 361). The “stream of motion” is only significant for a computational agent if the agent has an ontological conception of entities in that stream. This is supported by Sharon and Wynn’s psychological findings:

[A]dults watching someone perform a leap followed seamlessly by a bow are aided in the perception of two actions by their concepts of “leap” and “bow.” The fact that infants were unable to parse jumps and falls from an ongoing stream of motion suggests that they lack concepts for these basic-level actions (p. 361).

The nature of what makes an action “basic” has been dealt with in the philosophical literature as well (Baier, 1971).

2.1.1.4 Butterworth

Brian Butterworth (1999) puts forth and defends a “Mathematical Brain” hypothesis. This is a view that “the use of numbers is so universal that it is reasonable to infer that everyone is born with a Number Module” (p. 21). By Number Module, he is referring to a dedicated component for numerical cognition (specifically the recognition of numerosities). The idea of cognitive module comes from Jerry Fodor’s modular theory of cognition (Fodor, 1983). In Fodor’s view, the processing of numerosities is a “central process”. As such, numerical ability is a general-purpose skill, but not necessarily a fast automatic ability. The fast automatic abilities (like color perception) are called “cognitive modules” in Fodor’s theory. Butterworth argues against Fodor’s view to demonstrate the existence of a dedicated Number Module. He does this in order to argue for the ubiquity of numerical representations in human life. This supports his view that numbers were independently and spontaneously conceived across various cultures rather than originating from a single “inventor” (or a single culture) and spreading around the world.

Butterworth’s theory is number-centric and, as such, highlights the importance of counting. He identifies a variety of number types deployed in numerical cognition:

1. **Numerosities** A number associated with a collection. I call such numbers “cardinals”.
2. **Ordinal Numbers** Used for specifying positions in a sequence.
3. **Numerical Labels** Used to tag things for identification (e.g., a zip code). I call such numbers “nominals”.
4. **Fractions** The rational numbers (i.e., ratios of natural numbers).

5. **Measure Numbers** Answer the questions “How much?”, “How long”, “How heavy” etc. I call these “measure quantities binding to a unit sortal”.
6. **Cyclical Numbers** A set of numbers with a largest element that acts as a modulus and thus causes a repetition in the numbers. The prototype example is clock time. I do not deal with such numbers extensively in my theory.
7. **Infinite Numerosities** Cantorian transfinite cardinal and ordinal numbers.

It should be stressed that although these various conceptions are unified by being number types, they often originate in vastly different conceptual systems. For example, Butterworth indicates that ordinal numbers and nominal labels are often expressed by unrelated words in natural language.

Butterworth takes great care to demonstrate what he calls the “cognitive archeology” of numerical abilities and representations. Numbers have sprung up in all cultures and at all points in history (some of the earliest human artifacts are mathematical representations).

A well-studied phenomenon in mathematical psychology is a child’s transition between count-addition strategies in addition. This cognitive leap represents a spontaneous optimization of procedures and is observed almost universally *before* formal mathematical instruction.

Butterworth et al. (2001) provides a unique model of this transition, in which a comparison of addends is made before any counting begins. This develops into a “count-on-from-greater” strategy, which is guaranteed to be at least as fast (if not faster) than the count-on strategy.

As experience of addition increases, counting on from the larger addend could serve as the basis for the organization of facts in memory . . . It also implies that the process of solving a simple addition problem involves comparing the two addends to determine the larger and transforming, if necessary, a smaller-larger sum to a larger-smaller sum (p. 1009).

Butterworth’s model is relevant for our computational-agent implementation. First, it indicates that the notion of “greater-than” is developed as early as counting and count-addition. This is of particular interest, because we would like to have a computational agent explain the “greatest” component of a greatest common divisor. Secondly, it suggests that the commutativity of addition is discovered before formal arithmetic is taught.

I will defer further discussion of other computational models of count-addition strategy shifts until Chapter 5, where I discuss how this is modeled in SNePS.

2.1.2 Neuroscience

Recent advances in brain imaging techniques (e.g., fMRI and PET-scan) have led to significant results from the field of neuroscience. In the area of mathematical cognition, the neuroscientist is interested in the following questions:

- **Locating Math in the Brain:** What brain areas are active during mathematical problem solving? How does processing shift between brain areas responsible for different tasks?

- **Approximate vs. Exact Representation:** How does the brain handle the representation of small, exact quantities and large, approximate quantities?
- **Mental Number Line:** How do the architecture and memory limitations of the human brain impact the representation of the mental number line?

There is indeed some overlap between these questions and those posed by psychologists. The field of cognitive neuroscience is still quite young, and I suspect that several exciting results about human math cognition are not far off.²

2.1.2.1 Dehaene

Stanislas Dehaene (1992,1997) provides a broad theory of numerical cognition. One of his central arguments is for the dissociability of different numerical abilities and representations that are sometimes lumped together under the umbrella of the “number concept”. He uses evidence from patients with “aphasic acalculia” to show the following:

The processing of arabic numerals . . . can be dissociated neuropsychologically from the processing of verbal numerals . . . The neuropsychological approach has culminated in identifying dissociation of lexical and semantic transcoding processes (Dehaene, 1992, p 5)

He makes a case for an independent “number sense” that is dissociable from (and not purely generated from) the language system. This suggests that a computational cognitive agent may require special mechanisms that extend the abilities of a purely linguistic processing system.

Dehaene (1997, Ch 8) discusses the possibilities of using brain-imaging techniques to uncover brain processing during a mathematical task.

In the past few years, new tools . . . have begun to provide pictures of brain activity in living, thinking humans. With modern brain imaging tools, a short experiment is now sufficient to examine which brain regions are active while a normal subject reads, calculates, or plays chess. Recordings of the electrical and magnetic activity of the brain with millisecond accuracy allow us to unveil the dynamics of cerebral circuits and the precise moment they become active (p. 208).

Dehaene also discusses the *problem-size effect* in early arithmetic: “The time to solve single-digit addition or multiplication problems such as $2 + 3$ or 4×7 increases with the size of the operands”³(Dehaene, 1992, p 6). I will examine the problem-size effect in the context of a computational implementation of count-addition in Chapter 5.

²As the subfield of cognitive science that is most closely associated with the actual human “hardware” (i.e., the brain), we might expect that neuroscience has little to offer computational theories. However, very recent results in Anderson (2007) use ACT-R to build computational models that match actual brain-region functionality.

³See Ashcraft (1992) for various models of the response-time delay relating to the problem-size effect.

2.1.3 Philosophy

The philosophy of mathematics is chiefly concerned with three questions:

- **Ontology:** What is the ontological status of mathematical objects (numbers, sets, lines, etc.)?
- **Epistemology:** What are the knowledge conditions for mathematical propositions?
- **Semantics:** What is the meaning of mathematical expressions?

These questions have divided philosophers into “camps” (e.g., the ontology issue separates philosophers into realists and antirealists).

2.1.3.1 Shapiro

Despite working in a fully-intensional system like SNePS, I find the structuralist realism of Shapiro (1997) to be the most informative ontology for my theory. This is a realist position (i.e., a position that mathematical objects have an objective existence that can be learned and referred to intersubjectively). Similar positions have been elaborated by Isaacson (1994) and Resnik (1997). Shapiro makes clear “what is at stake” when choosing a position on the semantic question as well:

One strong desideratum is that mathematical statements have the same semantics as ordinary statements, or at least respectable scientific statements . . . scientific language is thoroughly intertwined with mathematical language. It would be awkward and counterintuitive to provide separate semantic accounts for mathematical and scientific language, and yet another account of how various discourses interact (p. 3).

This is an issue even in a theory of early cognition, because the child must square the meaning of statements presented formally (e.g., $1 + 2$) and statements presented in terms of concrete real-world objects (e.g., $1 \text{ cow} + 2 \text{ dogs} = ? \text{ animals}$).

Under the structuralist view, mathematical objects such as numbers are positions in instantiations of structures (places in instantiations of patterns for Resnik (1997)). A useful metaphor is to think of such objects as offices. When I make the assertion “The vice-president is next in the line of succession to the president”, I can make inferences about:

1. exemplars: all pairs of office-holders in these offices, from Washington and Adams to Bush and Cheney
2. future (possible) exemplars: the unspecified future holders of these offices
3. the structure itself: a part of the hierarchical organization of the American government regarding line-of-succession.

Another useful example (from Shapiro): When I assert “The shortstop is the hardest position in a baseball defense”, I can make a general inference about any player instantiating the position (e.g., for the particular player who may be filling that role for the New York Yankees). The instantiations of positions in a structure “inherit” the structural properties of the particular position within the

structure. Mathematical assertions such as “8 is divisible by 2” are really saying something about a relationship between positions in any instantiation of the natural-number progression (and, as it happens, any instantiation of the real-number progression). This assertion is saying something about a relationship between whatever may be playing the role of 8 and whatever may be playing the role of 2 in any instantiation.

Shapiro details three “flavors” of structuralism:

1. **Ante Rem Structuralism:** “Structures and their places exist independently of whether there are any systems of objects that exemplify them” (p. 9).
2. **Eliminative Structuralism:** “Statements [of mathematics] are not about specific *objects*. Rather each such statement is a generalization over natural-number *systems*” (p. 9)
3. **Modal Structuralism:** “Statements [of mathematics] are about all *possible* natural-number systems” (p. 10).

Shapiro aligns himself with *Ante Rem* structuralism, but I make no commitment to which of these positions is correct.

If mathematics is all about structures having an independent existence, then understanding mathematics amounts to understanding these structures. One philosophical reason I believe a computer can come to understand mathematics is that a computer can access the same exemplifications of structures as human beings.

2.1.3.2 Wittgenstein

Ludwig Wittgenstein (1939/1976) adopted a “meaning-is-use” approach to addressing the semantic issue in mathematics:

The use of the word “understand” is based on the fact that in an enormous majority of cases when we have applied certain tests, we are able to predict that a man will use the word in question in certain ways. If this were not the case, there would be no point in using the word “understand” at all (p. 23).

He brings up an important problem for a computational assessment of understanding by pointing out that a person (or computer) could just be programmed with a large lookup table of rules which determines the use of terms perfectly, but that understanding must be more than mere behavior.

Wittgenstein also addresses the notion of mathematical discovery:

There is no discovery that 13 follows 12. That’s our technique—we *fix*, we teach, our technique that way. If there’s a discovery—it is that this is a valuable thing to do (p. 83).

I believe that a great majority of early mathematics is just learning conventions and “proper use”, but I also believe that Wittgenstein underestimates (or ignores) intuition and genuine learning. Unfortunately, aspects of mathematical intuition are not usually made rigorous.⁴ Without such rigor, it is difficult to have computational models of mathematical intuition.

⁴An exception might be Kurt Gödel, who posited that mathematical intuition worked like one of the senses.

2.1.3.3 Glasersfeld

Ernst von Glasersfeld (2006) develops a constructivist position towards the philosophy of mathematics. Under this view:

[T]he *meaning* of both natural language and mathematical symbols is not a matter of ‘reference’ in terms of independent existing entities, but rather of subjective mental operations which, in the course of social interaction in experiential situations, achieve a modicum of intersubjective compatibility (p. 63).

That this view seems to go against the realism (certainly Platonic realism, and to a lesser degree structuralist realism) is not as important as the emphasis on mental operations and experiential situations. Indeed, I believe there is enough right and wrong about both realism and constructivism to warrant a metaphysics that incorporates both (e.g., the participatory view developed by Smith (1996)).

Glasersfeld also sets up some motivation for the present work:

[I]t seems reasonable to ask what the most elementary building blocks could be that might serve as a basis for the constitution of the mysterious structures or “objects” that mathematics develops. To pursue that quest requires empirical investigation, where “empirical” has its original meaning and refers to *experience*. But clearly it will not be sensory experience that matters, but the experience of mental operations.

[W]e have to devise a model of how elementary mathematical ideas could be constructed — and such a model will be plausible only if the raw material it uses is itself not mathematical (p. 64).

I believe that sensory experience is important, but equally important is an agent’s ability to abstract away from particular sensory experiences. Glasersfeld’s insistence that the abstraction should be from the non-mathematical to the mathematical foreshadows the discussion in chapter 6.

Glasersfeld emphasizes the importance of counting as the essential initial act involved in the development of number:

[I]f we did not count, it is unlikely that we should ever have arithmetic and mathematics, because without counting there would be no numbers (p. 64).

Even in the case of numbers that are higher than we could ever actually reach by counting, the tacit knowledge that there is a procedure by means of which, theoretically, we could reach them, constitutes the first (but by no means the only) characteristic of *number* as an abstract concept (p. 66).

A counting act that has the potential to generate arbitrarily large numbers “latently” gives the agent a conceptual framework to deal with all of the natural numbers.

2.1.4 Linguistics

Linguists study mathematics as a system of communication and investigate its relationship to verbal natural-languages. Broadly speaking, linguistics investigates these math cognition issues:

- **Associability of Math and Natural Language:** Given the dissociability of the two brain systems responsible for language and mathematics as given by neuroscience, how much overlap is there between the two cognitive modules?
- **Mathematical Discourse:** What makes certain discourses (or proofs) easier to process than others? How should mathematical statements be parsed when presented alongside natural language?
- **Cognitive Semantics:** How can linguistic devices such as conceptual metaphors and figure-ground schemata be applied to mathematical idea analysis.

2.1.4.1 Lakoff and Núñez

George Lakoff and Rafael Núñez (2000) provide a detailed basis for mathematical cognition using the device of conceptual metaphor. They believe that mathematics is a subjective, human creation, that has resulted from the human embodiment.

Mathematics as we know it is limited and structured by the human brain and human mental capacities. The only mathematics we know or can know is a brain-and-mind based mathematics . . . the only conceptualization that we can have of mathematics is a human conceptualization (p. 1–2).

This would seem to immediately fly in the face of any attempt at developing computational agents with *human* mathematical understanding. However, any implementation of such computational agents will likely be programmed by human beings. Those human programmers will provide these agents with some encoded form of *human* mathematics (since this is the only form of mathematics the programmers know, or can know). The problem of different embodiments for humans and computers can also be overcome. It may be the case that either a computational embodiment will result in a sufficiently similar mathematics to that of human beings or that the computational agent will need an interface through which to describe its own mathematics to the human. In terms of just mathematical ability, a pocket calculator already does this. The “binary” embodiment produces a binary numerical answer that is translated to a “human readable” format before it is displayed on the LCD.

Lakoff and Núñez describe the initial innate abilities of infants and show how these abilities can be built on to produce more complex mathematics. They claim that the central driving force behind this development is conceptual metaphor. Conceptual metaphors come in two varieties:

The first are what we call *grounding metaphors* — metaphors that allow you to project from everyday experiences (like putting things into piles) onto abstract concepts (like addition). The second are what we call *linking metaphors*, which link arithmetic to other branches of mathematics (p. 53).

This distinction strongly supports a bottom-up approach to designing computational agents with mathematical understanding. Even though we have not used conceptual metaphors in our cognitive agent's math lessons, the presence of linking metaphors from arithmetic to other branches of mathematics suggests that an agent will be able to answer questions from those branches *in terms of* its arithmetic understanding. Conceptual metaphors form the basis for what Lakoff and Núñez call an "Embodied Arithmetic". They claim that understanding of arithmetic comes from: "(1) the most basic literal aspects of arithmetic, such as subitizing and counting, and (2) everyday activities such as collecting objects into groups or piles, taking objects apart and putting them together, taking steps, and so on" (p. 54). Corresponding to these embodied activities, four grounding metaphors are given:

1. **Arithmetic Is Object Collection:** "arises naturally in our brains as a result of regularly using innate neural arithmetic while interacting with small collections of objects. The metaphor is so deeply ingrained in our minds that we have to think twice to realize that numbers are not physical objects and so do not literally have a size" (56).
2. **Arithmetic Is Object Construction:** "makes it possible to understand a number, which is an abstraction, as being 'made up', or 'composed of', other numbers, which are 'put together' using arithmetic operations" (65).
3. **The Measuring Stick Metaphor:** "when you form a blend of physical segments and numbers, constrained by the measuring stick metaphor, then within the blend there is a one-to-one correspondence between physical segments and numbers. The fateful entailment is this: Given a fixed unit length, it follows that for every physical segment there is a number" (70).
4. **Arithmetic Is Motion Along a Path:** "There is a simple relationship between a path of motion and a physical segment. The origin of the motion corresponds to one end of a physical segment; the endpoint of the motion corresponds to the other end of the physical segment; and the path of motion corresponds to the rest of the physical segment" (72).

The target domains for these metaphors give rise to very special meanings for one and zero:

... zero, in everyday language, can symbolically denote emptiness, nothingness, lack, absence, destruction, ultimate smallness and origin ... one [can take on] the symbolic values of individuality, separateness, wholeness, unity, integrity, a standard, and a beginning (p. 75).

These meanings can be used in non-mathematical domains or can serve to lend numerical cognition to an agent functioning in a non-mathematical domain.

Lakoff and Núñez then go on to describe how the human embodiment has influenced the language of arithmetic and representation of numerals:

Our linear, positional, polynomial-based notational system is an optimal solution to the constraints placed on us by our bodies (our arms and our gaze), our cognitive limitations (visual perception and attention, memory, parsing ability), and possibilities given by conceptual metaphor (p. 86).

The organization of other mathematical languages follows that of arithmetic. For example, a ‘3’ in a counting sequence, a coefficient ‘3’ in an algebraic equation, and a ‘3’ in a matrix are all related by their “threeness”. We take it for granted that, in any new branch of mathematics, the numerals will retain the same meaning. Lakoff and Núñez show how this is also the case for arithmetic operators. The ‘+’ sign brings some notion of summation to a branch of math, regardless of whether the operands are natural numbers, integers, fractions, imaginary numbers, or transfinite ordinal numbers. Often, ‘+’ will need to be reinterpreted and overloaded when given different operands. This requires an author to introduce and explain how the use of ‘+’ differs from the intuitive, arithmetic sense that has been formed by the grounding metaphors. The presence of such shared symbols across mathematical languages is why I claim that mathematics is expressed in a “family” of formal languages. Ideally, a mathematical agent that was expected to *learn* a new branch of mathematics should have some intuition of what ‘+’ means in its simple arithmetic sense.

It should be noted that, despite the cognitive foundation for the positional number system and for the privileged status of number zero, these were concepts that did need invention and acceptance into mathematics. In some sense, the broader multicultural mathematical community is much like an individual in that it ends up adopting the most useful conceptual tools and best practices. With the further developments made possible by conceptual tools such as zero and the positional number system, these concepts became indispensable.

For my research interests, the most significant commentary in Lakoff and Núñez (2000) is the discussion of procedural arithmetic and its relation to computability and understanding:

When we learn procedures for adding, subtracting, multiplying, and dividing, we are learning algorithms for manipulating symbols—numerals, not numbers. What is taught in grade school as arithmetic is, for the most part, not ideas about numbers but automatic procedures for performing operations on numerals—procedures that give consistent and stable results. Being able to carry out such operations does not mean that you have learned meaningful content about the nature of numbers, even if you always get the right answers!

There is a lot that is important about this. Such algorithms minimize cognitive activity while allowing you to get the right answers. Moreover, these algorithms work generally—for all numbers, regardless of their size. And one of the best things about mathematical calculation—extended from simple arithmetic to higher forms of mathematics—is that the algorithm, being freed from meaning and understanding, can be implemented in a physical machine called a computer, a machine that can calculate everything perfectly *without understanding anything at all* (emphasis added, p. 86).

This ability to manipulate numbers without a sense of their “meaning”, i.e., in a purely syntactic way is reminiscent of Shapiro’s (1977) suggestion to use a syntactic network to handle numerical operations as opposed to a semantic network. However, I believe that a computational agent can achieve understanding by treating numbers semantically, i.e., by considering their meaning in a given system. Lakoff and Núñez (2000) represents a “call to arms” to develop agents that *can* understand the algorithms that produce the “consistent and stable results”. I will claim that understanding and meaning can be drawn from the same mechanical device that performs the calculating.

2.1.4.2 Wiese

Heike Wiese (2003) speculates that several faculties of language set up numerical cognition in a very natural way. Both the linguistic and mathematical cognitive systems develop in parallel. Language is a tool with which correspondences can be made. There is a link formed between an internalized word and its external referent (or an internalized model of the external referent). Numbers, when treated as words, can also be used to make such correspondences.

The flexibility of numbers is demonstrated by their multiplicity of uses in their assignments to objects. Wiese brings up the three-way distinction made between number assignments: *nominal*, *cardinal*, and *ordinal*. Nominal number assignments are applied to an object and serve to identify the object. For all practical purposes, nominal assignments function as proper nouns (e.g., “Number three scored the most points in last week’s game”). Cardinal number assignments are applied to sets of objects and serve as an enumeration (or size) for the set. An ordinal number assignment is applied to an object in order to indicate its relative position in space and time. Nominal number assignments can be handled computationally by any agent capable of natural-language processing. Cardinal number assignments are also used as a “proof” of quantity for a set:

... if I ask you to prove to me that the pens on my desk are actually three, what you are most likely to do is to assign every pen a counting word, starting with ONE, and use the last counting word, THREE, to identify the cardinality of the entire set of pens (p. 90).

Wiese also shows that an object can simultaneously receive multiple assignments:

... the number that is assigned to a house does not only identify it, but can often also tell us something about its position among other houses (p. 39).

Wiese describes the act of counting linguistically. The counting act provides an unlimited set of identifiers that are immediately well-ordered

[Children] learn the *counting sequence* of their language, the set of words like ONE, TWO, THREE, etc. that are used for counting objects, but that also appear in a more basic usage, as part of a recited series of words ... you cannot only give me new counting words endlessly, but you can also tell their positions relative to each other (p. 68–69).

The sequence of identifiers is also unique (and this is the case in all natural languages):

There are no two counting words that share the same phonological representation; each occurs only once within the counting sequence, and this is exactly because counting words function as numbers (p. 71).

Thus, Wiese shows how number words, which are integrated into natural language, are fully functional as numbers.

Linguistically, when numbers serve to quantify nouns, the noun determines the scale of quantification.

So, for instance, the fruits I have here in a bowl on my desk can count as six fruits, or as two apples, three bananas, and one orange (p. 101).

2.1.5 Education

The pedagogical implications of cognitive science research in mathematics have also received significant attention. Math education researchers, as well as classroom teachers themselves, are interested in how to apply psychological results to improve student instruction. Some of the questions of interest from the educational perspective are:

- **How should the various “isms” be reflected in the curriculum?** Associationism, Gestaltism, behaviorism, and constructivism all point to different theories of learning and different styles of instruction.
- **What should be tested for, and how should it be tested for?** How can mathematical knowledge be measured by performance in producing the right results and performing the correct procedures?
- **What is the impact of using multiple representations for a single concept?** There is evidence that operational and structural representations for the same concept are essential to mathematical understanding. The interplay of different representations is therefore important to quality instruction.

2.1.5.1 Schoenfeld

Alan H. Schoenfeld (1987) presents a chronological review of how different cognitive theories have impacted mathematics education. He begins with the associationists. Associationists believe that bonds (i.e., associations between sets of stimuli and the responses to them) that “go together” should be taught together. “Translated into pedagogical terms their theoretical approach yielded ‘drill and practice’ ”(p. 3). This approach is associated with “rote learning” and flash cards. Associationism does not take into consideration how arithmetic facts are stored in memory.

Schoenfeld presents Gestaltism as a response to associationism. “Gestaltists believed in very rich mental structures and felt that the object of instruction should be to help students develop them. The main difficulty was that the [G]estaltists had little or no theory of instruction” (p. 4). The sorts of “rich mental structures” the Gestaltists seek mirrors the search for robust representations in a computational theory.

Behaviorism is then presented as a counter to gestaltism. Behaviorists believe that learning can be achieved by programmed instruction and heavy prompting, with an emphasis on small steps and careful sequencing of individual environmental interactions. These interactions would be behavioral observables (p. 5).

Constructivism in mathematical education, as inspired by Piaget (see §2.1.1.1), reaches the conclusion that mathematical reality is constructed from individual experiences and is based on the stage of development. This leads to the classic mistakes involving object permanence and conservation of volume at early developmental stages, and also to subtle systematic mistakes in arithmetic at later stages.

The contrast between these methodologies is one of where the research focus should be. To some degree, the associationist and behaviorist camps tend to favor external observables and ignore

the internal cognitive structure of an agent, so I do not believe they are as useful to a computational model of an individual student of mathematics.

Schoenfeld correctly emphasizes the tendency of education research focusing on the *product* of problem solving as opposed to the cognitive science stress on the *process* of problem solving:

Educational research has traditionally emphasized product, the bottom line being the correct answer or the number of problems a student, or group of students, can correctly answer . . . In general, “having an ability” has been defined as scoring well on a test for that ability . . . For the most part, classic educational research that claimed to explore “problem solving performance” did not examine what people were doing—or trying to do—when they worked problems (p. 8).

Worse than this, a focus on only the products of problem solving can lead to a student with a skewed perception of his or her own understanding. Schoenfeld discusses how certain students come to believe that a problem is understood only if it can be solved in under five minutes (p. 27). I will have more to say about the distinction between ability and understanding in the next chapter.

Schoenfeld criticizes Polya’s work on problem-solving (see §2.2.0.3 below) for being descriptive, rather than prescriptive:

[T]here is a huge difference between *description*, which merely characterizes a procedure in sufficient detail for it to be recognized, and *prescription*, which characterizes a procedure in precise enough detail so that the characterization serves as a guide for implementing the strategy.

As a result, Polya’s strategy descriptions actually correspond to several different prescriptions applicable across problem domains. For example, working with small instances in order to understand a general problem actually means different things in different problem contexts (see p. 19). Computational theories of problem solving are necessarily prescriptive because they must be implemented via a precise, unambiguous programming language. Thus, such theories can yield a more precise recommendation for a curriculum.

2.1.5.2 Sfard

Anna Sfard (1991) gives a detailed analysis of the operational and structural distinction underlying mathematical concepts.

. . . seeing a mathematical entity as an object means being capable of referring to it as if it were a real thing—a static structure, existing somewhere in space and time. It also means being able to recognize the idea “at a glance” and to manipulate it as a whole, without going into details . . . [I]nterpreting a notion as a process implies regarding it as a potential rather than actual entity, which comes into existence upon request in a sequence of actions . . . [T]here is a deep ontological gap between operational and structural conceptions (p. 4).

Both operational and structural information in my implementation is represented in the same knowledge base and uses the same representational structures (i.e., nodes in a semantic network). This allows the agent to treat a procedure as an object of belief. Thus, any SNePS solution to the ontological problem will not require a shift in representational structure.

Sfard notes the importance of considering mathematical objects structurally and operationally to mathematical understanding

...the ability of seeing a function or a number as both a process and an object is indispensable for a deep understanding of mathematics, whatever the definition of “understanding” is (p. 5).

...in order to speak about mathematical *objects*, we must be able to deal with *products* of some processes without bothering about the processes themselves (p. 10).

The resulting object becomes a replacement for the procedure that generated it. This is consistent with my implementation of counting (see Appendix). The agent uses a set of syntactic rules for counting up to a given number n . As a consequence of performing this procedure, the agent will believe that all of the numbers it has said in order to achieve its goal (i.e., to complete counting to n) are themselves numbers. Thus, the process of counting to n generates the usable objects $1 \dots n - 1$. The agent does not have to count to $n - 1$ to use it as a number, because it has already counted to $n - 1$ while counting to n .

Sfard describes a process of *condensation*, which allows for the reification of mathematical procedures. This is presented using a computational metaphor:

[condensation] is like turning a recurrent part of a computer program into an autonomous procedure: from now on the learner would refer to the process in terms of input-output relations rather than by indicating any operations. As in the case of computer procedures, a name might be given to this condensed whole. This is the point at which a new concept is “officially” born (p. 18–19).

The cognitive load is dramatically reduced when a student only has to think about the results of a process. This is something our implementation desperately needs. Currently, our agent stores all of the procedural information (e.g., how to count from 13 to 14) along with the declarative results (e.g., the digit strings representing the numbers 13 and 14). When it is not involved in counting, it would be best to ignore the counting procedures.

2.1.6 Anthropology

Anthropology has attempted to examine the place of mathematics in various cultures at various periods of human history. An anthropologist would be interested in tracing the development of the quantitative concepts and numeration systems as they migrated and were adopted. The nature of mathematical tools and artifacts are also of interest, as are social factors impacting mathematical reasoning.

Crump (1990) presents a review of existing research in the anthropology of numbers. In this work it is apparent that numerical cognition has a vast cultural impact, creating diverse influences

in: cosmology, ethno-science, economy, society, politics, time, games, art, architecture, music, poetry and dance.

2.1.6.1 Sapir and Whorf

The famous Sapir-Whorf hypothesis (Sapir, 1921; Whorf, 1956) is an anthropological claim that the language spoken by a set of people will directly determine the ways in which those people think and behave. This hypothesis is relevant to our research in two ways: (i) Mathematics is a form of language that (although more formal and uniform than natural language) is performed differently in different cultures (e.g., the procedural differences between Asian and non-Asian students when solving elementary arithmetic problems reported by Campbell and Xue (2001)) , and (ii) any computational agent we develop will be a product of its lessons, and thus constrained by both system formalisms and the representational choices of the programmer.

The Sapir-Whorf hypothesis is a divisive issue among cognitive scientists. Recently, it has been claimed that members of the Amazonian Pirahã tribe lack complex numerical cognition because they lack number words beyond two (Biever, 2004; Holden, 2004). If true, this would provide some evidence for the Sapir-Whorf hypothesis. However, this finding is still controversial. Interestingly though, a computational implementation is the perfect vehicle for simulating how an agent with two number-words might reason, a sort of predictive computational anthropology.

2.1.6.2 Hutchins

Edwin Hutchins (1995) has studied the operations of a naval crew in order to determine their communal use of information and their reliance on external media to augment their cognitive abilities. He makes several important points regarding the importance of external tools for mathematical cognition:

[T]ools permit us to transform difficult tasks into ones that can be done by pattern matching, by the manipulation of simple physical systems. These tools are used precisely *because* the cognitive processes required to manipulate them are not the computational processes accomplished by their manipulation. The computational constraints of the problem have been built into the physical structure of the tools (p. 170–171).

A computational cognitive agent should also be given access to tools that extend its cognitive abilities. This, along with the “extended mind” hypothesis of Clark and Chalmers (1998), is the impetus behind implementing an external calculator for the SNePS agent implementation.

Hutchins also addresses the possibility of human-computer collaboration:

By failing to understand the source of the computational power in our interactions with simple “unintelligent” physical devices, we position ourselves well to squander opportunities with so-called intelligent computers. The synergy of psychology and artificial intelligence may lead us to attempt to create more and more intelligent artificial agents rather than more powerful task-transforming representations.

2.2 Research in Mathematics

Surely, for as long as we have been doing mathematics, we have been thinking about *how* we do it, *why* we do it, and *what* it means. Unfortunately, this how, why, and what have only recently been properly treated as *scientific* subjects of inquiry. As a result, such reflections did not find their ways into the “polished”, published results of mathematicians. There have been notable exceptions:

What distinguishes Archimedes’ work in geometry from that of Euclid is that Archimedes often presents his method of discovery of the theorem and/or his analysis of the situation before presenting a rigorous synthetic proof. The methods of discovery of several of his results are collected in a treatise called *The Method . . . The Method* contains Archimedes’ method of discovery by mechanics of many important results on areas and volumes, most of which are rigorously proved elsewhere (Katz, 1998, p 111).

Even at our current position in mathematical history, it is the rigorous result that is considered to have lasting value. Textbooks are written so that the reader can follow the logical progression from one theorem’s proof to the next, from one abstract concept to the next, and from one subject matter to the next. All of the messy (yet creative) aspects of the mathematician’s labor are swept under the rug.

After a discovery has been completed and its ideas well-digested, one quite understandably wishes to go back and clean it up, so that it appears elegant and pristine. This is a healthy desire, and doing so certainly makes the new ideas much easier and prettier to present to others. On the other hand, doing so also tends to make one forget, especially as the years pass, how many awkward notations one actually used, and how many futile pathways one tried out (Hofstadter, 1995, p 21).

In this section, I review the work of a few mathematicians who introspectively examined their methods in a cognitive way.

2.2.0.3 Polya

George Polya (1945) was interested in the various kinds of useful problem-solving techniques.

Studying the methods of solving problems, we perceive another face of mathematics. Yes, mathematics has two faces; it is the rigorous science of Euclid, but it is also something else. Mathematics presented in the Euclidean way appears as a systematic, deductive science; but mathematics in the making appears as an experimental, inductive science. Both aspects are as old as the science of mathematics itself. But the second aspect is new in one respect; mathematics “in statu nascendi”, in the process of being invented, has never before been presented in quite this manner to the student, or to the teacher himself, or to the general public (p. vii).

He develops a four-part general strategy consisting of: (1) understanding the problem, (2) devising a plan, (3) carrying out the plan, and (4) looking back (to check the result and understand it). Thus mathematical understanding is required “before-the-job” and “after-the-job” of problem solving. Polya stresses the assimilation of what is newly understood after a problem:

Try to modify to their advantage smaller or larger parts of the solution, try to improve the whole solution, to make it intuitive, to fit it into your formerly acquired knowledge as naturally as possible (p. 36)

This insistence on modifying the solution demonstrates that performing a task is not necessarily a pathway to understanding. A result must be “made compatible” with an agent’s existing knowledge before it can be assimilated.

2.2.0.4 Mac Lane

Saunders Mac Lane (1981) developed a category theoretic stance towards the different branches of mathematics, in an attempt to show how each branch arose from an “originating” human activity:

[M]athematics started from various human activities which suggest objects and operations (addition, multiplication, comparison of size) and thus lead to concepts (prime number, transformation) which are then embedded in formal axiomatic systems (Peano arithmetic, Euclidean Geometry). These systems turn out to codify deeper and nonobvious properties of the various originating human activities(Mac Lane, 1981)

The following activities are given:

Activity	Branch of Mathematics
Counting	Arithmetic, Number Theory
Measuring	Real Analysis, Calculus
Shaping	Geometry, Topology
Forming	Symmetry, Group Theory
Estimating	Probability, Measure Theory, Statistics
Moving	Mechanics, Calculus, Dynamics
Calculating	Algebra, Numerical Analysis
Proving	Logic
Puzzling	Combinatorics, Number Theory
Grouping	Set Theory, Combinatorics

Mac Lane thus provides a potential explanation for the effectiveness of mathematics in science (and experience in general). A branch of mathematics is a source of understanding in part because it is encoded experience.

2.3 Research in Artificial Intelligence

The AI literature, while extensive, has relatively few studies of mathematical cognition (relatively few compared to other forms of reasoning, e.g., natural-language processing). I believe that there are two fundamental reasons for this: (1) mathematical cognition has only recently emerged as a serious topic in the cognitive sciences, and (2) computers and calculators have demonstrated a great deal of mathematical *ability*. Thus, some might feel that all there is left to study is the limits of this ability. What work there has been can be put into one of five general categories: automated theorem proving, production-system simulations, heuristic-driven models, connectionist models, and tutoring systems.

2.3.1 Automated Theorem Proving

The most common activity undertaken by human mathematicians is theorem proving. This deductive activity can be automated by encoding the set of premises (or hypotheses) as logical formulas and mechanically applying rules of inference until a desired conclusion is reached. In this application, computers have proven fruitful assistants to professional mathematicians (cf. Bundy 1983 for a review of automated theorem proving). Indeed, automated theorem proving has become a full-fledged topic of its own. Unfortunately, for our investigation of mathematical understanding, this topic does not have very much to offer. What is being automated by an automated-theorem-prover is the logic behind the act of proof, not the understanding of what a resulting proof means. Furthermore, automated theorem prover implementations pay very little attention to cognitive plausibility. The logical inference rule of resolution (a driving force behind many automated theorem provers) is very efficient, but very few humans would actually apply the rule to prove a theorem.

This claim runs against the tradition of “natural” deduction systems. Rips (1983) claims that untrained subjects produce proofs in a manner similar to that of a natural-deduction system. However, I believe the rules of such systems are only “natural” in the context of logical problem-solving. Following the mathematical foundations program, we might try to reduce any arithmetic problem to a logical problem. However, in undertaking this reduction, I believe the “naturalness” of such rule following would disappear.

Even though the techniques of theorem proving are a large part of the professional mathematician’s repertoire, they are not among the most common mathematical tools applied in everyday, “common sense” mathematical reasoning. The early mathematical abilities (such as counting and arithmetic) are more frequently applied by the non-mathematician, and are more central to the core of human intelligence. Thus, the logical rules used in the SNePS agent implementation are used to specify commonsense acts, rather than proof-theoretic inferences.⁵

⁵Even though, at a certain level of granularity, these acts consist of specific system-level inferences, the agent is not “aware” of the acts construed in this way.

2.3.2 Production-System Simulations

A production system is a computational model of cognition. The model stresses the distinction between short-term “working” memory and long-term memory. External stimuli enter working memory and interact with stored knowledge in long term memory. Problem solving can be simulated by developing a production system that behaves correctly when presented with the proper sequence of external stimuli.

2.3.2.1 Klahr

Production systems have been a fertile platform for the cognitive simulation of early mathematical activities. Klahr (1973) designed a production-system simulation of counting and subitization. The system worked on a principle Klahr called template matching:

Subitizing is viewed as a combination of template matching and sequential transfer from LTM [Long Term Memory] to STM [Short Term Memory] of both template “pieces” and number names (p. 533).

What Klahr would call “counting”, I would refer to as “enumeration”, namely, the matching of number names with corresponding entities in an agent’s perception. Klahr calls such entities tolerance-space atoms:

Template pieces are symbols representing tolerance space atoms (TSAs), the elementary unit of “countableness”. Somewhere in the system, there must be a class of symbols that represents what “countable” things are. Although the TSAs are defined as primitives, the decision about what things in the environment will be considered unitary objects is dependent upon such things as goal of the quantification attempt, the discriminatability and saliency of cues, and the current state of the system (p. 535).

My theory will utilize the philosophical notion of a sortal, similar to Klahr’s TSAs. A detailed computational account of how sortals can be used by agents to perform knowledge-level acts of embodied enumeration on perceptual representations is given in Chapter 5.

2.3.2.2 Fletcher and Dellarosa

Fletcher (1985) and Dellarosa (1985) have implemented production-system models of arithmetic word-problem solving. Fletcher’s program, WORDPRO, implements a theory proposed by Kintsch and Greeno (1985). A high-level description of the model is given:

Given the propositional representation of a text, WORDPRO constructs a bi-level representation which it then uses to derive the solution. The two levels are referred to as the text base and the problem model. The text base is an organized set of propositions. Such a representation is assumed to result from reading or listening to any text, regardless of the domain. The problem model is a non-propositional, domain-specific representation which drives the problem solving process (Fletcher 1985: 365)

Both levels of representation raise interesting research questions. First, how is a mathematical natural-language text best represented? Secondly, how can domain-specific information help the problem solver?

Dellarosa's program, SOLUTION, is an extension and enhancement of WORDPRO. Both programs demonstrate the power of simulating cognitive performance in a mathematical domain.

I do not believe that Klahr, Fletcher, or Dellarosa have used production systems in any particular way that would be incompatible with a semantic-network implementation. Unlike WORDPRO and SOLUTION, I would like my agent to be able to remember the steps it took so that it might answer questions when it is done (it is difficult to see how this might be done with just a trace of production-rule firings in SOLUTION). Also, I would like my agent to be able to reason metacognitively, that is, about its own reasoning process, to demonstrate its understanding.

2.3.3 Heuristic Driven Models

Many problems in AI can be boiled down to searching through a sizable search space for the solution to some particular problem. For many problems, the time complexity of an exhaustive search renders a brute-force solution unmanageable. Moreover, an exhaustive approach is very rarely used by human problem solvers. Even when applying a "trial and error" technique to solve a problem, a human agent will very rarely choose the "trials" in an arbitrary fashion. The heuristic represents a "rule of thumb" by which the potential worth of a line of reasoning (or a solution) can be evaluated. Heuristics allow the problem solver to reduce the search space and thereby focus the search. Inherent in this process is the potential need for backtracking. That is, a heuristic is neither a perfect metric of evaluation nor a guaranteed path to an optimal solution.

2.3.3.1 Lenat

Doug Lenat developed AM (the Automated Mathematician system) (Lenat, 1979; Davis and Lenat, 1982) and its successor EURISKO (Lenat and Brown, 1984) to investigate the process of mathematical discovery. The AM system was populated with an initial pool of set-theoretic concepts and heuristics. Lenat was most interested in seeing where the heuristics would guide the creation and investigation of new concepts:

[AM's] source of power was a large body of heuristics, rules which guided it toward fruitful topics of investigation, toward profitable experiments to perform, toward plausible hypotheses and definitions. Other heuristics evaluated those discoveries for utility and "interestingness", and they were added to AM's vocabulary of concepts (Lenat and Brown, 1984, p. 269)

AM was also guided by occasional user interactions in which the user was allowed to specify the interestingness of certain discoveries. Concepts in AM are represented as frames containing slots for: name, generalizations, specializations, examples, isa, in-domain-of, in-range-of, views, intuitions, analogies, conjectures, definitions, algorithms, domain/range, worth, interestingness. Some slots in a concept frame may be open. One of the ways in which AM applies heuristics is by considering how open slots might be filled.

AM was able to discover sophisticated mathematical concepts such as Goldbach's conjecture and the property of unique prime factorization. One of the things that held AM back was that it could not discover domain-specific heuristics while it discovered new mathematical domains. EURISKO attempted to add heuristic-discovery as a capability.

These results have not been without controversy. Ritchie and Hanna (1984) called the methodology of AM into question. Regardless of whether AM and EURISKO were as successful as initially claimed, I believe that these systems shed some light on computational math cognition. AM showed that from a minimal starting concept set, a system can produce concepts that surprise the human designer. Furthermore, it can do this without access to a very rich natural language system, proof theory, or complicated algorithm. These systems also spotlight the very different techniques appropriate for the context of mathematical discovery (as contrasted with the context of mathematical justification).

2.3.3.2 Ohlsson and Rees

Stellan Ohlsson and Ernest Rees (1991) present a nice heuristic-driven model of early mathematical learning that is very much relevant to our research:

We have implemented our theory in the Heuristic Searcher (HS), a computer model that learns arithmetic procedures. We have simulated (a) the discovery of a correct and general counting procedure in the absence of instruction, feedback, or solved examples; (b) flexible adaptation of an already learned counting procedure in response to changed task demands; and (c) self-correction of errors in multicolumn subtraction (p. 103).

The activities involved in *learning* how to count occur even earlier than the starting point of our implementation. Ohlsson and Rees are most interested in how conceptual knowledge can influence the acquisition and construction of procedures. In their model, learning can be done in a relatively isolated state (i.e., without instruction or feedback) by an agent who understands the "constraints" in a given problem space. In fact, such state constraints form the foundation of conceptual understanding for Ohlsson and Rees:

... we use the term *understanding* to refer to a collection of general principles about the environment, formulated as constraints on problem states

To understand a particular situation within a domain is to subsume that situation under the general principles for that domain ... An alternative view is that to understand a procedure is to know the purpose of each step in the procedure. Such an understanding is sometimes called a "teleological semantics" for the procedure (VanLehn and Brown 1980: 95). A related view is that to understand a procedure is to know the justification for the procedure, that is, the reasons why the procedure works. A complete theory of understanding would specify the nature and function of each of these types of understanding. We focus in this research on understanding as knowledge of the principles of a domain, because we are interested in the hypothesis that understanding the principles of arithmetic facilitates the learning of arithmetic procedures (p. 109).

This sets up a meaningful link between learning and understanding. Learning occurs when a state constraint is violated during the agent's problems solving attempt. Unfortunately, Ohlsson and Rees put too much emphasis on this type of learning, ignoring the kind of learning that can occur with metacognitive reflection (i.e., consciously considering the procedure that was carried out after it is over and deliberating about how to perform the task the next time it is performed). This is a form of learning I address in my computational implementation (see Chapter 5).

I speculate that “knowing the purpose of each step within a procedure” yields more than an understanding of the procedure. This is also the key to knowing when two different procedures are effectively doing the same thing. The “justification” for a procedure is what we are most interested in having an agent answer to. This is the heart of procedural understanding.

Ohlsson and Rees also describe how the procedural and conceptual components of arithmetic can come in two separate “doses”:

Models of empirical learning are quite successful in explaining human learning in cases in which a procedure is acquired in isolation from its conceptual basis . . .

Arithmetic procedures are constructed in order to produce (or reproduce) particular sequences of steps, with little attention to the mathematical meaning of those steps (p. 105).

Here, Lakoff and Núñez would interject that the “meaning” of the steps comes from counting and the grounding metaphors of arithmetic, but would agree that, because of the mechanical nature of arithmetic, the focus of attention is not on mathematical meaning.

Ohlsson and Rees voice the same frustration with the lack of computational models of mathematical cognition that I share:

In spite of the large amount of research devoted to the psychology and pedagogy of elementary arithmetic, and in spite of the increasing use of computer simulation in the study of educationally relevant task domains (Ohlsson 1988), only three computational models of the acquisition of arithmetic procedures have been proposed prior to the work reported here. The two process models — the strategy transformation model by Robert Neches and the procedure induction/repair model by Kurt VanLehn (p. 167).

I will have more to say about the heuristic models of Neches and Van Lehn when I discuss count-addition strategy change in Chapter 5.

2.3.4 Connectionist Models

Zorzi, Stoianov, and Umiltà (2005) presents a nice summary of connectionist models of numerical cognition.

[C]onnectionist simulations of mental arithmetic take the *associative* approach . . . that mental arithmetic is a process of stored fact retrieval. We will distinguish two types of modeling approaches: (a) learning models, in which the knowledge about arithmetic facts is acquired through a learning algorithm and stored in a distributed form; and (b) performance models, in which the architecture is hard-wired and set up by the modeler(s) according to specific representational and processing assumptions (p. 69).

The connectionist approach uses the computational formalism of an artificial neural-network, whereas my theory is in the symbolic AI tradition. In addition to this difference in underlying formalism, there is also a dramatic difference in the target phenomena that connectionist models are attempting to represent. Among these phenomena are:

- The representation of the mental number line. The mental representation of distance along the number line is not linear. It takes on a compressed logarithmic form. Subjects therefore exhibit effects when estimating distances based on numeric size.
- Response times in numerical comparison. There are various “problem size effects” impacted by the numeric size of arguments as well as distance between arguments. There are also correlations between problem size and error rates.
- Retrieval times. In the adult model, facts stored in memory are accessed with different response times.
- Simulation of artificial lesions to model dyscalculia. The neural network formalism includes weighted connections, making it possible to systematically simulate problems which might impact quantitative reasoning.

In both my symbolic approach and the various connectionist approaches there is a clear sense of individual differences between agents. For the connectionist, this arises because of differences in training data sets. In the symbolic approach, these differences are the byproduct of differences in the order and type of arithmetic lessons.

Connectionist networks may be better equipped than a purely symbolic system to model noisy or imprecise features of mathematical cognition, such as an agent’s ability to estimate large quantities (and the capacity to estimate in general). However, they have not been applied to the higher-cognitive purposes of mathematical understanding and explanation, where the symbolic approach appears to be more natural. However, as we shall see, SNePS agents are built with an architecture that permits low-level perception routines to be written in any paradigm the designer chooses. Connectionist approaches may be integrated with symbolic approaches to form hybrid agents.

2.3.5 Tutoring Systems and Error Models

Although my implementation is built as a single “self-serving” cognitive agent that is given the correct algorithms for arithmetic, several existing computational implementations come in the form of mathematical tutoring systems and models of human error cases. An important subclass of tutoring systems interact with a human user. These systems require an explicit model of the user and behave interactively to learn about the user’s needs and abilities. Error models attempt to highlight why particular mistakes are made by a human user during the performance of a task. These can often inform a tutoring system as to why a class of mistakes occurs.

2.3.5.1 Nwana

Hyacinth Nwana (1993) presents a series of collaborative tutoring systems called mathematical intelligent learning environments. These differ from traditional “drill and practice” computational tutors in that they do more than pose a series questions for the user and present knowledge about the solution as hints. They also differ from problem solving software developed to only to simulate problem solving situations (in the form of games, adventures, or stories). Rather, intelligent learning environments strive to provide informed guidance and feedback specific to the current user’s needs.

Computational theories of mathematical understanding would benefit in both modeling the user and creating a software environment that is most likely to help the user learn with understanding. LeBlanc (1993) (in Nwana (1993)) presents an environment based on theories of mathematical reading. His environment components EDUCE and SELAH provide feedback in the form that very closely simulates the grounding metaphor of object collection.

2.3.5.2 Brown and VanLehn

John Seely Brown and Kurt VanLehn (1982) have conducted some interesting work in modeling procedural “bugs” in arithmetic procedures. They treat such bugs as variants of correct arithmetic procedures:

[Incorrect] answers can be precisely predicted by hypothetically computing the answers to given problems using a procedure that is a small perturbation in the fine structure of the correct procedure (p. 118).

An attempt is then made to categorize the systematic errors that will result from following a “buggy” procedure. Some bugs in a routine for multi-digit subtraction include (see p. 119):

1. *Smaller-From-Larger*. Instead of borrowing, the student takes the top digit, which is smaller, from the bottom one, which is larger.
2. *Borrow-From-Zero*. When borrowing from a zero, the student changes the zero to a nine, but doesn’t go on to borrow from the next digit to the left.
3. *Diff-0 - N = N*. When the top digit is zero, the student writes down the bottom digit as the column answer.

Frequently, a composition of multiple bugs will also lead to systematic errors. The authors attempt to generate a series of errors for a given skill by programatically introducing bugs into correct procedures and to predict likely errors human students will make.

Although I am more interested in what an agent can do with *correct* arithmetic routines, this work says a lot about the computational approach towards mathematical cognition. In particular, it shows the strength of the “student as computer following a program” metaphor.

2.3.6 Word-Problem Solving Systems

Systems that accept natural-language input have been researched in the context of mathematical word-problem solving. The important additional capacity required in solving word problems, as opposed to simply producing arithmetic results, is the ability to see what kind of operation the word problem is calling for. Computationally, this task is approached by transforming a parse of the input into a usable representation and then applying arithmetic operations to this representation.

As such the natural-language capacity is simply a “front-end” module for the arithmetic processing. I believe there is an important semantic distinction between systems that can associate operational natural-language with arithmetic operations (e.g., “Alice gives Bob three apples” means “Add three apples to the number Bob has”) and systems that can actually perform the acts the language is referring to (e.g., “giving Bob three apples”). The former type of system simply makes an association between symbols denoting objects; the latter associates a symbol denoting an object with a symbol denoting an act. I will have more to say about this distinction later.

2.3.6.1 Bobrow

Bobrow (1968) developed the *STUDENT* system to solve algebra story problems given in natural language. Some examples of problems *STUDENT* can solve are:

- The distance from New York to Los Angeles is 3000 miles. If the average speed of a jet plane is 600 miles per hour, find the time it takes to travel from New York to Los Angeles by jet.
- The price of a radio is 69.70 dollars. If this price is 15 percent less than the marked price, find the marked price.
- The sum of two numbers is 111. One of the numbers is consecutive to the other number. Find the two numbers.

Bobrow decides to conflate language understanding and mathematical understanding, and assesses the overall understanding of the system using the following operational definition:

I have adopted the following operational definition of “understanding.” A computer *understands* a subset of English if it accepts input sentences which are members of this subset and answers questions based on information contained in the input. The *STUDENT* system understands English in this sense (p. 146).

Understanding in this sense makes a black box out of the internal operations of the system, including how the English input is parsed and transformed. However, Bobrow elaborates on three dimensions of understanding along which a system’s extent of understanding can be measured: syntactic, semantic, and deductive. Syntactic understanding can be measured by the breadth of natural language input the system can accept. If the subset of English is quite large, then the system will be more robust. Semantic understanding is measured by the degree to which the system can represent correspondences between words and objects. Although Bobrow is not perfectly clear

on the matter, a system that can map more words to the appropriate mathematical operations will have a greater semantic understanding. Deductive understanding is the system's ability to answer questions not explicitly given in the input. This adds an additional deductive step that the system must perform after parsing the input to determine what is really being asked for.

Bobrow stresses that a system should not only be evaluated for its capacity to understand along these three dimensions, but also by how easily the system's understanding can be extended and the degree to which the system must interact with the user (p. 150).

Several important methodological features can be taken from the design of the *STUDENT* system. First, it is unreasonable to assume that a system can take in input from the entire set of permissible English sentences. *STUDENT* works with a constrained subset and minimizes the language understanding and background knowledge to what is explicitly necessary for the problem:

The *STUDENT* program considers words as symbols, and makes do with as little knowledge about the meaning of words as is compatible with the goal of finding a solution to a particular problem (p. 155).

The *STUDENT* system utilizes an expandable store of general knowledge to build a model of a situation described in a member of a limited class of discourses (p. 158).

In my agent implementation, I will also use a minimal set of linguistic and background knowledge, allowing that each of these can be easily extended without interference.

It is also important to notice that the input is not only in natural language, but is in the form of a question. There is something to be "filled in", so the system must not only understand what the statement of the problem means, but what is being sought by the question. The significance of question answering as a measure of understanding will be discussed in the next chapter.

Chapter 3

Understanding and Explanation: Towards a Theory of Mathematical Cognition

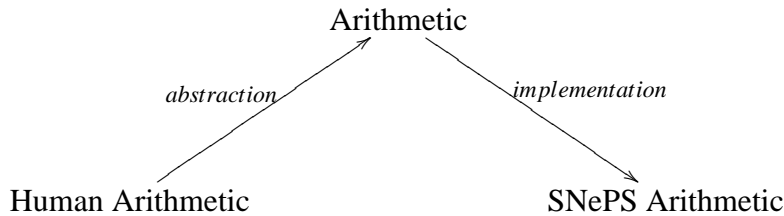
This chapter presents a high-level description of a computational theory of early mathematical cognition. This is done with an eye towards honoring the existing work presented in the previous chapter and with an eye toward formalizing my theory for a computational implementation. After outlining a series of claims my theory makes, I give an overview of the SNePS knowledge-representation, reasoning, and acting system and the GLAIR architecture that collectively serve as my agent implementation platform. I then present a characterization of mathematical understanding as a gradient of features, deriving the epistemic conditions for understanding from the classical justified-true-belief view of knowledge. Finally, I describe a method of exhaustive explanation that empirically tests for mathematical understanding.

3.1 Claims

The previous chapter showed that there are many approaches towards mathematical cognition. Each of these approaches is motivated by particular research goals and entrenched in particular disciplinary traditions. In this section, I make a series of claims about the nature of early mathematical cognition. These claims are all conducive to computational modeling, and many of them can be read off as requirements for the design of an agent. Collectively, these claims constitute my theory.

3.1.1 Multiple Realizability

Following Rapaport (1999,2005), I maintain that implementation is a three-place relation: I is an implementation of an abstraction A in a medium M . This construal of implementation yields a two-step process for developing computational theories of human cognition: (1) abstraction from the activities and representations associated with the human implementation, and (2) (re)implementation of the abstraction in a computational medium. If both of these steps are possible, then the abstraction is multiply realizable. The relevant picture of multiple realizability for arithmetic is as follows:



Step 1 is achieved by abstracting the essential features from the human case. This can be done by giving a logical analysis of the essential features. Step 2 is achieved by (re)implementing these features in a computational medium (such as SNePS/GLAIR). This can be done by writing a computer program.

The multiplicity of realizations is to be taken in several ways. First, the abstraction may be implemented in a computational system other than SNePS. Secondly, *different* SNePS implementations could give rise to *different* SNePS agents. In this respect, the abstraction is realizable in many different SNePS agents. Finally, the physical implementation media can vary to some degree. In the course of my work, I have run my agent implementation on Linux, Solaris, and Windows machines. Sometimes the machine was networked; sometimes the agent code was running locally. Under the relevant sense of implementation, each of these is also a distinct realization (i.e. each of these implementations counts towards multiple realizability).

- **Claim 1: A portion of early mathematical reasoning is multiply realizable.**

The portion in question can be measured by actual content covered, but also by the ease of extensibility (i.e., how easily the agent “scales up” from the implemented portion). It is my aim to cover counting and arithmetic. I hope to show the sufficiency of the SNePS/GLAIR platform for this purpose.

- **Claim 2: Mathematics is embodied.**

Although the implementation medium can be easily swapped for other similar media (e.g., swapping a Linux for MacOS), the relevant work presented in the previous chapter suggests embodiment does have an important role to play in the resulting agent. In particular, the (real or simulated) sensors and effectors present on the computational platform will steer the course of embodied activities that rely on perceiving and acting in the world. This becomes more dramatic if the perception and motor control aspects of embodiment are implemented in a robotic system as opposed to a simulated embodiment.

3.1.2 Representability

The grounding metaphors of arithmetic result from acting in the world. These meaningful acts must be represented by the agent and must also involve representations of the objects being acted upon and representations of the results of those actions. These representations are all located in the agent’s mind, but they depend (in a causal way) on the world and the agent’s perceptual field.

A critic of representations like Rodney Brooks may claim that the early stage of modeling grounding metaphors can be done without representations (although it would seem that representations are indispensable for full-blown arithmetic). The critic might say that mathematical behavior can be guided by perception and action alone. However, not all of the early representations need to be at the knowledge level (i.e., in a part of the agent that permits propositional representation). Even an unconscious, repeatable motor act can be treated as a representation.

- **Claim 3: Mathematics is represented publicly and privately.**

The representations used in early mathematical reasoning are constructed in a very subjective and agent-relative way but, at the same time, are alignable with (and end up flourishing within) an established, socially constructed system. I will describe how a SNePS agent can align its internal representation with natural language and public numeration systems in the next chapter.

- **Claim 4: Mathematical reasoning utilizes extended cognition.**

Tools for mathematical reasoning are plentiful. These range from marks on paper, to the abacus, to the pocket calculator, and include countless improvised, ad-hoc representations used to solve specific problems for specific agents. An agent situated in the world can affect reality in such a way that external tools can be configured and utilized to extend the cognitive capacity of the agent. This, of course, requires that the agent knows how to use the tool effectively. In order to be useful, external tools have to produce results in a predictable way, and the agent must be able to interpret these results. Proper interpretation of results includes the seamless integration of extended-cognition representations with each other and with existing belief representations.

- **Claim 5: Quantities can be usefully considered as a complex consisting of a number and a sortal.**

Although arithmetic fundamentally involves the manipulation of numbers, the grounding-metaphor acts have a fundamentally different “data type”. The agent is reasoning with a number of objects or a magnitude measuring a number of units. The representation of these objects and units needs careful treatment in a computational implementation. A theory of sortals and grounded quantities will be presented to address this claim in Chapter 5.

- **Claim 6: An agent must be able to shift between operational and structural meanings to understand mathematics.**

To represent the acts and objects is not enough. An agent must be able to effectively shift between different representations. Most notably, an agent needs to be able to alternate between operational and structural perspectives when dealing with processes (e.g., counting) and the results of those processes (e.g., a cardinality).

3.1.3 Performability

Semantics is usually taken to study the relationship between a domain of signs (qua abstract constituents of some language) and objects (in the actual world or possible worlds). The domain of signs is usually part of a socially constructed language (and it is certainly this way in the case of mathematics), which allows its users to pick out the target objects in a systematic, rule-governed way. In the case of mathematics, however, meaning is found in the acts generating signs (qua abstract constituents of the agent's language of thought). I want to draw attention to the correspondence between signs denoting acts and the behavior that performances of these acts produce (whether internal to the agent or in the external world). The relationship between act-denoting signs and behavioral products of performances is just as much an analysis of meaning as the relationship between signs and objects in the (traditional) sense.

- **Claim 7: Numbers obtain abstract meaning through the act of counting**

The activity-driven nature of mathematical cognition is present at the ground floor (or, as we shall see, on the middle floor). Properly applying the counting principles *during the act of counting* reliably results in an exemplification of a finite initial segment of the natural-number progression. Initially, it is only within the framework of the natural-number progression that natural-numbers have an “abstract” meaning. The way this is achieved in SNePS is described in the next chapter. However, several prenumerical (embodied) competences are essential in getting an agent to the counting stage.

- **Claim 8: Mathematical reasoning utilizes metacognition**

During problem solving, an agent needs a method of self-regulation and self-observation to prevent “wild-goose chases”. This involves reasoning online about which plans are being selected, reasoning about the efficiency of selected procedures, and reasoning about the context-appropriateness of procedures.

3.1.4 Empirical Testability

A theory is only as good as the evidence that supports it. Ideally, supporting evidence should come in the form of empirically testable claims. Building an agent and testing its performance against a set of requirements is an important method for testing the claims made by a computational theory.

- **Claim 9: Understanding can be treated as an analyzable gradient of features**

This claim will be discussed in detail below. An important aspect of the features in question is that they be empirically testable.

- **Claim 10: The ability to justify results through the method of exhaustive explanation demonstrates mathematical understanding.**

The method of exhaustive explanation to be described in §3.5 allows an agent to demonstrate various aspects of its understanding. Such an explanation utilizes various inference techniques and capabilities. As such, this method may be considered as constituting several experiments. Exhaustive explanation is used to test the depth of an agent's understanding of a particular concept.

- **Claim 11: The ability to apply quantitative reasoning to non-mathematical domains demonstrates mathematical understanding.**

The ubiquity of quantitative reasoning in commonsense reasoning implies that a part of mathematical understanding involves the ability to import mathematical knowledge into various contexts. The number of contexts an agent can successfully apply quantitative reasoning to is used to test the breadth of an agent’s understanding.

3.2 SNePS

SNePS is a knowledge-representation, reasoning, and acting system (Shapiro, 1979; Shapiro and Rapaport, 1987; Shapiro and Rapaport, 1995). SNePS 2.7 (Shapiro and SNePS Implementation Group, 2008), the version of SNePS I will be using throughout this work, is implemented in ANSI Common Lisp. SNePS can be viewed as a system that simultaneously supports various formalisms; it is a semantic network, a logic, and a frame system. In any of these guises, SNePS represents the first-person beliefs of a computational cognitive agent called Cassie. SNePS is distinguished from other formalisms by being a fully-intensional propositional system. SNePS is *fully intensional* in that every term in the logic of SNePS (or node in a SNePS network) denotes a unique intensional entity. No term in the logic of SNePS denotes an extensional entity. SNePS is *propositional* in that propositions are represented as first-class objects in the logic of SNePS. This means that a proposition can appear in an argument position of other propositions. SNePS makes a distinction between asserted and unasserted propositions. The former are believed to be true by Cassie, and the latter she has no truth commitment to (see (Shapiro and SNePS Implementation Group, 2008) for more specific details).

SNePS provides two interface languages for the agent developer: SNePSUL and SNePSLOG. SNePSUL has a Lisp-like syntax that exposes the relations that hold between intensional entities. These entities are represented by nodes, and the relations by directed labeled arcs connecting nodes. SNePSLOG has a syntax resembling first-order predicate logic. “Under the hood”, SNePSLOG is translated into SNePSUL. My agent implementation is done exclusively in SNePSLOG, but I will occasionally include a semantic network representation when I believe that displaying the relational structure of Cassie’s belief space is useful.

3.2.1 Overview of SNePS

SNePS is a very robust, flexible, and extensible agent implementation platform that comprises several subsystems. In the sections that follow, I briefly give an overview of the components of SNePS and describe how they will be used in implementing my theory.

3.2.1.1 GLAIR

GLAIR, the Grounded Layered Architecture with Integrated Reasoning, is an architecture for embodied SNePS agents (Hexmoor and Shapiro, 1997; Shapiro and Ismail, 2003). This architecture

has been implemented in various physically embodied agents (robots) and agents with simulated embodiments (softbots). GLAIR is composed of three layers:

- *Knowledge Layer (KL)*: Contains Cassie’s beliefs, plans, and policies. Implemented in SNePSLOG.
- *Perceptuo-Motor Layer (PML)*: Contains Cassie’s primitive actions and low-level functions for controlling her embodiment and perception. Implemented in Lisp.
- *Sensori-Actuator Layer (SAL)*: Low-level routines for sensing and affecting the outside world. This layer is sometimes omitted or simulated in softbots. Implemented in the particular device implementation language(s) of the selected embodiment.

The PML is actually divisible into three finer-grained sub-layers, but I will defer discussing this until chapter 5.

Using GLAIR makes Cassie an agent that can be situated in the real-world or any number of simulated virtual worlds. This allows her to perform concrete embodied activities that will impact her abstract KL beliefs.

GLAIR enforces a methodological knowledge-perception dualism by separating Cassie’s mental representations (the KL) and her bodily representations (the PML and SAL). In theory, the same mind can be attached to various implementations of a body and *vice versa*.

3.2.1.2 SNIP

SNIP, the **SNePS Inference Package**, enables inferential operations over the beliefs in Cassie’s network. This allows SNePS to serve as a logical rule-based system. Commands for finding and deducing nodes (representing propositions) allow the user to ask Cassie questions and are the foundation for a suite of inference techniques. These include:

- *Rule-based inference (also called Formula-based inference)*: Natural-deduction style inference rules for the introduction and elimination of logical connectives.
- *Path-based inference*: When SNePS is construed as a semantic network, a relation between nodes in a given network may be inferred by the presence of a certain path between those nodes.
- *Slot-based inference*: When SNePS propositions are construed as frames, this inference method allows an agent to infer from a proposition-valued frame another frame with a subset of slots filled.

Paths in SNePS networks are ordered sequences of arc-labels between nodes. Paths are specified using a syntax similar to regular expressions (Shapiro, 1978). Path-based techniques can be used as a model of unconscious inference for SNePS agents (Shapiro, 1991). This is an “unconscious” activity, because the newly inferred relation is added to the agent’s belief space without an explicit knowledge-level deduction. Path-based inference takes advantage of the fact that Cassie’s knowledge is stored in a semantic network, in which a node’s “meaning” is determined by its position

in the network relative to other nodes (Quillian, 1968; Rapaport, 2002). Rule-based inference is construed as “conscious” reasoning, because it generates new KL beliefs from existing KL beliefs.

3.2.1.3 SNeRE

SNeRE, the **SNePS Rational Engine**, is Cassie’s subsystem for acting. SNeRE provides a set of dedicated constructs for representing acts. Unlike propositions, acts are subject to performance. However, an act can also occur as a term within a proposition. This allows Cassie to hold beliefs about her actions and to reason about them before, during, and after their performance. There are three kinds of acts in SNeRE:

- *User Primitive Acts*: A set of user-defined acts representing the fundamental routines for the agent. These are implemented in Lisp.
- *SNeRE Primitive Acts*: Mental acts to believe or disbelieve a proposition and a set of system-defined acts used to sequence, iterate, and select over user-primitive acts and mental acts. These are implemented in SNePSLOG.
- *Complex¹ Acts*: An act that is neither user-primitive nor SNeRE-primitive. Complex acts require plans to carry them out. A plan in SNePS usually consists of a series of SNeRE-primitive acts applied to user-primitive acts. These are also implemented in SNePSLOG.

The SNeRE constructs I will use for my agent implementation, along with their semantics, are as follows²:

- `ActPlan(a1, a2)`: The way to perform the act `a1` is to perform the act `a2` (p. 66).
- `Effect(a, p)`: The effect of performing the act `a` is that the proposition `p` will hold (p. 66).
- `believe(p)`: `p` is asserted. If the negation of `p` is believed, then the negation is disbelieved and `p` is asserted (p. 65).
- `disbelieve(p)`: The proposition `p`, which must be a hypothesis (i.e., an axiom for the system), is unasserted (p. 65).
- `snsequence(a1, a2)`: Perform `a1`, and then perform `a2`.
- `withsome(?x, p(?x), a(?x), da)`: Deduce which entities (if any) satisfy the open proposition `p(?x)` (here, `?x` denotes a variable term). If any do, nondeterministically choose one, say `e`, and perform the act `a(e)`. If no entity satisfies `p(?x)`, perform the act `da` (p. 65).

¹A more fine-grained distinction is made between “composite” and “defined” complex acts in (Shapiro et al., 2007). I am using the terminology of (Shapiro and SNePS Implementation Group, 2008).

²Note that some of these acts are not used in their most general form (Shapiro and SNePS Implementation Group, 2008)

- `snif({if(p1, a1), else(a2)})`: Deduce whether `p1` holds. If it does, then perform the act `a1`, otherwise, perform the act `a2` (p. 65).

SNeRE also has constructs for policies. An adopted policy represents a condition under which an action will be taken. I will make use of the following policy:

- `wheneverdo(p, a)`: If SNIP forward chains into `p`, perform the act `a`.

What is most important for our current purposes is that Cassie reasons *while* acting, and she leaves a trail of beliefs as an act unfolds. This trail of propositions represents an episodic memory of the particular act that was performed.

3.2.1.4 SNeBR

SNeBR, the **SNePS Belief Revision** subsystem, is a component of SNePS devoted to maintaining the consistency of Cassie’s belief space (see Shapiro and SNePS Implementation Group (2008)). In some cases, SNeBR allows Cassie to resolve contradictions on her own. When this is not possible, SNeBR prompts the user when Cassie derives a contradiction during inference. The user has a choice of which beliefs to discard in an attempt to restore consistency. Because SNePS is a paraconsistent logic, Cassie can hold contradictory beliefs without inferring everything from this contradiction.

3.2.2 SNePS for Mathematical Cognition

SNePS has traditionally been used as a knowledge-representation system in support of natural language applications (e.g., Shapiro (1989), Rapaport and Ehrlich (2000)). However, the recent trends in SNePS agent development have included a greater emphasis on acting and embodiment (e.g., Shapiro et al. (2007)). As was noted above, acting and embodiment are central to my computational theory of mathematical cognition.

SNePS is well suited for a math-capable agent, because procedural and declarative information is stored side-by-side in the same way (i.e., propositionally) and in the same knowledge base.

SNePS allows for metacognition in the form of metapropositions (i.e., second order beliefs about beliefs) (see (Shapiro et al., 2007) for a survey of metacognitive techniques used by SNePS agents).

SNePS agents also produce beliefs *while* reasoning. These beliefs can serve as an episodic memory of agent’s experiences. In my implementation, a difference in experiences will result in a different belief space, and this has some impact during explanation.

Finally, SNePS provides the user the ability to ask questions. This is an absolutely essential feature, because it provides an empirical way to test my theory by probing Cassie’s knowledge and demonstrating her inferential capacity.

3.3 Understanding

The necessary and sufficient conditions for mathematical understanding have never been precisely fixed in mathematics education, the philosophy of mathematics, or mathematics itself. The term “understanding” is notoriously imprecise because it is used in many disciplines, and the precision of mathematical activities does not alleviate this problem for the term “mathematical understanding”. Despite a vagueness and ambiguity of meaning, understanding is universally described as a desirable feature. Students, teachers, and mathematicians alike strive to understand the abstract subject matter of mathematics.

Understanding is sometimes taken to be related to (or even synonymous with) knowledge, but usually something that goes beyond knowledge. However, specifying how knowledge is a prerequisite for understanding or how the two are distinct can be done in many ways. The conditions for knowledge are much better studied than those of understanding. In §3.4, I will examine how the classical epistemic view of knowledge as justified true belief impacts our characterization of mathematical understanding.

Understanding is also taken to be related to ability (or performability). To understand is to be able to *do* something. This “something”, whatever it is, happens in parallel with learning and must be done afterward as well.

Like both knowledge and ability, understanding changes as beliefs and situations change and as new abilities are learned. In §3.5.3, I will consider what it means to understand the same thing in different ways.

A general theory of understanding, whatever it might look like, may still not find applicability when the objects to be understood are those of mathematics. In this section, I will *not* undertake the project of *defining* mathematical understanding, but instead will consider the features of mathematical understanding that can be exhibited and empirically tested for in both human and computational agents.

We know *a priori* that whatever features of understanding go into a computational theory must be built from the constructs of the computational medium and will be limited by the abilities of the system. All SNePS agents serve as models of computational theories only insofar as they represent the right beliefs, perform the right acts, and are embodied in a suitable way. As such, a successful theory and implementation will elicit a pattern of belief, action, and embodiment that *behaviorally* demonstrates its mathematical understanding.

3.3.1 Modes of Understanding

Some of the ambiguity surrounding understanding involves the ontological category of the thing being understood. One might have: an *understanding that X*, which fixes *X* as a proposition; an *understanding how to X*, which fixes *X* as an act; an *understanding of X*, which fixes *X* as a “concept” (or, perhaps more usefully, as a basic-level category as per Rosch (1978)), or an *understanding what X means*, which fixes *X* as a symbol or expression in a language (in our case, a formal language). Collectively I will call these the different “modes” of understanding. The following are some concrete examples :

Mode	Example
understanding that X (declarative)	understanding that $2 + 3 = 5$
understanding how to X (procedural)	understanding how to add 2 and 3
understanding of X (conceptual/categorical)	understanding of addition understanding of numbers
understanding what X means (semantic)	understanding what '2' means understanding what '+' means understanding what ' $2 + 3 = 5$ ' means

In addition to these, there are several usages of “understanding” that correspond directly to mathematical question-answering. Of these, “understanding why X” will be most important. We will discuss this later in the context of explanation. Also, “understanding when to X” can be considered as a form of contextual or situational understanding. We will discuss this later in the context of metacognition.

One might claim that noticing these distinct modes only amounts to noticing an artifact of the English language, i.e., noticing something about contexts in which the *word* “understanding” is used. However, a careful analysis of these modes of understanding reveals that they demand very different things from a cognitive agent.

Declarative understanding presupposes that an agent can detect when a proposition holds or is the case (or, if it is a negative proposition, to detect whether a proposition does not hold or is not the case). Depending on the kind of proposition, this presupposes vastly different abilities. Even restricting ourselves to mathematical propositions, we can see that the propositions expressed by P_1 : “ $2 + 3 = 5$ ” and P_2 : “A collection of two apples and three oranges is a collection of five fruit” have different verification requirements. Furthermore, an agent might understand a particular fact expressed by P_2 based on some isolated experience, without correlating this experience with the abstraction expressed by P_1 . This example seems to suggest that there is a greater degree of declarative understanding when the perceivable becomes abstractable.

Procedural understanding presupposes that an agent can retrieve or form a plan for a particular act. Interestingly, this does not entail that the act is performable by the agent. For example, without actually being able to fly, I may understand in great detail the procedure used by birds to fly. For this reason, we might say that there is a declarative aspect of procedural understanding (e.g., “Knowing that A can be done by doing X ”) and a non-declarative aspect (e.g., “Knowing how to do A in the current context with my current embodiment”). Also associated with this non-declarative aspect of procedural understanding is something which might be called qualitative understanding, i.e., knowing “what it feels like” to perform the act. Procedural understanding can also be demonstrated: (1) when the agent understands the role of an act in a larger act, (2) when the agent knows the preconditions or effects of an act, and (3) when the agent can relate one way of performing the act to other ways of performing the act.

Conceptual or categorical understanding may presuppose a certain degree of declarative understanding if the concept or category in question is a (potentially abstract) thing, or it may presuppose a degree of procedural understanding if the concept or category is a (potentially abstract) operation. This may include a broad understanding of the concept’s or category’s extension.

Semantic understanding is best studied in the context of natural-language understanding. Understanding a language is understanding the meaning of a linguistic object: a word, a sentence, or an entire discourse. Rapaport (1995) construes semantic understanding as “understanding in terms of”:

Semantics and correspondence are co-extensive. *Whenever* two domains can be put into a correspondence (preferably, but not necessarily, a homomorphism) one of the domains (which can be considered to be the *syntactic domain*) can be understood in terms of the other (which will be the *semantic domain*) (p. 60).

It is up to the agent to understand syntactic-domain entities using the semantic-domain entities by noticing (or, more accurately, constructing) the correspondence between the domains.

3.3.2 Understanding in SNePS

Of particular importance to a SNePS implementation is the fact that each of the modes of understanding can be represented propositionally (i.e., a reduction can be made from each mode to the propositional mode), semantically (i.e., using a syntactic and semantic domain), and subjectively (i.e., in a first-person way).

The SNePS ontology³ includes things like propositions (subject to belief or disbelief), acts (subject to performance), and policies (subject to adoption or unadoption), but all of these are expressed propositionally (i.e., every well-formed formula in the logic of SNePS is a proposition). As such, our theory should capture the various modes of understanding in a propositional way. By performing such a reduction, understanding becomes a propositional attitude.⁴

The modes of understanding can be expressed propositionally as follows:

- All cases of declarative understanding are already propositional, because understanding that X is the case implies that X is a proposition.
- All cases of procedural understanding in the form “understanding how to X ” are expressed as “understanding that the way to X is Y ”, where Y is a plan for doing X . Although X is still not a proposition, that Y is a way of doing X can be expressed as a proposition.
- Cases of conceptual or categorial understanding can be expressed as an understanding of propositions in which the concept or category and the members of its extension participate.
- Cases of semantic understanding involving a term can be expressed as an understanding of propositions in which that term participates.
- Cases of semantic understanding involving propositions, i.e., where one proposition is understood in terms of another, are already propositional.

³As of SNePS 2.7

⁴Unlike the propositional attitudes of knowing and believing, I have not been able to find a logic with an understanding operator.

If anything is lost in this reduction, it is the non-declarative aspect of procedural understanding as described above (i.e., “understanding how to perform an act with the current embodiment and in the current context”). However, it is important to note that the empirical evidence of non-declarative, procedural understanding is bound to be the agent exhibiting the right sorts of behaviors. SNePS is distinguished as a system that allows for acting. Thus, the agent can exhibit such acts *and* explain what it is doing (to cover the part of procedural understanding that is reducible to propositional understanding). This will be discussed further below.

SNePS is a fully-intensional system that represents the mind of a computational cognitive agent. The agent’s mental entities are denoted by terms in the language of thought (i.e., one of the SNePS interface languages). It is therefore necessary to treat understanding in a subjective way within our theory. It may be argued that understanding is not purely subjective: that, like meaning, it may have a social component. However, this has not been the way understanding is treated in the literature:

All [authors] agree that understanding is a mental experience. Understanding is always ‘in the head’. While the meaning is public, at least for some authors, understanding remains private. The confrontation of understandings through social interactions and communications are only steps in a personal process of understanding; they can give an impulse for a change in understanding, but whether the change will be introduced or not remains an individual problem (Sierpinska, 1994, 22–23)

3.3.3 Characterizations of Understanding

Before developing an account of understanding in SNePS, it will be useful to consider various characterizations. In this section, I will briefly examine how particular philosophers, educators, and AI researchers have treated the term ‘understanding’.

3.3.3.1 Logical Characterizations

Wittgenstein gives a precise characterization of propositional understanding in the *Tractatus*:

To understand a proposition means to know what is the case, if it is true. (One can therefore understand it without knowing whether it is true or not.) One understands it if one understands its constituent parts (4.024).

The early Wittgenstein held propositions to be “picture-like” entities that “show their sense” (4.022). Although this isn’t quite the view of propositions endorsed in a symbolic system like SNePS, it is still useful to consider Wittgenstein’s definition. Symbolically, we might represent the definition for agent A and propositions p, q , and r , all of which are distinct (i.e. $p \neq q$, $q \neq r$, and $p \neq r$) as:

$$\forall A, p, r [Understands(A, p) \equiv (\exists q ((p \supset q) \wedge Knows(A, p \supset q)) \wedge (ConstituentOf(r, p) \supset Understands(A, r)))]$$

The first conjunct on the right hand side of this equivalence suggests that an agent must know some consequences of any proposition that is to be understood (because it would seem unrealistic to require the agent to know *all* consequences), and the second conjunct handles the recursive nature of understanding.

I translate Wittgenstein's claim of "knowing what is the case" if p is true to knowing some non-trivial consequence of p . It would seem to be very implausible to *know* what is the case if " $2 + 2 = 4$ " is the case, because "what is the case" if this is true includes every mathematical theorem! Even with the existential quantifier on q , it is clear that there must be a further restriction on q , namely that it is "immediately relevant" in some sense to p .

I completely agree with the idea expressed in the second conjunct. If propositions are given a semantics that ensures compositionality, it will be through understanding the terms occurring in the proposition (and the objects they refer to) that we come to understand the proposition. It also suggests (akin to (Rapaport, 1995)) that some propositions must fail to have constituent parts and must be understood in terms of themselves. Such propositions are part of domains that can be understood in terms of themselves. This gives a basis to understanding any proposition. Stuart Shapiro (personal communication) points out that the basis of understanding may not be propositional at all, but, rather, it may be some sensory structure. This seems quite plausible. However, if there is no way to admit propositions *about* this sensory structure into an agent's belief space, then the agent cannot be expected to realize the source of its understanding.

I am not taking Wittgenstein as a philosopher in isolation; this characterization is also used by Rudolph Carnap:

Suppose we want to investigate a given sentence with a view towards establishing its truth-value. The procedure necessary to this end can be divided into two steps. Clearly we must, to begin with, understand the sentence; therefore *the first step* must consist in establishing the meaning of the sentence. Here two considerations enter: on the one hand, we must attend to the meanings of the several signs that occur in the sentence (these meanings may perhaps be given by a list of meaning-rules arranged e.g. in the form of a dictionary); and on the other, we must attend to the form of the sentence, i.e., the pattern into which the signs are assembled. *The second step* of our procedure consists in comparing what the sentence says with the actual state of affairs to which the sentence refers. The meaning of the sentence determines what affairs are to be taken account of, i.e. what objects, what properties and relations of these objects, etc. By observation (understood in the widest sense) we settle how these affairs stand, i.e., what the facts are; then we compare these facts with what the sentence pronounces regarding them. If the facts are as the sentence says, then the sentence is true; otherwise, false (Carnap, 1958, p 16).

It may seem that, in the case of mathematics, only the first of Carnap's steps can be carried out, since there is no (external) state of affairs to which the signs refer. What state of affairs does the mathematical sentence " $2 + 2 = 4$ " ask one to "take account of"? I believe there are differing states of affairs that can be used to account for such a sentence, and that understanding such a sentence

is not to fix its meaning unambiguously.⁵

3.3.3.2 Educational Characterizations

Bisanz and LeFevre (1992) also reach the conclusion that understanding is defined “inconsistently, ambiguously, or narrowly” (p. 113) and that this yields an unclear relationship between cognitive processing and mathematical tasks. The authors develop a framework in which to evaluate understanding based on student performance in a “contextual space”:

The two dimensions of this space are the type of activity involved (applying, justifying, and evaluating solution procedures) and the degree of generality with which these activities are exercised. These two aspects can be used to construct a “profile” that reflects the contexts in which an individual shows various forms of understanding (p. 113).

This approach has several advantages: it treats understanding as a gradient of features, it captures the “show your work” motivation to justify solutions, and it captures the difference between solving instances correctly and solving the general problems correctly. The implication is that testing for the various forms of understanding requires a variety of tasks to be set up. This mirrors the “breadth” requirement from the discussion of claim 11 above.

Bisanz and LeFevre (1992) also suggest that to produce a demonstration of the depth of understanding (see claim 10 above), the right sort of tasks must be devised:

After researchers or teachers have identified the domain of understanding, the next step typically is to select or develop a task to elicit relevant behavioral evidence on whether the student understands the domain (p. 116).

Thus, to demonstrate that my computational agent understands the domain of arithmetic, I will develop a GCD task which requires the agent to invoke much of its arithmetic understanding.

The authors place a heavy burden on justification for demonstrating understanding. However, they warn against the attribution of misunderstanding for a student who cannot produce such a justification:

When individuals provide an adequate justification for a procedure, it often is reasonable to assume that they have explicit, accessible knowledge about the underlying principles . . . failure to provide an adequate justification does not imply that the person lacks the knowledge in question; he or she simply may have difficulty verbalizing that knowledge (p. 120).

The adequacy of justifications should vary in the computational case in the same way that it does in the human case. In the computational case, however, the designer-eye view of the agent may

⁵Much more could be said about the relation of understanding to various philosophical “isms”: semanticism, inferentialism, representationalism, and deductivism. However, such an investigation is beyond the scope of this dissertation.

detect non-verbalizable features that underpin a justification (e.g., perceptions of objects the agent has never seen before but can still count).

Sierpinska (1994) provides the longest investigation of mathematical understanding from the educational perspective. She distinguishes between acts of understanding, which are taken to be instances of understanding arising in particular situations, and processes of understanding, which are “lattices” of acts of understanding unfolding over time. Although the terminology is a bit confusing (an *act* of understanding suggests that understanding is something subject to performance), I believe the distinction is an important one. “An act of understanding is an experience that occurs at some point in time and is quickly over” (p. 2). Processes of understanding require the identification (what is being understood), discrimination (what isn’t being understood), generalization (how is this situation a case of other situations), and synthesis (searching for a common link or unifying principle) of the target concept for an act of understanding (p. 56).

Sierpinska also makes a distinction between “understanding X” and “understanding with X”:

When it is said in ordinary language that a certain person has understood something, an X, it may mean that X is indeed the object of his or her understanding, or that he or she has understood something else of which X is seen as forming the ‘essence’ or most important feature; he or she has understood something else ‘on the basis of X’ (p. 3).

This is the sense of understanding one says things like “I understand geometry” (i.e., understanding things on the basis of geometry qua subject matter). This distinction shows that acts of understanding involve two domains (see the discussion of Rapaport’s theory in the next section).

Importantly, Sierpinska links early mathematical understanding to embodied action:

Another category of representations seems to impose itself as one studies understanding of mathematics in younger children. Very often they behave as if their understanding was in their fingers rather than in their minds. In their acts of understanding, the intention of understanding seems to be directed towards an immediate action. It is based on some kind of ‘feeling’ of an activity which has to be performed here and now (p. 51).

This echoes the sentiment of embodied action given by Lakoff and Núñez (2000), and, despite Sierpinska’s skepticism at computational approaches (pp. 70–71), there is nothing inherently uncomputational about these findings. What we should take from this is that understanding requires an acting agent in all of its realizations.

Greeno (1991) links the number sense with a specific flavor of understanding that he calls “situated knowing in a conceptual domain”. This is an “environmental” position that Greeno contrasts with the information-processing approach:

In the information-processing view, concepts and relations are represented in cognitive structure, and reasoning proceeds through activation of connected representations and interpretation of activated structures. In the environmental view, knowing a set of concepts is not equivalent to having representations of the concepts but rather involves

abilities to find and use the concepts in constructive processes of reasoning. Representations of concepts and procedures can play an important role in reasoning, as maps and instructions can help in finding and using resources in a physical environment. The person's knowledge, however, is in his or her ability to find and use the resources, not in having mental versions of maps and instructions as the basis for all reasoning and action (p. 175).

As is becoming a common theme by now, I take this as indicating that the information-processing view must be expanded to incorporate action and embodiment in order to attain this environmental "situated knowing". Mere representations are not enough, nor is what a user can derive or prove from those representations; however what the agent can *do* with its representations to help it perform in the world is important. This is suggestive of viewing understanding as a potential application of knowledge. A good analogy would be to say knowledge is potential energy, which is latent and unobserved until it becomes kinetic energy.

Greeno's representations involve mental models, which he characterizes in a non-propositional way:⁶

A mental model is a special kind of mental representation, in that the properties and behavior of symbolic objects in the model simulate the properties and behavior of the objects they represent rather than stating facts about them. Reasoning with a mental model differs from reasoning with a representation in the form of sentences or formulas that express propositions. A model is a mental version of a situation, and the person interacts within that situation by placing mental objects in the situation and manipulating those symbolic objects in ways that correspond to interacting with objects or people in a physical or social environment (p. 177).

A mental model of the current situation is indeed useful to have as part of the agent implementation (and it will be part of my agent implementation in the form of the PML, see Chapter 5). This imagistic, manipulable representation of a situation may also be sufficient for understanding in the situated knowing sense. However, a mental model is not enough for a *demonstration* of understanding. An agent that needs to explain its actions must link up its mental model with some form of propositional representation.

Greeno also discusses how quantities function in mental models and how the numeric part of quantities can be "detached" from the quantity:

In the domain of quantities and numbers, a quantity is an amount of something. Any physical object or event has many quantitative properties. Each quantitative property can be described numerically by assigning a unit of measure. Numbers then, are included in descriptions of quantitative descriptions of quantitative properties of objects. Given a choice of units, a number is a property of an object or event. Numbers are also conceptual objects in their own right. They can be considered either as elements of the system of numbers or as reifications of properties. In reasoning about quantities,

⁶Non-propositional in classical approaches towards propositions, not, for example, as Wittgenstein's picture view of propositions (see §2.1.3 above).

both of these modes of cognition occur. For example, in the calculations for a problem, numerical operations are carried out that apply to the numbers regardless of their quantitative embeddings (p. 186).

This matches up nicely with claim 5 given above. The independence of numbers from quantities in cognition is actually a process that takes significant development. Once an agent is able to abstract away from the realm of physical objects, numbers are understood in a fundamentally different way. I will have much more to say about quantities in Chapter 5.

3.3.3.3 Computational Characterizations

Rapaport (1995) presents a computational theory of understanding using SNePS as an example. Understanding is presented as a recursive relation between two domains. One of these is understood in terms of the other. The essence of semantic understanding is noticing the correspondence and being able to use it. An important consequence of this “two-domain” view is that not all domains will match up in a way that is useful to an agent. Even though it may be possible to match up a multiplication domain with both an addition domain and a domain consisting of Beethoven’s symphonies, this does not entail that multiplication can be understood in terms of both addition and Beethoven’s symphonies. In the latter case, the correspondence simply isn’t useful for the agent.

There are remarkably few computational characterizations for understanding and mathematical understanding in particular. This is one of the primary motivations for conducting this research. However, there has been some commentary on the similarity between human and computational instances of understanding:

Computers can use certain symbols to denote numbers because they are manipulated by arithmetical procedures and used as loop counters, address increments, array subscripts, etc. Thus the machine can count its own operations ... [t]he way a machine does this is typically very close to the core of a young child’s understanding of number words—they are just a memorised sequence used in certain counting activities. So: ‘S refers to a number, for U’ = ‘S belongs to a class of symbols which U manipulates in a manner characteristic of counting, adding, etc.’ (Sloman, 1985)

3.4 From Knowledge to Understanding

One of the central goals of epistemology is providing the necessary and sufficient conditions for knowledge. The view that knowledge is justified true belief (i.e., that justification, truth, and belief are necessary and mutually sufficient for knowledge) is among the earliest and most popular of these views (see Steup (2006) for a review). Theories of knowledge that accept the necessity and mutual sufficiency of the justification, truth, and belief conditions (henceforth JTB theories) have been challenged (most notably by Gettier (1963)). I will take as a working assumption that these conditions are at least necessary for knowledge.

Knowledge representation (KR), considered as a branch of AI, does not dwell on knowledge conditions. Under a JTB theory, it would not always be clear “whose” knowledge is being repre-

sented in many KR formalisms. Knowledge could be attributed to: a programmer, user, or knowledge engineer; a cognitive agent; several agents; or even a society. While there are some finer distinctions (such as the distinction between expert and common knowledge), KR takes knowledge as the *object* of representation without requiring all three JTB conditions to be given.

3.4.1 Formalizations of JTB

KR systems often use formal representation languages. Thus, the utility of JTB theories to KR is dependent on how each of the conditions is formalized. In other words, a KR system that takes seriously a JTB theory of knowledge must provide a formal theory of justification, truth, and belief before claiming that it can represent knowledge.

3.4.1.1 Belief

Belief is the most internal, private, and subjective of the JTB conditions. It also seems that believing a proposition should be an early step on the path to understanding it (perhaps preceded only by understanding what the proposition means).⁷ Belief is often treated as a propositional attitude. The formal characterization of belief most compatible with my computational theory is called representationalism. Representationalists view that a belief is a stored proposition:

A sentence in the language of thought with some particular propositional content P is a “representation” of P. On this view, a subject believes that P just in case she has a representation of P that plays the right kind of role—a “belief-like” role—in her cognition. That is, the representation must not merely be instantiated somewhere in the mind or brain, but it must be deployed, or apt to be deployed, in ways we regard as characteristic of belief (Schwitzgebel, 2006).

This need for deployment binds belief to an agent’s behavior. This is the dispositional aspect of belief:

Traditional dispositional views of belief assert that for someone to believe some proposition P is for that person to possess one or more particular behavioral dispositions pertaining to P (Schwitzgebel, 2006).

An agent believes P if it behaves as if P were the case. However this raises a puzzle for beliefs in the propositions of arithmetic. What would it mean to behave like $2 + 2 = 4$ were the case? What would it mean to behave like $2 + 2 = 5$ were the case? Such a characterization seems to presuppose that the agent is attending to arithmetic while behaving. Otherwise, why would arithmetic beliefs be relevant to behavior? I will set aside these behavioral issues and focus on the representation of belief. SNePS has a very natural and robust formalism for belief representation, as well as: meta-belief representation (i.e., beliefs about beliefs), support for *de dicto* and *de re*

⁷Rapaport (personal communication) notes that often, in higher mathematics, a proof of a theorem is easier to come by when the theorem is believed.

belief report distinctions (Rapaport, Shapiro, and Wiebe, 1997), and a representation mechanism for propositions not believed by the agent (but which nevertheless need representing).

It is also important to note that belief is defeasible and bound to change. This implies that a characterization of understanding that extends upon the belief condition will yield agents that do not understand phenomena in the same way throughout their development. New beliefs can result in new ways of understanding things.

3.4.1.2 Truth

Truth is the most external, public, and objective of the JTB conditions. To say that a proposition is true is to say something about reality (however that tricky term might be resolved).

Three dominant theories of what truth consists in are:

- Correspondence: “agreement, of some specified sort, between a proposition and an actual situation”.
- Coherence: “interconnectedness of a proposition with a specified system of propositions”.
- Pragmatic Cognitive Value: “usefulness of a proposition in achieving certain intellectual goals” Moser (1999)(p. 274).

One of the most popular formalizations of truth was given by Tarski (1944). Tarski replaced the notion of truth with the broader notion of satisfiability in a model:

We define a three place relation—called *satisfaction*—which holds between a formula, a model, and an *assignment of values to variables*. Given a model $M = (D, F)$, an *assignment of values to variables* in M ... is a function g from the set of variables to D . Assignments are a technical device which tell us what the free variables stand for. By making use of assignment functions, we can inductively interpret *arbitrary* formulas in a natural way, and this will make it possible to define the concept of truth for *sentences* (Blackburn and Bos, 2005, p. 12)

The sentences belong to a formal language. This interpretation of satisfaction is very useful computationally and, thus, this has remained one of the standard theories in formal semantics.

3.4.1.3 Justification

Justification bridges the internal condition of belief and the external condition of truth. For an agent, a justification is really just a further set of beliefs, but for these beliefs to count as a justification, they must conform to some objective standard: evidence, proof, or satisfying an interrogator. There are many ways in which a proposition may be given justification. Among these are: perception, memory, deduction, and introspection. Epistemology divides theorists into two major camps: evidentialists, who claim that justification is having evidence for a proposition, and reliabilists, who claim that propositions are justified if they originate from a reliable source. Among evidentialists, there are also differing positions on whether internal or external evidence should be primary (Steup, 2006).

The justification condition has eluded formalization and incorporation into many theories.

Considerations of cognitive agents' *justifications* for their beliefs has not recently been of central concern to formal computational analyses of knowledge ... however, once the appropriate logical foundations for knowledge- and belief-representation are determined, the issue of justification ought once again to become a major area of research (Rapaport, Shapiro, and Wiebe, 1997).

Despite the fact that the justification condition has received the greatest attention in epistemology it lacked a formal representation (Artemov and Nogina, 2005)

There are exceptions. Chalupsky and Shapiro (1994) provide a formal definition for justified belief (relative to an agent model) in a subjective logic based on the logic of SNePS.

In the mathematical domain, justification is most often associated with proof. Although proof-as-explanation is indeed a staple of advanced mathematical reasoning, it is not the commonsense approach to justification we are after. An explanation should be represented as a set of acts which operate over beliefs, these may be acquired much later than the beliefs that need justification.

3.4.2 JTB in SNePS

SNePS can be viewed as providing a formal theory for each of the JTB conditions “out-of-the-box” (i.e., without further modification of SNePS). The clearest is the belief condition (cf. Rapaport, Shapiro, and Wiebe (1997)). A SNePS agent believes a proposition to be true if and only if that proposition is asserted. The truth condition has not been as central in SNePS because it is intended to model a cognitive agent in the first person (a cognitive agent that might, among other things, maintain *false* beliefs). In place of truth, there is a well developed notion of consistency in SNePS. A SNePS agent that encounters a contradiction during reasoning is able to identify the inconsistent beliefs that led to the contradiction.

Two SNePS features that can be seen as natively implementing the justification condition are support sets and inference traces.

Every asserted proposition in SNePS is either a hypothesis (i.e., a belief without need of further support) or derived (i.e., a belief depending on the beliefs from which it is inferred). The support set of a hypothesis is just a singleton set including only the hypothesis. The support set of a derived belief is a set including all of the beliefs from which the derived belief was inferred. The following example illustrates support sets in SNePS:⁸

```
wff1: Human(Aristotle) {<hyp, {wff1}, {}>}
wff2: all(x) (Human(x) => Mortal(x)) {<hyp, {wff2}, {}>}
wff3: Mortal(Aristotle) {<der, {wff1, wff2}, {}>}
```

wff1 denotes “Aristotle is human” and the support set $\{\langle \text{hyp}, \{\text{wff1}\}, \{\}\rangle\}$ indicates that it is a hypothesis. wff2 denotes “All humans are mortal” and the support set $\{\langle \text{hyp}, \{\text{wff2}\}, \{\}\rangle\}$ indicates that it is also a hypothesis. The belief expressed by wff3 is obtained from wff1 and wff2 after asking `Mortal(Aristotle)?`. The support set of wff3 is $\{\langle \text{der}, \{\text{wff1}, \text{wff2}\}, \{\}\rangle\}$.

⁸Support sets can be viewed in SNePS 2.7 by executing the `expert` command in SNePSLOG.

This indicates that $wff3$ is derived from $wff1$ and $wff2$. $wff1$ and $wff2$ are the support set for the derived belief $wff3$.

I later will consider how asking a SNePS agent a question can be considered a trigger for justification. In this example, the agent believes $wff3$ *because* it believes $wff1$ and $wff2$. A belief in $wff1$ and $wff2$ is a *justification* for believing $wff3$ (see also Chalupsky and Shapiro (1994)).

Inference tracing is another feature of SNePS that can be construed to satisfy the justification condition of knowledge.⁹ A SNePS user can enable a trace of an agent’s reasoning as it tries to infer new beliefs. For the above example, this trace includes:

```
I wonder if Mortal(Aristotle) holds
I wonder if Human(Aristotle) holds
I know wff1 Human(Aristotle)
Since wff2 and wff1, I infer Mortal(Aristotle).
```

Like support sets, this output demonstrates the dependence of $wff3$ on $wff1$ and $wff2$.

Both support sets and inference tracing are cognitively inaccessible to SNePS agents. After the performance of a particular action, there are no beliefs *about* support sets or what was “said” during inference tracing. In this way, these features represent “third-person” justifications for the agent designer. The explanation-as-justification view we will develop treats explanation as a conscious, first-person act, and will not suffer from cognitive inaccessibility.

3.4.3 First-person knowledge

I wish to make the move from knowledge as justified true belief to knowledge as explainable consistent assertion.¹⁰ The primary reason for this is that, as we have seen above, each of the latter conditions is more applicable to first-person knowledge and each has a very precise meaning in SNePS. Under this view of knowledge, a SNePS agent *knows* $wffN$ if and only if the following three conditions hold:

1. $wffN$ is asserted in the agent’s belief space: $wffN! \in BS$
2. $wffN$ is consistent with the agent’s belief space: $\{wffN\} \cup BS \not\perp$
3. $wffN$ is explainable from the agent’s belief space: $\exists p \text{ ActPlan}(\text{Explain}(wffN), p)$

Asserted propositions are either hypothetical or derived (or both) in the agent’s belief space. Consistent assertions are those that “survive” inferences without contradiction in the agent’s belief space *at a certain time*. Thus, consistency (and also knowledge) is evaluated at a “time slice” using a “snapshot” of an agent’s belief at the time of inference. According to condition 3, an explainable proposition is one for which the agent can deduce a plan for the act `Explain wffN`.

⁹Inference tracing can be enabled in SNePS 2.7 by executing the `trace inference` command in SNePSLOG.

¹⁰William Rapaport (personal communication) suggests that the term ‘assertion’ is a near synonym for ‘belief’, only with less anthropomorphic overtones.

3.4.4 Understanding as a Gradient of Features

I will claim that the leap from demonstrating knowledge (in the first-person sense I have been describing) to demonstrating understanding is made when explainable consistent assertions *are explained in a particular way*. I call this method *exhaustive explanation*, and it is described in the next section.

The method of exhaustive explanation relies on a specific treatment of understanding, namely, treating understanding as a gradient of features. Rather than singling out any particular feature that would lead an observer to ascribe understanding to a cognitive agent, the gradient of features approach attributes a greater understanding to an agent if they have more of these features and lesser understanding to an agent that exhibits fewer of these features. If these features are empirically testable, it will allow for the comparison of different computational agents and the comparison of computational and animal agents. This is, of course, subject to the weight one wishes to give to particular features of understanding.

The features of mathematical understanding tested for during an exhaustive explanation are:

- The ability to explain complex acts in terms of simple acts.
- The ability to use natural-language semantics in explaining the meaning of mathematical concepts.
- The ability to appeal to external tools during an explanation.
- The ability to perform embodied acts during an explanation.
- The ability to explain the role of a concept in the entire belief space.
- The ability to choose between different ways of performing an act based on context.

Each ability is a demonstrable feature of understanding if it can be exhibited by an agent during explanation.

3.5 Exhaustive Explanation

There must be something to explanation over and above having (and representing) reasons. Producing an explanation is an act and, therefore, a proposition is explainable only when such an act is available and performable.

In what follows, I will give an overview of the commonsense mathematical explanation technique that I have been referring to as “exhaustive explanation”.¹¹ The SNePS implementation of this technique is carried out in the next two chapters. Exhaustive explanation is formal, which will make it suitable for a computational implementation, but is also *commonsense* in that it will not require proof-theoretic techniques employed by professional mathematicians. Instead, it will be an *action-theoretic* explanation technique in the following senses:

¹¹For a different perspective on a SNePS agent’s explanation of its plans, see (Haller, 1996).

- The theory will treat explanation as an act that must be performed by a cognitive agent.
- Actions and their effects are the principal object of the explanation.
- The cognitive agent will be allowed to perform on-line non-explanatory actions during an explanation to improve its answers.

Among other things, this will mean that there will be intersubjective differences between explanations of any given proposition. Different agents will begin an explanation with different background knowledge and with a different set of experiences (most notably, a different set of mental acts performed before the explanation). This may lead to differing explanations for the same proposition (see 3.5.3).

3.5.1 A Turing Test for Mathematical Understanding

One of the early “holy grail” standards of AI was the Turing test (Turing, 1950). The Turing test unfolds as a question-and-answer dialogue between an interrogator and either a human or computer subject. If the computer succeeds in convincing the interrogator (and perhaps a wide sampling of other interrogators) that it is a human subject it has “passed” the Turing test. In his original paper, Turing was interested in devising a behavioral test to answer the question “can machines think?”, or, rather, to replace that question with one whose answer would be clearer. For our present purposes, we are after a behavioral test of understanding (specifically early mathematical understanding). I believe this can also be achieved through a certain sort of question-and-answer dialogue, namely, an explanatory question-and-answer dialogue.

Although the Turing test has traditionally been viewed as a test for natural-language competence, Turing never restricted the subject matter of the interrogator’s questions. In fact, Turing includes a mathematical question among the questions in his original imitation game:

Q: Add 34957 to 70764.

A: (Pause about 30 seconds and then give as answer) 105621 (p. 434).

There is much about this interaction that is interesting. First of all, Turing implies that mathematical questions should remain fair game. Moreover, posing mathematical questions might have been a good strategy for the interrogator. Computers have always been hailed for their speed of calculation; a rapid answer to certain mathematical problems would have been a telling feature of a computer subject. Indeed, 30 seconds is eons for a modern computer. Secondly, the interrogator is not really asking a question (i.e., what is the sum of 34957 and 70764?) but issuing a command. Finally, the incorrect sum given as the answer reeks of “humanhood” and, if anything, will help to fool the interrogator.

For a test of mathematical understanding, the success criterion of fooling the interrogator must be de-emphasized. Instead we should sacrifice the desire of human-like answers given at human-like speeds in favor of comprehensive explanatory answers, i.e., answers that, if given by a human student, would lead a teacher to claim that the student had mathematical understanding. I therefore will make the following distinction to differentiate the success criteria of a Turing test:

- *Imitative Turing test*: A computational agent strives for “natural” human-like responses, given in the time it would take a human to make such a response, and perhaps even giving incorrect answers where humans typically make mistakes.¹²
- *Exhaustive Turing test*: A computational agent strives to answer an interrogator’s questions by producing a full justification of its beliefs. It strives to demonstrate its understanding in any way possible.

The technique of exhaustive explanation can be specified in terms of this exhaustive Turing test. The interrogator will be the SNePS user (and thus already know they are dealing with a computational agent) and will interact with the agent by asking questions (using a constrained subset of written English) at a command prompt.¹³ The agent will need to represent the questions in a consistent and systematic way. The SNePS representation I will use for questions is discussed in the next chapter. Once the question is represented, the agent will need to classify the kind of question it is: “what”, “how”, and especially “why” questions will be of central importance.

The “exhaustive” aspect of exhaustive explanation will come in the way the agent answers the question. Once the agent has determined what kind of answer the question is seeking, it chooses from a variety of inference techniques to answer the question. These techniques correspond exactly to the abilities given above as features of mathematical understanding. Importantly, the agent should answer in such a way that there is a “natural” next question for the interrogator to ask.

3.5.2 The Endpoint of an Explanation

During an exhaustive explanation, an agent must navigate across its knowledge, starting from the original question and moving towards antecedently inferred propositions. An arithmetic explanation will be most effective (i.e., will satisfy the questioner) if the agent moves “towards” simpler concepts. Ideally, the agent will eventually reach propositions for which the questioner needs no explanation. Even more ideally, the agent may reach propositions for which *no* questioner needs an explanation. Such propositions have been given various names (with subtle differences) including: axioms, basic truths, hypotheses, self-evident propositions, and first principles. The objective existence of such propositions, beyond those chosen by convention, is debatable. However, there is certainly a place where a particular agent must stop its explanation. I will refer to these “endpoints” of explanation as *cognitive axioms*. Clearly, the set of propositions that are cognitive axioms will differ from subject to subject, and this is very much determined by the acts the agent has performed.

Corcoran (in press) is skeptical of the possibility of such backward tracing to reach first principles:

Even though the overwhelming majority of mathematical cognitions were held to be the results of inference from axioms or first principles, not one example of such backwards tracing has been presented and no one has ever proposed a criterion for deter-

¹²Johnson-Laird (2006) has demonstrated certain logical problems where even trained logicians tend to make systematic mistakes!

¹³SNePS can also be integrated with various speech-to-text software packages that would allow for a verbal interrogation, but I wish to avoid the implementation complexity required for such a system.

mining of a given belief whether it is an axiom or an inference. Needless to say, I am skeptical concerning the hypothesis that knowers have the capacity to trace each of their cognition-producing chains of reasoning back to cognitive intuitions (Corcoran, in press).

Corcoran takes “first-principles” to be intuitions that are not inferred from other propositions. Indeed, it is doubtful that a single objective set of first principles would be cognitively accessible to an entire community of knowers. Russell (1907/1973) sheds some light on this:

[T]he propositions from which a given proposition is deduced generally give the reason why we believe the given proposition. But in dealing with the principles of mathematics, this relation is reversed. Our propositions are too simple to be easy, and thus their consequences are generally easier than they are. Hence we tend to believe the premises because we can see that their consequences are true, instead of believing the consequences because we know the premises to be true (pp. 273–274).

In light of Russell’s other work “the principles of mathematics” can be interpreted as a formal axiomatic system meant to capture first principles. A proposition like “ $2 + 2 = 4$ ” is too simple to be easy.

This all suggests a picture in which the cognitive axioms are the starting point for inferences we make in mathematics as well as the abstracted end points of particular classes of experiences (e.g., the grounding metaphors of arithmetic: object collection, object construction, motion along a path). This is illustrated in Figure 3.1: This is an “hourglass” view of inferences and experiences.

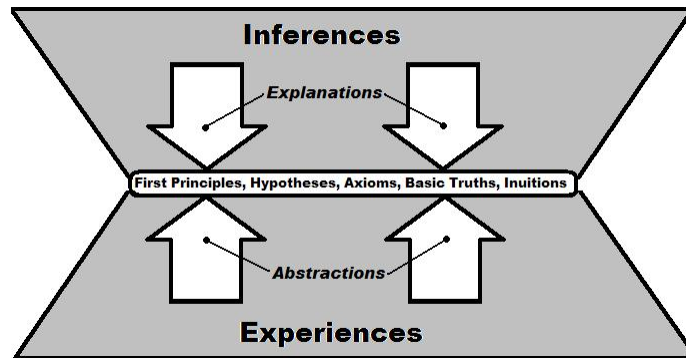


Figure 3.1: Two paths back to first principles.

Abstractions of various experiences narrow towards a set of cognitive axioms, and from these abstractions the agent expands its mathematical beliefs. Thus, an “explanation back to first principles” may involve both a reduction of complex concepts to simpler ones, eventually bottoming out at the cognitive axioms, or an account of how a set of experiences is abstracted to arrive at the cognitive axioms. Thus construed, explanation “inverts” the direction of inference and experience.

3.5.3 Multiple Justifications

As we shall see in the coming chapters, arithmetic operations can be understood in an external, embodied way that involves the agent acting, perceiving, and (often) altering the world, or in an internal, abstract way that requires the agent to rephrase an abstract operation in terms of simpler arithmetic operations and counting routines.

Even within the broad categories of abstract arithmetic and embodied arithmetic, there exist a plurality of ways to understand an operation. For example, we can consider two abstract ways in which an agent might understand the proposition expressed by “ $6 \div 3 = 2$ ”. An agent may understand division in terms of iterated subtraction. It would then be justified in believing the proposition in terms of the propositions expressed by “ $6 - 3 = 3$ ” and “ $3 - 3 = 0$ ”. However, the agent could understand division equally well in terms of inverted multiplication. In this way, “ $6 \div 3 = 2$ ” is justified by the proposition “ $3 \times 2 = 6$ ”. Both iterated subtraction and inverted multiplication can be further understood on the basis of facts stemming from iterated addition. This is illustrated in Figure 3.2.

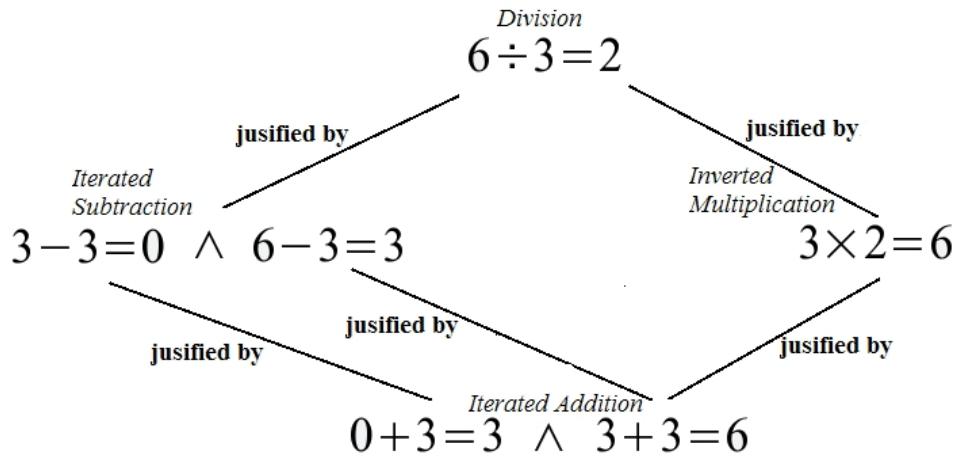


Figure 3.2: Ways of understanding division.

There is a direct correspondence between a way of understanding division and a way of doing division. That is, a quotient can effectively be found by iterated subtraction or by inverted multiplication. Thus, whatever we take understanding an arithmetic operation to mean, we must consider the ways in which the agent in question performs its arithmetic. This is consistent with the characterization of procedural understanding mentioned earlier.

Less salient for an agent performing division are facts like $0 + 3 = 3$ and $3 + 3 = 6$, which justify the propositions of iterated subtraction and inverse multiplication. The demands of attention during both the performance of arithmetic operations and justification of results are likely to be significant. However, the fact that both ways of understanding division “reconnect” to iterated addition in an agent’s “web of understanding” is significant because it provides a computational way of demonstrating that the two ways of *performing* are functionally equivalent.

How would an agent discover that one of the ways of doing division was functionally equivalent to another way of doing division¹⁴? One answer is that the agent might observe the input-output behavior of each way of doing division for some small inputs and verify that they match up. A more sophisticated agent could reason about the structural relationships between the operations. Suppose, in our example, that the agent understands division in terms of iterated subtraction and is shown the new method of division by inverted multiplication. It might then reason as follows:

- inverted multiplication is really inverted iterated addition because multiplication is really just iterated addition.
- inverting iterated addition is functionally equivalent to iterating inverted addition.
- iterated inverted addition is really just iterated subtraction.

Through this reasoning, the agent has transformed the unknown way of doing division (inverted multiplication) into the known way (iterated subtraction) by reasoning about the operations and the processes of iteration and inversion. Unfortunately, this metacognitive reasoning is very sophisticated and a far cry from the abilities available in early mathematical cognition.¹⁵

3.5.4 Behavior and Procedural Understanding

By limiting a Turing test interrogation to questions and answers expressed linguistically, we restrict our inquiry to the sorts of understanding that can be reduced to propositional understanding.¹⁶ However, as I noted in §3.3.2, there is a behavioral aspect of procedural understanding that cannot be reduced to propositional understanding. Wittgenstein (1922/2003) famously said “Whereof one cannot speak, thereof one must be silent” (*Tractatus*, 7). However, it might have been better to say that whereof one cannot speak, thereof one must act! To capture the “un-sayable” aspect of understanding that calls for action, we must augment the Turing test¹⁷ and allow the agent to act. In such an augmented test, the interrogator becomes an observer and the agent becomes an actor.¹⁸ This method provides a more thorough probe of an agent’s understanding by allowing the agent to both say *and* do things.

First, if we are going to consider behaviors as part of the test, we need some precise way to characterize behaviors. For SNePS agents, it is very natural to define a behavior as a process¹⁹ invoked by the SNePSLOG command `perform a` at the prompt (with `a` denoting an act). One

¹⁴This kind of question, raised by Rapaport (1995), was one of my original motivations for studying mathematical understanding.

¹⁵Note that such metacognitive reasoning is not the only option an agent might have. An imagistic mental model might provide another approach to seeing the two procedures as doing the same thing. However, the plausibility of such models also come under pressure when scaling up from early mathematical cognition.

¹⁶An interesting analysis of pre-linguistic thinking is given by Bermúdez (2003)

¹⁷A similar move is made by Bringsjord, Caporale, and Noel (2000) with their “Total Turing test”.

¹⁸Of course, unlike Turing’s initial separate-room teletype layout, this augmented test assumes that the interrogator can perceive the agent’s behavior.

¹⁹Here I mean process in the metaphysical sense, rather than in the technical computer science sense. I will assume processes are fundamental and will not discuss them further here.

may object that it is “cheating” to have a command prompt at all because, in this case, it is not really the agent behaving autonomously. I will not worry about this objection here, because behaviors could be triggered in a much more natural way (e.g., as a result of perceiving something or to achieve some goal) to address agent autonomy.

Now we can consider the different possible interactions between an interrogator (qua observer) and an agent (qua actor). It is easier to do this if we consider the interrogator as another SNePS agent and take a “god’s eye” view of both agents. This allows us to talk of about the interrogator’s beliefs. Figure 3.3 shows a set of possible interactions. In case (i), the interrogator identifies the

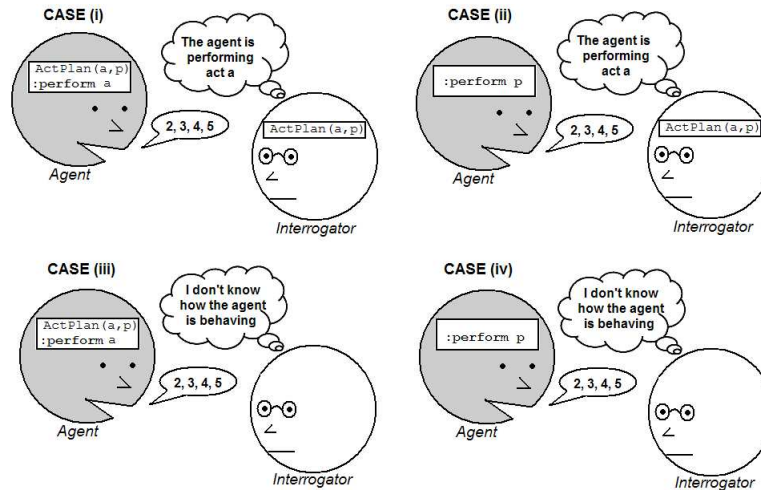


Figure 3.3: Possible interactions between an interrogator and acting agent.

behavior as act *a* because the interrogator knows that *p* is a way of doing *a*. In case (ii), the interrogator also identifies the behavior as act *a*, but the agent does not know that by doing *p* they are doing *a*. This can be viewed as a misidentification on the part of the interrogator, or, perhaps as the agent fooling the interrogator. Alternatively, the interrogator can remain skeptical and just claim that the agent is just doing *p*, but such an interrogator would have to say the same thing in case (i). In case (iii), the interrogator is only able to say that the agent is behaving in some unspecified way because the interrogator has no name for that behavior, even though the agent does. Although this case is not of much concern for the domain of early mathematical cognition, it is possible that a computational agent could classify its behaviors in such a way that certain interrogators could not identify the acts being performed. Finally, in case (iv), the interrogator is again just able to note that the agent is behaving in some way and, this time, the agent also has no named act to associate with its behavior.

These cases can be formalized as follows.

Let \mathbf{KL}_i be the knowledge layer of the interrogator agent *i*.

Let \mathbf{KL}_c be the knowledge layer of the agent Cassie *c*.

Let $perform_x(p)$ be a function from the domain of SNePS acts performable by agent *x* to the domain of behaviors \mathbf{B}_x demonstrated by agent *x* while acting, such that $perform_x(p)$ is the behavior

of agent x when the SNePSLOG command `perform p` is invoked.

Let $\llbracket Bel(x,y) \rrbracket = \llbracket x \rrbracket$ holds belief $\llbracket y \rrbracket$.

Define $\mathbf{ACTPERF}_x \subset \mathbf{B}_x$ to be a set of behaviors for agent x such that for any agent y (with x and y possibly denoting the same agent), act a , and plan p , the following two properties hold:

- $\text{ActPlan}(a, p) \in \mathbf{KL}_x \supset \text{perform}_x(p) \in \mathbf{ACTPERF}_x$
- $\text{ActPlan}(a, p) \in \mathbf{KL}_y \supset \text{Bel}(y, \text{perform}_x(p)) \in \mathbf{ACTPERF}_x$

In other words, $\mathbf{ACTPERF}_x$ is that set of agent behaviors that constitute performances of named acts for the agent performing them and are believed to be performances of named acts for agents observing the behavior who also know the name of the acts. Now we consider four cases using this formalism:

- **Case (i).** Both Cassie and the interrogator believe that p is a way of doing a :
 $\text{ActPlan}(a, p) \in \mathbf{KL}_i$ and $\text{ActPlan}(a, p) \in \mathbf{KL}_c$.
 In this case $\text{perform}_c(p) \in \mathbf{ACTPERF}_c$ and $\text{Bel}(i, \text{perform}_c(p)) \in \mathbf{ACTPERF}_c$.
- **Case (ii).** The interrogator believes that p is a way of doing a , but Cassie does not (even though she can perform p):
 $\text{ActPlan}(a, p) \in \mathbf{KL}_i$ and $\text{ActPlan}(a, p) \notin \mathbf{KL}_c$.
 In this case $\text{perform}_c(p) \in \mathbf{B}_c - \mathbf{ACTPERF}_c$ and $\text{Bel}(i, \text{perform}_c(p)) \in \mathbf{ACTPERF}_c$.
- **Case (iii).** The interrogator does not believe that p is a way of doing a , but Cassie does:
 $\text{ActPlan}(a, p) \notin \mathbf{KL}_i$ and $\text{ActPlan}(a, p) \in \mathbf{KL}_c$.
 In this case $\text{perform}_c(p) \in \mathbf{ACTPERF}_c$ and $\text{Bel}(i, \text{perform}_c(p)) \in \mathbf{B}_c - \mathbf{ACTPERF}_c$.
- **Case (iv).** Both Cassie and the interrogator do not believe that p is a way of doing a (even though Cassie can perform p):
 $\text{ActPlan}(a, p) \notin \mathbf{KL}_i$ and $\text{ActPlan}(a, p) \notin \mathbf{KL}_c$.
 In this case $\text{perform}_c(p) \in \mathbf{B}_c - \mathbf{ACTPERF}_c$ and $\text{Bel}(i, \text{perform}_c(p)) \in \mathbf{B}_c - \mathbf{ACTPERF}_c$.

These interaction cases are by no means exhaustive. If the interrogator has two names for a given act (e.g., $\text{ActPlan}(a1, p) \in \mathbf{KL}_i$ and $\text{ActPlan}(a2, p) \in \mathbf{KL}_i$) then they may attribute either $a1$ or $a2$ to the agent's behavior. There may be many cases in which the agent's acts don't correspond to observable behaviors (e.g., as is the case with mental acts).²⁰ Another issue arises when considering the objects external to the agent involved in behaviors. Here, I have considered the acts apart from the objects over which the acts are performed. However, the interrogator may fail to label the act *because* they are unfamiliar with the performance of a sort of act with the object(s) the agent is employing (e.g., if I wash my hands with broth, not many people would say I am washing my hands).

Despite these simplifications, I believe that a function like $\text{perform}_x(p)$ would be interesting to investigate. Even though $\text{perform}_x(p)$ does not operate in line with classical semantic theories (e.g., it is not like the λ of lambda calculus), it is a useful tool in measuring procedural understanding. Unfortunately, a further investigation is beyond the scope of this dissertation.

²⁰This would involve the interrogator discriminating between a mental act and no behavior at all. This "NULL" behavior is only detectable from a god's eye view

Chapter 4

Abstract Internal Arithmetic

An embodied theory of mathematical cognition should present a unified view of how we do mathematics “in our heads” and how we do mathematics “in the world”. In early mathematical cognition, embodied routines over physical objects form the basis for the arithmetic we eventually do “in our heads”. The latter I will call *abstract internal arithmetic*. In this chapter, abstract internal arithmetic is presented in isolation, taking for granted that it is abstractable from an embodied external arithmetic (this part of the story is put off until the next chapter).

I first provide the syntax and semantics for the SNePS representations I utilize in implementing abstract internal arithmetic. A taxonomy is given according to which arithmetic routines can be classified. I focus on a class of “semantic routines” to develop a set of count-based routines in a bottom-up way. These routines exhaust the four basic arithmetic functions: addition, subtraction, multiplication, and division. To empirically demonstrate a depth-of-understanding, a case study is described in which the agent attempts to demonstrate its understanding of the greatest common divisor (GCD) of two natural numbers. To achieve this, a question and answer dialogue is carried out. This dialogue is intended to induce an exhaustive explanation from Cassie. I discuss how each type of question is parsed and represented in SNePS. Depending on the type of question, different inference techniques are employed by the agent. I will focus on the techniques of *procedural decomposition* and *conceptual definition*.

4.1 Representations for Abstract Internal Arithmetic

In SNePS, the structure of representations is set up by the designer before running an agent. While active, Cassie perceives, acts, and deduces new beliefs within her framework of representations. In my implementation, Cassie does not alter her representations on the fly, so the representation choices determine, in large part, Cassie’s ultimate effectiveness.

As mentioned in the previous chapter (see §3.2), SNePS represents intensional entities. According to Shapiro and Rapaport (1987) (pp. 268–269), intensional entities must satisfy five criteria:

1. *Non-substitutability in referentially opaque contexts.*
2. *Indeterminacy with respect to some properties.*

3. *Intensional entities need not exist.*
4. *Intensional entities need not be possible.*
5. *Intensional entities can be distinguished even if they are necessarily identical.*

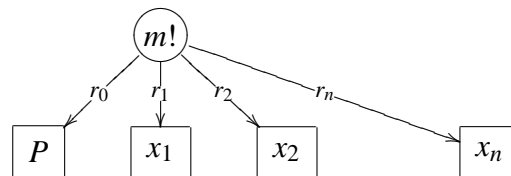
Cassie’s representations for arithmetic will fulfill all of these criteria. For a few examples, consider: Cassie may believe “ $1 + 1 = 2$ ” without believing “ $\text{gcd}(8,6) = 2$ ” (thus, we cannot just substitute for ‘2’ and a fact like “ $\text{gcd}(8,6) = 1 + 1$ ” is something Cassie must discover); Cassie can conceive of $\frac{1}{0}$, which cannot exist or be possible even in realist ontologies!; and, as Shapiro and Rapaport note, Cassie can entertain different intensional objects for “the sum of 2 and 2” and “the sum of 3 and 1”.

4.1.1 A Case-Frame Dictionary for Mathematical Cognition

In this section, I provide the syntax and semantics for the most widely used representations in my agent implementation. The schematic form of a wff is called a case-frame; each case-frame below is given in terms of SNePSLOG syntax, the `define-frame` command that generates the frame, and the semantic network structure associated with the frame. The syntax of the `define-frame` command is as follows:

`define-frame P(r_0, r_1, \dots, r_n)`

which enables the system to assert wffs of the form $P(x_1, x_2, \dots, x_n)$. Such an assertion generates the following network structure:



The nodes P, x_1, x_2, \dots, x_n are called *base nodes*. Base nodes have no arcs emanating from them. The node $m!$ is called a *molecular node*. Molecular nodes have arcs emanating from them. The ‘!’ in the molecular node $m!$ indicates that this is an asserted proposition. Optionally, if the relation r_0 is set to `nil` in the `define-frame` command, then there will be no arc r_0 pointing to node P .

I have tried to keep the number of case-frames to a minimum while representing a cognitively plausible amount of information for each arithmetic task Cassie must perform. The semi-formal semantics and a sample usage of each case-frame is also provided. In giving the semantics, I use the double-bracket symbol to distinguish between nodes and the entities the nodes represent. If x is a node, $\llbracket x \rrbracket$ denotes the entity represented by x .

4.1.1.1 Counting Case-Frames

Number

Syntax:

Definition	SNePSLOG	Network
<i>define-frame</i> Number(<i>class member</i>)	Number (<i>x</i>)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is a number. Numbers in my theory are positions in a structure exemplifying a finite initial segment of the natural number progression. Cassie constructs this structure during counting (see below) and each term (*qua* position) reached in the process is asserted as being a number using this case-frame.

Sample Use: Number (*n3*) asserts that *n3* is a number.

As a convention, I will write n_x to denote the number corresponding to the Arabic numeral x . However, in the implementation, this convention is not guaranteed to be followed. In one mode of operation, Cassie will follow this convention, so, for example n_3 will denote the number three. In another mode of operation, Cassie will obtain randomly named symbols via Lisp's `gensym` function to denote numbers. These represent the numerons of Cassie's private language (or "mentalese"). These symbols will be unknown to the designer between runs of the agent. All of Cassie's mathematics can be performed on the basis of these private numerons. For further discussion on why this sort of representation is important, see §7.3.

Successor

Syntax:

Definition	SNePSLOG	Network
<i>define-frame</i> Successor(<i>nil succ pred</i>)	Successor (<i>y</i> , <i>x</i>)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket y \rrbracket$ is the (immediate) successor of $\llbracket x \rrbracket$ in the natural-number progression.

Sample Use: Successor (*n3* , *n2*) asserts that n_3 is the successor of n_2 .

NumeralOf

Syntax:

Definition	SNePSLOG	Network
<i>define-frame NumeralOf(nil numeral number)</i>	<code>Numeral(x, y)</code>	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is the Arabic numeral associated with the number $\llbracket y \rrbracket$. This case-frame allows Cassie to associate her (internal) numbers with the numerals of a public-communication number-system. In my implementation, this is the Arabic decimal system. A similar construct would be necessary for interfacing with (e.g.) the Roman numerals (I, II, III, IV, etc.) or the English number words ('one', 'two', 'three', 'four', etc.).

Sample Use: `NumeralOf(3, n3)` asserts that 3 is the Arabic numeral for the number `n3`.

NumberName

Syntax:

Definition	SNePSLOG	Network
<i>define-frame NumberName(nil name number)</i>	<code>NumberName(x, y)</code>	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is the English number-name associated with the number $\llbracket y \rrbracket$. This case-frame allows Cassie to associate her (internal) numbers with the linguistic names of a public-communication number-system.

Sample Use: `NumberName(three, n3)` asserts that `three` is the English number-name for the number `n3`.

Zero

Syntax:

Definition	SNePSLOG	Network
<i>define-frame Zero(nil zero)</i>	<code>Zero(x)</code>	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is the number zero.

Sample Use: `Zero(n0)` asserts that `n0` is the number zero.

As was briefly mentioned in §2.1.4.1, the number zero has a special meaning for an agent. For

Cassie, zero comes up in describing the base case of recursively implemented arithmetic acts. In order to pick out which of Cassie's nodes refers to zero, it is handy to have a distinguished case-frame like `Zero`. However, the agent designer cannot assume that `n0` will denote the number zero just because of our convention. What we need is for Cassie to assert `Zero` for whatever numeron corresponds to the numerlog 0. This is achieved by:¹

```
perform withsome(?n,
                NumeralOf(0, ?n),
                believe(Zero(?n)),
                Say('`The numeral 0 is not associated with any number`'))
```

This guarantees that Cassie picks out the right numeron for zero even between runs where `gensym` establishes different symbols for zero.

4.1.1.2 Arithmetic Acts

The following is a case-frame schema for every two-argument arithmetic act. The uniformity of representation is maintained across all of the specific ways of performing a general routine (e.g., count-addition and addition have the same argument structure).

Arithmetic Act Schema

Syntax:

Definition	SNePSLOG	Network
<i>define-frame</i> A (<i>action object1 object2</i>)	A (<i>x, y</i>)	

Semantics: $[[m]]$ is a functional term² representing the act consisting of the action $[[A]]$ with arguments $[[x]]$ and $[[y]]$.

Sample Use: `perform Add(n2, n3)` is the SNePSLOG invocation to perform the act of adding `n2` and `n3`.

Additionally, the implementation makes use of three primitive acts `Say`, `SayLine`, and `SayLine+`, which produce properly formatted output of utterances Cassie makes while acting. These are included only as a convenience for the SNePS user. If these were stripped out of the implementation (thus making it an implementation of *purely* mental arithmetic), the user could still determine Cassie's beliefs using SNIP deductions.

If the task at hand were to design an agent that is simply able to do arithmetic, it would not be necessary to use many more case frames than the ones already defined. However, the needs of an agent who must explain itself in demonstration of its understanding needs some more machinery. Specifically, we need a way for an agent to link together results of various mathematical acts. I

¹The `withsome` SNeRE primitive was defined in §3.2.

²Shapiro and Rapaport (1987) have also called these structured individuals.

will describe my representational solution, result, and evaluation frames, in the next section, but first I wish to motivate the need for these frames by discussing the nature of mathematical acts.

I believe mathematical acts are an interesting ontological category of acts, but this claim needs some elaboration and justification. It is not clear at all what makes an act mathematical, and it is also unclear what makes a non-mathematical act non-mathematical. Dijkstra (1974) points out some features of mathematical “activity”:

With respect to mathematics I believe, however, that most of us can heartily agree upon the following characteristics of most mathematical work: (1) Compared with other fields of intellectual activity, mathematical assertions tend to be unusually precise. (2) Mathematical assertions tend to be general in the sense that they are applicable to a large (often infinite) class of instances. (3) Mathematics embodies a discipline of reasoning allowing such assertions to be made with an unusually high confidence level . . . when an intellectual activity displays these three characteristics to a strong degree, I feel justified in calling it “an activity of mathematical nature”, independent of the question whether its subject matter is familiar to most mathematicians (Dijkstra, 1974).

Dijkstra’s three criteria of precision, generality, and confidence collectively point towards a fourth criterion for an activity to be considered mathematical. This might be called the purposiveness criterion: when performing a mathematical act, an agent expects the performance of the act to produce a particular result. Moreover, this result is not a trivial past-tensing of the act, i.e., the effect of act A is not simply “having done A ”, but instead, it is the production of something that can be considered a result.

Mathematical acts are not detached from acting in general. For example, consider a child performing an act of subtraction for the problem $9 - 1$. The child may be doing this for the purpose of obtaining the answer 8, or for the purpose of updating their belief about how many planets there are in our solar system (after the recent decision that Pluto is not a planet³). The act of updating the number of planets is done *by* subtraction, an act which produces a result (i.e., a difference) for the larger act of updating a belief about our solar system.

$\text{Effect}(a, p)$ is Cassie’s belief that, after performing act a , proposition p will hold. One of the necessary and sufficient conditions for a being an addition act is that a produces sums. In general, one of the necessary and sufficient conditions of any mathematical act is that it produces a particular result.

4.1.1.3 Evaluation and Result Case-Frames

Result

Syntax:

³As of this writing, this result is still somewhat controversial!

Definition	SNePSLOG	Network
<i>define-frame</i> Result(<i>nil resultname arg1 arg2</i>)	Result(x, y, z)	

Semantics: $\llbracket m \rrbracket$ is a functional term representing the result named $\llbracket x \rrbracket$ with arguments $\llbracket y \rrbracket$ and $\llbracket z \rrbracket$. Since a Result is not a proposition, it must occur as a constituent part of a larger wff. Results are treated as *procepts* as given in (Gray and Tall, 1994). This is discussed further in the next section.

Sample Use: Result(Sum, n2, n3) is a term denoting the procept “the sum of 2 and 3”.
NOTE: Result case-frames are not used by themselves. Because Results are not propositions, they can only occur in other propositions in Cassie’s belief space. This type of usage is illustrated by m2 in Figure 4.1.

Evaluation

Syntax:

Definition	SNePSLOG	Network
<i>define-frame</i> Evaluation(<i>nil result value</i>)	Evaluation(x, y)	

Semantics: $\llbracket m \rrbracket$ is the proposition that the result $\llbracket x \rrbracket$ evaluates to $\llbracket y \rrbracket$. Evaluation frames effectively decouple the representation of the (proceptual) result from its evaluation. The benefits of this representation will be described below.

Sample Use: Evaluation(Result(Sum, n2, n3), n5) asserts that the sum of n2 and n3 is (i.e., evaluates to) n5.

Evaluated

Syntax:

Definition	SNePSLOG	Network
<i>define-frame</i> Evaluated(<i>property object</i>)	Evaluated(x)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ has a value. This is used as a “helper” frame to the Evaluation frame by indicating that a (proceptual) result has a value that Cassie knows. This can be expressed logically by:

$$\forall r, a_1, a_2 (\text{Evaluated}(\text{Result}(r, a_1, a_2)) \supset \exists v \text{Evaluation}(\text{Result}(r, a_1, a_2), v))$$

4.1.1.4 Results as Procepts

Gray and Tall (1994) discussed the notion of a “procept” in arithmetic. A procept is a complex symbol that can be ambiguously interpreted as a process or a concept. The expression “2 + 3” can be thought of as the process of adding 2 and 3 or as another “name” for 5 (a concept).⁴ While most theories of reference frown on ambiguity, the ability to fluidly oscillate between “summing” and “the sum” is actually quite useful for a cognitive agent (also see the operational-structural distinction given by Sfard (1991)). The proceptual nature of “2 + 3” is not just a mark of the economy of (mathematical) language, but allows an agent to be flexible in interpreting more complex expressions (i.e., expressions in which multiple procepts may occur). As procepts, result frames play very different roles before and after an arithmetic act. Before performing an arithmetic act, a result frame represents an object of thought that must be evaluated (presumably by the act). After performing an arithmetic act, a result frame functions as a descriptive name for the value obtained by the act. It is also a residue of the act that generated the value.

A procept may be deployed in either the public-communication languages utilized by an agent or in the agent’s language of thought. Due to the uniqueness principle in SNePS, a result frame cannot *denote* both a process and a concept, but is *associable* with both a process and a concept as follows:

- $\text{Effect}(\text{Add}(n_2, n_3), \text{Evaluated}(\mathbf{Result}(\mathbf{Sum}, n_2, n_3)))$. Procept $\mathbf{Result}(\mathbf{Sum}, n_2, n_3)$ associated with the process (act) of addition.
- $\text{Evaluation}(\mathbf{Result}(\mathbf{Sum}, n_2, n_3), n_5)$. Procept $\mathbf{Result}(\mathbf{Sum}, n_2, n_3)$ associated with the number (concept) n_5 .

As such, Cassie only deals with “factored” procepts; a procept is either explicitly related to an act or to a number. After performing the act of adding 2 and 3, Cassie will have the subnetwork in Figure 4.1 in her belief space. This synthesizes the Evaluation, Evaluated, and Result case-frames and will be usable in the context of question answering. Another way to interpret this representation is to say that it intensionally separates an equation like $x + y = z$ into its right and left hand sides.

Decoupling results from evaluations solves a “cart-before-the-horse” representation problem for Cassie. Before performing an arithmetic act, Cassie does not know the value of the result, even though she expects the result to have some value. In an earlier implementation, the result and its evaluation were stored in a single case-frame $\text{Result}(r, x, y, z)$ which meant that the operation r applied to x and y yielded z . However, this representation makes it impossible to

⁴Also, there may be a “neutral” reading of “2 + 3” as “two plus three” that does not commit an agent to either a process or a concept.

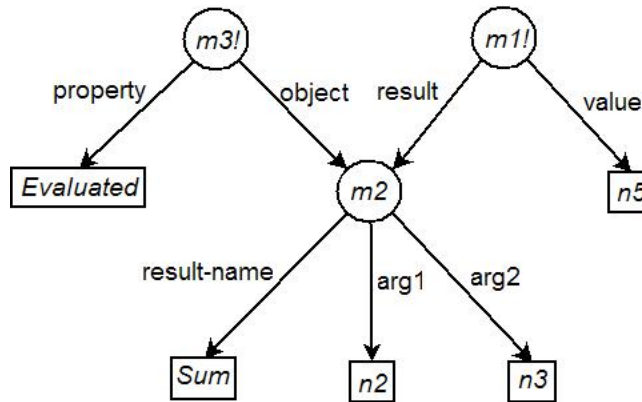


Figure 4.1: Cassie's belief that $2 + 3 = 5$

represent the expected effect of an act without knowing the value of z in advance. Thus, the game would be given away if we represented the effect as:

$$\text{all}(x, y, z) (\{ \text{Number}(x), \text{Number}(y) \} \ \&=> \ \text{Effect}(\text{Add}(x, y), \text{Result}(\text{Sum}, x, y, z))).$$

Notice that this representation fails because it asserts that *everything* is the result of adding two numbers. We simply have nothing to say *a priori* about z . The existential quantifier might be used in the consequent, but this is not supported in SNePS 2.7. Using a Skolem function (i.e., using $f(x,y)$ in place of z , above) is not as elegant as a decoupled proceptual solution.

Another benefit of the decoupled proceptual representation is that it provides a nice way of scaling the theory from natural numbers to rational numbers. Also, using result and evaluation frames, we can represent undefined results (e.g., $8 \div 0$). The agent should initially *expect* such divisions to have results, and the value needs an intensional representation even if there is no extension associated with it. These issues will be discussed in the conclusion.

4.2 A Taxonomy of Arithmetic Routines

I distinguish three categories of early mathematical routines.

- **Semantic:** Characterized by a dependence on previously learned routines, natural-language semantics (when the meaning of a concept is fixed in part by a linguistic description), and action-theoretic semantics (when the meaning of an operation is fixed in part by an activity performed on external entities, for example, the link between the operation of division and the act of evenly sharing candies between friends).
- **Syntactic:** Characterized by formal operations performed on positional number systems (e.g., the decimal number system) using column-wise operations (e.g., carrying and borrowing).

- Extended: Characterized by reliance on a trusted outside source (e.g., a calculator or teacher) and usually involving non-mathematical acts such as pushing buttons or asking questions.

These categories do not form sharp boundaries around mathematical routines. Indeed, most routines will require elements from all three categories. For example, syntactic multiplication requires the previously learned routine of syntactic or semantic addition (this is a feature of semantic routines). Syntactic multiplication may also require an agent to make marks on paper to keep track of intermediate results (this is a feature of extended routines). However, we would classify syntactic multiplication as being of the syntactic type because its main feature is performing column-wise operations on numbers positioned in a particular way. These operations are done without regard to the semantic significance of each operation and without reliance on the extended medium to directly provide the solution.

An agent must synthesize results produced by routines in each of these categories. Whether the result $129 + 23 = 152$ is discovered by paper and pencil, by calculator, or by counting, the result must be represented in a uniform way. Despite this consistent representation, not all routines are created equally when it comes to understanding. I will claim that semantic routines are the most important for an agent's understanding and for an agent's demonstration of understanding.

4.3 Semantic Arithmetic

In this section, I will demonstrate how a series of acts can implement the basic four functions of arithmetic (i.e., addition, subtraction, multiplication, and division) in such a way that each can be understood in terms of simpler acts, eventually bottoming out at a counting routine. The counting routine can be thought of as a *base* case of action, and each of the basic arithmetic functions is a *recursive* combination of counting routines. Collectively, I will refer to this series of acts as “semantic arithmetic” because it is the essence of understanding-in-terms-of as described in the previous chapter.

Semantic arithmetic is Cassie's way of understanding all of arithmetic with a counting routine. However, it is not necessarily a reduction of arithmetic understanding to understanding the procedure of counting. An agent might accidentally perform the correct sequence of count acts necessary for an arithmetic operation *without* even knowing it has performed the arithmetic operation. Understanding with counting is not just the correct sequence of count acts; the agent must also understand how the different procedures are related to one another (i.e., how to procedurally decompose one act into others).

4.3.1 Counting

Counting is the procedural generator of the natural numbers. Quine (1969) had the intuition that all there is to numbers is what is done to them:

Arithmetic is, in this sense, all there is to number: there is no saying absolutely what the numbers are; there is only arithmetic (p. 45).

Counting requires a cognitive agent that can generate a succession of unique mental symbols and can associate each unique symbol with a symbol in a public-communication language. Gelman and Gallistel (1978) call the former symbols “numerons” and the latter “numerlogs”. Since I am adopting a structuralist ontology, absolutely any node in Cassie’s KL could serve as a numeron. Human children usually learn counting before they learn what counting is “useful” for (or, more precisely, what they learn how counting is publicly meaningful), treating the act of counting as a sort of nursery rhyme. In nursery rhymes, there is a correct first word to say and there is always a correct next word to say (i.e., a unique successor). Counting only becomes meaningful when the mental numerons are aligned with the publicly accepted numerlogs.

Cassie counts using two primitive actions `ConceiveInitialElement` and `ConceiveSuccessor`. `ConceiveInitialElement` creates a base node that will denote the number zero for Cassie.⁵ This is a primitive action, and it is performed once. Every time `ConceiveSuccessor` is performed, Cassie creates a new base node and relates it to the previously created base node using the `Successor` relation. Repeating this performance results in a structure instantiating a finite initial segment of the natural-number progression. The `ConceiveOrdering` action performs the `ConceiveInitialElement` act followed by a user specified number of `ConceiveSuccessor` acts. The symbols in this ordering are Cassie’s private numerons.

`Number` is asserted of each symbol as it is created.

When Cassie has conceived of an ordering, she has not properly counted *in* a particular enumeration system. To do this, she must associate each numeron with a numerlog of a public-communication language. This process is illustrated in Figure 4.2. As the Figure shows, Cassie can make associations with multiple public-communication languages, e.g., numerals and number names. Numerals are generated in the PML, and the association with numbers is made via the `tell-ask` interface. `tell` is a command that is available in Lisp (and, hence, the PML) that permits an assertion to be made in the KL. `ask` is a command that permits a deduction to be made from Lisp. This sort of generation of numerals treats Lisp as a “syntactic network” that interfaces with Cassie’s semantic network (Shapiro, 1977).

In the current implementation, number names must be entered manually. It is possible, however, to have a large set of KL rules for handling the generation of number names, including the standard English morphology for these names (e.g., 173 is ‘one’ ‘+hundred’ ‘seven’ ‘+ty’. ‘three’).⁶ Cassie does not produce any number-name outputs in my implementation, so number-name generation is beyond the scope of the dissertation.

Cassie begins with a very “bare-bones” and abstract representation of numbers. Numbers never stop being nodes, but those nodes participate in a richer and richer network as Cassie performs different activities. It is in this way that numbers become more and more meaningful to her.

⁵Human children do not start with a “zero *qua* number” concept. A fuller account of the ontogeny of the zero concept would need to include non-mathematical notions of “absence”, “lack”, and “emptiness”.

⁶Wiese (2003) gives a formal account of such a generator (see pp. 304–313).

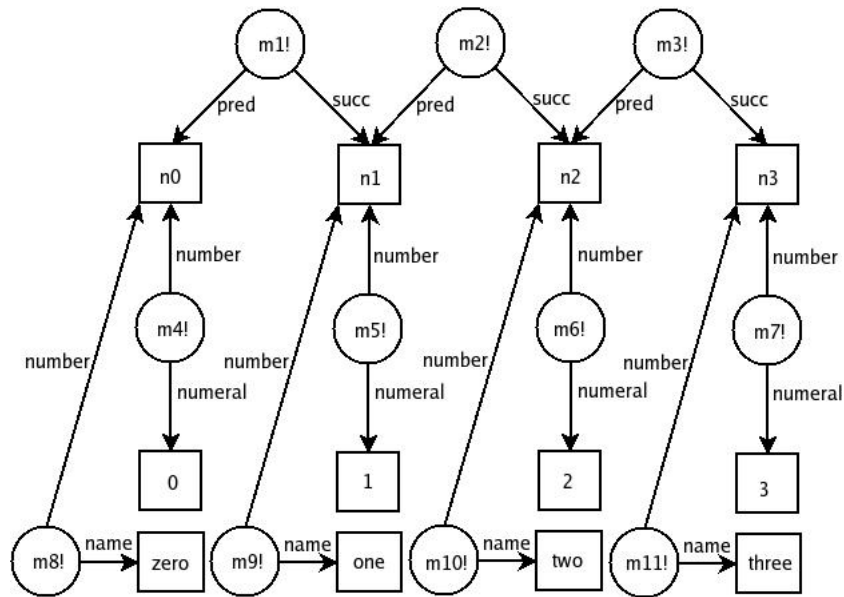


Figure 4.2: Numeron-numerlog association. m_1, m_2 , and m_3 establish the finite initial segment of the natural-number progression up to three. m_4, m_5, m_6 , and m_7 associate this progression with the numerals 1, 2, 3, and 4. m_8, m_9, m_{10} , and m_{11} associate this progression with the number names.

4.3.1.1 The Greater-Than Relation

For Cassie, what is the natural-number progression a progression of? Initially, it is just a progression of symbols. Importantly, Cassie can count *things* with these symbols, i.e., make number assignments (see Chapter 2, §2.1.4.2). That is, the symbols can be associated in a one-to-one way with objects being counted in a stable order (this will be dealt with in great detail in the next chapter). Gelman and Gallistel (1978) indicate that once the one-to-one and stable-order “how to count” principles are present, the cardinal principle may be accessible, i.e., treating the natural-numbers as a succession of potential cardinalities increasing in size (see Chapter 2, §2.1.1.2).⁷

The development of the cardinal principle is dealt with extensively by Gelman and Gallistel (1978). Each number is a potential endpoint of a counting routine. An agent that is well practiced in making cardinal number assignments will notice that in enumerating a collection of size n , all of the cardinalities $1 \dots n - 1$ are passed. After this is noticed for particular concrete collections, an agent can abstract that passing smaller cardinalities on the way to larger ones will occur when enumerating any sort of thing. The agent abstracts from the collection sizes and considers only the numbers themselves (i.e., the number component of the quantity). The number line becomes a ruler of potential sizes for potential collections. The greater-than $>$ relation is used to compare these numbers treated as sizes.

The initial Successor relationship Cassie applies while counting can be used to produce the

⁷Note however, that more recent research (Clements and Sarama, 2007) indicates that the one-to-one and stable order principles may be necessary but insufficient for the emergence of the cardinal principle.

GreaterThan relation by applying path-based inference. The relevant case-frames are:

```
define-frame Successor(nil succ pred)
define-frame GreaterThan(nil greater lesser)
```

The SNePS relations `greater` and `lesser` that serve as arc-labels in Cassie’s semantic network, can be inferred from a particular orientation of nodes in Cassie’s network. A path in SNePS is a sequence of arcs, nodes, and inverse-arcs (i.e., going against the direction of the arc). Two path-based inference rules are needed:

```
define-path greater (or greater (compose succ (kstar (compose pred- ! succ))))
define-path lesser (or lesser (compose pred (kstar (compose succ- ! pred))))
```

The first rule says that a `greater` SNePS relation can be inferred whenever there is a path that includes the `greater` relation (the trivial case), or when there is a path consisting of a `succ` arc followed by zero or more paths consisting of an inverse `pred` arc, an asserted node `!`, and a `succ` arc. The second rule says that a `lesser` SNePS relation can be inferred whenever there is a path that includes the `lesser` relation (the trivial case), or when there is a path consisting of a `pred` arc followed by zero or more paths consisting of an inverse `succ` arc, an asserted node `!`, and a `pred` arc.

Used in tandem, these two rules can be used to infer a belief involving the `GreaterThan` relation. This is illustrated in Figure 4.3. When a number y is said after a number x in the counting

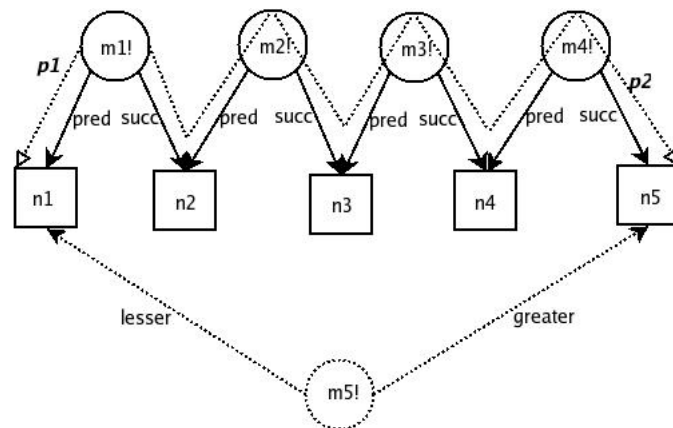


Figure 4.3: Path based inference of $5 > 1$. The *lesser* arc of $m5$ is implied by path $p1$ (from $m1$ to $n1$) and the *greater* arc of $m5$ is implied by path $p2$ (from $m1$ to $n5$).

sequence, this pair of rules imply that $y > x$. This inference is a “subconscious” one for Cassie in that she does not use KL rules to compare the numbers, but relies instead on the implicit structure of the network as given by counting. However, this does not preclude the agent designer from developing a “conscious” comparison act in the KL. In fact, when an agent uses syntactic routines, the numeric representations cry out for a lexicographic comparison to determine which number is greater. The important point is that Cassie can support various techniques for attributing the `GreaterThan` relation, but all such relationships are captured under a common case-frame.

4.3.2 Count-Addition

Count-addition is Cassie's gateway to arithmetic. When children learn to add by counting, they progress through a series of successively more efficient strategies. These strategy shifts happen in conjunction with a shift from concrete representations towards more abstract representations. A model of strategy shifts will be discussed in great detail in the next chapter. Here, I will describe one method of count-addition for abstract internal arithmetic known as the COUNT-ON technique.

To compute the sum of two numbers x and y , Cassie says the first addend x and proceeds to count for y many more numbers: Although I have stated the algorithm iteratively, it is much more

Algorithm 1 Count-Addition of $x + y$

Say x
 Say $x + 1, x + 2, \dots, x + y$

natural to implement it recursively using SNeRE. The expression of these routines as non-recursive iterations, although more cognitively plausible, imposes a great deal of implausible representational machinery (e.g., predicates tracking each intermediate value which drastically slow Cassie's reasoning). What is more important is that the acts themselves can be computationally expressed in SNeRE. The complex acts⁸ involved are `CountFromFor` and `CountAdd`:

```

;;;
;;; To count from x for y numbers, count from x+1 for y-1 numbers.
;;;
all(x,y)({Number(x),Number(y)} &=>
  ActPlan(CountFromFor(x,y),
    withsome(?xpl,
      Successor(?xpl,x),
      withsome(?yml,
        Successor(y,?yml),
        snif({if(Zero(?yml),believe(FinalSum(?xpl))),
          else(CountFromFor(?xpl,?yml)})),
        Say("I don't know how to count add these numbers")),
      Say("I don't know how to count add these numbers")))).

;;;
;;; To Count-Add x and y, count from x for y numbers.
;;;
all(x,y)({Number(x),Number(y)} &=>
  {ActPlan(CountAdd(x,y),
    ssequence(SayLine+"Count Adding",x,y),
    ssequence(CountFromFor(x,y),
      withsome(?z,
        FinalSum(?z),
        ssequence(believe(Evaluation(Result(CountSum,x,y),?z)),
          disbelieve(FinalSum(?z))),
        SayLine("I could not determine the sum.")))))},
  Effect(CountAdd(x,y),Evaluated(Result(CountSum,x,y))))).

```

The recursion operates as a sequence of calls to `CountFromFor`: To count from x for y numbers, count from the successor of x for the predecessor of y many numbers (i.e., count from $x + 1$ for $y - 1$ many numbers). To count from $x + 1$ for $y - 1$ many numbers, count from $x + 2$ for $y - 2$ many numbers, and so on. Eventually, Cassie reaches the case where she is counting from $x + y$ for 0 numbers. When this occurs, Cassie asserts that the number arrived at is the answer using the `FinalSum` case-frame.

⁸This does not include the variations of this act demanded by the unique-variable binding-rule (UVBR), see §4.5

$\llbracket \text{FinalSum}(x) \rrbracket$ asserts that $\llbracket x \rrbracket$ is the sum reached by the current (i.e., attended to) act of count-addition.

The `CountAdd` act then asserts that this number is the value of the procept "the `CountSum` of x and y ". The effect of performing this act is that Cassie will believe that this procept will have a value.

Cassie will be given a variety of ways to perform each arithmetic routine. We need a way form an association between a general act (i.e., addition) and a specific way of performing the act (i.e., count addition). To link up the fact that this way of finding a sum is a sum *simpliciter* and that this way of adding is an addition *simpliciter*, the following rules are used:

```

;;;
;;;Every counted sum obtained by this method is a sum.
;;;
all(x,y,z)(Evaluation(Result(CountSum,x,y),z) => Evaluation(Result(Sum,x,y),z)).

;;;
;;;CountAdd is a plan for adding.
;;;
all(x,y)({Number(x),Number(y)} &=> ActPlan(Add(x,y),CountAdd(x,y))).

```

These rules play an important role for the part of Cassie's KL responsible for explanation because they link routine-specific results with general results and plans for performing an act with the general act.

This routine builds on counting by allowing Cassie to begin her count from an arbitrary spot on her mental number-line. The `Successor` frame can be used to determine not only the successor of a number, but also its predecessor. In the recursive implementation, this allows the first addend to increase as the second is decreasing. This is a model of the "finger-transfer" method children use, i.e., lowering one finger on the left hand for each finger that is raised on the right hand.

4.3.3 Count-Subtraction

The ability to "reverse" directions on the number line by attending to predecessors gives rise to count-subtraction. To compute the difference $x - y$, Cassie says x and counts down for y many numbers:

Algorithm 2 Count-Subtraction of $x - y$

Say x

Say $x - 1, x - 2, \dots, x - y$

The implementation of count-subtraction involves the `CountDownFromFor` and `CountSubtract` complex acts and is similar in structure to that of count-addition:

```

;;;To count down from x for y numbers, count down from x-1 for y-1 numbers.
all(x,y)({Number(x),Number(y)} &=>
  ActPlan(CountDownFromFor(x,y),
    withsome(?xml,
      Successor(x,?xml),
      withsome(?yml,
        Successor(y,?yml),
        sniff({if(Zero(?yml),believe(FinalDifference(?xml))),
          else(CountDownFromFor(?xml,?yml)})),

```

```

        believe(FinalDifference(x))))).

;;;
;;; To Count-Subtract y from x, count down from x for y numbers.
;;;
all(x,y)({Number(x),Number(y)} &=>
  {ActPlan(CountSubtract(x,y),
    snif({if(GreaterThan(y,x),
      ssequence(
        SayLine("I cannot subtract a larger number from a smaller number."),
        believe(Evaluation(Result(CountDifference,x,y),undefined))}),
      else(ssequence(SayLine+"Count Subtracting",x,y),
        ssequence(CountDownFromFor(x,y),
          withsome(?z,
            FinalDifference(?z),
            ssequence(
              believe(Evaluation(Result(CountDifference,x,y),?z)),
              disbelieve(FinalDifference(?z))),
            SayLine("I could not determine the difference. "))))))}),
    Effect(CountSubtract(x,y),Evaluated(Result(CountDifference,x,y))))).

```

The major difference is that Cassie first utilizes the `GreaterThan` relation to determine whether she is trying to subtract a larger from a smaller number. This is simply done for the sake of the user. The natural numbers are not closed under subtraction, and, thus, if she tried to subtract y from x with $y > x$, she would eventually try to deduce a predecessor for 0. Not finding such an entity in her KL, she would report that she could not determine the difference.

Using the recursive act `CountDownFromFor`, Cassie subtracts y from x by subtracting $y - 1$ from $x - 1$, $y - 2$ from $x - 2$, ... 0 from $x - y$. The number $x - y$ is the basis of the recursion and is marked as the result using `FinalDifference`.

`[[FinalDifference(x)]]` asserts that `[[x]]` is the difference reached by the current (i.e., attended to) act of count-subtraction. If, in the course of acting, Cassie ever finds herself subtracting a larger number from a smaller one, she will assign the value `undefined` in the value position of the evaluation frame.⁹

The effect of performing this act is that Cassie will believe that this procept will have a value. As with count-addition, these acts are bound to the general acts for subtraction:

```

;;;
;;; Every counted difference obtained by this method is a difference.
;;;
all(x,y,z)(Evaluation(Result(CountDifference,x,y),z) => Evaluation(Result(Difference,x,y),z)).

;;;
;;; CountSubtract is a plan for subtracting.
;;;
all(x,y)({Number(x),Number(y)} &=> ActPlan(Subtract(x,y),CountSubtract(x,y))).

```

4.3.4 Iterated-Addition Multiplication

With addition and subtraction expressed in the simpler routine of counting, it is now possible to phrase multiplication and division in terms of addition and subtraction, respectively. Multiplication can be expressed as iterated addition, by considering the product $x \times y$ as:

⁹The semantics of `[[undefined]]` is *roughly* that an undefined value of which Cassie cannot predicate anything else.

$$\underbrace{x + x + \dots + x}_{y-1 \text{ additions}}$$

The algorithm can be completely stated using previously defined operations $+$, $-$, and $>$:

Algorithm 3 Iterated-Addition multiplication of $x \times y$

```

set PRODUCT  $\leftarrow$  0
ITERATIONS  $\leftarrow$   $x$ 
while ITERATIONS  $>$  0 do
    PRODUCT  $\leftarrow$  PRODUCT +  $x$ 
    ITERATIONS  $\leftarrow$  ITERATIONS - 1
end while

```

We must be particularly careful with the semantics of iteration. In the instruction “to find $x \times y$, add x to itself $y - 1$ times”, we are not iterating the *same* act $y - 1$ times (i.e., we are not computing $x + x$ a total of $y - 1$ times. Instead, we need to repeat an act of the same form $y - 1$ times. This act takes the resulting sum of each addition and makes it an argument in the subsequent addition.

The embodied analog for iteration in the domain of object collection is an accumulated enumeration as object collections are merged. In the domain of object construction, iterating is adding pieces to an ever larger construction. In iterated-addition multiplication, the added collection or piece is a temporary representation of the product as it is accumulated or constructed.

The SNePS implementation of iterated-addition multiplication use two complex acts AddFromFor and AddMultiply:

```

;;;Base case of add multiplication. To add from x for 0 iterations
;;;just stay at x. Also, believe appropriate SumProduct for unit
;;;multiplication
all(x,y)({ProductResultSoFar(x),Zero(y)} &=>
    ActPlan(AddFromFor(x,y),
        withsome(?one,
            Successor(?one,y),
            believe(Evaluation(Result(SumProduct,x,?one),x)),
            Say("Problem asserting the unit multiplication")))).

;;;Recursive case of add multiplication. To repeat the addition of w
;;;for z iterations, add w to itself and repeat the process for z-1
;;;iterations.
all(w,x,y,z)({NumToAdd(w), Successor(z,y),ProductResultSoFar(x)} &=>
    ActPlan(AddFromFor(x,z),
        snsequence(disbelieve(ProductResultSoFar(x)),
            snsequence(Add(x,w),
                withsome(?p,
                    Evaluation(Result(Sum,x,w),?p),
                    snsequence(believe(ProductResultSoFar(?p)),
                        AddFromFor(?p,y)),
                    SayLine(`I could not determine the sum`)))))).

;;;
;;; To Add-Multiply x and y, add x to itself y-1 times
;;;
all(x,y)({Number(x),Number(y)} &=>
    {ActPlan(AddMultiply(x,y),
        snsequence(SayLine+"Add Multiplying",x,y),
        snsequence(believe(ProductResultSoFar(x)),

```

```

ssequence(believe(NumToAdd(x)),
ssequence(withsome(?p,Successor(y,?p),AddFromFor(x,?p),
    Say("I don't know how to add-multiply these numbers")),
ssequence(withsome(?z,
    ProductResultSoFar(?z),
    ssequence(believe(Evaluation(Result(SumProduct,x,y),?z)),
        disbelieve(ProductResultSoFar(?z))),
    SayLine("I could not determine the product.")),
    disbelieve(NumToAdd(x)))))),
Effect(AddMultiply(x,y),Evaluated(Result(SumProduct,x,y))))}.

```

To multiply x and y , Cassie adds from x for a number of iterations equal to the predecessor of y . This involves invoking `AddFromFor`, whose recursive plan is to update the temporary `ProductResultSoFar`, and adding x before performing `AddFromFor` for one less iteration.

$\llbracket \text{ProductResultSoFar}(x) \rrbracket$ represents the belief that $\llbracket x \rrbracket$ is the temporary value Cassie associates with the product (for the multiplication she is attending to). $\llbracket \text{NumToAdd}(x) \rrbracket$ denotes the belief that $\llbracket x \rrbracket$ is the number being added successively to produce the product.

The basis plan for this recursion is invoked when Cassie has zero more additions to perform. The `SumProduct` then contains the final result. The basis case of the recursion also asserts the unit product $x \times 1$, because such a product is an iteration of zero additions (in other words, to multiply x by 1 involves “doing nothing” in terms of addition). The product $x \times 0$ is undefined using this procedure. All this means is that Cassie does not understand this product *in terms of* addition.

The addition “subroutine” used in the act of `AddFromFor` is specified in its general form `Add`. Rather than forcing Cassie to use `CountAdd` (one of the ways she knows how to perform `Add`), the multiplication act treats the method of addition as a black box. Thus, for every way Cassie knows how to perform `Add`, she has a different way of performing `AddMultiply`.

We see that it is possible to have all of these acts bottom out at counting. Furthermore, the use of the counting `Successor` relation is also valuable in expressing information for internal and temporary structures (e.g., the number of iterations to go). The effect of performing this act is that Cassie will believe that this procept has a value. Again, the link between general and specific multiplication must be made:

```

;;;
;;;Every sum product obtained by this method is a product.
;;;
all(x,y,z)(Evaluation(Result(SumProduct,x,y),z) => Evaluation(Result(Product,x,y),z)).

;;;
;;;AddMultiply is a plan for multiplying.
;;;
all(x,y)({Number(x),Number(y)} &=> ActPlan(Multiply(x,y),AddMultiply(x,y))).

```

4.3.5 Iterated-Subtraction Division

Division, the inverse of multiplication, can be expressed as repeated subtraction, the inverse of repeated addition. The method of division by repeated subtraction was used by Euclid (see *Elements*, VII.1). The quotient $x \div y$ is the number of times y can be subtracted from x (updating the value of x to this result) until this value reaches zero.

This is an effective procedure only if the quotient is a natural number or if $y \neq 0$. Cassie’s implementation addresses these cases.

Algorithm 4 Iterated-Subtraction division of $x \div y$

```
QUOTIENT  $\leftarrow$  0
set DIVIDEND to  $x$ 
while DIVIDEND  $\neq$  0 do
  DIVIDEND  $\leftarrow$  DIVIDEND  $-y$ 
  QUOTIENT  $\leftarrow$  QUOTIENT +1
end while
```

The SNePS implementation of iterated-subtraction division consists of the complex acts `SubtractUntilZero`, and `SubtractDivide` as follows:

```
;;; Base case of subtracting until zero
all(x,z)({Number(x),NumToSubtract(x),QuotientSoFar(z)} &=>
  ActPlan(SubtractUntilZero(x,x),
    snsequence(Subtract(x,x),
      withsome(?s,
        Successor(?s,z),
        snsequence(believe(QuotientSoFar(?s)),
          disbelieve(QuotientSoFar(z))),
        SayLine("I could not determine the successor")))).

;;;
;;; Recursive case of subtracting until zero
;;;
all(x,y,z)({Number(x),NumToSubtract(y),QuotientSoFar(z)} &=>
  ActPlan(SubtractUntilZero(x,y),
    snsequence(Subtract(x,y),
      snsequence(withsome(?s,
        Successor(?s,z),
        snsequence(believe(QuotientSoFar(?s)),
          disbelieve(QuotientSoFar(z))),
        SayLine("I could not determine the successor")),
      withsome(?r,
        Evaluation(Result(Difference,x,y),?r),
        SubtractUntilZero(?r,y),
        SayLine("I could not determine the difference")))).

;;;
;;; To Subtract-Divide x by y, subtract y from x until the difference
;;; is 0.
;;;
all(x,y)({Number(x),Number(y)} &=>
  {ActPlan(SubtractDivide(x,y),
    sniff({if(Zero(y),snsequence(Say("I cannot divide by zero."),
      believe(Evaluation(DifferenceQuotient,x,y),undefined))),
    else(snsequence(SayLine+"Subtract Dividing",x,y),
      withsome(?z,Zero(?z),
        snsequence(believe(QuotientSoFar(?z)),
          snsequence(believe(NumToSubtract(y)),
            snsequence(SubtractUntilZero(x,y),
              snsequence(withsome(?q,QuotientSoFar(?q),
                snsequence(believe(Evaluation(Result(DifferenceQuotient,x,y),?q)),
                  disbelieve(QuotientSoFar(?q))),
                  SayLine("I could not determine the difference."))),
                disbelieve(NumToSubtract(y))))),
              SayLine(" I could not determine the difference.")))))},
    Effect(SubtractDivide(x,y), Evaluated(Result(DifferenceQuotient,x,y)))).
```

After checking to see that y is not zero, Cassie sets the quotient to zero and begins subtracting y

from x with the recursive `SubtractUntilZero` act. The recursive plan of `SubtractUntilZero` stores the accumulating quotient in `QuotientSoFar` until the basis plan which updates this value one more time.

$\llbracket \text{QuotientSoFar}(x) \rrbracket$ represents the belief that $\llbracket x \rrbracket$ is the temporary value Cassie associates with the quotient (for the division she is attending to).

The final value of `QuotientSoFar` is set to the quotient. The effect of performing this act is that Cassie will believe that this precept will have a value. As with the other arithmetic acts, a link to the general acts are given by:

```

;;;
;;;Every difference quotient obtained by this method is a quotient.
;;;
all(x,y,z)(Evaluation(Result(DifferenceQuotient,x,y),z) => Evaluation(Result(Quotient,x,y),z)).

;;;
;;;SubtractDivide is a plan for dividing.
;;;
all(x,y)({Number(x),Number(y)} &=> ActPlan(Divide(x,y),SubtractDivide(x,y))).

```

4.3.6 Inversion Routines

Subtraction and division can be specified in terms of addition and multiplication without iteration as specified above. Subtraction can be expressed as inverted addition and division can be expressed as inverted multiplication. In the case of subtraction, Cassie can compute the difference $x - y$ by finding the z that satisfies $y + z = x$:

Algorithm 5 Inverted-Addition subtraction of $x - y$

Find z such that $y + z = x$

Similarly, the quotient $x \div y$ can be computed by finding the z that satisfies $y \times z = x$:

Algorithm 6 Inverted-Multiplication division of $x \div y$

find z such that $y \times z = x$

The implementation of subtraction and division through inversion is much the same. Here is the SNePS implementation of inverted-multiplication division:

```

;;;
;;; MultIterate multiply every number from 1 to y-1 by y until the
;;; result is x
;;;
all(x,y,z)({Number(x),Number(y),QuotientCandidate(z)} &=>
  ActPlan(MultIterate(x,y),
    snsequence(Multiply(y,z),
      snif({if(Evaluation(Result(Product,y,z),x),
        snsequence(believe(Evaluation(Result(InvMultQuotient,x,y),z)),
          disbelieve(QuotientCandidate(z))),
        else(withsome(?s,
          Successor(?s,z),
          snsequence(disbelieve(QuotientCandidate(z)),
            snsequence(believe(QuotientCandidate(?s)),
              MultIterate(x,y))),

```

```

SayLine("I can't perform the division")))))).

;;;
;;; To InvMult-Divide x by y, find a number z that, when multiplied by y,
;;; yields x.
;;;
all(x,y,z)({Number(x),Number(y),Zero(z)} &=>
  {ActPlan(InvMultDivide(x,y),
    ssequence(SayLine+"Inverse Multiply Dividing",x,y),
    withsome(?z,
      (Evaluation(Result(Product,y,?z),x) or Evaluation(Result(Product,?z,y),x)),
      believe(Evaluation(Result(InvMultQuotient,x,y),?z))),
      ssequence(believe(QuotientCandidate(z),
        MultIterate(x,y))))),
    Effect(InvMultDivide(x,y),Evaluated(Result(InvMultQuotient,x,y)))}).

```

Cassie first checks whether she already has computed the relevant inverse product. If so, she does not need to do any more work. If not, she performs `MultIterate`, which tries successive values of $z = 0, 1, \dots, n$ where n is the largest number Cassie has counted to. The implementation of inversion routines is not as efficient in SNePS, but they do represent alternative ways to express subtraction and division in terms of simpler routines.

4.4 Cognitive Arithmetic Shortcuts

A computational model of mental arithmetic should account for the performance differences between classes of problems where “shortcut” methods can be applied and those where they cannot. The following is a list of such shortcuts:

Operation	Shortcut
Addition	$n + 1 = m$ where <code>Successor(m, n)</code> $n + 0 = n$
Subtraction	$n - 1 = p$ where <code>Successor(n, p)</code> $n - 0 = n$ $n - n = 0$ $n - p = 1$ where <code>Successor(n, p)</code>
Multiplication	$n * 0 = 0$ $n * 1 = n$
Division	$n / 1 = n$ $n / n = 1$

The use of such shortcuts is indicative of a deeper understanding of arithmetic because it requires an agent to consider the form of the problem while planning how to solve it (Carr and Hettinger, 2003). In order to deploy such methods, the agent must notice a regularity between inputs and outputs for a particular operation and essentially rewrite the general procedure for the class of problems. Moreover, the agent must understand that the shortcut applies *only* to the subset of problems in a proper form.

These cognitive shortcuts may be introduced to Cassie in such a way that she will always prefer the shortcut method over other methods. This is done with a special case-frame called

Prefer which will be described in detail in the context of count-addition strategy change in the next chapter. The top-level SNeRE plans for short-cut arithmetic are:

```

;;;
;;;establish shortcuts as specific ways of doing general operations
;;;
all(x)(Number(x) =>
  {ActPlan(Add(x,n0),ShortcutAdd(x,n0)),
    Prefer(ShortcutAdd(x,n0)),
    ActPlan(Add(x,n1),ShortcutAdd(x,n1)),
    Prefer(ShortcutAdd(x,n1)),
    ActPlan(Subtract(x,n0),ShortcutSubtract(x,n0)),
    Prefer(ShortcutSubtract(x,n0)),
    ActPlan(Subtract(x,n1),ShortcutSubtract(x,n1)),
    Prefer(ShortcutSubtract(x,n1)),
    ActPlan(Subtract(x,x),ShortcutSubtract(x,x)),
    Prefer(ShortcutSubtract(x,x)),
    ActPlan(Multiply(x,n0),ShortcutMultiply(x,n0)),
    Prefer(ShortcutMultiply(x,n0)),
    ActPlan(Multiply(x,n1),ShortcutMultiply(x,n1)),
    Prefer(ShortcutMultiply(x,n1)),
    ActPlan(Divide(x,n1),ShortcutDivide(x,n1)),
    Prefer(ShortcutDivide(x,n1)),
    ActPlan(Divide(x,x),ShortcutDivide(x,x)),
    Prefer(ShortcutDivide(x,x)),
    Effect(ShortcutAdd(x,n0),Evaluated(Result(ShortcutSum,x,n0))),
    Effect(ShortcutAdd(x,n1),Evaluated(Result(ShortcutSum,x,n1))),
    Effect(ShortcutSubtract(x,n0),Evaluated(Result(ShortcutDifference,x,n0))),
    Effect(ShortcutSubtract(x,n1),Evaluated(Result(ShortcutDifference,x,n1))),
    Effect(ShortcutSubtract(x,x),Evaluated(Result(ShortcutDifference,x,x))),
    Effect(ShortcutMultiply(x,n0),Evaluated(Result(ShortcutProduct,x,n0))),
    Effect(ShortcutMultiply(x,n1),Evaluated(Result(ShortcutProduct,x,n1))),
    Effect(ShortcutDivide(x,n1),Evaluated(Result(ShortcutQuotient,x,n1))),
    Effect(ShortcutDivide(x,x),Evaluated(Result(ShortcutQuotient,x,x)))}.

all(x,y)({Number(x),Successor(y,x)} &=>
  {ActPlan(Subtract(x,y),ShortcutSubtract(x,y)),
    Prefer(ShortcutSubtract(x,y)),
    Effect(Subtract(x,y),Evaluated(Result(ShortcutDifference,x,y)))}).

```

Using SNePS, it is possible to write plans for *fully-grounded* acts (i.e., specific acts whose arguments are all base nodes such as $\text{Add}(n1, n1)$), *partially-grounded* acts (i.e., acts whose arguments include base nodes and variable nodes such as $\text{Add}(x, n1)$), and *totally-ungrounded* acts (i.e., general acts whose arguments are all variables such as $\text{Add}(x, y)$). These cognitive shortcuts take advantage of the support for all three types of acts.

The plan for each of the shortcut acts consists of an automatic belief in the answer based on the form of the problem. Here are some examples:

```

;;;x + 0 = x
;;;x - 0 = x
;;;x - x = 0
;;;x * 0 = 0
;;;x * 1 = x
;;;x / 1 = x
;;;x / x = 1
all(x)(Number(x) =>
  {ActPlan(ShortcutAdd(x,n0),
    ssequence(SayLine+("Shortcut Adding",x,n0),
      believe(Evaluation(Result(ShortcutSum,x,n0),x)))),
    ActPlan(ShortcutSubtract(x,n0),
      ssequence(SayLine+("Shortcut Subtracting",x,n0),

```

```

        believe(Evaluation(Result(ShortcutDifference,x,n0),x))),
ActPlan(ShortcutSubtract(x,x),
        ssequence(SayLine+("Shortcut Subtracting",x,x),
        believe(Evaluation(Result(ShortcutDifference,x,x),n0))),
ActPlan(ShortcutMultiply(x,n0),
        ssequence(SayLine+("Shortcut Multiplying",x,n0),
        believe(Evaluation(Result(ShortcutProduct,x,n0),n0))),
ActPlan(ShortcutMultiply(x,n1),
        ssequence(SayLine+("Shortcut Multiplying",x,n1),
        believe(Evaluation(Result(ShortcutProduct,x,n1),x))),
ActPlan(ShortcutDivide(x,n1),
        ssequence(SayLine+("Shortcut Dividing",x,n1),
        believe(Evaluation(Result(ShortcutQuotient,x,n1),x))),
ActPlan(ShortcutDivide(x,x),
        ssequence(SayLine+("Shortcut Dividing",x,x),
        believe(Evaluation(Result(ShortcutQuotient,x,x),n1))))).

```

Cognitive shortcuts are linked into the explanation system just like the other routines and are available to Cassie during an explanation.

4.5 UVBR and Cognitive Plausibility

The logic of SNePS enforces a unique-variable binding rule (UVBR). This means that any two variables differing in their syntactic representation are taken to denote different mental entities (Shapiro, 1986). Furthermore, a single syntactic representation (a variable or non-variable) denotes exactly one mental entity regardless of how many times it occurs in a well-formed formula. Thus, for example, `Connected(x,y)` expresses a relation of “connected” holding between two different entities and does not allow for an object to be “self-connected”. Supposing `x` and `y` are variables ranging over some domain D , then if x is bound to some particular c , y effectively ranges over the domain $D - \{c\}$.¹⁰

At face value, UVBR seems to place a burden on the agent’s belief space because it necessitates the introduction of additional arithmetic rules. For example, the general act for dividing x and y is given by a SNePSLOG wff of the form:

wffD: `all(x,y)({Number(x),Number(y)} => ActPlan(Divide(x,y),...)).`

However this says nothing about dividing a number by itself. To handle that case we must add the rule:

wffDuvbr: `all(x)(Number(x) => ActPlan(Divide(x,x),...)).`

While this leads to a proliferation of specific “UVBR-handling” rules in the agent’s belief space, there is a plausible philosophical reason to adopt UVBR as well as a plausible psychological reason.

First we examine the philosophical justification for adopting UVBR. Shapiro (personal communication) has pointed out that UVBR was defended by Wittgenstein (1922/2003) in the *Tractatus*:

¹⁰The order of binding of x and y is an implementation detail in SNePS.

Roughly speaking: to say of *two* things that they are identical is nonsense, and to say of *one* thing that it is self-identical with itself is to say nothing. I write therefore not $f(a, b) \wedge a = b$ but $f(a, a)$ (or “ $f(b, b)$ ”) and not $f(a, b) \wedge \neg(a = b)$ but $f(a, b)$ (5.5303–5.531).¹¹

Since I want to adopt a structuralist ontology for mathematics, a number in my theory is a position in a structure exemplifying the natural-number progression. To say of two positions in such an exemplifying structure that they are the same position is indeed nonsense and to say of a given position that it is self-identical is indeed vacuous of any information.

Now we turn to a psychological justification for the plausibility of UVBR. In the previous section, I discussed certain cognitive “shortcut” routines for performing automatic arithmetic when a given problem is in a certain form. Interestingly, there is a direct correlation between the UVBR-handling rules and the cognitive shortcut routines. The fact that dividing a number by itself requires $wffDuvbr$ is a psychological “hint” to an agent that this is a special form of division. The salience of $wffDuvbr$ in the agent’s belief space (since it is a top-level rule like $wffD$) lays the foundation for learning a general shortcut, namely that any number divided by itself will yield a quotient of 1. This sort of learning requires that the agent notices the following:

Whenever $wffDuvbr$ is used in the act of division of x by itself, then
 Evaluation(Result(Quotient, x, x), $n1$) is the form of the result, where
 $n1$ denotes the number 1.

4.6 Syntactic Arithmetic

Although count-based arithmetic routines are rich in meaning for a young learner, such routines do not scale up easily. A strain is placed on cognitive resources as operand size increases. This leads to a higher sensitivity to error. It is very likely that errors will occur even when adults rely on counting for very large quantities (e.g., the disputed totals and subtotals of Florida votes in the 2000 US election). The system of small subitizable cardinalities represented on the fingers can no longer be utilized in the same way. Arithmetic only becomes scalable to larger cardinalities by introducing a new representation for numbers (e.g., a place-value number system) and new routines to operate over such representations (e.g., place-value arithmetic routines).

Precision is attainable in place-value arithmetic routines largely because an agent can perform a sequence of precise operations in the familiar small-number system. The results of these operations must be combined in a well-defined way that respects the compositional semantics of the place-value system.

Place-value arithmetic routines are characterized by the following features:

- place-value representations of operands (e.g., decimal notation)
- operands written in a column-wise fashion on an extended medium (e.g., pencil markings on paper)

¹¹I have replaced Wittgenstein’s conjunction sign “.” by \wedge .

- sequential column-wise operations over places
- column-wise accrual of an intermediate or final results
- special written symbolization for temporary variables (e.g., carries, borrows, remainders)

Syntactic arithmetic is not as central to exhaustive explanation as semantic arithmetic (see the Appendix for the SNeRE implementations). However, there are several representational issues worth noting if the theory will scale up.

4.6.1 Multidigit Representations

Syntactic routines rely on a positional number system in which the position of a marker within a syntactic structure is inherent to its meaning in a larger structure. Semantic routines operate over numbers in their “native” context of meaning, i.e., as positions natural-number progression.

Fuson (1998) describes a series of conceptual supports for representing multidigit numbers. She describes a series of proto-multidigit representations that precede the full fledged conceptual system. These include:

- *Unitary*: 53 is represented as 53 “ones”.
- *X-TY*: 53 is represented as 50 “ones” and 3 “ones”.
- *Count-By-Tens*: 53 is represented as 5 groups of “ten ones” and 3 “ones”.
- *Place-Value*: 53 is represented as 5 “tens” and 3 “ones”.

Each of these representations is treated as a quantity, i.e., a number bound to a sortal. The sortals in question are units (abstractly considered). Each of these stages of multidigit representation is representable using the same mechanism used for quantity representation described in the next chapter. However, since I am not interested in modeling the transition to multidigit representations, I use a digit-string representation of multidigit numbers instead.

A digit-string is a recursive structure built up from the functional term defined by `define-frame S(nil head tail)`. To represent 1234 using a digit-string representation, we write `S(S(S(1, 2), 3), 4)`. The traversal of this structure, along with the proper semantics, uniquely specifies the numerical value of the digit-string. Digit-string representations can be associated with Cassie’s counting numerons via the `DigitStringOf` case frame. `[[DigitStringOf(S(S(S(1, 2), 3), 4), n1234)]]` is the proposition that `S(S(S(1, 2), 3), 4)` is the digit-string for the number `n1234`.

4.6.2 Extended Arithmetic

External tools for arithmetic are plentiful. Humans use pencil and paper, the abacus, and pocket calculators to assist in problem solving and as an aid to memory. Cassie should also have access to external tools for these same reasons.

Extended arithmetic involves a translation of a problem into the system required by the external tool. To properly use a calculator, one needs to know how to press buttons, what each

button represents, and how to read off an answer. A calculator will display information in a public communication-language because it is intended to be useful for a community of users.

Although extended cognition relies on external tools, there is an important difference between extended-arithmetic routines and the embodied-arithmetic routines. In an embodied-arithmetic act, the interactions with the objects in the world are the mechanism through which arithmetic is performed. The acts of collecting, constructing, measuring, and moving are the vehicle for arithmetic. In an extended-arithmetic act, the interactions with external tools requires significant interpretation on the part of the agent as to what the “outputs” of these tools signify. An act of pressing buttons on a calculator is not the vehicle for arithmetic; the real arithmetic work is being done *by* the calculator internally.

That having been said, there is clearly an overlap between extended and embodied routines with respect to certain external tools. For example, the stones of an abacus could be used for count-adding some small subitizable quantity (literally the number of stones). In this respect, using the abacus would be an embodied routine. However, the different rows of an abacus could be used to stand for different “places” in a place-value numeration system. In this case, the value of outputs can only be determined by an agent with access to the place-value convention. In this respect, using the abacus would be an extended routine.

Given this similarity between extended and embodied acts, I believe that a SNePS model of extended arithmetic should follow the architecture of embodiment (i.e., GLAIR). We can designate a set of Lisp functions as the *extended layer* that interacts with the KL in much the same way that the PML does. This is illustrated in Figure 4.4. Currently, the only implemented component of Cassie’s extended layer is a Lisp calculator. The `tell-ask` interface is deployed to import numeral representations for Cassie’s numerons (because the calculator (qua public tool) expects numeral inputs) and to return evaluations to Cassie’s KL. However, as the figure suggests, Cassie might also be provided with a set of Lisp variables to represent a “scratch-pad” to augment her KL beliefs. A benefit for cognitive modeling is the fact that the extended layer makes Cassie more efficient, just as extended cognition makes humans more efficient.

The implementation of the Lisp calculator is provided in the Appendix. As with semantic and syntactic arithmetic routines, the calculator routines are made available for both explanation and action.

4.7 Questions in SNePS

It is possible to ask a SNePS agent any question that can be expressed as a SNePSLOG wff with zero or more free variables. In this section, I will classify several kinds of questions of interest for an exhaustive explanation. First, we will need some definitions:

- A **question** is a SNePSLOG wff containing zero or more free variables terminated by the ‘?’ sign.
- An **answer** to a question is an asserted SNePSLOG wff in which all of the question’s free variables have been filled by terms, or else the answer is the terms themselves, or the answer is “I don’t know” if the agent cannot find terms to bind to the question’s free variables.

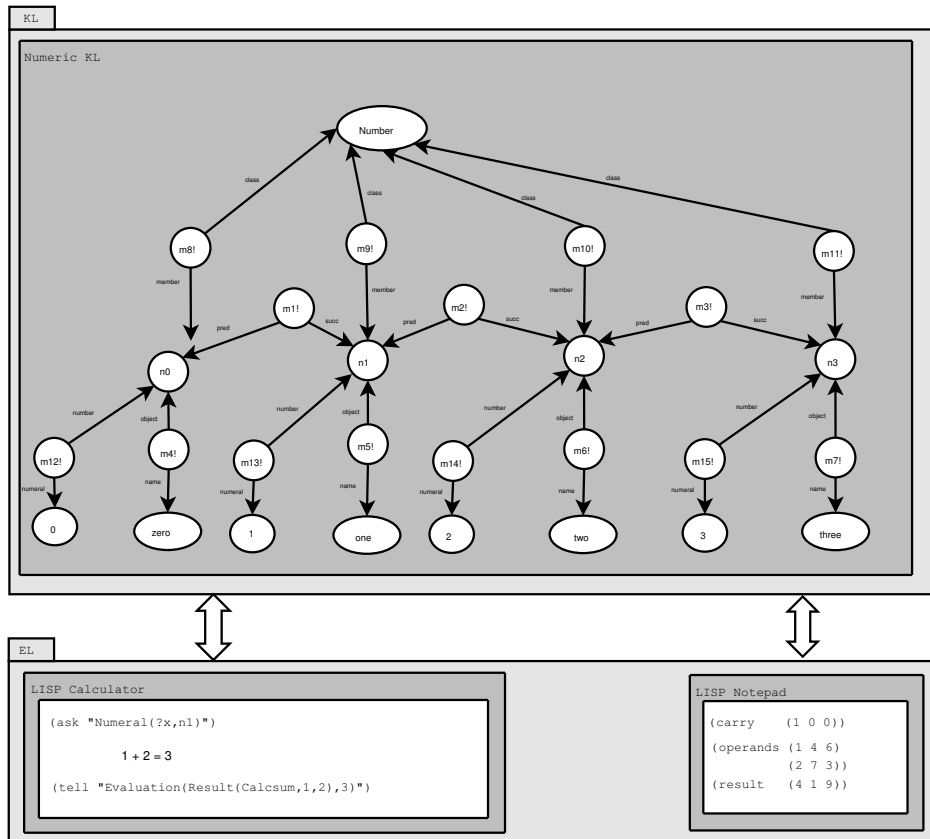


Figure 4.4: Representing an extended layer (EL), and its interaction with the knowledge layer (KL).

The free variables in the questions have usefully been called “queriables” (Belnap and Steel, 1976), indicating that an agent is seeking to fill those positions in the question form to construct the answer.

Under this formulation, it is very natural to formalize several types of questions in SNePS using the evaluation and result frames described above.

- *Is/Does* $2 + 3 = 5$? translates to $\text{Evaluation}(\text{Result}(\text{Sum}, n2, n3), n5)?$.
- *What is* $2 + 3$? translates to $\text{Evaluation}(\text{Result}(\text{Sum}, n2, n3), ?x)?$.
- *How can you find the result of* $2 + 3$? translates to $\text{Effect}(?a, \text{Evaluated}(\text{Result}(\text{Sum}, n2, n3)))?$.

The *is/does* question asks whether or not the evaluation is factual (for the agent being asked). The answers to such questions are represented by fully-ground (i.e., variable-free) SNePSLOG terms. The answer to a *what*-question is the value associated with a particular result frame. This is a declarative belief (e.g., knowing *that* $2 + 3 = 5$). The answer to a *how*-question is an act whose effect is an evaluated result-frame. This is a procedural belief (e.g., knowing *how* to compute $2 + 3$).

4.8 Why Questions and Procedural Decomposition

Why-questions are the crux of an explanation. While the other type of questions seem to be seeking a yes/no answer, a result, or a procedure, why-questions are explanation-seeking.

We have seen that what an inquirer is looking for in trying to answer other “wh-” questions is the ultimate conclusion. In contrast, when an inquirer is trying to cope with a why-question, she is looking for the argumentative bridge between initial assumptions and the given ultimate conclusion. The bridge is “the answer” in that it is what is sought (Hintikka and Halonen, 1995).

Thus, questions of this form¹² will be the crux of exhaustive explanation:

- *Why is $2 + 3 = 5$?*

Not surprisingly, why-questions are not as easily formalized in SNePS. As a first attempt at formalization, we may just say that answering the why-question is a function of answering the what-question, is/does-question, and how-question. After all, if an agent knows what the evaluation is, that the evaluation is factual, and how to compute the evaluation, it already has most of the ingredients for answering the why-question. However, we must remember that why-questions are explanation-seeking. Having all of the ingredients for an explanation is not an explanation. Cassie must be given an act that puts these pieces together in the right way. Because why-questions are explanation seeking, they are parsed into imperative commands in SNePSLOG. As such, they are not treated like questions at all, but as an act given to perform. “Why is $2 + 3 = 5$?” would be parsed as:

```
perform Explain(Evaluation(Result(Sum,n2,n3),n5))?
```

I will refer to Cassie’s mechanism for answering why-questions as “procedural decomposition”. The flow of Cassie’s deductions is shown in Figure 4.5. Why questions contain a presupposition. For example, the question “Why is $2 + 3 = 5$?” presupposes that “ $2 + 3 = 5$ ”. So Cassie first asks herself an is/does question. Is the result factual, or is the interrogator attempting a trick? If she has performed the relevant act, she will begin by saying that she believes the presupposition; otherwise, she will claim that she does not believe the proposition. This might be the case for two reasons: (1) she has not performed a relevant arithmetic act, or (2) she has performed a relevant act but has arrived at a different answer. Response (2) may seem overly cautious, but is the result of the fact that Cassie is not told that an arithmetic result has exactly one evaluation. So, for example, the fact that “ $2 + 3 = 5$ ” does not preclude that “ $2 + 3$ ” might evaluate to something else.

Cassie then attempts to deduce an act whose effect is to evaluate the result in question. In our example, she attempts to find acts ?a to satisfy the well-formed formula `Effect(?a, Evaluated(Result(Sum,n2,n3))`). In our example, the querable ?a might be filled by `Add(n2,n3)`. For each such act, Cassie states the ways she knows of performing the general act. This is achieved by finding the plans ?p1 for the general act ?a. In our

¹²Other formalizations of why-questions are given by (Bromberger, 1992; Mancosu, 2001).

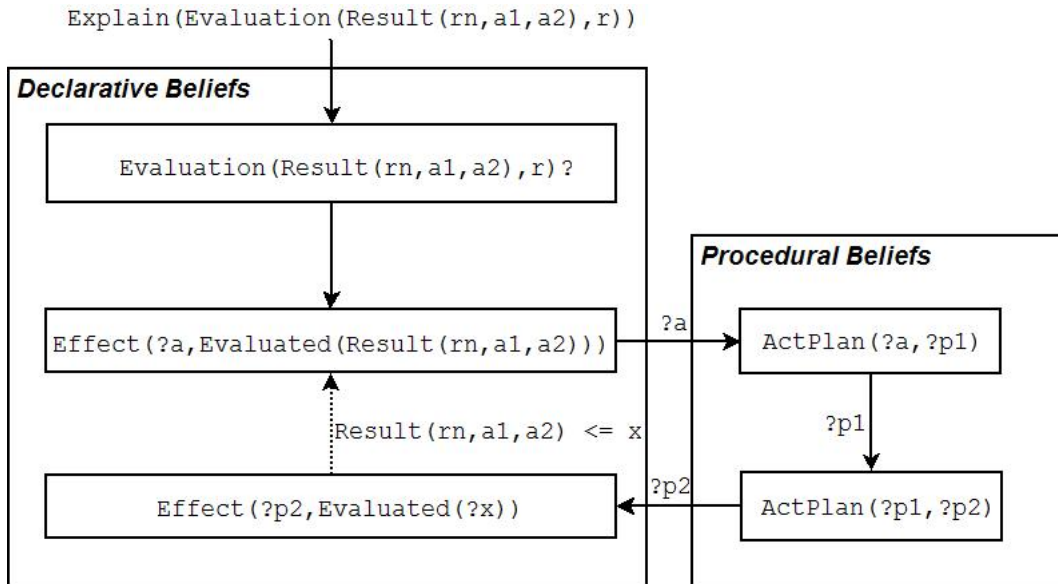


Figure 4.5: Procedural Decomposition

example, the querable $?p1$ might be filled by $\text{CountAdd}(n2, n3)$. If necessary, another deduction of this same form will move from the specific plan $?p1$ to a still more specific way of performing $?p1$, namely, $?p2$. This extra step is not necessary for the GCD case-study described below. Finally, Cassie deduces the effect of performing the specific plan $?p1$ (or $?p2$ if the extra deduction is performed). In our example, the effect of $\text{CountAdd}(n2, n3)$ is $\text{Evaluated}(\text{CountSum}, n2, n3)$. This tells the interrogator *why* the plan $?p1$ for the general act $?a$ is relevant for the evaluation of the result.

Importantly, Cassie's answer to the why question should prompt a logical next question for the interrogator. The resulting procept that is evaluated (marked $?x$ in Figure 4.5) suggests a new question to ask Cassie, based on this new procept. To take up a new example: if Cassie is asked "Why does $2 \times 3 = 6$?", she will say that she knows the result is correct (if she has performed the relevant multiplication), and that one way she knows of evaluating 2×3 is by performing AddMultiply which leaves a SumProduct evaluated. This naturally suggests that the interrogator should probe Cassie for her knowledge of sums.

The SNePS implementation of procedural decomposition consists of three acts: IsFactual , ActExplain , and Explain , as follows:

```
all(rn,a1,a2)({ResultName(rn),Number(a1),Number(a2)} &=>
  ActPlan(ActExplain(Result(rn,a1,a2)),
    withsome(?a,
      Effect(?a,Evaluated(Result(rn,a1,a2))),
      ssequence(SayDescrip(?a),
        ssequence(SayLine("And I can do this by performing"),
          withsome(?p,
            ActPlan(?a,?p),
            ssequence(SayDescrip(?p),
              ssequence(Say("Which has the effect(s) of"),
                withsome(?e,
                  Effect(?p,?e),
                  SayDescrip(?e),
```

```

        SayLine("I don't know the effect(s) of that")),
        SayLine("I don't know any plans for that"))),
    SayLine("I don't know how to evaluate this result"))).

```

```

all(rn,a1,a2,r)({ResultName(rn),Number(a1),Number(a2),Number(r)} =>
  {ActPlan(IsFactual(Evaluation(Result(rn,a1,a2),r)),
    sniff({if(Evaluation(Result(rn,a1,a2),r),
      SayLine("I know that the result is correct")),
      else(SayLine("I don't yet know that the result is correct"))})),
    ActPlan(Explain(Evaluation(Result(rn,a1,a2),r)),
      ssequence(IsFactual(Evaluation(Result(rn,a1,a2),r)),
        ssequence(SayLine("Here's HOW I can find the result"),
          ActExplain(Result(rn,a1,a2))))))}).

```

4.9 Conceptual Definitions

Being able to answer why-questions by applying procedural decomposition will lead Cassie towards her cognitive axioms. However, why-questions do not cover all of the needs for exhaustive explanation. Although we have seen what-questions applied to results, there is another form of what-question that seeks a conceptual definition. These include questions like: “what is a number?”, “what is 2?”, “what is addition?”, “what is a divisor?”. Being able to answer such questions demonstrates an understanding of the *meaning* of the concept involved.

Being able to give such a conceptual definition based on an agent’s current beliefs is at the heart of the contextual-vocabulary acquisition (CVA) project (Rapaport and Ehrlich, 2000). I will use similar techniques but instead apply them to a KL with mathematical beliefs.

Using conceptual-role semantics (Rapaport, 2002), the meaning of a concept as represented by a base node is the position of the node within a semantic network. As the network around a node gets more complex, the node automatically acquires new meanings. Computationally, this implies that there are “non-local” effects on the meaning of any given item, i.e., as new paths from a given node to other nodes develop, the given node acquires new meaning without itself being updated in any way. Under this characterization, any meaning is temporary. A node *b*, in network *N1* may have a different meaning than the node *b* in network *N2* (with $N1 \neq N2$), because *N1* and *N2* contain different paths leading to or from *b*. These differing paths indicate differing relationships between *b* and the other nodes in the network. Cassie’s network changes over time (i.e., as she acquires new beliefs, as she is acting, as she discards pre-existing beliefs).

We may not expect a human to answer the “deep questions” that conceptual definitions demand. However, things like “number”, “two”, and “divisor” do occur in the dictionary, and most people can come up with a dictionary-like definition if pressed. Thus, they should be a part of an exhaustive explanation.

The SNePS implementation of conceptual definition is Cassie’s complex act `Define`:

```

all(x)(ConceptToDefine(x) => ActPlan(Define(x),
  ssequence(GetClassMembers(x),
    ssequence(GetClassContainment(x),
      ssequence(GetActsWithArgumentType(x),
        ssequence(GetOperationsWithResultType(x),
          ssequence(GetMannerOfRelationships(x),

```

GetGeneralResultType(x)))))).

Each of the component acts in the plan is a primitive act that relies on path-based inferences:

- `GetClassMembers` lists members of `x`, if `x` is a class.
- `GetClassContainment` lists the classes of which `x` is a member.
- `GetActsWithArgumentType` lists the acts whose antecedent restrictions demand argument type `x`.
- `GetOperationsWithResultType` lists all the specific result types corresponding to the general result type `x`.
- `GetMannerOfRelationships` lists the ways in which an act `x` can be performed.
- `GetGeneralResultType` lists the general type of result associated with specific result type `x`.

If any of these inferences come up empty, Cassie is silent about the particular inference and adds nothing to the definition. Unlike the CVA technique, which provides different definition algorithms for nouns and verbs, this same act is performed for all conceptual definitions.

A more complicated sort of conceptual question is one with an implicit arbitrary reference. This can be stated as a how question “How do you add?”, which is elliptical for “How do you add any numbers?”. SNePS 2.7 does not support arbitrary references (the next version of SNePS, SNePS 3 will). However, in Chapter 7, I briefly consider how such questions might be handled.

4.10 Questions in Natural Language

Natural-language competence is usually taken to be a mark of general intelligence, and, even in the arithmetic domain, it makes sense to provide a front-end through which Cassie can parse questions expressed in natural language.

All of the evaluation questions I am interested in can be generated by the following grammar:

```
<QUESTION> := <ISDOES> <EVAL> | <WHY> <EVAL> | <WHAT> <RES> |
             <HOW><RES> | <DEFINE> <CONCEPT> ;
<ISDOES> := is | does ;
<WHY> := why <ISDOES>;
<WHAT> := what is;
<HOW> := how <DOCAN> you <COMPUTE>;
<DEFINE> := define
<DOCAN> := do | can;
<COMPUTE> := ((compute | find) <RES>) | <GENERALACT>;
<GENERALACT> := ((add | multiply) <NUMERAL> and <NUMERAL>) |
                (subtract <NUMERAL> from <NUMERAL>) |
                (divide <NUMERAL> by <NUMERAL>);
<EVAL> := <RES> = <NUMERAL>;
<RES> := <NUMERAL> <OP> <NUMERAL>;
<OP> := + | - | * | / ;
```

Since this is an NL front-end, the operands are expressed in the public-communication language as numerals.

A simple recursive-descent parser is designed to handle this grammar. To avoid complicating the parser with lookahead issues, e.g., “What is 2?” is a conceptual definition question and “What is $2 + 3$?” is a what-question, I have used the terminal “define” to generate conceptual definition questions. The parser is invoked by the primitive action `answerQuestion` given a well-formed question. Cassie first classifies the type of question, then represents the question in SNePSLOG as indicated above. For is/does, what, and how questions, the `askwh` interface is used to query the KL for an answer. For why questions and conceptual definitions, `tell perform` is used to invoke the acts `Explain` and `Define` respectively.

4.11 A Case Study: Greatest Common Divisor

The task of finding the greatest common divisor (GCD) of two natural numbers will be used to test Cassie’s depth of understanding of abstract internal arithmetic. Although GCD is not a concept usually associated with early mathematical cognition, there are several benefits for using it as a target problem:

- **Sufficient Complexity:** Many solutions to the GCD problem require Cassie to use several arithmetic routines. GCD allows her to demonstrate her understanding of those routines. However, a solution to the GCD problem need not include the advanced conceptualizations of algebra or calculus.
- **Multiple Solution Methods:** There are many algorithms for finding a GCD. These vary in terms of intuitiveness, efficiency, and technique (e.g., GCD can be solved arithmetically or geometrically).
- **Natural Language Intuitions:** People who are not familiar with GCD know that a greatest common divisor is a greatest common “something”. This ends up yielding a common-sense algorithm for GCD based on the meanings of “greatest” and “common”.
- **Interface to Post-Arithmetic Concepts:** To solve a GCD problem, an agent must use some non-arithmetic representations and acts. Solving a GCD problem shows that the agent can “scale up” from arithmetic.

The presence of multiple solution methods for GCD immediately implies that some methods will be easier to understand (and explain) than others. One of the original motivations for this research was to demonstrate how an agent might determine that a non-intuitive GCD algorithm was also a GCD algorithm (this problem was originally posed in (Rapaport, 1988)). See §3.5.3 above for a discussion of multiple justifications.

The exhaustive explanation demonstration given below uses only semantic arithmetic routines. However, when all arithmetic routine categories are included in the run, Cassie will explain herself in a non-deterministic way. That is to say, her explanations may differ for the same question. This is due to the fact that the SNeRE act for `Explain` utilizes the `withsome` SNeRE-primitive act, which finds and applies its variable non-deterministically.

4.11.1 Human Protocols

Observing and interacting with a (non-mathematician) human subject while the subject is solving a GCD problem is a natural way to extract a commonsense algorithm. In doing this, it is important to remember that we are not after a simulation of the human subject's behavior, but to get at the commonsense techniques applied to the problem. A series of these interactions, which I call "human protocols", was performed using simple GCD problems (i.e., GCD problems that used only one or two-digit arguments).

A typical interaction in such protocols is as follows. **JU** has a college level understanding of mathematics and computer science, but has been out of school for five years. I am **AG**. **JU** is looking at a paper with the numbers 12 and 16 written on it:

AG: Do you know how to find the greatest common divisor of two numbers?

JU: I think I used to, but I don't remember what divisors are.

AG: A "divisor" of a number x is a number you can divide x by evenly.

JU: What do you mean by "evenly"?

AG: I mean there is no remainder when you divide.

JU: OK.

AG: So what is the greatest common divisor of 12 and 16?

JU: 4.

AG: How did you get that answer?

JU: I just found the biggest divisor for both.

AG: Write down the divisors of both numbers.

JU: [Writes 1, 2, 3, 4, 6 next to 12 and 1, 2, 4, 8 next to 16]

AG: Now write the divisors common to both numbers.

JU: [Circles 1, 2, and 4] and 4 is the biggest.

As is evident in this protocol (and typical for many human protocols), the human subject must sometimes be prodded into being exhaustive (often being told to write something down or asked to explain their answer). This underlies the importance of acting and asking "why" questions during an explanation. Also, it is clear that human subjects like **JU** take shortcuts. For example, **JU** did not write out 12 and 16 as divisors. This is likely due to the fact that 12 and 16 were already present on the page (i.e., perceptually available to **JU** and part of **JU**'s extended cognition). Other subjects would skip over writing 1 as a divisor when it was clear that the arguments were relatively prime.

The initial interaction consists of a clarifying dialogue about divisors in which **JU** "builds up" a definition of "divisor" from simpler concepts. In trying to pin down what I mean by the term, **JU** is forced to deal with a new term "evenly". This is clarified as meaning "with no remainder". Currently, Cassie is implemented in such a way so that she has sufficient background knowledge to determine a term's definition without a clarifying dialogue. However it is possible that she could computationally detect when one of her definitions is insufficient or does not match up with the interrogator's usage of the term.

JU is able to "zero-in" on a meaning for GCD and a quick "back-of-the-envelope" solution procedure. Based on a series of similar protocols with different subjects, I have found that most people quickly notice that a greatest common divisor is a greatest common *something*, and only

need clarification or a reminder for what divisors are. I believe that this is due to the high frequency of using “greater” and “common” in everyday natural-language situations as compared with the more technical term “divisor”. Thus, subjects are able to quickly import natural-language meanings into the task (e.g., greatest as “biggest” and common as “shared feature”) and to integrate these meanings with the technical meaning of “divisor”. Moreover, natural language seems to steer the solution procedure: Find the divisors, find the common divisors, find the greatest of the common divisors. This suggests that a commonsense algorithm for GCD should be guided, in part, by natural-language.

4.11.2 A Commonsense Natural-Language GCD Algorithm

As exemplified by the human protocols in the previous section, when asked to find the GCD of two natural numbers x and y , most non-mathematicians who learned GCD at some point, I have found, come up with a “back-of-the-envelope” algorithm similar to the following:

1. List the divisors of x
2. List the divisors of y
3. List the divisors common to the two lists above
4. Circle the greatest number from the list of common divisors

This “common-sense” algorithm isolates the semantics of “greatest” and “common” (terms which occur frequently in natural language) from divisor-finding. I would speculate that most non-mathematicians would respond with a similar algorithm (although I have not subjected this hypothesis to a formal trial), and it would be more surprising to see a non-expert using a recursive version of the Euclidean GCD algorithm:

```
gcd(x,y)
  if y=0 return x
  else return gcd(y, x mod y)
  end if
```

This more efficient and optimized algorithm lacks the intuitive semantics of the common-sense algorithm.

I will call the the commonsense algorithm a “natural language” GCD algorithm (or NLGCD), because it relies on the intuitive NL semantics of the words “greatest” and “common”. By performing such an algorithm alongside the semantic-arithmetic routines, Cassie will be in a position to perform an exhaustive explanation. In the next section, I consider what an idealized version of such an explanation might look like for the concept of GCD.

4.11.3 An Idealized Dialogue

To probe the understanding of a cognitive agent who believes that the GCD of 8 and 6 is 2, we can imagine the following idealized question and answer dialogue unfolding:

Q1: Why is 2 the greatest common divisor of 8 and 6?

A1: 2 is the greatest of the common divisors of 8 and 6.
 Q2: Why is 2 a common divisor of 8 and 6?
 A2: 2 is a divisor of 8 and 2 is a divisor of 6.
 Q3: Why is 2 a divisor of 6?
 A3: There is a number that, when multiplied by 2, gives 6, and that number is 3.
 Q4: Why is 2 times 3 = 6?
 A4: Multiplication is repeated addition: 2 plus 2 is 4; 4 plus 2 is 6
 Q5: Why is 2 plus 2 = 4?
 A5: When I count from 2 for two numbers I end up at 4.
 Q6: How do you know that you will end up at 4?
 A6: I counted two groups of apples, with 2 apples in each, ending up with 4 total apples.
 Q7: What is 2?
 A7: It is a number and the greatest common divisor of 8 and 6.
 Q8: What is a number?
 A8: Some examples are 2,4,6 and 8... It is something that can be counted, added, multiplied... and something that can be the result of finding a greatest common divisor

Questions Q1 through Q5 are all why questions. The idealized agent procedurally decomposes each result until it arrives at certain cognitive axioms. Once it has reached these axioms (e.g., Q6), it must appeal to embodied experience. This sort of appeal will be described in the next chapter. Questions Q7 and Q8 are conceptual definition questions.

This idealization is often not reproduced by humans. I have found that most people will stop somewhere along the line of explaining a complex act in terms of simpler acts and give “It just is” answers (especially when asked things like “Why is 2 times 3 = 6?”). Nevertheless, I think such a dialogue is what we should strive for from a computational agent because at each point a result is justified in terms of simpler, antecedently understood procedures. Humans tend to pack away the procedural knowledge after an arithmetic operation has been understood at a higher level of abstraction. For example, once the multicolumn addition is learned, it seems unnatural to cite counting as a method of addition (see Sfard (1991) for a discussion of this sort of reification).

The dialogue shows the different sources of meaning used in a mathematical justification. Natural language semantics can be used to address Q1 and Q2. Procedural decomposition is used for Q3, Q4, and Q5. An empirical embodied activity is cited for Q6. Finally, conceptual-role semantics are applied to answer Q7 and Q8 (for a further discussion of these semantic sources, see (Goldfain, 2006)).

Again, the point here is not to produce a faithful rendition of an actual dialogue with a particular human. The point is to probe an agent to answer questions in such a way that the answers *if they had occurred in a dialogue* would leave no skeptic unconvinced that the agent understands GCD.

4.11.4 Implementation

NLGCD is implemented in SNePS using three complex acts and a helper act `UpdateDivisorCandidate`. This act is responsible for moving the agent’s attention from one potential divisor to its successor.

```

all(d)(DivisorCandidate(d) =>
  ActPlan(UpdateDivisorCandidate(d),
    withsome(?dpl,
      Successor(?dpl,d),
      snsequence(disbelieve(DivisorCandidate(d)),
        believe(DivisorCandidate(?dpl))),
      SayLine("I don't know the successor")))).

```

CreateDivisorList is Cassie's way of listing the divisors of each of the arguments. This includes a plan which establishes that every number is a divisor of itself (when performed, this act replicates the behavior of **JU** above) and a plan for determining the divisors by successively dividing all other numbers up to the given argument by the candidate number (represented using the DivisorCandidate case frame). For exhaustive explanation, all such routines are decomposed into semantic routines. Divisors found are memorized via the DivisorOf case frame.

```

all(nx)({Dividend(nx),DivisorCandidate(nx)} &=>
  ActPlan(CreateDivisorList(nx),
    snsequence(believe(DivisorOf(nx,nx)),
      snsequence(SayLine("Done with divisor list"),
        snsequence(disbelieve(DivisorCandidate(nx)),
          disbelieve(Dividend(nx)))))).

all(nx,d)({Dividend(nx),DivisorCandidate(d)} &=>
  ActPlan(CreateDivisorList(nx),
    snsequence(Divide(nx,d),
      withsome(?q,
        (Number(?q) and Evaluation(Result(Quotient,nx,d),?q)),
        snsequence(Say(d),
          snsequence(SayLine(" is a divisor "),
            snsequence(believe(DivisorOf(d,nx)),
              snsequence(UpdateDivisorCandidate(d),
                CreateDivisorList(nx))))),
          snsequence(Say(d),
            snsequence(SayLine(" is not a divisor"),
              snsequence(UpdateDivisorCandidate(d),
                CreateDivisorList(nx)))))).

```

The list of common divisors is easily ascertained using a conjunctive condition for the withall act (i.e., all DivisorOf numbers that appear on both lists are common divisors). This yields the act CreateCommonList:

```

all(nx,ny)({Number(nx),Number(ny)} &=>
  ActPlan(CreateCommonList(nx,ny),
    withall(?div,
      (DivisorOf(?div,nx) and DivisorOf(?div,ny)),
      believe(CommonDivisorOf(?div,nx,ny)),
      SayLine("problems creating common list")))).

```

The act FindGreatestCommonDivisor iterates through the common divisors and successively compares each to the one currently believed to be the greatest. The value is initially set to one, which is the GCD for relatively prime arguments:

```

all(nx,ny)({Number(nx),Number(ny)} &=>
  ActPlan(FindGreatestCommonDivisor(nx,ny),
    snsequence(SayLine("Now finding GCD"),
      snsequence(believe(Evaluation(Result(GCD,nx,ny),n1)),
        withall(?div,
          CommonDivisorOf(?div,nx,ny),
          withsome(?g,
            Evaluation(Result(GCD,nx,ny),?g),
            snif({if(GreaterThan(?div,?g),
              snsequence(disbelieve(Evaluation(Result(GCD,nx,ny),?g)),

```

```

        believe(Evaluation(Result(GCD,nx,ny),?div))),
        else(believe(Evaluation(Result(GCD,nx,ny),?div))))},
        SayLine("I can't find the GCD")),
        SayLine("GCD found")))).

```

Finally, the top-level complex act for this algorithm is NLGCD:

```

;;;ActPlan for NLGCD
all(nx,ny,z)({Number(nx),Number(ny),Zero(z)} &=>
  ActPlan(NLGCD(nx,ny),
    withsome(?one,
      Successor(?one,z),
      snsequence(believe(DivisorCandidate(?one)),
        snsequence(believe(Dividend(nx)),
          snsequence(CreateDivisorList(nx),
            snsequence(believe(DivisorCandidate(?one)),
              snsequence(believe(Dividend(ny)),
                snsequence(CreateDivisorList(ny),
                  snsequence(CreateCommonList(nx,ny),
                    FindGreatestCommonDivisor(nx,ny))))))))),
      SayLine("Problem with NLGCD.")))).

```

4.11.5 Cassie's Exhaustive Explanation of GCD

The following is an annotated demonstration of Cassie performing the NLGCD algorithm with inputs n8 and n6 and exhaustively explaining herself afterward. The justification is not identical to the idealized dialogue presented above, but is based on the actions Cassie actually performed during the algorithm.

```
: perform NLGCD(n8,n6)
```

First Cassie builds a divisor list for 8:

```

Subtract Dividing n8 n1
Count Subtracting n8 n1
Count Subtracting n7 n1
Count Subtracting n6 n1
Count Subtracting n5 n1
Count Subtracting n4 n1
Count Subtracting n3 n1
Count Subtracting n2 n1
Count Subtracting n1 n1
n1 is a divisor
Subtract Dividing n8 n2
Count Subtracting n8 n2
Count Subtracting n6 n2
Count Subtracting n4 n2
Count Subtracting n2 n2
n2 is a divisor
Subtract Dividing n8 n3
Count Subtracting n8 n3
Count Subtracting n5 n3
I cannot subtract a larger number from a smaller number n2 n3.
n3 is not a divisor
Subtract Dividing n8 n4
Count Subtracting n8 n4
Count Subtracting n4 n4
n4 is a divisor
Subtract Dividing n8 n5
Count Subtracting n8 n5
I cannot subtract a larger number from a smaller number n3 n5.

```

```

n5 is not a divisor
Subtract Dividing n8 n6
Count Subtracting n8 n6
I cannot subtract a larger number from a smaller number n2 n6.
n6 is not a divisor
Subtract Dividing n8 n7
Count Subtracting n8 n7
I cannot subtract a larger number from a smaller number n1 n7.
n7 is not a divisor
Done with divisor list

```

Now she computes the divisor list for 6 and computes the GCD:

```

Subtract Dividing n6 n1
Count Subtracting n6 n1
Count Subtracting n5 n1
Count Subtracting n4 n1
Count Subtracting n3 n1
Count Subtracting n2 n1
Count Subtracting n1 n1
n1 is a divisor
Subtract Dividing n6 n2
Count Subtracting n6 n2
Count Subtracting n4 n2
Count Subtracting n2 n2
n2 is a divisor
Subtract Dividing n6 n3
Count Subtracting n6 n3
Count Subtracting n3 n3
n3 is a divisor
Subtract Dividing n6 n4
Count Subtracting n6 n4
I cannot subtract a larger number from a smaller number n2 n4.
n4 is not a divisor
Subtract Dividing n6 n5
Count Subtracting n6 n5
I cannot subtract a larger number from a smaller number n1 n5.
n5 is not a divisor
Done with divisor list
Now finding GCD

```

Now we can ask Cassie some questions. The grammar only supports arithmetic acts, but GCD is placed in an Evaluation frame like all of the arithmetic results:

```

: Evaluation(Result(GCD,n8,n6),?x)?

wff4602!: Evaluation(Result(GCD,n8,n6),n2)

CPU time : 0.40

```

Now we ask Cassie to what the common divisors of 8 and 6 are:

```

: CommonDivisorOf(?x,n8,n6)?

wff4587!: CommonDivisorOf(n1,n8,n6)
wff4585!: CommonDivisorOf(n2,n8,n6)

CPU time : 0.38

```

We can also ask what the divisors of 8 and what the divisors of 6 are as two questions:

```

: DivisorOf(?x,n8)?

wff2343!: DivisorOf(n4,n8)
wff1340!: DivisorOf(n2,n8)

```

```
wff215!: DivisorOf(n8,n8)
wff201!: DivisorOf(n1,n8)
```

CPU time : 0.40

```
: DivisorOf(?x,n6)?
```

```
wff4098!: DivisorOf(n3,n6)
wff4041!: DivisorOf(n2,n6)
wff3980!: DivisorOf(n6,n6)
wff3966!: DivisorOf(n1,n6)
```

CPU time : 0.41

Now we exercise the natural-language interface and force Cassie to do some procedural decomposition:

```
: perform answerQuestion("why is 8 / 2 = 4?")
```

```
Parsing <QUESTION>
Parsing <WHY>
Parsing <EVAL>
Parsing <RES>
Parsing <ARG1>
Parsing <OP>
Parsing <ARG2>
Parsing =
Parsing <NUMBER>
```

```
Question in SNePSLOG:perform Explain(Evaluation(Result(Quotient,n8,n2),n4))
```

I know that the result is correct

Here's HOW I can find the result

```
wff1338: Divide(n8,n2)
```

And I can do this by performing

```
wff1358: SubtractDivide(n8,n2)
```

Which has the effect(s) of wff1364!: Evaluated(Result(DifferenceQuotient,n8,n2))

CPU time : 10.28

```
: perform answerQuestion("why is 8 - 4 = 4?")
```

```
Parsing <QUESTION>
Parsing <WHY>
Parsing <EVAL>
Parsing <RES>
Parsing <ARG1>
Parsing <OP>
Parsing <ARG2>
Parsing =
Parsing <NUMBER>
```

```
Question in SNePSLOG:perform Explain(Evaluation(Result(Difference,n8,n4),n4))
```

I know that the result is correct

Here's HOW I can find the result

```
wff2455: Subtract(n8,n4)
```

And I can do this by performing

```
wff2510: CountSubtract(n8,n4)
```

Which has the effect(s) of wff2516!: Evaluated(Result(CountDifference,n8,n4))

CPU time : 10.74

```
: perform answerQuestion("why is 4 - 4 = 0?")
```

```
Parsing <QUESTION>
Parsing <WHY>
Parsing <EVAL>
Parsing <RES>
Parsing <ARG1>
Parsing <OP>
Parsing <ARG2>
Parsing =
Parsing <NUMBER>
```

```
Question in SNePSLOG:perform Explain(Evaluation(Result(Difference,n4,n4),n0))
```

```
I know that the result is correct
Here's HOW I can find the result
  wff2396: Subtract(n4,n4)
And I can do this by performing
  wff2619: CountSubtract(n4,n4)
Which has the effect(s) of wff2625!: Evaluated(Result(CountDifference,n4,n4))
```

```
CPU time : 10.57
```

Finally, we ask Cassie some conceptual definition questions:

```
: perform answerQuestion("define 2.")
```

```
Parsing <QUESTION>
Parsing <DEFINE>
```

```
Question in SNePSLOG:perform ssequence(believe(ConceptToDefine(2)),Define(2))
```

```
2 is a
(Numeral)
```

```
CPU time : 7.73
```

```
: perform answerQuestion("define n2.")
```

```
Parsing <QUESTION>
Parsing <DEFINE>
```

```
Question in SNePSLOG:perform ssequence(believe(ConceptToDefine(n2)),Define(n2))
```

```
n2 is a
(Number)
```

```
CPU time : 8.12
```

```
: perform answerQuestion("define CountDifference.")
```

```
Parsing <QUESTION>
Parsing <DEFINE>
```

```
Question in SNePSLOG:perform ssequence(believe(ConceptToDefine(CountDifference)),Define(CountDifference))
```

```
CountDifference is a
(ResultName)
```

```
CPU time : 7.85
```



```
: perform answerQuestion("define Subtract.")

Parsing <QUESTION>
Parsing <DEFINE>

Question in SNePSLOG:perform snsequence(believe(ConceptToDefine(Subtract)),Define(Subtract))

Subtract can be performed via the following acts
(CountSubtract)

CPU time : 7.75

: perform answerQuestion("define Number.")

Parsing <QUESTION>
Parsing <DEFINE>

Question in SNePSLOG:perform snsequence(believe(ConceptToDefine(Number)),Define(Number))

Number has the following class membership
(n10 n9 n8 n7 n6 n5 n4 n3 n2 n1 n0)
Number is an argument for the following acts
(CountFromFor CountAdd Add CountDownFromFor CountSubtract Subtract AddMultiply Multiply SubtractDivide Divide)

CPU time : 7.86

:
```

Chapter 5

Embodied External Arithmetic

In the previous chapter, I examined the depth of Cassie’s understanding by focusing on the GCD problem in particular. In the current chapter, my aim is to demonstrate my theory’s breadth of applicability. It would be a weak theory indeed if Cassie’s representations and routines were only applicable to understanding GCD. Importantly, the KL representations developed in the previous chapter are redeployed for embodied arithmetic. This demonstrates that the abstracted and the embodied components of arithmetic can be integrated in a single agent.

Understanding in early mathematical cognition hinges (in part) on the embodied actions and perceptions of an agent situated in the world. A consequence of this situatedness is that an agent will have a particular perspective into the world and will act on objects and perceive phenomena unique to its “corner of reality”. Reality provides different agents with vastly different experiences. Yet the applicability and power of mathematics is precisely its ability to unify a large set of experiences under a common description. Somehow, with a different set of “sandbox” experiences, I was able to abstract the same early mathematics as children in sandboxes halfway around the world. The embodied component of a theory of mathematical cognition should explain why this is the case.

In this chapter, I address the embodied external arithmetic component of my theory. I am using the term *external* in two senses. First, some acts described in this chapter require the agent to represent and collect information from beyond its knowledge layer. As such, information must flow from Cassie’s “body” (i.e., the lower layers of GLAIR) up to her mind (i.e., the knowledge layer). Second, some acts described in this chapter are performed for general-purpose problem solving. This is, in some sense, “external” to (pure) mathematics and “externally” applicable. As such, these acts are contrasted with acts like finding the GCD, which is usually done as part of a larger mathematical task.

In § 5.1, I will describe how the representations developed in the previous chapter can be extended to account for quantities, the fundamental representation Cassie uses for embodied arithmetic. In § 5.2, I describe how Cassie performs the task of enumerating external objects. In § 5.3, I discuss a suite of general applications for an agent that can make number assignments. Finally, in §5.4, I describe a model of count-addition strategy change.

5.1 Quantities

The basic building block of embodied arithmetic is the quantity. The term quantity is ambiguous. Quantity is sometimes used as a synonym for number. In this sense, a quantity answers questions of the form “How many X ?”, where X is a unit. This is not how I will use the term. For embodied arithmetic, it will be important to represent the relevant unit X *as part of* the notion of quantity. I will characterize the unit X as a *sortal* (a technical term that will be explained in §5.1.4). For now, it will suffice to take sortals to be countable things like “apples”, “inches”, and “cups of water”. I will use the term ‘quantity’ to refer to a complex that includes both a number, a sortal, and a relationship that binds the number to the sortal. I also take the term ‘quantity’ to be a ‘multitude’ of *discrete* entities rather than a ‘magnitude’, which is usually taken to be a continuous extent. The magnitude/multitude distinction is clear in the English expression of quantity, where it is manifested as the count-noun/mass-noun distinction (see below).

The embodied activities of object collection, object construction, and measurement involve reasoning about quantities in this sense. At the very minimum, any representation for quantities so taken must account for the following:

1. Representing the number.
2. Representing the sortal.
3. A mechanism for binding the number and sortal, sometimes called a *number assignment* (Wiese, 2003).

These three requirements present a representational challenge beyond the representation of just a number or sortal. Embodied activities demand an additional requirement:

4. A mechanism for associating a quantity with perceptual symbols derived from sensory input.

The symbolic representation of a quantity should be somehow grounded in perception or somehow visualized in terms of previous episodes of perception.

5.1.1 Representations of Quantity

Since quantities can appear in natural-language (NL) utterances, they should be somehow handled in NL processing systems and any knowledge-representation and reasoning systems that attempt to represent NL. In this section, several formalisms for representing quantities are reviewed to determine how they might inform a computational representation for embodied arithmetic.

5.1.1.1 Category Systems

Quantity in philosophy (specifically in metaphysics) has been treated as a part of category systems.¹ Category systems are sometimes also used as ontologies, i.e., as theories of what exists

¹See (Thomasson, 2004) for a good survey of category systems; the category systems discussed in this section are drawn from this article.

<i>Categories</i>	
Substance	Quantity
Quality	Relation
Place	Date
Posture	State
Action	Passion

Table 5.1: Aristotelian Categories.

and how existing things are related. To be fair, these systems were not intended to be applied to computational representations, but they do indicate some of the issues involved in characterizing quantity.

Aristotle considered Quantity to be one of the ten highest “uncombined” categories of being, taking each of the categories to be a constituent of reality (Aristotle, *Categories*, trans. Ackrill 1963) . These categories are given in Figure 5.1.

Thus, for Aristotle, quantities are simple unanalyzable constituents of reality. Furthermore, such a categorization points to what quantities are *not*, namely, they are not things like relations, substances, or places. Although Aristotle’s ontological commitment is crystal clear, this categorization is not particularly useful for representing finer-grained distinctions about quantities.

Kant made Quantity a basic conceptual category, with sub-categories Unity, Plurality, and Totality (Kant, 1787/1958). His categories are given in Figure 5.2.

<i>Category</i>	<i>Subcategory</i>
Quantity	Unity
	Plurality
	Totality
Quality	Reality
	Negation
	Limitation
Relation	Inherence and Subsistence
	Causality and Dependence
	Community (reciprocity)
Modality	Possibility
	Existence
	Necessity

Table 5.2: Kantian Categories.

Husserl (1913/2000) included “quantity-like” categories in his object category system distributed across the three categories of Unit, Plurality and Number. His categories are given in Figure 5.3.

Husserl also developed a framework for categories of meaning that would correlate with the object categories, but there is no clear correlate for quantity-like entities in this correlated category

<i>object categories</i>
object
state of affairs
unit
plurality
number
relation

Table 5.3: Husserl’s Object Category-System

system.

In a modern category system, such as the one given by Hoffman and Rosenkrantz (1994), the category of Quantity must be built up from unfamiliar pieces. Their categories are given in Figure 5.4.

<i>Category</i>	<i>Subcategory</i>	<i>Sub-subcategory</i>
Entity	Abstract	Property Relation Proposition
	Concrete	Event Time Place Substance Limit Collection Privation Trope
		Material Object Spirit

Table 5.4: Category System of Hoffman and Rosenkrantz.

In such a system, it is hard to identify quantity-like entities (e.g., what has happened to the category of “Number”?), but at the same time, there seems to be replacement categories that might work (e.g., Proposition, Collection). Quantity, in the way we need to represent it for embodied arithmetic, does not occupy a single area in such a system, because the representation we are after has components that are both abstract and concrete.

From these examples, we see that the category Quantity has had a volatile journey in metaphysics. It has gone from being a basic constituent of reality in Aristotle’s realism, to being a basic but divisible subjective phenomenon in Kant’s conceptualism, to being a non-basic category distributed (or “built-up”) from other categories in Husserl’s descriptivism, to becoming utterly lost in the categories of a modern ontology. This is not a criticism against any of the category systems, but the overall trend indicates that some explanation of quantities in category systems is required, especially since many computational systems rely on ontologies built up from such categorizations.

5.1.1.2 First-Order Logic

Naïvely, we might decide to treat quantities in logic as numbers predicated of sortals. The sentences: (1) “There are red apples on the table” and (2) “There are three apples on the table” share the same linguistic form, so the naïve approach would be to represent the color term and the numeric term in the same way. For (1), we would expect to see a representation such as:

$$\exists x(\text{CollectionOf}(x, \text{Apple}) \wedge \forall y(\text{MemberOf}(y, x) \supset \text{Red}(y)) \wedge \exists z(\text{Table}(z) \wedge \text{On}(x, z)))$$

With the semantics of $\text{Red}(y)$ being something like “Red” is a property of (object) y . If “red” and “three” work in the same way, we might expect the following representation for (2):

$$\exists x(\text{CollectionOf}(x, \text{Apple}) \wedge \forall y(\text{MemberOf}(y, x) \supset \text{Three}(y)) \wedge \exists z(\text{Table}(z) \wedge \text{On}(x, z)))$$

The representation $\text{Red}(y)$ makes sense because “redness” (if it is treated as a property at all) is an intrinsic property. The property of “being red” belongs to each apple in the collection. However, “threeness” cannot be an intrinsic property of any apple in the collection, and it would be quite difficult to specify the semantics of $\text{Three}(y)$ in (2).

There is also a method for representing sentences such as “There are (exactly) three apples” in first-order logic with equality and the usual connectives ($\wedge, \vee, \neg, \supset$):

$$\exists x_1 \exists x_2 \exists x_3 (\text{Apple}(x_1) \wedge \text{Apple}(x_2) \wedge \text{Apple}(x_3) \wedge \forall y (\text{Apple}(y) \supset (y = x_1 \vee y = x_2 \vee y = x_3)))$$

Such a schema is an elegant logical convention. It tells us how to produce an appropriate FOL formulation for sentences containing exact quantities. However, this leads to several problems when used as a cognitive representation. This representation introduces a term for each individual. The implausibility of this as a cognitive thesis for a quantity representation is evident from several points of view. (1) It would appear harder for a computational cognitive agent to conceive of sentences containing “100 birds” than those containing “3 birds”, the former representation requiring 100 objects of thought and the latter taking only three; yet, are we really straining to think of 100 distinct birds when someone utters sentences containing “100 birds”? (2) It is not even clear that we *can* entertain 100 objects of thought. Psychological evidence has pointed to two mental number systems, one for the small, precise, subitizable cardinalities, and one for large, approximate cardinalities (Feigenson, Dehaene, and Spelke, 2004). We can conceive of sentences containing “100 senators” and proceed to reason with our conception without being able to name all (or any) of the senators. We can be sure about the exactness of a quantity without having a mental concept (or logical term) for each senator.² These logical representations do not tell us how a quantity should combine syntactically to meet our representational requirements.

Logicians and linguists have noted some of these issues and have proposed a generalization of the traditional quantifiers \forall and \exists . This has led to a class of such generalizations called *generalized quantifiers* (Westerstahl, 2005). Relevant among these are numerical quantifiers. SNePS actually

²Stuart Shapiro suggested this useful example to me.

provides a numerical quantifier `nexists` with the following syntax and semantics (Shapiro and SNePS Implementation Group, 2008):

`nexists(i, j, k)(x)({P1(x), ..., Pn(x)}: {Q(x)})` means there are k individuals that satisfy $P1(x) \wedge \dots \wedge Pn(x)$ and, of them, at least i and at most j also satisfy $Q(x)$.

Despite the utility of such a representation, it treats i , j , and k as natively supported constructs (rather than as numerons generated during counting).

5.1.1.3 Natural Language

NL provides several clues as to how human cognition represents quantities. From the perspective of computational systems, NL utterances are raw, pre-representational cognitive content, often providing better insight into human cognitive organization than the formal, polished, logical representations of these utterances.

Several NLs (including English) make a distinction between count-nouns (e.g., ‘apple’) and mass nouns (e.g., ‘water’). This distinction is marked by a syntactic restriction on how such nouns can be used in quantities. E.g., “three apples” is grammatical; “three waters” is not: One must say things like “three cups of water”. X is a count noun in questions of the form “How many X ?” and a mass noun in questions of the form “How much X ?”. Such a feature of language invites us to participate in the activity of counting by organizing the world into the countable and the uncountable.

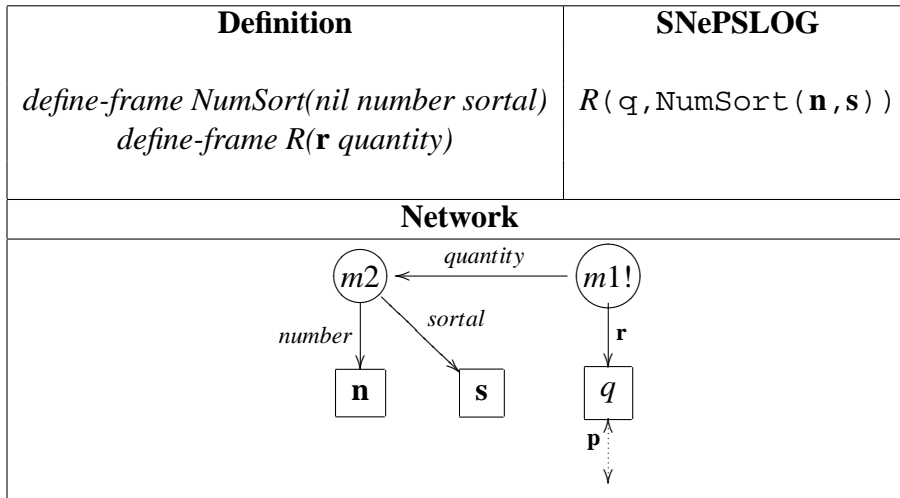
An interesting (and surprisingly universal) feature of how count nouns can participate in quantities is that quantity is never systematically marked by closed-class forms (i.e., quantity is not lexicalized). Languages may have a form for singular (apple = 1 apple), plural (apple+s = multiple apples), and, in some languages, paucal (a few apples), but never a form such as “applethree” to mean “three apples” (Talmy, 2000). Cognitively primitive structures are encoded in closed-class forms (such as affixes and adpositions) and not in open-class forms (such as nouns, verbs, and adjectives) (Talmy, 2000). Some closed-class constructions may be used for concepts up to a small quantity: the terms unicycle (1 wheel), bicycle (2 wheels), and tricycle (3 wheels) lexicalize the number of wheels in the vehicle. Yet there is no systematic construction for all quantities, and such exceptions usually terminate at a very small cardinality. Even though the world provides objects like 16-wheeler trucks, they are not usually referred to by terms like “hexidecicycle”. As with modern category systems, the open-class construction of quantities in numeral systems (1,2,3,...) and number-name systems (“one”, “two”, “three”, ...) points to the conclusion that quantities are not a primitive concept.

The distinction of unity and plurality is fundamental to human cognition. This is encoded in the singular/plural distinction in most of the world’s languages. It also found its way into the *Elements* of Euclid. Euclid’s ontology was such that 1 was a *unit* and 2,3,4,... were *numbers*. Both the unit and the numbers were geometrically conceived as distances, making number theory a theory of line segments in Book VII of the *Elements* (trans. Heath, 2002)

5.1.2 Quantities for Embodied Arithmetic

To satisfy our four representation requirements (see §5.1), and to utilize the insights from the previous section, we need a grounded representation for quantity that works in service of embodied acting. If the semantics for arithmetic really does arise from embodied activities, we need to consider how a quantity will be represented by an embodied agent that is situated in the real world and capable of perception.

The four representational requirements suggest a formalization of quantity as a four-tuple. If n is a number, s is a sortal, r is a particular two-place relation that holds between n and s , and p is a mechanism for perceiving or imagining the relationship given by $r(n, s)$, then $\langle n, s, r, p \rangle$ is a quantity. The general form of the SNePS representation of a quantity $q = \langle n, s, r, p \rangle$ is as follows:



The general semantics for this formalism is as follows: $\llbracket m1 \rrbracket$ is the proposition that $\llbracket q \rrbracket$ is a quantity of type $\llbracket r \rrbracket$ of $\llbracket n \rrbracket$ $\llbracket s \rrbracket$ s grounded by relation $\llbracket p \rrbracket$. In the following sections, I will discuss the role each of the components n, s, r , and p have to play and the contribution each makes towards the semantics of quantities.

5.1.3 Natural Numbers in Quantities (n)

As we have seen in the previous chapter, an effective counting routine is just a method for generating a progression of unique symbols. The complete semantics of these symbols is given simply by their occurrence in a certain place in the progression. For the purposes of communication, these symbols can then be systematically associated with a set of numerals or a set of number-names. The numeral and number-name systems are both guided by a grammar for generating new numerals and number-names. Importantly, once the association with numerals or number-names has been made, an agent can identify a number without executing the counting routine. Both of these symbolic systems are equipped with a syntax and semantics. Syntactically, 132 specifies a number, because the digits ‘1’, ‘3’, and ‘2’ are numerals (which are associated with numbers in a counting routine). Semantically, 132 is “1 hundred plus 3 tens plus 2”. Thus, a computational agent can

conceive of a number without needing to symbolize all of its predecessors in the counting routine. This avoids the “100 senators” problem we saw earlier.

The primacy of the counting routine is somewhat challenged by the (embodied) phenomenon of subitization (the automatic and arguably innate recognition of cardinality) exhibited by human infants and several non-human primates. It has been argued that subitization is simply an organizing principle used to assist counting routines (Gelman and Gallistel, 1978). Adopting this position, we can simply assume an agent capable of performing a counting routine for our present purposes. I will have more to say about subitization in §5.2.1.

5.1.4 Sortals (*s*)

I have imported the technical term *sortal* to refer to the discrete entities a quantity is taken to be a number of. Sortals have various definitions in philosophy:

A sortal: (1) gives a criterion for counting the items of that kind, (2) gives a criterion of identity and non-identity among items of that kind, (3) gives a criterion for the continued existence of that kind, (4) answers the question ‘what is it?’ for things of that kind, (5) specifies the essence of things of that kind, (6) does not apply to parts of things of that kind (Grandy, 2006)

These are separate consistent definitions for the term ‘sortal’, and they each capture a relevant useful feature for a task such as embodied enumeration.

I will add the following computationally relevant definition to the list (not claiming to capture all of the nuances of the six given definitions):

- *S* is a *sortal predicate* (or *sortal*) for a cognitive agent *A* and object *x* if, whenever *A* believes $S(x)$, *A* has a decidable procedure *P* for generating a non-empty set of distinguishing properties of *x*.
- *S* is a *count sortal predicate* (or *count sortal*) for cognitive agent *A* and object *x* if *S* is a sortal predicate for *A* and *A* has a decidable procedure *P'* for counting (or measuring) *x* in some sensory modality *M*.

This makes the discrimination of sortals an agent-relative ability and intimately ties this ability to the agent’s particular embodiment, sensory capacities, and beliefs.

As an example, `Apple` is a sortal for Cassie if she can infer from her KB the properties that distinguish apples from anything else, including, shape, color, and size. `Apple` is a count sortal for Cassie if, in addition to being a sortal for her, she is embodied and can use the set of inferred properties to distinguish and count apples in the actual world through some sensory modality (e.g., through a vision system).

Because sortals are agent relative, there will often be disagreement between agents as to the applicability of a sortal. Sortals such as “game” and “chair” may have precise application conditions for a particular agent even though there are no publicly agreed upon necessary and sufficient conditions for the application of the sortal.

Xu (2007) has investigated the use of sortals in cognitive science. She makes a more fine-grained distinction between distinguishing properties, breaking these down into *object individuation*, which enables an agent to answer the question “how many?”, and *object identification*, which enables an agent to answer the question “which one?”. Object identity, so characterized, is distinct from object identity through time.

If an object seen at a different time is labeled with a different count noun, ‘Look, a dog!’, a mental symbol is then created to represent the sortal concept DOG. These sortal concepts provide the basic criteria for individuation and identity: an object that falls under the sortal RABBIT cannot be the same object as one that falls under the sortal DOG. In this sense, the acquisition of basic-level sortal concepts depends on acquiring basic-level count nouns (Xu, 2007).

I am not as concerned with this fine-grained distinction, but it should be noted that SNePS has been used to develop an agent capable of tracking an object’s identity over time for the task of identifying perceptually indistinguishable objects (Santore and Shapiro, 2004).

I will say a sortal is a *unit sortal* if it (implicitly or explicitly) makes reference to a known standard. Unit sortals can be metric if the known standard is a fixed unit of measurement (e.g., inch, second, pound) or topological if the known standard is not dependent on objective precision (e.g., a block on a map in “three blocks away”, a story of a building in “three stories high”). A sortal is a *collection sortal* if makes reference to a prototypical member of the class (e.g., “apples”) and that member is not a unit. Quantities, when conceived of as a number bound to a sortal, can be used to modify a (second) collection sortal: “three-lego piece”, “three-man team”, “three-egg omelet”, “three-story building”, “three-inch cut”. I will call these sortals *construction sortals*. Also, a mass noun can be quantized by collecting it in units, as in “three cups of water” or “three pounds of sugar”. I call these *quantity complexes*.

5.1.5 Relations Between Numbers and Sortals (r)

To draw out the cognitive distinction between quantities such as “three apples”, “three inches”, “three-Lego piece” and “three cups of water”, it is not enough to make the classification of unit sortals, collection sortals, construction sortals, and quantity complexes. It is also important to consider the activities in which the different kinds of sortals are deployed. Unit sortals tend to be applied during measurement activities. Collection sortals are used in object collection activities. Construction sortals are needed for the activity of constructing a unit sortal from a collection (e.g., putting three Lego pieces together to form a three-Lego piece). Quantity complexes occur in activities where mass nouns must be quantized (or, activities involving “unit-excerpting” (Talmy, 2000))

These distinctions are captured in the relation r (and the associated case-frame R). The functional term $\text{NumSort}(n, s)$ “relates” the number n and the sortal s in the SNePS “arc-label” sense of relation, but this is not enough. The relation r specifies the way in which the agent is considering the relationship between n and s to hold. I claim that the relation is properly represented as an arc-label rather than as a node (recall that this is a model of “subconscious” representation

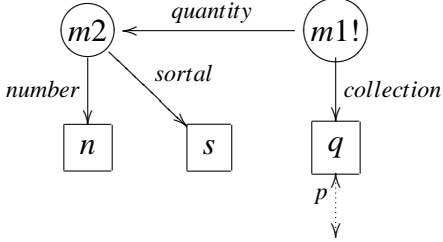
in SNePS), because I believe the subtle distinctions between sortals are not as salient in an agent's attention as the activity the quantity is being used for.

5.1.5.1 Collections

The SNePS representation for collections is as follows:

Collection

Syntax:

Definition	SNePSLOG
<i>define-frame NumSort(nil number sortal)</i> <i>define-frame Collection(nil collection quantity)</i>	$\text{Collection}(q, \text{NumSort}(n, s))$
Network	
	

Semantics: $\llbracket m1! \rrbracket$ is the proposition that $\llbracket q \rrbracket$ is a collection of $\llbracket n \rrbracket$ $\llbracket s \rrbracket$ s. The agent considers $\llbracket q \rrbracket$ to consist of $\llbracket n \rrbracket$ discrete objects that have a sufficient resemblance to a prototypical object of sort $\llbracket s \rrbracket$.

Sample Use: $\text{Name}(b, \text{'Lego'})$ and $\text{Collection}(q, \text{NumSort}(n3, b))$ asserts that q is a collection of three Legos.

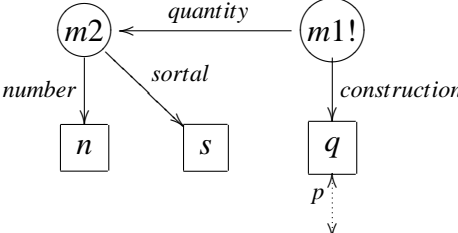
Collections are unique in that their constituents need not occur contiguously in space or time. There is also some associated notion of order-invariance with respect to the constituents of a collection (this is also a property of sets in set theory).

5.1.5.2 Constructions

The SNePS representation for constructions is as follows:

Construction

Syntax:

Definition	SNePSLOG
<i>define-frame NumSort(nil number sortal)</i> <i>define-frame Construction(nil construction quantity)</i>	Construction(<i>q</i> , NumSort(<i>n</i> , <i>s</i>))
Network	
 <pre> graph TD m2((m2)) -- quantity --> m1!((m1!)) m2 -- number --> n[n] m2 -- sortal --> s[s] m1! -- construction --> q[q] q -.-> p[...] </pre>	

Semantics: $\llbracket m1! \rrbracket$ is the proposition that $\llbracket q \rrbracket$ is a construction of $\llbracket n \rrbracket$ $\llbracket s \rrbracket$ s. The agent considers $\llbracket q \rrbracket$ to consist of a single contiguous object with $\llbracket n \rrbracket$ discrete sub-parts that have a sufficient resemblance to a prototypical object of sort $\llbracket s \rrbracket$.

Sample Use: Name(*b*, 'Lego') and Construction(*q*, NumSort(*n3*, *b*)) asserts that *q* is a construction (e.g., a block) composed of three Legos.

A quantity is viewed as a construction when the constituent members are either spatially or temporally contiguous, or there is an otherwise “tight” relationship holding between constituents that is created by the agent. An agent considering a quantity as a construction is considering the whole as the sum of its parts (i.e., analytically), or as “summing” the parts to make up that particular whole (i.e., synthetically).

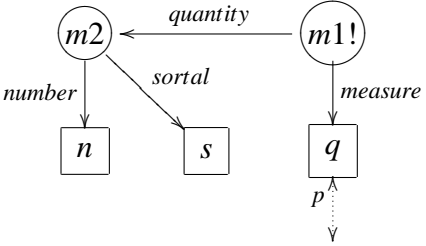
It seems quite natural to ask how a unit becomes a unit in the first place. Following Kitcher (1984), I will claim that the needs of a community of cognitive agent drives standardization. Standardization of a unit across a community allows the members of that community to measure quantities in a common, non-arbitrary way. This implies that there is some domain of target entities whose measurement would be useful to the community. This “need for a description” might be addressed in imprecise ways by particular individuals. The individual might use a collection sortal as an ad-hoc description to stand in for the measurement. For example, without dragging out the measuring tape, I might say that my door is “two-people wide”. A person is a collection sortal that may be identified by a range of “widths”. Despite the imprecision of this collection sortal description, I can still be understood by a competent speaker of English. Constructions can thus be seen as occupying an intermediate step in the standardization of unit sortals from collection sortals. The constituents of collections are not precise enough for measurements (e.g., the king’s foot). As a result, collection sortals are brought together in a precise constructions (e.g., a 12-inch ruler marked off every length of the kings foot), and finally become public standards (e.g., the reproducible 12-inch ruler).

5.1.5.3 Measures

The SNePS representation for measures is as follows:

Measure

Syntax:

Definition	SNePSLOG
<i>define-frame NumSort(nil number sortal)</i> <i>define-frame Measure(nil measure quantity)</i>	$\text{Measure}(q, \text{NumSort}(n, s))$
Network	
 <pre> graph TD m2((m2)) -- quantity --> m1!((m1!)) m2 -- number --> n[n] m2 -- sortal --> s[s] m1! -- measure --> q[q] q -.-> p[p] </pre>	

Semantics: $\llbracket m1! \rrbracket$ is the proposition that $\llbracket q \rrbracket$ measures $\llbracket n \rrbracket$ $\llbracket s \rrbracket$ s. The agent considers $\llbracket q \rrbracket$ to be a single object capable of measuring other objects in units of $\llbracket n \rrbracket$ $\llbracket s \rrbracket$.

Sample Use: $\text{Name}(b, \text{'Lego'})$ and $\text{Measure}(q, \text{NumSort}(n3, b))$ asserts that q is a three-Lego block.

Sortals used in measurements are usually considered to be standardized units. The act of measurement expresses a relationship between physical entities, specifically, a relationship between a publicly accepted standard and a target object to be measured. It is understood that such a measurement is not a complete description of the target, but rather a partial description of a particular feature of the target. This is immediately useful for a cognitive agent because it provides a common metric for comparison between things that may otherwise have very little in common. World War II and the time spent on a doctoral dissertation may both be measured by four years. The distance from the Earth to the Moon can be given in miles and so can the distance from Buffalo to Syracuse. The common unit “miles” is a signal to the agent that the numeric part of the quantities can be compared.

Moles (1977) has pointed out that acts of measurement yield meanings for abstract arithmetic

Measurement permits the investigator to deal with both magnitudes and relationships between magnitudes. In other words, we are interested in how many, how much, and how long and the relationships among them. The use of numbers per se does not aid in the process of uncovering patterns within data. The connections between phenomenal relationships and the mathematical relationships used to represent them should be displayed. Unless this is done, it is difficult, if not impossible, to know what

the mathematical manipulations mean (p. 239).

Measuring is an act that can take place in various sensory modalities and is often performed with the aid of measuring tools. Measuring tools vary in complexity and must be “calibrated” in some way, or otherwise constructed so as to guarantee standardization to the unit. For example, a foot-long ruler is usable as a measure of feet insofar as it can be guaranteed to be a foot in length. Requiring a second measuring tool for rulers invites a regress, since the second measuring tool would itself require measurement. However, in the act of measurement, a cognitive agent is usually after approximate precision. This is due not only to the limitations of constructing measuring tools, but also to the limitations of the agent in precisely “reading off” measurement quantities from a measuring device. There is a hierarchy of precision for the set of measuring tools available to a cognitive agent. The more precise tools are exactly those that most closely conform to the standardized unit.

5.1.6 Grounding Quantities in Perception (p)

In order to perform embodied acts such as collecting, constructing, and measuring with external objects, the agent needs a way to link what it is reasoning about to what it is perceiving. The relationship p between the KL symbol denoting a quantity q and the perceived or imagined representation of the quantity residing in the PML is best described in terms of the symbol-grounding problem. The symbol-grounding problem (Harnad, 1990) requires that a cognitive agent can associate all of the symbols it uses with meanings (i.e., provide its symbols a complete semantics in a non-circular way). To avoid a regression in which meaningless symbols are only given meaning in terms of further meaningless symbols, several approaches to the symbol-grounding problem involve grounding symbols in sensorimotor action and perception, which can be thought of as “subsymbolic”; thus, a symbol system can be grounded in a non-symbolic system. Shapiro and Ismail (2001) have presented a SNePS approach to symbol grounding called *anchoring*:

Anchoring is achieved by associating (we use the term “aligning”) a KL term with a PML structure, thereby allowing Cassie to recognize entities and perform actions, but not to discuss or reason about the low-level recognition or performance (Shapiro and Ismail, 2001)

Anchoring is similar to the approach I present below in that it establishes associations between the KL and the lower GLAIR layers.³

I don’t believe that grounding the symbolic in the non-symbolic is actually necessary. I am sympathetic with Rapaport’s (1995) theory of syntactic semantics in which a complete semantics can be theoretically given by symbol manipulation (i.e., syntax) alone and with Barsalou’s (1999) perceptual symbol systems in which all agent percepts can be considered symbolically. However, any proposed solution to the symbol grounding problem must deal with quantities in some way.

Roy (2005) has given a thorough account of how a cognitive robot might ground its language in embodied sensory experience. He posits that using grounded language is a catalyst for communication. However, the scope of the language that can be grounded is restricted:

³See also (Jackendoff, 1987) for an approach to associating linguistic and visual information.

Communication gets off the ground because multiple agents can simultaneously hold beliefs grounded in common external entities such as cups of coffee.

I take beliefs about the concrete, physical world of objects, properties, spatial relations, and events to be primary. Agents can of course entertain more abstract beliefs, but these are built upon a physically grounded foundation (Roy, 2005)

5.1.6.1 Challenges

The grounding of a quantity such as “three apples” is problematic because it relies in part on the abstraction “three”. This can also be construed as a problem of reference. I can point to an apple, I can point (or otherwise gesture) to a set of three apples, but I cannot point to three. Now, it might be pointed out that I also cannot point to apple or point to red *qua universals*, because these are also abstractions. However, I can point to *an* apple and *a* red thing as specific instances. With the case of numeric reference, it seems difficult to specify what pointing to “an instance of” three would mean without specifying the sortal the instance is three *of*. Numbers in this respect are a “double” abstraction. Russell (1918/1983) has said:

[A]ll numbers are what I call logical fictions. Numbers are classes of classes, and classes are logical fictions, so that numbers are, as it were, fictions at two removes, fictions of fictions. (p. 270)

If we replace “fictions” with “abstractions”, this claim suggests that the number component of quantity is more abstract than the sortal it binds to.

Despite this apparent difficulty, an agent can come to have many experiences *involving* “three”, none of which *is* the abstraction. These include writing and reading the numeral ‘3’, observing collections of three items, and identifying the third position in a sequence. Analogy (in Roy’s terms) or conceptual metaphor (Lakoff and Núñez, 2000) may be sufficient to generate the abstraction from such sensory experiences, but we can no longer rely on the similarity of experiences from agent to agent (in the way we might, for example, expect a similarity of experience for a physical object such as an apple). If agent A has had experiences involving three apples (but never three oranges), and agent B has had experiences involving three oranges (but never three apples), both agents should still be able to abstract the same “three” from their experiences.

Learning sortal terms (e.g., what can be counted as an apple) does require some abstraction (e.g., experiences with red and green apples), but these are always abstractions over experiences with objects that an agent can find and subsequently internalize through perception (e.g., locating and pointing to green apples).

Grounding linguistic expressions involving quantities relies on grounding the sortal term *and* grounding the number term (i.e., the abstraction). A single sensory experience may be enough to ground a sortal term, but the grounding of a number, an abstraction that can be applied in a quantity to *all* countable things, requires several sensory experiences with at least a few countable things. Otherwise, the agent cannot know that numbers can be applied to a variety of things.

Another difficulty in grounding quantities arises from the fact that a physical object can be *taken* to correspond to many quantities. Frege (1884/1974) has said:

While I am not in position, simply by thinking of it differently, to alter the colour or hardness of a thing in the slightest, I am able to think of the Illiad either as one poem, or as 24 books, or as some large number of verses (p. 28).

Again, we see that choosing the number and sortal determines the correct grounded quantity. Fortunately, our representation mechanism allows for binding multiple quantities to the same object. Frege's example can be given in SNePS by:

```
;;;The (proper) name of i is Illiad
Name(i,'Illiad').
;;;The name of (i.e., word for) p is poem
Name(p,'Poem').
;;;The name of (i.e., word for) b is book
Name(b,'Book').
;;;The name of (i.e., word for) v is verse
Name(v,'Verse').

;;;The Illiad taken as one poem
Collection(i,NumSort(n1,p)).
;;;The Illiad taken as twenty-four books
Collection(i,NumSort(n24,b)).
;;;The Illiad taken as 15693 verses
Collection(i,NumSort(n15693,v)).
```

Because of the uniqueness principle, every occurrence of *i* picks out the same node, and this node can be associated with the very same perceived (or visualized) object.

Another challenge of grounding quantities in perception arises from the very nature of perception. Perceptual information is often noisy. In just the sensory modality of vision we encounter several pragmatic obstacles: imperfect lighting, occluded objects, and distortion. Also, perceptual data is fleeting. Thus, an agent can only have limited experience with each perceptual field; in a sense, it has an attention span. This places a restriction on the PML that is not present at the KL.

Finally, there are issues with how collections should be treated over time. Embodied acts are not instantaneous. Suppose an agent is collecting apples and currently has a collection of two apples. Call this collection *C*. Suppose the agent adds three more apples to the collection. Is it still the same collection *C*? Or perhaps it is a new collection *C'*. Or perhaps, as each new apple is added, there is a new collection generated *C*₁, *C*₂, and *C*₃. Should a collection with a single constituent be represented as a collection? What if that sole constituent is removed? Should the empty collection still be represented as such? Each of these questions poses a representational issue of collection identity over time.

5.1.6.2 Grounded Quantities in SNePS

I do not claim to have a general solution to the symbol-grounding problem, or even a general solution for the grounding of quantities. However, I believe the SNePS representation addresses several of the issues raised in the previous section. To focus explicating how quantities are grounded in perception, I will not dwell on the issues of collection identity over time raised in the last section. Nevertheless, I believe such issues could be accommodated for in SNePS, because previous work has addressed acting and reasoning in time (Ismail and Shapiro, 2000), and the representational issues surrounding collections (Cho, 1992).

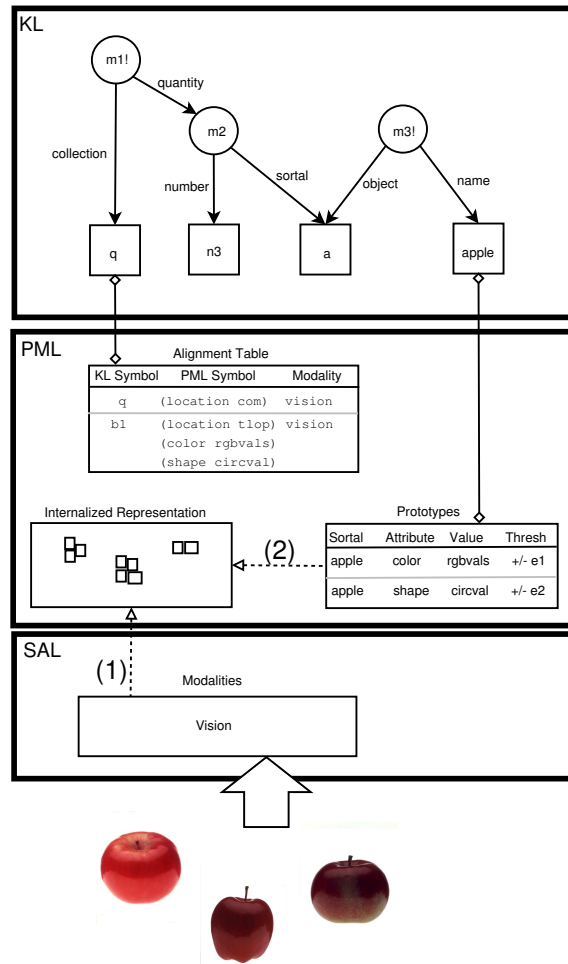


Figure 5.1: Two grounding relations for the quantity 3 apples: (1) Perceived and (2) Visualized

I will consider two ways in which p might ground a quantity: *perception* and *visualization*. An illustration of this mechanism is given in Figure 5.1. Both of these methods require Cassie to coordinate her KL symbols with her PML symbols. To do this, a data structure known as the *alignment table* is used. The alignment table associates a KL base node with a PML attribute-value pairing and a sensory modality in which such a pairing is applicable. In the example in Figure 5.1, the quantity q (“three apples”) is aligned (i.e., has an entry in the alignment table) with the PML attribute-value pair (location com) in the modality of vision. The location attribute indicates that the value com (a center-of-mass) will help to identify the quantity q as it appears in the visual field. The perceptual field for any given modality is represented by whatever data structures are most appropriate. This is an “internalized” representation of reality because it is dependent on a particular agent’s sensors (i.e., its embodiment). For the visual field, a natural data-structure would be a two-dimensional array of pixel-brightness values. A base node may have more than one entry in the alignment table, and each entry need not be limited to a single sensory modality. Also illustrated in the figure is the fact that a constituent $b1$ of the collection (not

illustrated in the KL), can occur in the alignment table. This demonstrates some different attribute-value pairs that can be useful in picking out a constituent of a collection, including the top-left object-pixel value (`tlop`) for tracking the location of an object, red-green-blue pixel brightness values (`rgbvals`) for tracking color, and a circularity value (`circval`) for tracking shape. The modality information specifies which SAL method of perceptual-data acquisition is appropriate. Using the alignment table, the perceptual data for a given base node is free to vary in the PML as new perceptual information becomes available while leaving the KL relatively stable.

Another PML mechanism is necessary for determining which objects in the perceptual field belong to which sort. This is a table of *prototypes* and, for each KL sortal, includes prototypical attribute-value pairs for perceived instances of that sort. These prototypical values may be obtained in various ways (e.g., averaging over past perceptual encounters with objects of the sort), and can be updated with each new perceptual encounter with objects of the sort. Also included with each entry is threshold error value `Thresh` that specifies the range around the prototypical value an object of that sort can be permitted to vary and still be called an object of that sort. This value is also subject to change with further experience.

More of the specifics for these mechanisms will be given in the context of embodied enumeration below. However, we are now in a position to describe the two methods of grounding a quantity. When a quantity is grounded in perception (method (1) in Figure 5.1), the alignment table associates the quantity q with an internalized representation that originated in an act of perception. I.e., the act invokes the sensory apparatus in the SAL. When a quantity is grounded in a visualization (method (2) in Figure 5.1), the alignment table associates the quantity q with an internalized representation that originated in an act of visualization. Such an act would entail the creation of prototypical objects (via the *prototype* values). In other words, the “faked” perceptual data is simply “imagined” by the agent without activating any sensors in the SAL.

Now that we have a detailed account each of the four components of quantity $q = \langle n, s, r, p \rangle$, we can deploy quantities in an embodied act.

5.2 Embodied Enumeration and Its Applications

The abstract component of Cassie’s explanations bottom out at counting routines. As we saw in the previous chapter, nothing besides primitive counting acts are needed to compute sums, differences, products, quotients, and even GCDs. There is, however, a great distinction between the act of saying (or thinking about) the number names and counting *things* with numbers. I will call the latter act *enumeration*, and when the things being counted are concrete physical objects external to the agent’s body, I will call this *embodied enumeration*. This is the backbone of the object-collection grounding-metaphor as given by Lakoff and Núñez (2000). Just as counting is a gateway to abstract arithmetic, embodied enumeration is a gateway to an embodied arithmetic, i.e., a set of acts that correspond to the four basic operations. In embodied arithmetic, addition can be viewed as object collection, subtraction as object removal, multiplication as iterated collection, and division as sharing objects among people.

In this section, I will focus on the act of embodied enumeration and describe how it can be implemented in Cassie with the help of grounded quantities.

5.2.1 Models of Enumeration

The responsibilities assigned to low-level (subconscious) and high-level (conscious) processing give rise to two different models of enumeration. An early production-system implementation of both models was given by Klahr (1973). Both models are illustrated using the case of a single die in Figure 5.2. Under the model shown in Figure 5.2 (a), low-level processing uses pattern

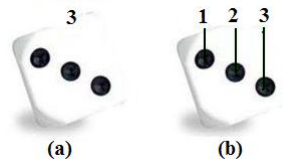


Figure 5.2: Two models of enumeration

recognition to automatically make a cardinal assignment to the number of dots⁴ on the face of the die. This ability is known as subitization in the literature (Clements, 1999) and, in human beings, is considered an innate ability for small numerosities. A low-level pattern recognition routine must be coupled with a high-level addition routine that accumulates the subitized results.

Under the model shown in Figure 5.2 (b), low-level processing uses object recognition to isolate each individual dot on the die. The agent uses a high-level routine that assigns a number to each dot as it is detected and marked in the visual field. The cardinal assignment for the set of dots is complete once the final dot has been assigned a number; this number is the cardinality for the entire set. This is the model I will use for Cassie because it will allow me to demonstrate both the high level KL-processing and the low-level PML processing. An implementation of this model requires the following from the agent:

1. A system that coordinates the low-level object-recognition routine and the high-level cardinal assignment routine.
2. The ability to mark the counted entities in the sensory (e.g., visual) field.
3. The ability to direct attention to a new entity.
4. The ability to assert of each entity counted that it is a constituent of the accumulating collection.
5. The ability to detect when all target entities in the sensory field have been assigned a cardinal number.
6. The ability to tag the entire set with a cardinal number.

⁴This sortal can be referred to by “dots” or “eyes”, even though the proper term is “pips”.

5.2.2 Implementation

Cassie is given the task of enumerating apples in her visual field. Cassie's embodiment is simulated, with the visual field consisting of three two-dimensional arrays of pixel-brightness values (one array each for red, green, and blue). I am most interested in the KL-level impact of the simulated embodiment and grounding of quantities. Thus, I have made several simplifying assumptions about the agent embodiment and environment. I limit Cassie to the modality of vision and the distinguishing properties color and shape for apples. Each of the test images contains apples displayed in a characteristic pose with no occlusion (no overlapping apples) and good lighting. The SAL layer of GLAIR is not required since the embodiment is only simulated. The agent model of attention is also very primitive. Thus, unlike human foveal vision which has a low-resolution periphery and a high-resolution center, the entire visual field is represented with equal precision.

I am able to make such assumptions because I am simulating Cassie's embodiment. It should be noted that such assumptions must be discarded when an agent is acting and reasoning in the real world, in real-time, with limited memory and sensory apparatus. However, recent efforts in robotics have included teaching robots to enumerate objects (Vitay, 2005). Such an effort would require an expansion of my theory to address more PML issues. However, because GLAIR imposes a methodological dualism, there is nothing (in principle) preventing a more robust PML from being attached to Cassie's KL.

Unlike abstract internal arithmetic, a variety of SNePS representations are necessary to fully model embodied enumeration. Consider the variety of mental entities needed during the enumeration of apples:

1. The grounded quantity, including:
 - The numbers (qua numerons as given in the previous chapter)
 - The sortal *apple* (a universal of which there are particular instances).
 - The collection relationship between the number three and the sortal apple which Cassie is taking the quantity to be.
 - The perceived quantity.
2. The prototypical apple, along with its prototypical shape and color.
3. The current apple being attended to, along with its particular shape and color.
4. The particular apples that are instances of the sortal and are constituents of the collection.

Suppose we consider a particular apple in a collection. This is represented as: an array of pixel-brightness values in the PML, an object being attended to in the PML, an instance of a sortal (if it resembles the prototype closely enough) in the KL, and a constituent of the collection in the KL. This distribution of representations must be coordinated by Cassie.

5.2.3 KL

Cassie’s knowledge of apples will be represented in the KL. Among this knowledge, we will expect to find relevant beliefs about the shape and color of apples. Realistically, we should expect Cassie to also have knowledge involving apples that is completely irrelevant to enumeration. For example, she may believe that, in the context of the story of *Genesis*, Eve gave Adam an apple. This will not interfere in any way, because the name “apple” can be applied to various non-mathematical ways in the KL.

Along with the counting case-frames presented in the previous chapter, the following case-frames are used to model the domain:

Name

Syntax:

Definition	SNePSLOG	Network
<i>define-frame Name(nil object name)</i>	Name (x , y)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is the word or proper name for $\llbracket y \rrbracket$ in English.

Sample Uses: Name (b1 , Albert) asserts that the proper name of b is Albert.

Name (b2 , cup) asserts that the English word for b2 is cup.

InstanceOf

Syntax:

Definition	SNePSLOG	Network
<i>define-frame InstanceOf(nil particular universal)</i>	InstanceOf (x , y)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is a particular instance of the universal $\llbracket y \rrbracket$.

Sample Use: InstanceOf (x , uApple) asserts that x is a particular instance of the universal uApple.

HasProperty

Syntax:

Definition	SNePSLOG	Network
<i>define-frame HasProperty(nil object property)</i>	HasProperty(x, y)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ has (i.e., instantiates) the (universal) property $\llbracket y \rrbracket$.

Sample Use: HasProperty(b, uRed) asserts that b has the property uRed.

ConstituentOf

Syntax:

Definition	SNePSLOG	Network
<i>define-frame ConstituentOf(nil constituent collection)</i>	ConstituentOf(x, y)	

Semantics: $\llbracket m \rrbracket$ is the proposition that $\llbracket x \rrbracket$ is one of the constituents of the collection $\llbracket y \rrbracket$.

Sample Use: ConstituentOf(b, c) asserts that b is a constituent of collection c.

Color

Syntax:

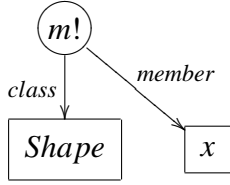
Definition	SNePSLOG	Network
<i>define-frame Color(class member)</i>	Color(x)	

Semantics: $\llbracket m \rrbracket$ is the proposition that the universal $\llbracket x \rrbracket$ is a color.

Sample Use: Color(uRed) asserts that uRed is a color.

Shape

Syntax:

Definition	SNePSLOG	Network
<i>define-frame Shape(class member)</i>	Shape(x)	

Semantics: $\llbracket m \rrbracket$ is the proposition that the universal $\llbracket x \rrbracket$ is a shape.

Sample Use: Shape(uRound) asserts that uRound is a shape.

It is first necessary to distinguish between universals and their English names:

```
;;;The name of Red (the universal) is 'Red'  
Name(uRed,Red).  
  
;;;The name of Round (the universal) is 'Round'  
Name(uRound,Round).  
  
;;;The name of Apple (the universal) is 'Apple'  
Name(uApple,Apple).
```

This separates the linguistic entities that may be present in Cassie's KL from the universals involved in enumeration. The universal uApple will play the role of the sortal in the quantity.

Next, we bind the distinguishing properties to their categories:

```
;;;Red is a color  
Color(uRed).  
  
;;;Round is a shape  
Shape(uRound).
```

These representations do not draw out the distinction between (for example) colors qua universals (e.g., red) and color instances (e.g., the redness of that apple). This is largely because the color instances will be accounted for in the PML.

Now we represent the rule "All apples are red and round" as:

```
all(x)(InstanceOf(x,uApple) => {HasProperty(x,uRed),HasProperty(x,uRound)}).
```

Collectively, the previous three beliefs represent general knowledge about apples that will guide Cassie's expectation of the kinds of objects she is attempting to recognize. This rule models Cassie as lacking a belief that apples can be green. This disjunctive distinguishing property can be accounted for in SNePS, but I will avoid it in this presentation. As it stands, Cassie would simply not classify green apples as being apples.

The KL implementation of embodied enumeration uses two complex acts: a top level act EnumerateApples and helper act UpdateCollection. The former could easily be generalized for any sortal, and the latter could be generalized for any quantity, but I present the more specific acts to shorten the exposition. These acts are implemented as follows:

```

ActPlan(UpdateCollection(q),
  withsome({?n,?m},
    (Collection(q,NumSort(?n,uApple)) and Successor(?m,?n)),
    snsequence(disbelieve(Collection(q,NumSort(?n,uApple))),
      believe(Collection(q,NumSort(?m,uApple)))),
    Say("Something is wrong."))).

all(i)(Image(i)=>
  ActPlan(EnumerateApples(i),
    snsequence(BuildEnumFrame(uApple),
      snsequence(believe(Collection(q,NumSort(n0,uApple))),
        snsequence(Perceive(i),
          snsequence(withsome({?c,?cn},
            (TargetColor(?c) and Name(?c,?cn)),
            DistinguishByColor(?cn),
            SayLine("I don't know what color apples are")),
          sniterate({if(ObjectsLeft(i),
            withsome({?s,?sn},
              (TargetShape(?s) and Name(?s,?sn)),
              snsequence(DistinguishByShape(?sn),
                snsequence(AddToCollection(Apple,q),
                  snsequence(UpdateCollection(q),
                    snsequence(AttendToNewObject(i),CheckIfClear(i))))),
                SayLine("I don't know what shape apples are"))),
            else(SayLine("Enumeration Complete")))))))))).

```

It will be helpful to trace through the act:

1. The primitive action `BuildEnumFrame(uApple)` is performed first. With this action, Cassie imagines a prototypical apple by asserting it (from the PML) as an instance of the universal sortal `uApple`. This allows her to infer the color and shape of this prototypical apple, `uRed` and `uRound` respectively. The predicates `TargetColor` and `TargetShape` are applied to these to distinguish them as the relevant distinguishing properties.
2. Cassie then forms a belief that the quantity `q` she is about to enumerate is currently a collection of zero apples.
3. `Perceive` is invoked with an image name (an artifact of the representation that can be ignored). This results in a PML perceptual representation of what Cassie sees.
4. The name of the target color is deduced and is passed to the primitive action `DistinguishByColor`. This effectively filters the visual field leaving only red objects.
5. Cassie then checks if there are any candidate objects left to enumerate. If there are none, she declares that her enumeration is complete.
6. If there are still candidate objects in her visual field, Cassie applies the primitive action `DistinguishByShape`. This routine considers each object in turn and determines the circularity of each.
7. At this point, Cassie decides whether or not a candidate from the previous step should be called an apple and, if she determines she has recognized an apple, she creates a new KL node to represent the instance she has perceived with the `AddToCollection` primitive action. This is followed by an invocation of the complex act `UpdateCollection`, which updates the number component of the quantity to its successor.

8. Cassie then marks the candidate as accounted for and shifts her attention to the next candidate object using the `AttendToNewObject` primitive action. Going back to Step 5, this process will stop when she has marked all items as counted.

It is apparent that much of the real work is being done by Cassie’s perceptual system.⁵ The counting component is isolated to the `UpdateCollection` primitive action. Cassie accrues the quantity as she is performing the enumeration. Thus, at any point, she has a value for the number of “apples enumerated so far”.

This act illustrates two different methods of applying distinguishing properties. The distinguishing property of color is applied to the entire perceptual field in parallel (i.e., without iterating over target objects), whereas the distinguishing property of shape is applied serially.

5.2.4 PML

The PML simulates Cassie’s embodiment by providing low-level routines for perception and motor control. This is her “window” on the world, an interface between the simulated external objects and the KL. We consider each sublayer of the PML in turn:

5.2.4.1 PMLa

The PMLa is the layer in which we implement Cassie’s primitive actions. The PMLa is written in Lisp and utilizes the `SNePS define-primaction` function to create primitive actions and the `attach-primaction` function to associate each primitive action with a KL base node that denotes the action. All information flowing from the Cassie’s “body” to her “mind” passes through the PMLa. The PMLa is responsible for coordinating the low-level routine of object recognition and the high-level routine of counting.

5.2.4.1.1 BuildEnumFrame The `BuildEnumFrame` primitive act is given a `sortal` and performs inferences to determine the relevant distinguishing properties for instances of this `sortal` in the modality of vision. For vision, Cassie uses shape and color as distinguishing properties. The sensory modality used for the embodied enumeration determines which kinds of properties will be useful. For example, suppose Cassie had to enumerate Scrabble tiles by placing her hand into a bag and drawing them out. Then the distinguishing properties of shape, size, and texture would be most appropriate, but color would not be useful.

Cassie first creates a prototypical instance of the `sortal` whose name is a concatenation of the string “proto” and the `sortal` name. In the case of enumerating apples, this instance is called *protouApple*. Then, using the `tell` interface, this instance is asserted into the KL. At this point, it can be deduced that *protouApple* is red and round. These relevant properties are determined using the `askwh` interface, which captures KL deductions and returns them to the PML. The relevant color `uRed` returned is tagged with `TargetColor(uRed)`, and the relevant shape `uRound` is tagged with `TargetShape(uRound)`.

⁵In fact, for human agents, there is even more “work” to be done at the knowledge level. For example, such work includes the construction of a spatial plan for sweeping across the perceptual field.

This process represents Cassie subconsciously bringing-to-bear the relevant features for object recognition.

5.2.4.1.2 Perceive The `Perceive` primitive act is given the name of an image stored in PMLb. This is a very rudimentary model of Cassie deciding at the KL what she wants to look at. It is assumed that the relevant image has all of the enumeration targets (i.e., that Cassie does not have to look somewhere else to finish counting).

The acquired image is implemented as a Lisp class and consists of three arrays of pixel-brightness values, one for red, one for green, and one for blue. The image is kept very small (a 16 by 16 matrix) so that pixel values can be displayed on screen for the user. However this can be scaled up if the user does not require the output to fit on the screen.

5.2.4.1.3 DistinguishByColor Given the name `Red` deduced from the universal `uRed`, the `DistinguishByColor` primitive act first accesses the `*COLOR-PALETTE*` Lisp hash to obtain prototypical red-green-blue values for `Red`. These values, along with a threshold error for the property of color is passed along to the PMLb function `threshold-classify` (described below). The three arrays are replaced by a single binary array (i.e., an array with only values of 0 and 1) where every candidate-object pixel is labeled with a 1, and every non-object pixel is labeled with a 0.

5.2.4.1.4 DistinguishByShape Given the name `Round` deduced from the universal `uRound`, the `DistinguishByShape` primitive act first accesses the prototypical compactness for round objects. The candidate object whose top-left object pixel has the smallest values is attended to. A chain-code representation of this object's shape is computed via the PMLb function `clockwise-chain-code`. The prototypical compactness value (1.0 for a circle) and object chain-code are then passed to the `object-gamma` PMLb function to determine the candidate object's circularity.

5.2.4.1.5 AddToCollection Given a sortal name and collection, the `AddToCollection` primitive act creates a new KL base-node to represent a newly recognized instance of the sortal by concatenating the sortal name and a randomly generated identifier. So, for example, a newly recognized apple may get the identifier `Apple1234`. It is then asserted that this instance is a constituent of the collection being accumulated: `ConstituentOf(Apple1234, q)`. Another assertion is made to indicate that `Apple1234` is an instance of the universal `uApple`. Finally, the new object is entered into the alignment table along with its top-left object-pixel coordinates. The alignment table is implemented as a Lisp hash `*ALIGNMENT-TABLE*`. In principle, the top-left object-pixel values could be used for Cassie to "point" to each constituent of the collection in her perceptual field.⁶

⁶In an earlier implementation, Cassie also tracked the ordinal position of each new constituent of the collection in terms of which object was counted n th. This was removed to reduce the KL complexity. As such Cassie is a model of an agent who has mastered the order-invariance counting principle (Gelman and Gallistel, 1978).

5.2.4.1.6 `AttendToNewObject` The `AttendToNewObject` primitive act effectively marks each object as it is enumerated and incorporated into the collection. This is done by removing it from the perceptual field via the `wipe-object` PMLb function.

5.2.4.1.7 `CheckIfClear` The `CheckIfClear` primitive act checks whether there any candidate target objects left in Cassie’s perceptual field. When this is the case, this is the termination condition for embodied enumeration.

5.2.4.2 PMLb

The PMLb is implemented in Lisp and has no direct access to the KL. In this layer of GLAIR, I have implemented Cassie’s vision system. This includes a representation of matrices as lists of lists in Lisp along with routines to access a pixel given its coordinates and converting to and from the matrix representation. In what follows, I describe the functions called by the primitive acts.

5.2.4.2.1 `threshold-classify` As described above, the 2D image is stored as three 2-dimensional arrays of pixel brightness values: a red array, a green array, and a blue array. The color value at pixel (i, j) is given by the triple $(red(i, j), green(i, j), blue(i, j))$.

The prototypical feature for “an apple’s redness” is also a red-green-blue (RGB) triple, which we shall write as $(red_{apple}, green_{apple}, blue_{apple})$. The first test is to find the absolute difference between each pixel component-color and the corresponding prototypical component value. A summation of these values gives the component error for pixel (i, j) :

$$Err(i, j) = (red_{apple} - red(i, j)) + (green_{apple} - green(i, j)) + (blue_{apple} - blue(i, j))$$

A binary 2D array $Binary(i, j)$ is then created based on a predefined error threshold for color $Thresh_{color,apple}$.

$$Binary(i, j) = \begin{cases} 1, & \text{if } Thresh_{color,apple} \leq Err(i, j) \\ 0, & \text{otherwise} \end{cases}$$

The binary array represents the candidate object pixels after being filtered for the target color. An idealized example of such an image is given in Figure 5.3. Realistically, the binary array will be far more noisy, with stray candidate pixels needing to be rejected by future tests.

5.2.4.2.2 `clock-wise-chain-code` Shape is represented using an 8-direction chain-code representation (Sonka, Hlavac, and Boyle, 1999). The chain code for a contiguous set of object pixels is a list of cardinal directions (i.e., N,NE,E,SE,S,SW,W,NW) on a path around the border of the object. We begin our chain code from the top left object pixel. A chain-code representation is illustrated in Figure 5.4. The utility of the chain code representation is that it provides a syntactic representation of shape that can be used in further operations. Given a chain code (d_1, d_2, \dots, d_n)

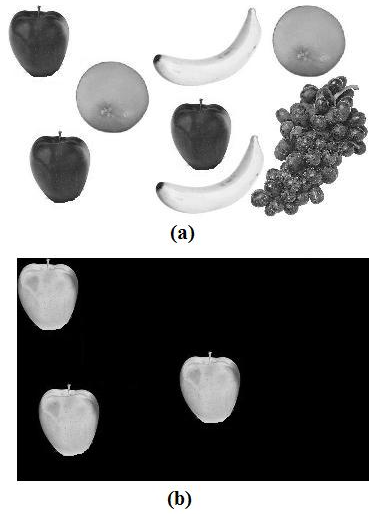


Figure 5.3: (a) Original 2D image (b) Ideal extracted object pixels

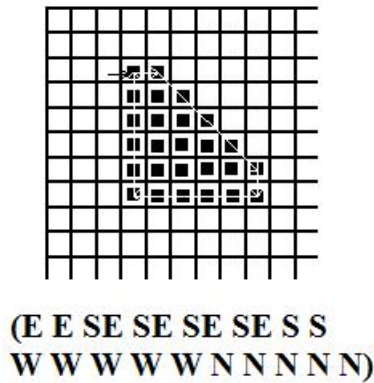


Figure 5.4: Chain-code representation of shape

for an object x , we have the perimeter $P(x)$ is:

$$P(x) = \sum_{i=1}^n l_i$$

where l_i is given by:

$$l_i = \begin{cases} \sqrt{2}, & \text{when } d_i = NE, SE, NW, SW \\ 1, & \text{when } d_i = N, E, S, W \end{cases}$$

It is also straightforward to compute the area of an object given a chain-code representation. We determine a bounding box around the object and sum up the number of object pixels inside this region.

5.2.4.2.3 object-gamma Given the perimeter $P(x)$ and the area $A(x)$, the circularity of object x is given by its compactness $\gamma(x)$ (Sonka, Hlavac, and Boyle, 1999):

$$\gamma(x) = 1 - \frac{4\pi A(x)}{P(x)^2}$$

The value of $\gamma(x)$ tends towards 0 as the shape of x tends towards a perfect circle. We classify the object as “circular enough to be an apple” based on whether the distance of the target object from perfect circularity falls under a predefined threshold. Circularity in the PML corresponds with the relevant KL concept “round”. However, we do not want to check for *perfect* circularity in the target object, but an “apple’s circularity”. This would be achieved by retrieving the prototypical compactness for an apple $compact_{apple}$ and using it instead of 1 in the measure of “apple” circularity:

$$\gamma_{apple}(x) = compact_{apple} - \frac{4\pi A(x)}{P(x)^2}$$

Like the “redness of apples”, this value is learned through experience with apples (including the act of enumeration). Also like color, the candidate object can be classified based on some threshold $Thresh_{compact,apple}$.

5.2.4.2.4 wipe-object When an object is determined to be an apple (i.e., its color and shape fall under the threshold values), the object must be “marked” as counted in the visual field. This is done by removing it from the binary matrix representation of the image. The removal of object pixels given a top-left object-pixel and a chain code is a non-trivial procedure. The spatial extension of the object is represented by three values: $delta - i$, the number of pixels down from the top-left object pixel the object occupies, $delta - j$, the number of pixels to the right of the top-left object pixel the object occupies, and $delta - neg - j$, the number of pixels to the left of the top-left object pixel the object occupies. This is illustrated in Figure 5.5. This procedure is not general, as it

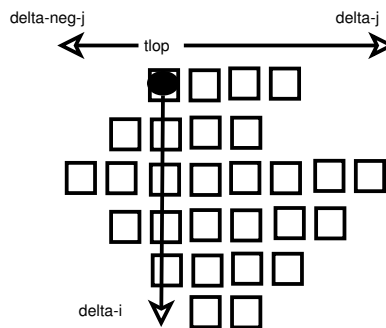


Figure 5.5: The delta values computed from an object’s top-left object-pixel and its chain code.

does not work for objects with a bottom concavity, but, since our assumption is that apples will be presented in a characteristic pose, this method is sufficient.

5.2.4.3 PMLc

The PMLc includes Cassie's image acquisition mechanism. Images in JPEG format are exported to a C Source format using the GIMP (GNU Image Manipulation Project). A small C program writes out the pixel values in the RGB array form expected by Lisp. These arrays are then pasted by hand into the Lisp PMLc. The two relevant classes for images are implemented as Lisp classes:

```
(defclass BinaryImg ()
  ((width :accessor img-width :initarg :width)
   (height :accessor img-height :initarg :height)
   (pixels :accessor img-pixels :initarg :pixels)))

(defclass RGB ()
  ((width :accessor img-width :initarg :width)
   (height :accessor img-height :initarg :height)
   (red-pixels :accessor img-red-pixels :initarg :red-pixels)
   (green-pixels :accessor img-green-pixels :initarg :green-pixels)
   (blue-pixels :accessor img-blue-pixels :initarg :blue-pixels)))
```

In future implementations, this process should be automated using the foreign-function call interface provided in Lisp. Another alternative is to code the PMLc in Java and utilize the JavaSNePS API (available in SNePS 2.7) for image acquisition.

5.2.5 Sample Run

The following sample run puts all of the pieces together for the task of embodied enumeration. In this interaction, Cassie enumerates a collection of two apples. The demonstration should be self-explanatory.

```
: perform EnumerateApples(img1)
Acquiring image img1
RED PIXELS:
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 128 254 254 254 254 254 254 254 254 254 254 254)
(254 254 128 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 128 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 128 254 128 254 254)
(254 254 254 254 254 254 254 254 254 254 128 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
GREEN PIXELS:
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 0 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 0 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 254 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 0 0 254 254 254)
(254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
```

```
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
```

BLUE PIXELS:

```
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 0 0 0 0 254 254 254 254 254 254 254 254 254 254)
(254 0 0 0 0 254 254 254 254 254 254 254 254 254 254)
(254 254 0 0 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 0 0 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 254 254 0 0 0 0 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
(254 254 254 254 254 254 254 254 254 254 254 254 254 254 254)
```

Distinguishing by color Red with threshold 100

Prototype for color is: (255 0 0)

Binary Image:

```
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0)
(0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0)
(0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0)
(0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0)
(0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0)
(0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

Distinguishing by shape Round with threshold 0.4

Prototype for shape is compactness = 1.0

Chain code for top-left object is (E SE S SW NW W N NE)

Circularity for top-left object is -0.34753005695632955d0

Adding KL constituent to collection.

Updating alignment table.

KLSym: Apple3425 PMLSym: (location (1 2)) Modality Vision

Clearing top-left object

New Binary Image:

```
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0)
(0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0)
(0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0)
(0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0)
(0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0)
```

```
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

I detect more objects to enumerate
Distinguishing by shape Round with threshold 0.4
Prototype for shape is compactness = 1.0
Chain code for top-left object is (E SE E S S SW W N NW SW N N NE)

Circularity for top-left object is 0.11479907880955154d0
Adding KL constituent to collection.

Updating alignment table.
KLSym: Apple3836 PMLSym: (location (5 9)) Modality Vision
Clearing top-left object

New Binary Image:
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

Enumeration Complete

CPU time : 0.36

After Cassie completes the enumeration, we can ask her some questions about the collection. First, we can ask her to describe the collections in her belief space:

```
: Collection(?x,?y)?  
  
wff127!: Collection(q,NumSort(n2,uApple))
```

CPU time : 0.00

We can also ask her to describe the constituents of the collection and the instances of the universal (sortal) apple she holds beliefs about:

```
: ConstituentOf(?x,q)?  
  
wff122!: ConstituentOf(Apple3836,q)  
wff112!: ConstituentOf(Apple3425,q)
```

CPU time : 0.01

```
: InstanceOf(?x,uApple)?  
  
wff123!: InstanceOf(Apple3836,uApple)  
wff113!: InstanceOf(Apple3425,uApple)  
wff94!: InstanceOf(protuApple,uApple)
```

CPU time : 0.00

Here, `wff94` represents Cassie’s belief about the prototypical apple she conceived while performing `BuildEnumFrame`.

5.2.6 Representing Experience with a Sortal

Cassie judges a particular apple to be an instance of the sortal `uApple` on the basis of its resemblance to a prototypical instance. This was achieved by considering an error threshold for each distinguishing feature. However, the embodied enumeration of apples should be counted as experience with the sortal apple and should impact (however slightly) Cassie’s prototypical values for an apple’s distinguishing properties. A natural way to do this is to average the instance values for color and shape into the prototypical value. A more robust implementation would create sub-prototypes for the different types of apples as Cassie has experiences involving the different type (e.g., Granny Smith apples are green, red delicious apples are more square than circular).

5.3 Assignment Numeracy and its Applications

Embodied enumeration is a special form of a cardinal number assignment. As mentioned above (see §2.1.4.2) number assignments are one of the fundamental uses of numbers. When making a cardinal number assignment, an agent will mark off each element of a counted set with a successive number in the natural-number sequence. When the final element is counted, the agent “assigns” the resulting number to the entire set. This number is the cardinality of the set, and acts as the *size* of the set. It is the cardinal number, and not the counting procedure that generated it, that makes its appearance in language. We say things like: “I saw seventeen birds on the lawn” instead of “I saw one bird on the lawn, then a second bird on the lawn, . . . then a seventeenth bird on the lawn”. When making an ordinal number assignment, an agent marks off particular events as they unfold in time or objects as they occur in space. Thus, an agent could mark a car as being the *third* to cross the finish line after counting a first and second car to cross the finish line. An agent could mark a car in a parking lot as being the *third* from the left by starting at the leftmost car and counting cars as it moves its gaze right. We shall say that an agent is *assignment numerate* (or has *assignment numeracy*) if it can properly perform both cardinal and ordinal number assignments.⁷

In what follows, I will describe some applications of assignment numeracy involving the fundamental cognitive ability of reasoning about actions. I will then provide a formalization of two problem domains in which assignment numeracy yields a solution method. These domains are non-mathematical (or, at least not overtly mathematical), and are intended to demonstrate a breadth of applicability for Cassie’s quantitative reasoning.

⁷I am purposely downplaying an agent’s ability with nominal number assignments (e.g., “Bus number 3”) because such assignments are meant to be treated simply as identifiers. As such, they are more properly a part of computational theories of natural language.

5.3.1 Self-Action Enumeration

Actions are among the most important abstract entities that an agent can count. This point is made with some humor in a *Peanuts* cartoon by Charles Schultz. A girl named Lucy is holding a jump rope and talking to a boy named Rerun:

- **Lucy:** See Rerun? It's a jump rope ... (she starts jumping) you twirl the rope, and you jump up and down like this ... then you count how many times you jump ...
- **Rerun:** Why?

What this cartoon gets at is that it is utterly natural for children to enumerate repetitive actions, but it is somewhat unnatural for them to wonder why they do so. Once a child can deploy such enumerations in service of cardinality assignments, they can characterize the complex act (e.g., jumping rope) in terms of a number of iterations of a sub-act (e.g., times jumped over the rope). This forms the basis of cardinal comparisons (e.g., “How many times can you jump rope?”) and permits comparison of all “countable” actions *as if they were object collections*.

An iterated sub-act will only be enumerable if it has certain “objectual” characteristics:

- *Identification Criteria:* A way to distinguish a performance of the sub-act from either a non-performance of the sub-act or the performance of a different sub-act within the complex act
- *Individuation Criteria:* A way to distinguish a single (token) performance of the sub-act within the complex act (i.e., the sub-act must have a clear “boundary”).

There is evidence that children begin to individuate actions from a stream of continuous motion well before they grasp the principles of counting (Wynn, 1996; Sharon and Wynn, 1998), so applying action enumeration is just a matter of transferring this skill to a self-actions. I will call this task self-action enumeration (SAE).

In general, action enumeration involves an “actor” agent performing the actions and a “counting” agent enumerating the actions. SAE is thus a special case of action enumeration in which the actor agent *is* the counting agent. This fact makes self-action enumeration easier to implement than general action enumeration. Since an agent will know when it is initiating an action at the knowledge layer, it will be able to count initiated actions without any feedback from the external world. An agent responsible for general action enumeration (or self-action enumeration of *completed* external actions) will require sensory feedback from its perception or proprioception. Such feedback is a confirmation that the initiated act completed successfully.

Two distinct modes of self-action enumeration can be identified: *conscious counting* and *post-action counting*. In the conscious counting mode, the agent is informed that one of its actions is to be counted *before* it has performed the first such action. The impact of such a request is twofold: (i) while on the job, the agent must augment its original plan for performing an act by adding an additional counting step to its procedure, and (ii) if asked after the job, the agent must be able to retrieve the the number of times an action was performed. Because the agent is keeping track of the action count while it is performing the actions, it will also be able to answer questions of the form “How many times did you do action A?” while still on the job.

In the post-action counting mode, an agent is in the middle of performing a set of actions (or has entirely completed the actions) when the request to enumerate is made. Since the agent has not been attending to the number of actions it has performed, it must look for some “residue” of action to serve as evidence that the action took place. This requires an agent that can reason about what external effects or states in the environment will indicate that the action took place, or to otherwise consult an internal episodic memory of the act.

5.3.1.1 Implementation

For my implementation, I have developed a version of the conscious counting mode of self-action enumeration. I will reuse the `Collection` case-frame as given above. This time, the sortal is the act to be consciously counted, and the quantity represents the collection of such acts. The variable tracking this quantity is `SAEResult` (i.e., the self-action enumeration Result). This is a special variable, so Cassie can only have one conscious-count active at any given time. This is a cognitively plausible assumption.

A user indicates that Cassie should consciously count one of her complex acts `a` by asserting `ConsciousCounting(a)!`. The ‘!’ indicates that forward inference is performed immediately after `ConsciousCounting(a)` is asserted. Next, the command `perform ConsciousCountingSetup(a)` must be invoked to have Cassie augment her original plan with a counting step. This complex act is implemented as follows:

```
;;;To set up a conscious count of the agent action a, the designer
;;;has the agent perform ConsciousCountingSetup(a) before the job.
;;;The agent discards any plan p for achieving act a and replaces
;;;p with an augmented plan that sequentially increments the count
;;;of a and then performs the original p
all(a,p)({ConsciousCounting(a), ActPlan(a,p)} &=>
  ActPlan(ConsciousCountingSetup(a),
    snsequence(believe(Collection(SAEResult,NumSort(n0,a))),
      snsequence(disbelieve(ActPlan(a,p)),
        believe(ActPlan(a,snsequence(IncrementActionCount(a),p))))))).
```

Cassie first initializes the collection `SAEResult` to zero performances of `a`. She then temporarily disbelieves the plan `p` for performing `a` in favor of a count-augmented plan that includes the step `IncrementActionCount(a)`. This complex act is implemented as follows:

```
;;;Plan for incrementing the conscious count of action a.
all(a,m,n)({Collection(SAEResult,NumSort(m,a)),Successor(n,m)} &=>
  ActPlan(IncrementActionCount(a),
    snsequence(disbelieve(Collection(SAEResult,NumSort(m,a))),
      believe(Collection(SAEResult,NumSort(n,a)))))).
```

This is a straightforward update of the current number of performances of `a` from `m` to its successor `n`. The agent can be asked at any moment how many times it has performed act `a`: `Collection(SAEResult,?x)?`. At some point, the user will want to tell the agent to stop its conscious count of `a`. This is done with `StopConsciousCounting(a)`:

```
;;;Plan for terminating the conscious count. The agent no longer
;;;believes that it is consciously counting act a. It discards
;;;the count-augmented procedure for plan p and restores p as the
;;;plan associated with act a.
all(a,p)({ConsciousCounting(a), ActPlan(a,snsequence(IncrementActionCount(a),p))} &=>
  ActPlan(StopConsciousCounting(a),
    snsequence(withsome(?n,
```

```

Collection(SAEResult,NumSort(?n,a)),
SayLine+"The result of SAE is ",?n,a),
SayLine("I could not perform SAE for the act.")),
snsequence(disbelieve(ConsciousCounting(a)),
snsequence(disbelieve(ActPlan(a,snsequence(IncrementActionCount(a),p))),
believe(ActPlan(a,p)))))).

```

This complex act first deduces the number of times *a* has been performed and reports it to the user. Cassie then disbelieves that *a* is still being consciously counted and reinstates the original plan *p* for performing *a*.

The user may still want to have the result of the conscious count of *a* available after the count has happened. If this value is no longer needed, the complex act `ClearSAEResult` can be called:

```

;;;Plan for clearing the conscious count for act a.
all(a,n)(Collection(SAEResult,NumSort(n,a)) =>
  ActPlan(ClearSAEResult(a),
    disbelieve(Collection(SAEResult,NumSort(n,a))))).

```

This act simply removes the quantity associated with `SAEResult`.

5.3.1.2 Sample Run

In this section, I present a sample run of conscious counting. I first give Cassie an act called `Greet()` which consists of either saying “hi” or saying “hello” (the choice is made non-deterministically). The setup is as follows:

```

: define-frame Greet(action)
Greet(x1) will be represented by {<action, Greet>, <nil, x1>}

CPU time : 0.00
: ActPlan(Greet(),do-one({Say("hi"),Say("hello")}))

wff226!: ActPlan(Greet(),do-one({Say(hello),Say(hi)}))

CPU time : 0.00
: ConsciousCounting(Greet())

wff239!: ActPlan(ConsciousCountingSetup(Greet()),
  snsequence(believe(Collection(SAEResult,NumSort(n0,Greet()))),
  snsequence(disbelieve(ActPlan(Greet(),do-one({Say(hello),Say(hi)}))),
    believe(ActPlan(Greet(),
      snsequence(IncrementActionCount(Greet()),
        do-one({Say(hello),Say(hi)})))))))

wff227!: ConsciousCounting(Greet())
wff226!: ActPlan(Greet(),do-one({Say(hello),Say(hi)}))

CPU time : 0.01
: perform ConsciousCountingSetup(Greet())

CPU time : 0.08

```

At this point, Cassie has adopted a count-incremented plan for the performance of `Greet()`. We can verify that the current count is initialized to zero as follows:

```
: Collection(SAEResult,?x)?  
wff89!: Collection(SAEResult,NumSort(n0,Greet()))  
CPU time : 0.00
```

Now we have Cassie perform `Greet()` a few times and check with Cassie how far her conscious count has gotten:

```
: perform Greet()  
hi
```

```
CPU time : 0.05
```

```
: perform Greet()  
hello
```

```
CPU time : 0.09
```

```
: Collection(SAEResult,?x)?  
wff137!: Collection(SAEResult,NumSort(n2,Greet()))  
CPU time : 0.00
```

She correctly answers that she is up to two performances of `Greet()`. Despite this interruption, Cassie can continue with her conscious counting:

```
: perform Greet()  
hi
```

```
CPU time : 0.07
```

```
: perform Greet()  
hi
```

```
CPU time : 0.07
```

```
: perform Greet()  
hello
```

```
CPU time : 0.07
```

```
: Collection(SAEResult,?x)?  
wff319!: Collection(SAEResult,NumSort(n5,Greet()))
```

At this point we can stop the conscious count of `Greet()` and clear the `SAEResult`:

```
: perform StopConsciousCounting(Greet())  
The result of SAE is n5 m81
```

```
CPU time : 0.05
```

```
: perform ClearSAEResult(Greet())
```

```
CPU time : 0.03
```

```
: Collection(SAEResult,?x)?
```

```
CPU time : 0.00
```

5.3.2 Ordinal Assignment to Self-Action

Ordering is a fundamental mechanism behind the temporal demarcation of events. An agent must be able to act and reason *in* time and *with* time (i.e., with temporal knowledge). Some problematic aspects of temporal reasoning are discussed by Ismail and Shapiro (2000). The problem of temporal reasoning is extremely important in agent design. Large logical frameworks have been developed solely to address temporal reasoning (Shanahan, 1999).

A numerate agent has the ability to tag sequential actions with natural numbers. Such ordinal number assignments impose the ordering of the natural numbers onto the ordering of actions. This approach avoids a “system clock” timestamp for actions, which is itself an ordinal mapping made to a global (and perhaps less cognitively plausible) set of numbers. We are after an agent that knows what it did first, second, and third, not one that knows what it did at noon, 12:15, and 12:30. Indeed, the latter ability can easily be added if the agent makes a mapping from the actions to global time. As such, the ability to tag actions with numbers *is* a model of time. Given any pair of actions, the before-after distinction can be made simply by applying the less than-greater than comparison on the ordinals assigned to those actions.

In my SNePS implementation of ordinal assignment to actions, the user must use the `OrderedSnsequence` instead of the built-in SNeRE control act `snsequence` when the ordinal tagging of actions is desired.

```
;;An OrderedSnsequence of two actions. If a is primitive then tag it
;;with the next available ordinal, otherwise, if a is complex, then
;;just perform a. Next, check if b is primitive. If it is, then
;;check if a was primitive (i.e., check if an ordinal was used up on
;;a), if it was then give b the next available ordinal p, otherwise
;;give b the ordinal n. If b is complex, just perform b.
all(a,b,m,n,p)({Successor(n,m),Successor(p,n),
                OrdinalNumberReached(m),OrderableAct(a),
                OrderableAct(b)} &=>
ActPlan(OrderedSnsequence(a,b),
snsequence(snif({if(PrimitiveAct(a),
                snsequence(a,
                snsequence(believe(OrdinalPosition(a,m)),
                snsequence(disbelieve(OrdinalNumberReached(m)),
                believe(OrdinalNumberReached(n))))),
                else(a)}),
snif({if(PrimitiveAct(b),
                snsequence(b,
                snif({if(PrimitiveAct(a),
                snsequence(believe(OrdinalPosition(b,n)),
                snsequence(disbelieve(OrdinalNumberReached(n)),
                believe(OrdinalNumberReached(p))))),
                else(snsequence(believe(OrdinalPosition(b,m)),
                snsequence(disbelieve(OrdinalNumberReached(m)),
                believe(OrdinalNumberReached(n))))))}),
                else(b)))))).
```

This plan handles the four separate ways in which a designer might sequence two acts: (i) a primitive followed by a primitive, (ii) a primitive followed by a complex, (iii) a complex followed by a primitive, and (iv) a complex followed by a complex.

Ordinal assignments are stored in the `OrdinalPosition` predicate. The semantics of `OrdinalPosition(a,n)` is that act `a` is the n^{th} (primitive) act performed. This allows for two types of after-the-job questions:

- *What* questions. For example, “What is the third act you performed?” is equivalent to the SNePSLOG `OrdinalPosition(?x,3)?`
- *When* questions. For example, “When did you say ‘Albert’?” is equivalent to the SNePSLOG `OrdinalPosition(say('Albert'),?x)?`

Ordinals are only assigned to primitive actions and not to SNeRE primitive acts or complex acts. However, the possibility of having an agent correctly respond to ordinal-number questions at different scales of action complexity is quite intriguing. For one thing, humans will usually respond to such questions at the action complexity level specified by the context of discourse. Thus, the question “What is the first thing you did this morning?” might be answered by either “I opened my eyes” or “I drove to school” with equal validity. However, the question “What did you do before going to class?” is better answered by “I drove to school” than by “I opened my eyes”. This is because the act of driving to school and the act of going to class are both at roughly the same level of complexity. The problem is how to correctly assign an ordinal once the complexity level of the action is known. The matter is further complicated for a SNeRE-based agent, since primitive actions can be “factored” into different groupings (by different designers) to achieve effectively the same complex act.

I believe that there are two solutions to the problem of assigning ordinals at different granularities of action complexity. The first is to have the agent give the ordinal position of an action relative to some other action (Rapaport, personal communication). Thus, an agent might say “I opened my eyes first, relative to getting out of bed” and might say “I drove to school first, relative to going to class”. This solution introduces a new problem. The agent must either be given the granularity of action complexity in the question (e.g., “What is first thing you did this morning, relative to X ?”), or else must figure out the granularity from context (this ability, by itself, might make for a long research project).

Another solution is to saddle the designer with the burden of deciding what actions should receive ordinal assignments. The designer and agent are in a design partnership, and this may be a good place for the designer to step in. There are two fundamental choices to be made in the design of any SNeRE-based agent:

1. Which agent actions should be primitive?
2. How should agent actions be grouped as complex actions?

The first question is both philosophical (Baier, 1971) and practical. Primitive actions in SNePS agents are written in Lisp, stored at the PML of GLAIR, and form the building blocks of all

complex actions available to an agent. Because all agent activities can be decomposed into a sequence of performed primitive actions, it makes sense to form an ordinal sequence of these. This sequence will give the agent access to the progression of its most basic functions. Our current implementation handles the formation of the primitive action ordinal sequence.

The second question forces the designer to decide which groupings of primitive actions (and groupings of mixtures of primitive and complex actions) should be given `ActPlans`. The designer must make an ontological commitment as to which groupings of primitive actions are themselves worthy of being called actions. I believe that this level of agent activity, namely, the level of the `ActPlan`, also deserves an ordinal sequence. The `ActPlan` thus becomes an *a priori* contract between the designer and agent. When a designer writes `ActPlan(a, p)` in an agent’s source code, they are saying that the plan for performing act *a* is *p* *and* that act *a* is worth ordering.

Another issue, closely related to the ordinal assignment problem, is how the agent will linguistically relate two complex actions when answering a temporal question. Given two complex acts, *a* and *b*, with f_a being the ordinal of the first primitive act of *a* (possibly after decomposition of the first complex act of *a*), f_b being the ordinal of the first primitive act of *b*, and l_a and l_b being the ordinals of the last primitive acts of *a* and *b* respectively, we have six possible orientations of the two complex acts. Associated with each orientation is a different linguistic description relating the events:

Primitive Act Structure	Linguistic Description
$l_a \leq f_b$	<i>a</i> is before <i>b</i> .
$l_b \leq f_a$	<i>a</i> is after <i>b</i> .
$f_a \leq f_b \leq l_a$	<i>a</i> begins before <i>b</i> begins and ends after <i>b</i> begins.
$f_b \leq f_a \leq l_b$	<i>b</i> begins before <i>a</i> begins and ends after <i>a</i> begins.
$f_a \leq f_b$ and $l_b \leq l_a$	<i>b</i> happens during <i>a</i>
$f_b \leq f_a$ and $l_a \leq l_b$	<i>a</i> happens during <i>b</i>

With the last two cases, the use of word “during” does not suggest the presence or absence of a causal link between *a* and *b*. If a speaker wishes to indicate the presence of such a causal connection, they might say “I was doing *a* *by* doing *b*”, whereas, if the speaker wanted to indicate the absence of a causal connection, they might say “I was doing *a* *while* doing *b*”.

5.3.3 The Jail-Sentence Problem

An agent with assignment numeracy can also solve real-world problems in which quantitative information is used to make a qualitative decision. An interesting class of such problems arises when the ordinal structure of the natural numbers (and the corresponding ordinal relation “>”) is mapped to a qualitative domain. Consider a situation in which an agent is presented with the following information:

- The jail sentence for theft is 1 year in jail.

- The jail sentence for murder is 50 years in jail.
- The jail sentence for *blicking* is 25 years in jail.

Here, “blicking” represents an activity which is unknown to the agent. Suppose that the agent is asked what it knows about blicking. Two answers seem immediately plausible: (1) blicking is a crime, and (2) blicking is worse than theft but not as bad as murder. An agent might infer (1) from contextual information (this is a problem involving crimes, such as theft and murder) or background knowledge (only crimes receive jail sentences). But how might an agent come to believe conclusion (2)? Where did the qualitative relations “worse-than” and “not-as-bad-as” come from?

I suggest that an agent comes to such a conclusion because it has a rule in its knowledge base that “maps” the ordering of the natural numbers (along with the ‘>’ relation that orders them) onto the domain of crimes (along with the ‘worse-than’ relation). In other words, the agent has access to the rule: “The worse the crime, the longer the jail sentence”. This rule may be formalized as follows:

$$\forall x, y, m, n ((Crime(x) \wedge Crime(y) \wedge JailSentence(x, m) \wedge JailSentence(y, n) \wedge m > n) \supset WorseThan(x, y))$$

In general, given a unary predicate P that fixes x and y in some domain Δ , a binary relation L that links members of Δ with quantities, a binary quantitative ordering relation $Quant$, and a binary “qualitative” relation $Qual$, a schema for ordinal-mapping rules can be expressed as:

$$\forall x, y, m, n ((P(x) \wedge P(y) \wedge L(x, m) \wedge L(y, n) \wedge Quant(m, n)) \supset Qual(x, y))$$

Such rules expand the meaning of qualitative relations by imposing the structure of the ordinals between qualitative relation parameters. How ordinal-mapping rules end up in an agent’s knowledge-base is another matter. In the ideal case, the agent is simply given the rule; this is our current simplifying assumption. More realistically, the agent will have to induce the generalization from several examples. To implement ordinal mapping onto qualitative domains, Cassie will make use of the `GreaterThan` relation that orders Cassie’s numerons.

5.3.4 The Breakfast-Always-Served Problem

Relational information is packed away in the meanings of terms in a natural-language. Consider the English word ‘breakfast’ as used in an expression one might find at a restaurant: “Breakfast always served”. To a human reasoner, it is immediately clear what the sign means, namely, that the sort of food one would have in the morning is served all day. However, as immediately obvious as this might appear, it is a reading that requires some inference. The word ‘breakfast’, like many words in English, is polysemous.⁸ ‘Breakfast’ can mean either (1) the first meal of the day or (2)

⁸I am grateful to Len Talmy (personal communication) for this example.

the type of food usually served for the first meal of the day. Both meanings yield a separate reading of “Breakfast always served”. Using (1), we obtain “The first meal of the day is always served”, and using (2), we obtain the intended reading.

How would a computational cognitive agent reject the first reading? This is a problem of word-sense disambiguation. A listener needs some method of picking out an intended sense of a polysemous word. In this particular example, we are dealing with the ordinal assignment “first meal of the day”. By reasoning about the ordinal structure of “meals of the day” and about experiences at a restaurant where someone is served their second meal of the day, Cassie can detect that there is a problem with the reading generated by (1).

For the implementation, suppose the *Pancake House* is a restaurant with a sign reading “Breakfast always served”. Cassie’s KB might include the following:

- wff1: Breakfast, lunch, and dinner are meals.
- wff2: An agent can eat one meal in one location at one time.
- wff4: Breakfast is the first meal of the day.
- wff7: A meal is an event.
- wff8: Events can be ordered.
- wff9: If an event is nth in an order, it has ordinal number n (with respect to the other events in the order)
- wff16: If two events are in the same ordinal position in the same ordering, they are the same event.
- wff17: If two events do not occur at the same time, they are not the same event.
- wff26: B1 is a breakfast.
- wff28: B2 is a breakfast.
- wff33: Albert ate B1 at 8am 2/6/06 at home.
- wff37: Albert ate B2 at 3pm 2/6/06 at the pancake house.

From this information, Cassie’s SNeBR subsystem can detect an inconsistency:

A contradiction was detected within context default-defaultct.

The contradiction involves the newly derived proposition:

wff38: `EquivEquiv({B2,B1})`

`{<der,{wff1,wff4,wff7,wff8,wff9,wff16,wff26,wff28},{wff17,wff33,wff37}>}`

and the previously existing proposition:

wff39: `~EquivEquiv({B2,B1})`

`{<der,{wff17,wff33,wff37},{wff1,wff4,wff7,wff8,wff9,wff16,wff26,wff28}>}`

The B1 and B2 that were served must be breakfast *qua* type of food, because, when considered as breakfast-the-event, a contradiction can be derived.

5.4 A Model of Count-Addition Strategy Change

I claim that embodied enumeration, discussed earlier in this chapter, is a gateway to the rest of embodied arithmetic. In the previous chapter, I made a similar claim about count-arithmetic routines. The earliest of these arithmetic techniques is count-addition. Interestingly, count-addition usually is initially employed as an embodied external routine and only later becomes an abstract internal routine. In this way, count-addition forms a bridge between embodied arithmetic and abstract arithmetic. I will address a more unified view of the “two arithmetics” in the next chapter, but here I wish to sketch a more elaborate model of count-addition than the one presented in the previous chapter.

In this section, I will take ‘count-addition’ to refer to a class of strategies that invoke counting as the primary method for computing the sum of two positive natural numbers. Count-addition manifests itself as a progression of successively more efficient routines performed over successively more abstract representations of the counted objects. As such, count-addition should not be thought of as a static algorithm, but rather as a set of improving strategies traversed by a cognitive agent.

5.4.1 Strategies

Childrens’ count-addition strategies have been well studied by cognitive scientists. In particular, researchers have focused on transitions between strategies (Fuson, 1982; Secada, Fuson, and Hall, 1983; Butterworth et al., 2001) and the spontaneous creation of novel efficient strategies from the modification of existing routines (Groen and Resnick, 1977; Neches, 1987; Jones and VanLehn, 1994). The variety of count-addition strategies can be seen as resulting from the early position of count-addition in a child’s mathematical development. The child is leaving behind a domain of concrete physical objects and moving to the more abstract domain of mental arithmetic, and still, it is precisely acting on concrete physical objects (e.g., fingers or blocks) that provides some semantics for the abstract operations of arithmetic. The demands of arithmetic force a cognitive agent to master an efficient counting routine.

I will focus on three strategies: COUNT-ALL (*CA*), COUNT-ON (*CO*), and COUNT-ON-FROM-GREATER (*COF >*).⁹ These strategies are described in the next section.

When using the *CA* strategy for computing $m + n$, the agent forms a collection of m objects and a collection of n objects (in the variation of *CA* presented by Neches (1987), these collections must be counted out initially). Both collections are assumed to be perceptually available and somehow separated in the perceptual field. The agent counts the first collection by assigning natural numbers to each object in the first collection: $1, 2, \dots, m$. The agent then counts the objects in the second collection $m + 1, m + 2, \dots, m + n$. The agent interprets the final number assigned as the sum. Although *CA* is an early strategy, it relies on several prerequisite skills: object recognition (or, at

⁹This is the terminology of Fuson (1982) and Butterworth et al. (2001). Elsewhere in the literature (Neches, 1987; Jones and VanLehn, 1994) these strategies have been called SUM, FIRST, and MIN, respectively, but these names do not emphasize the centrality of the counting act in the addition.

the very least object segmentation), the ability to mark off items in the perceptual field as they are counted, and the ability to coordinate the current total with the progression through the perceptual objects. Also, it is assumed that the agent can use its embodiment to help attend to its position in the counting process (e.g., by pointing at the physical objects or turning its gaze).

The *CO* strategy for computing $m + n$ is to say the number m and then count on through the second collection of objects $m + 1, m + 2, \dots, m + n$. This reduces the number of steps from *CA* and, since saying m becomes a shorthand for counting $1, 2, \dots, m$, removes the need to attend to the first object collection (or to even have one present). When using this strategy, it is assumed that the agent has a mechanism of tracking the second addend n (e.g., by raising fingers or in memory) to know when the count-addition is complete.

The *COF* > strategy for computing $m + n$ is to first compare m and n and to apply the *CO* strategy from the larger of the two: $\max(m, n), \max(m, n) + 1, \dots, m + n$. This produces an improvement on the *CO* strategy proportional to the absolute difference of the addends $|m - n|$.

The shift from one of these to another can take place independently of the type of target being counted. However, there is a tendency to shift from concrete objects towards abstract objects while the shift from *CA* to *COF* > is taking place.

There are several variations on intermediate strategies discussed in the literature (see Jones and VanLehn (1994)) and none of the strategy shifts are “all-or-nothing” (i.e., children will sometimes adopt a less efficient strategy even after discovering a more efficient one). What is remarkable about these strategy shifts is that they occur in very young children and in a very uniform, natural, and unconscious manner. Furthermore, the order of strategy change, from *CA* to *CO* to *COF* >, is stable across subjects. The psychological data suggest that *CO* and *CO* > are “spontaneously” invented by the agent without formal instruction (Neches, 1987; Groen and Resnick, 1977).

5.4.2 Computational Models

Strategy change is a powerful cognitive tool and, as such, computational models of early mathematical cognition should be able to account for these transitions. This account is bound to vary across computational cognitive architectures. The seemingly spontaneous character of the changes in strategy can be conveyed computationally in terms of emergent phenomena. For example, given a computational agent A capable of *CO*, there must be properties of A from which the *COF* > strategy emerges. Furthermore these properties should not be *COF* > or a simple restatement thereof, but should be more fundamental.

Neches (1987) describes a production-system model of count-addition strategy change using a technique called Heuristic Procedure Modification (HPM). The core idea is to enhance performance by detecting patterns in the trace of existing procedures and then transforming the procedure. This pattern matching involves determining the conditions for the application of certain heuristics. When these conditions are found to hold, modifications are made to the existing procedure, including: retrieving an item rather than recomputing it, eliminating the computation of an output that no other procedure uses as an input, and trying to use a method that involves less effort when operating on the same input. It is clear that in this model the spontaneous emergence arises

from the application of these heuristics (170).

Jones and VanLehn (1994) also present a production system model capable of modeling count-addition strategy change. It is called General Inductive Problem Solver (GIPS). GIPS combines a general problem solving algorithm with a probabilistic learning algorithm. Using both of these algorithms, GIPS is capable of symbol-level learning, which does not increase the deductive closure of the system's knowledge, and knowledge-level learning, which involves a change to the knowledge base and deductive capacity (15). The main finding of the work is that children do not invent *CO* strategies in response to a gap in knowledge (i.e., an impasse), but as a gradual replacement to earlier strategies. Hence this is also a model that stresses emergence over explicit instruction.

Both of these models are successful in that they do what the authors set out to do. However, the authors background issues of embodiment and abstraction change in the target objects (e.g., in both models, an external object, an internalized prototype, a finger, and a number-word would just be treated in more or less the same way). Also, both models are applicable to production system theories of acting and development. Although a popular paradigm, there are other computational paradigms that are sufficient for the modeling task and can expose different issues in the strategy shift.

5.4.3 Abstraction of Counting Targets

Steffe and colleagues (1983) have classified childrens' counting routines according to the kind of target items being counted. Ordered from most concrete to most abstract these targets are: perceptual, figural, motoric, verbal, and abstract. Any of these target types can be deployed in conjunction with any one of the count-addition strategies. This creates a matrix of possible strategy-target pairs to model (e.g., a *CA* strategy over figural items, a *COF >* strategy over abstract items). However the general tendency is to prefer more abstract representations as counting (and count-addition) becomes more familiar.

As in other areas of cognitive development, the child's progress in counting is marked by decreasing dependence on perceptual material. The first step in that direction is the ability to count figural representations of perceptual items (i.e., visualized images), which, though presented in the context of the task, are not perceptually available at the moment (36–37)

The perceptual-to-figural representation shift is highlighted here as an important first step. Each of the target types can be represented using our implementation platform, but below it will suffice to examine the differences between perceptual and figural target representations.

5.4.4 Commutativity, Comparison, and Cardinalities

Three significant aspects of the the *COF >* strategy are its apparent reliance on the commutative principle (i.e., that $m + n = n + m$), the comparison step in the strategy, and the cardinal principle. The literature is mostly silent about the relationship between the *CO* to *COF >* transition and

an agent's understanding the commutativity of addition. If commutativity did not hold, then the *COF* > strategy would simply not work. There are two plausible situations regarding the agent's state at the time of transition: (1) the agent has not yet grasped the general principle of commutativity but has noticed an inferential stability of results for past count-additions involving the given numbers, or (2) the agent has fully grasped the principle of commutativity for addition and is making the initial comparison while conscious of addend order irrelevance. Cowan and Renton (1996) have suggested that children's application of addend-swapping strategies (such as *COF* >) can occur without explicit knowledge of commutativity. In such cases, the agent is *obeying* commutativity without understanding the general principle. Thus, because the shift to *COF* > can happen at a developmentally early age, it makes more sense to model an agent in situation (1).

The overall savings afforded by the *COF* > strategy become more dramatic as the value $|m - n|$ increases. This "problem size effect" (Butterworth et al., 2001) is a good example of how an agent may be sensitive to training examples. An agent is much less likely to realize a difference in the count additions of 2+3 (3 steps) and 3+2 (2 steps) than in the count additions of 2+8 (8 steps) and 8+2 (2 steps). To become aware of this difference, an agent needs to be able to make comparisons of more than just the addends.

The overhead cost of comparing the addends may be a road block to the transition between the *CO* and *COF* > strategy. An agent must either convince itself that the additional cost of comparison is not significant compared to the steps saved in *COF* >, or it must simply improve its processing of comparison. But since the addends in count-addition may be represented at various levels of abstraction (as described above), the latter may be more easily achieved if physical object collections are present, allowing the agent to make a quicker quantity judgment. The presence of this road block may suggest why the transition is not "all or nothing" as described above.

Fuson (1982) links the need for concrete object collections with a child only having a partial understanding of cardinal principle. This indicates that the cardinal principle plays an important role in the process of abstracting from concrete (object collection) addends. An agent can treat a number as simultaneously representing a process (counting up to a number) and a concept (the size of a collection) will have a more compact representation of the problem domain.

Before describing the strategy shifts, it is worth considering a list of desiderata for any such computational model:

- New strategies should emerge from earlier strategies without explicit instruction and without drastic changes in agent organization and representation.
- Strategy shifts should not require the explicit representation of commutativity.
- The cardinal principle should be represented when modeling the *CO* and *COF* > strategies.
- A model should operate over representations at various levels of abstraction.

5.4.5 CA to CO

The plan for the CA strategy applied to $m + n$ can be expressed as a sequence of two counts. These counts can take place over the abstract representations, but, following Fuson (1982), I model the agent using CA as not yet having mastered the cardinal principle and requiring object collections to represent both addends. As such the entire process can be done with grounded quantities. This essentially involves two embodied enumerations as described above.

If the agent is working with perceptual units, there is additional non-counting time taken for object recognition, the arrangement of addends in the perceptual field, the sweep of attention across the perceptual field, and the coordination of updating the current count and “marking” each target object as having been counted. If the agent is working with figural units then some time might be saved in object recognition (since the perceptual field will retrieve prototypes), but cannot save much more of the non-counting time.

Thus, the move away from CA is just a matter of saving time combined with acquiring the cardinal principle. The cardinal principle can be represented by placing quantity arguments in result positions of the evaluation frame:

```
Evaluation(Result(CountSum,b1,b2),b3)
Collection(b1,NumSort(n3,apple))
Collection(b2,NumSort(n2,apple))
Collection(b3,NumSort(n5,apple))
```

5.4.6 CO to COF >: Metacounting

An intuitive approach to the CO to CO > transition is to consider the number of steps each takes (this was hinted at above). From the agent’s perspective this requires two abilities: (1) the ability to count self-actions (SAE as given above) and (2) the ability to attend to two counts at once, the count of targets and the “metacount” of the number of steps (counting actions).

Using SAE, Cassie metacounts the steps of count-addition. The transition can be made when the agent notices that the number of steps corresponds to the second addend. Counting on from m to n will take n steps. As with the commutative principle, the node representing n will play a double-role, as the second argument to count-addition and as the result of the metacount.

Metacounting seems to bring up attentional issues that might lead to mistakes. The agent can compensate for this in various ways. Fuson (1982) illustrates a “matching the count” technique in which the second addend is tracked raising fingers as each number is reached, as well as an auditory “double count” technique “eight, nine is one, ten is two, eleven is three, twelve is four, thirteen is five” (74).

5.4.7 CO to COF >: Duration Estimation

Another plausible explanation for the transition is that the agent gets a sense of the different time requirements for $m + n$ and $n + m$. By this I do not mean an agent using an explicit “stopwatch”, but rather applying an internal (embodied) clock to estimate an act’s duration. Meck and Church (1983) proposed an accumulator model that acts very much like such an internal clock. Ismail and Shapiro (2001) used a similar “bodily feedback” approach to timekeeping in the context of SNePS.

The duration estimation method of transition from CO to $COF >$ is modeled in Cassie as follows. A primitive act `timed-do` is written that invokes a PML timer while the agent performs an act. When the act is completed the internal timer returns a (real-valued) duration that is added to the KL. As a base-node, the numerical value has no independent KL meaning, but maintains its meaning at the PML (i.e., a bodily feeling of how long an act took).

All complex acts are performed via the `do-one` SNeRE primitive act. By default, this function nondeterministically selects between the plans for the given act. However, the user is free to rewrite this primitive in order to revise this behavior. A more deliberative *do-one* is written to select plans that the agent prefers. At the KL, Cassie has a rule that preference should be given for acts that previously had a shorter duration.

Both the implementation of `timed-do` and the deliberative re-implementation of `do-one` can be found in the Appendix. Cassie embodies the desirable features of count-addition strategy change model. Importantly, Cassie’s representation foregrounds the shift in the abstraction of targets. This shift occurs in parallel with count-addition strategy change and has not been prominently discussed in the relevant literature. Nevertheless, there are aspects of the model that could be made more realistic. For example, there is nothing in the model to differentiate counting objects in the world (e.g., apples) from counting objects on the body (e.g., fingers). The familiar and proprioceptive nature of finger-counting could also be modeled at the PML.

5.5 Summary

This chapter attempts to cover a lot of ground. The formalism of grounded quantities provides a suitable representation for embodied applications. Such applications include: embodied enumeration, self-action enumeration, and ordinal assignment to self actions. The representations developed are also sufficient in handling several commonsense reasoning tasks (e.g., the “breakfast always served” problem and “the jail sentence” problem). Finally, I considered the transitions in count-addition strategy within the larger context of the shift from embodied to abstract arithmetic. In the next chapter, I formalize the embodied and abstract aspects of the theory in an attempt to present a unified picture of Cassie’s arithmetic.

Chapter 6

A Unified View of Arithmetic

The previous two chapters showed that an agent can perform an arithmetic act using either (1) an abstract activity over a mental representation, or (2) a physical action over external objects, as well as the perceptual representatives of external objects. The purpose was to show that both abstract internal arithmetic and embodied external arithmetic can be understood computationally. However, the resulting picture is somewhat disunified. Abstract internal arithmetic proceeds without sensory input (with “eyes and ears” closed) with the agent just sequencing counting routines in the appropriate way. Embodied arithmetic is done “in the moment” and “in the world” with the agent attending to what it perceives as it acts and expecting that the act will yield a configuration of reality that can be perceived as the result.

Understanding either abstract internal arithmetic or embodied external arithmetic is a significant achievement for an agent. But, granting that a particular agent has grasped both domains, the question becomes: is this enough? Is it enough for an agent to learn and understand both of these domains without making the connection between the two? This comes down to a question of *degree* of understanding. An agent that can understand the connection between the embodied and the abstract has more understanding than one that does not. For theorists like Lakoff and Núñez (2000) however, understanding the source and target domains of a conceptual-metaphor mapping is insufficient for a full understanding of the metaphor. Take for example the metaphor “love is a journey”. Understanding the domain of human relationships and the domain of transportation is not enough to understand that, in the metaphor, love plays a role in the human relationship domain similar to a journey in the transportation domain. This is a further observation that yields deeper understanding. Moreover, the grounding metaphors of arithmetic should serve as a *tool* for an agent trying to learn abstract arithmetic. Thus, understanding the isomorphism between, say, a collection of objects in an accumulation and a number in an addition is essential.

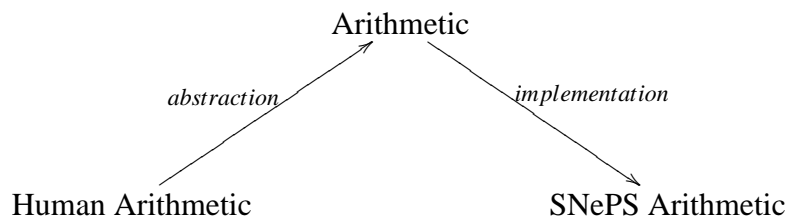
This chapter unifies and generalizes the results of the previous two chapters. I take a large part of the philosophy of embodied mathematics seriously (Lakoff and Núñez, 2000, Chapter 16) and simultaneously hold that arithmetic ability is multiply realizable (in particular, it is realizable in a computational agent). Anyone holding both of these positions should be able to give a functional analysis of embodied arithmetic and demonstrate how this analysis might yield an appropriate agent implementation. If successful, this functional analysis should fill in some details that Lakoff and Núñez (2000) leave underspecified. Although they specify the source and

target domains for grounding metaphor mappings, and although they describe the origin of the metaphors in sensorimotor actions and explain why the arithmetic procedures in both domains are effective (i.e., inference preserving across domains), they do not directly say how the agent makes the metaphoric mapping *through* action. I believe it is not enough to say that there is a mapping, but the mapping must be analyzed “as it is happening”.

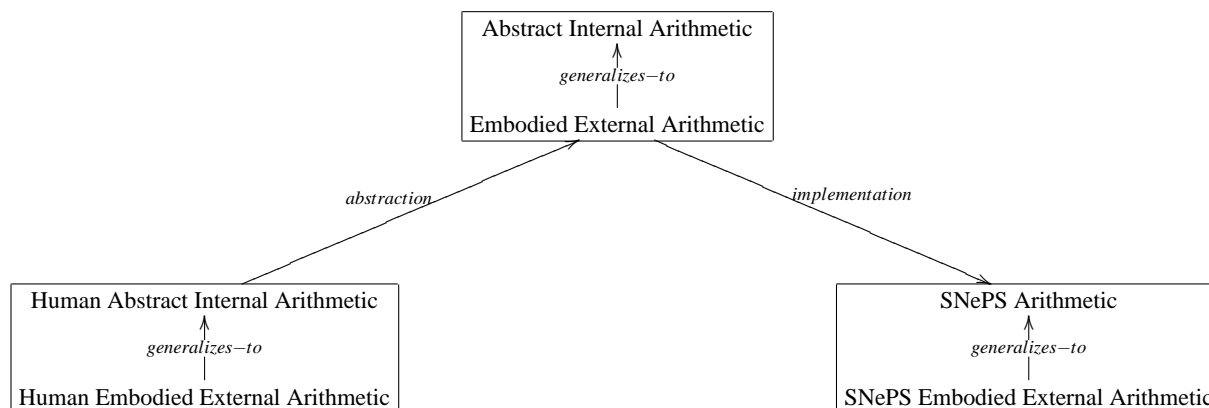
The analysis should be general enough to apply to any suitable computational implementation, not just a SNePS implementation. Thus, I will first review what I mean by the multiple realizability of arithmetic.

6.1 Multiple Realization

What does an implementation of arithmetic look like? What are the general characteristics of *any* such implementation? Recalling the discussion of multiple realizability in §3.1.1, implementation is a three-place relation: I is an implementation of an abstraction A in a medium M . This construal of implementation yields a two-step process for developing computational theories of human cognition: (1) abstraction from the activities and representations associated with the human implementation, and (2) (re)implementation of the abstraction in a computational medium. The relevant picture for arithmetic is as follows:



A crucial feature of the human case is that humans are able to generalize from embodied external arithmetic into abstract internal arithmetic. In this chapter, I attempt to show that the process of generalizing from embodied external arithmetic into abstract internal arithmetic is itself multiply realizable. This generalizability is given by a theory of human mathematical cognition. The abstraction and implementation of this generalizability is described by this diagram:



The abstraction step is to consider how a human cognitive agent generalizes from embodied experience. This invariably will include some notion of “mapping” representations from the embodied domain to the abstract domain. The implementation step requires the designer to computationally express a theory of how the mapping takes place. The implementation of this abstracted feature is influenced in part by the outlook that the designer takes towards agent design. Vera and Simon (1993) point out that embodied situated action can be given a suitable symbolic interpretation even though several researchers have claimed that symbolic AI approaches are not suitable for this task. I am sympathetic to this view, and I see no *prima facie* obstacle to using a fully-symbolic system in the analysis. This is certainly a biased view, but, at this point in computational math cognition research, I believe it is important to develop computational agents and to observe their strengths and weaknesses. Hopefully, such agents will “wear the designer’s bias on their sleeves”. Importantly, if someone is inclined to develop a non-symbolic (or partially symbolic) implementation, it will be an empirical matter to weigh the pros and cons of each implementation.

6.2 The Abstraction of Arithmetic: A Functional Analysis

The task at hand is to make precise how embodied external arithmetic becomes internally meaningful for a particular agent. This is done in the context of a fully-symbolic system from an agent-centric perspective.

6.2.1 Agent

First, we need a logical characterization of an embodied agent. The essential feature that gives an agent its “agency” is a capacity to initiate acts. As I have mentioned in previous chapters, acts are important ontologically (see §4.1.1.2) and cognitively (see §3.5.4). An agent is embodied insofar as these acts are initiated within a specific physical framework of sensory apparatus, effectors, and representation mechanisms. For the purposes of this logical characterization, I will reduce acting

to the notion of computing a mathematical function, and reduce all aspects of the embodiment to sets of symbols.

Definition 1. An **embodied agent** A is a three-tuple $\langle \mathbf{KL}, \mathbf{PML}, \mathbf{SAL} \rangle$ such that:

- **KL** is an ordered pair $\langle \mathbf{KLSYM}, KLFUNC \rangle$. In SNePS, **KLSYM** includes a set of symbols denoting beliefs, acts, and policies.
- **PML** is a seven-tuple $\langle \mathbf{PMLSYM}, PMLFUNC_k, PMLFUNC_s, PMLFUNC_e, PERCEPTS, \mathbf{MODALITIES}, ALIGN \rangle$.
- **SAL** is a four-tuple $\langle \mathbf{SENSORSYM}, \mathbf{EFFECTORSYM}, SALFUNC_s, SALFUNC_e \rangle$.
- **KLSYM**, **PMLSYM**, **SENSORSYM**, and **EFFECTORSYM** are mutually disjoint sets of symbols.
- $KLFUNC$ is a set of partial functions from **KLSYM** to **PMLSYM**. In SNePS, $KLFUNC$ is a set of primitive acts.
- $PMLFUNC_k$ is a set of partial functions from **PMLSYM** to **KLSYM**. In SNePS, $PMLFUNC_k$ is a set of functions performed by invoking the `tell-ask` interface from Cassie’s PML (i.e., unconscious inferences).
- $PMLFUNC_s$ is a set of partial functions from **PMLSYM** to **SENSORSYM**. In SNePS, $PMLFUNC_s$ is a set of Lisp functions responsible for initiating active perception (e.g., looking, listening).
- $PMLFUNC_e$ is a set of partial functions from **PMLSYM** to **EFFECTORSYM**. In SNePS, $PMLFUNC_e$ is a set of Lisp functions responsible for invoking the activation of effectors (e.g., grasping, moving, pushing).
- $SALFUNC_s$ is a set of partial functions from **SENSORSYM** to **PMLSYM**. In SNePS, $SALFUNC_s$ is a set of hardware-specific functions responsible for sensation or passive perception (e.g., seeing, hearing).
- $SALFUNC_e$ is a set of partial functions from **EFFECTORSYM** to **PMLSYM**. In SNePS, $SALFUNC_e$ is a set of hardware-specific functions responsible for proprioception or other forms of bodily feedback resulting from action.
- $PERCEPTS$ is a binary relation on **PMLSYM**. I.e., $PERCEPTS \subset \mathbf{PMLSYM} \times \mathbf{PMLSYM}$
- $\mathbf{MODALITIES} \subset \mathbf{PMLSYM}$. In SNePS, **MODALITIES** is a set of distinct “channels” of sensory information available to an agent.

- *ALIGN* is a partial function from **KLSYM** to *PERCEPTS* × **MODALITIES**. In SNePS, *ALIGN* is the alignment table; a representation of the perceived entities an agent is actively attending to.

Figure 6.1 provides an illustration of the functional relationships between the sets of symbols. The circles in the figure represent the symbol sets and the block arrows represent the sets of partial

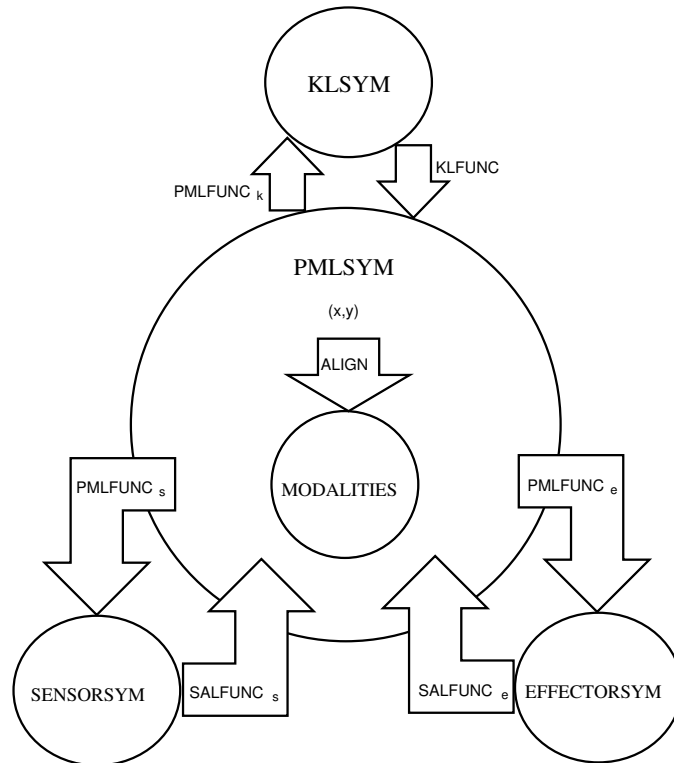


Figure 6.1: A functional description of GLAIR.

functions. This is a model of GLAIR agents stripped of the typical interpretations of the layers.¹ What remains is just the functional relationship of the layers to one another. GLAIR suggests a methodological dualism for the agent designer in which the “mind” is associated with the KL and the “body” is associated with the union of all of the PML layers and the SAL. However, insofar as these are just sets of symbols, partial functions, and relations, a monist interpretation is possible by considering all of these in a single set. In fact, the number of “layers” is also generalizable.

6.2.2 Embodied Experience

I now examine the logical structure of certain embodied acts that serve as the originating experiences for numeracy. The first problem we must face is how to describe what it means for an

¹For example, there is no distinction made between KL symbols denoting acts, propositions, or policies. There is no interpretation of attached primitive actions.

embodied agent to perform an act on external objects. Is our logical language supposed to capture the subjective experience or the objective fact? As I stated above, I want to perform this analysis from the perspective of an agent. This entails that some aspect of the originating experiences may be vastly different between subjects and perhaps even prone to error, but, nevertheless, such experiences are part of an agent’s history. It is the totality of this history that determines the abstractability of embodied arithmetic.

First we must describe what an embodied act is for a particular agent. I distinguish between two species of embodied acts, affecting acts and sensing acts:

Definition 2. Let $A = \langle \mathbf{KL}, \mathbf{PML}, \mathbf{SAL} \rangle$ be an embodied agent. Let $a \in \mathbf{KLSYM}$ denote α for A . Then α is an **embodied affecting act** for A if and only if:

- $\exists f, g (f \in \mathbf{KLFUNC} \wedge g \in \mathbf{PMLFUNC}_e \wedge g(f(a)) \in \mathbf{EFFECTORSYM})$.

and α is an **embodied sensing act** for A if and only if:

- $\exists f, g (f \in \mathbf{KLFUNC} \wedge g \in \mathbf{PMLFUNC}_s \wedge g(f(a)) \in \mathbf{SENSORSYM})$.

α is an **embodied act** for A if and only if α is an embodied affecting act or an embodied sensing act (but not both).

The agent is **aware of the embodied act** α if and only if, in addition to α being either an embodied affecting act or an embodied sensing act, the following condition also holds:

- $\exists p, h, i (p \in \mathbf{SENSORSYM} \wedge h \in \mathbf{SALFUNC}_s \wedge i \in \mathbf{PMLFUNC}_k \wedge i(h(p)) \in \mathbf{KLSYM})$

The agent **performs the embodied act** α if it computes $g(f(a))$ and is **aware of its performance of the embodied act** α if it also computes $i(h(g(f(a))))$.

α does not denote any symbol in the object language (i.e., it is not a member of any of the sets specified in Figure 6.1). Rather α is a metalanguage symbol (for the purposes of this functional analysis) that denotes the computation of $i(h(g(f(a))))$ by agent A . The performance of an act is the computation of a function.

The performance of an act for an embodied agent requires a functional “path” through the embodiment. A distinction is made between acts in which the agent is affecting the external world and those in which the agent is sensing the external world. The performance of an embodied act requires only a “downward” functional path through the layers (i.e., computing $g(f(a))$) while the awareness condition requires a “downward” path, to the invoke the body to perform an act, followed by an “upward” functional path, to sense that the act was performed (i.e., computing $i(h(g(f(a))))$). Hexmoor et al. (1993) call the “downward” functional path the “control flow”, and the upward functional path the “data flow”.

Despite this elaboration, we are still missing a fundamental feature of embodied action, namely, the set of external objects that the agent affects and senses. Strictly speaking, an agent’s embodied action operates over internalized representations of external objects. As Kant might say, these are phenomenal objects, and there remains a gap between an internalization that has been “filtered”

through our perception and the noumenal object qua constituent of reality. However, a theory of mathematical cognition is essentially unaffected by how we interpret its objects. For the agent, there is no distinction between external embodied experience and what we might call *internal* embodied experience. Both involve operations over phenomenal objects. Whether those objects are simulations or constituents of reality (i.e., noumenal) will not affect the agent’s understanding. There are obviously more things to say about the nature of phenomenal objects, but for now, let us assume that they are internalized symbols that result from embodied sensing acts.

Before formalizing a phenomenal object, there is a small notational issue to address. One problem concerning the set-of-symbols formalization of the layers of GLAIR is that sets are statically defined. Often, as a result of perception, Cassie’s PML will contain new items. Thus, a proper formalization of the PML should include time-indexed sets such as $\mathbf{PMLS\!Y\!M}_{t_0}$ and $\mathbf{PMLS\!Y\!M}_{t_1}$ for representing the contents of the PML at time t_0 and at time t_1 respectively. However, I will avoid this notation and assume that a set like $\mathbf{PMLS\!Y\!M}$ can grow or shrink in size without changing its name. Importantly, this allows us to include the addition of a symbol to $\mathbf{PMLS\!Y\!M}$ as part of the definition for phenomenal object.

Definition 3. ω is a **phenomenal object** for an embodied agent A if and only if, as a result of performing some embodied sensing act, $\omega \in \mathbf{PMLS\!Y\!M} \wedge \exists p(\omega = i(p) \wedge i \in \mathbf{PMLFUNC}_k)$.

Now it is possible to broadly characterize the token instances of the originating experience. I will distinguish particular experiences using the subscript p .

Definition 4. A **particular embodied experience** E_p is three-tuple $\langle A, T, \alpha(\Omega) \rangle$ where A is an embodied agent, α is an act (expressed as a function of arity n) performed by A on a set of phenomenal objects $\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$ (such that ω_i corresponds to argument i for α) over time interval T .

To be used in a meaningful way for arithmetic, a particular embodied experience must correspond to one of the source domains of grounding metaphors (see Lakoff and Núñez (2000)); i.e., α must be an instance of object collection, object construction, measurement, or motion along a path. Furthermore, the elements of Ω should not only be phenomenal objects, but should correspond to grounded objects in the sense developed in the last chapter. As such, they should be able to serve as grounded quantities in the formal sense developed in §5.1. Functionally, a grounded object may be described as follows:

Definition 5. A **grounded object** o is a four-tuple $\langle b, a, v, m \rangle$ such that $b \in \mathbf{KLS\!Y\!M}$, $\langle a, v \rangle \in \mathbf{PERCEPTS}$, $m \in \mathbf{MODALITIES}$, and $\mathbf{ALIGN}(b) = \langle \langle a, v \rangle, m \rangle$.

We now need a way to represent how an agent forms classes of similar embodied experiences that it can generalize from. I will use the subscript g to indicate that an experience is generalized. This can be broadly characterized as follows:

Definition 6. A **generalized embodied experience** E_g is a three-tuple $\langle A, EP, G \rangle$ such that A is an embodied agent, EP is a set of particular embodied experiences and G is an equivalence relation on EP .

The equivalence classes of G will serve as the *types* of experiences that correspond to the source domains of grounding metaphors. What is abstracted away is the time interval in which the experience EP took place and the particular objects involved in those experiences (as well as the sortal predicates applied in each particular experience). This gives us some indication of what G should look like. It should, for example, relate two particular experiences if the acts involved enumerations that resulted in the same cardinality using the same sensorimotor structure. This can also be the basis of a general “algorithm for α ” (i.e., an act type that is effective for all objects).²

6.2.3 From Embodied Experience to Abstract Facts

In the previous section, I traced a functional path over the gap between the noumenal objects of the real world and the phenomenal objects in an agent’s perception. Phenomenal objects make external data available for internal manipulation. Acting on phenomenal objects yields a set of particular experiences that are further abstracted into generalized embodied experience. Now, to form a unified theory, we have to cross a different gap: how do generalized embodied experiences become associated with arithmetic facts?

In the framework of this chapter, we are looking for a certain “bridge” function:

- Let $EG = \{E_{g1}, E_{g2}, \dots, E_{gk}\}$ be a set of generalized embodied experiences.
- Let $P \subset \mathbf{KLSYM}$ be a set of propositions representing arithmetic facts.
- Let $BRIDGE$ be a function from EG to P .

The whole process is illustrated in Figure 6.2. The leftmost boxes in the figure represent particular embodied experiences. These are generalized into the equivalence classes of G shown in the center of the figure. These generalized experiences are abstracted into “ $3 + 2$ ” by $BRIDGE$.

It is clear from the figure that $BRIDGE$ is not a one-to-one function, because different generalized experiences can map to the same arithmetic fact. $BRIDGE$ is also not an onto function. This is because some arithmetic facts (e.g., $-3 \times -3 = 9$) do not have a clear embodied correlate and thus are not mapped to by any generalized embodied experience. This is because arithmetic facts such as $-3 \times -3 = 9$ are not simply the result of abstracting from the grounding metaphors (represented in my theory as EG , the domain of the $BRIDGE$ function), but rather the result of applying conceptual blends and linking-metaphors to the originating experiences (Lakoff and Núñez, 2000). Thus, the range of $BRIDGE$ represents the subset of arithmetic facts abstractable from the grounding metaphors alone.

A theory that unifies embodied external arithmetic and abstract internal arithmetic must spell out the details of this $BRIDGE$ function. The previous two chapters can be seen as spelling some

²In undertaking such an analysis, I am implicitly assuming that no phenomenal objects arise out of “direct experience” (against the views expressed by Brooks, Gibson, and, to a lesser degree, Barsalou). As such, every experience is filtered through internal processing and mediated by internal representations. Notice, however, that phenomenal objects occurring at the PML are in some sense “more directly” experienced than those occurring at the KL, since the former require the composition of fewer functions.

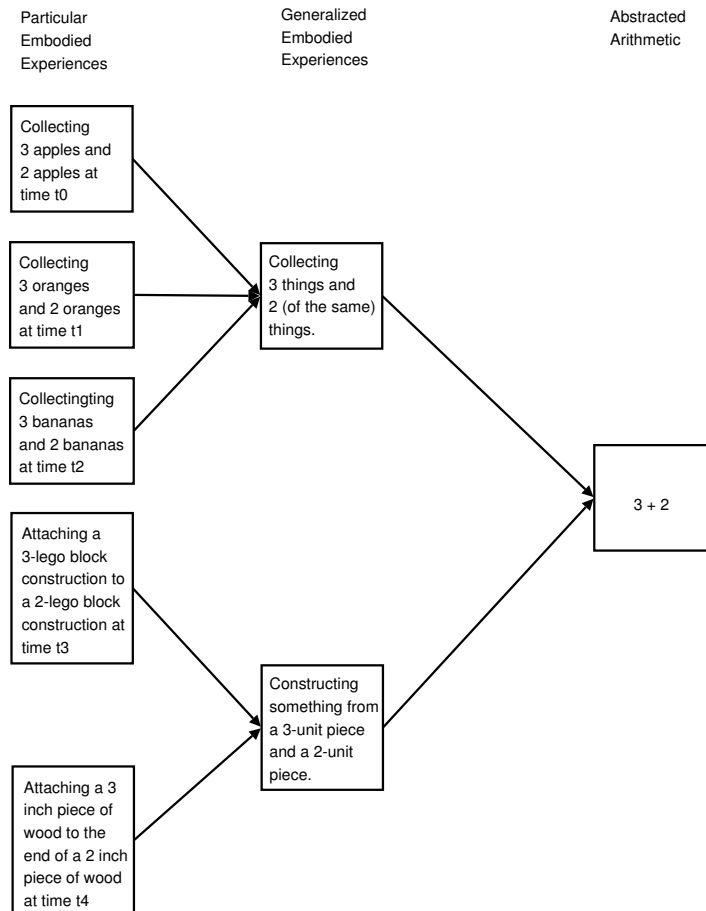


Figure 6.2: The generalization of “3 + 2”.

of this out for a SNePS implementation. Implicit in this is that there is some work for the *BRIDGE* function to do. One could take a reductionist position and claim that all mathematics is just embodied external mathematics. However, this does not account for the propositions of higher mathematics, which express truths that are very difficult to express using just embodied activities. Consider the concept of GCD. One could view GCD in a completely embodied way as follows: The GCD of natural numbers x and y is the maximum number of objects that can be evenly shared among both a group of x people and a group of y people.³ However, suppose the task was to show that the GCD of two Fibonacci numbers was equal to the Fibonacci number indexed by the GCD of those numbers:

$$gcd(F_m, F_n) = F_{gcd(m,n)} \text{ (with } F_1 = 1, F_2 = 2, F_n = F_{n-1} + F_{n-2})$$

In thinking about this task, it becomes very difficult (if not impossible) to attend to the object-

³Note that this formulation uses only the “basic” sortals OBJECT and PERSON suggested by Xu (2007) to be innate.

sharability conception of GCD. This suggests that, after *BRIDGE* has mapped a generalized embodied experience to an arithmetic fact, the generalized embodied experience can be (at least temporarily) abandoned.

6.2.4 *BRIDGE* in SNePS

The specifics of the *BRIDGE* function can be described in various ways, including the use of cognitive theories of conceptual metaphor or analogy. Importantly, there is nothing to suggest that these theories cannot be expressed computationally.

In theory, the bridge from embodied arithmetic to abstract arithmetic for Cassie is easily specified. The abstract arithmetic routines of Chapter 4 are implemented separately from the embodied arithmetic routines of Chapter 5. This means that Cassie can perform a particular count addition over numerons that are constituents of grounded quantities or over numerons that are not. The performance of the `CountAdd` act is blind to whether or not its arguments are just base nodes in the progression formed by `Successor`, or if they are base nodes in such a progression *and* playing the number role in several grounded quantities. Thus, Cassie can abstract away from embodied arithmetic by “ignoring” any sub-networks pertaining to grounded quantities. This is illustrated in Figure 6.3. By ignoring the embodied representation, Cassie is working with just the

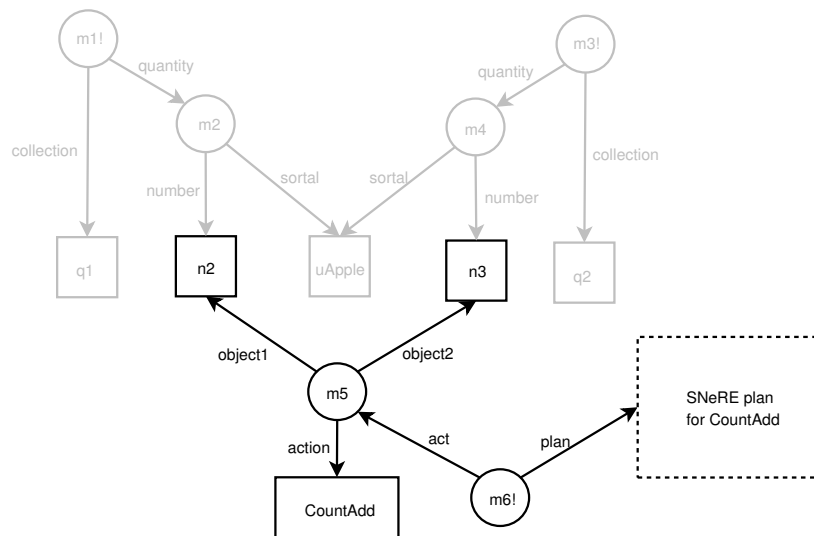


Figure 6.3: The act `CountAdd` ignores the (grayed out) embodied representation.

numerons. This effectively cuts the tie with perception and allows her to act solely on the basis of the numerons.

Although developing such an attention model (in which part of the network is ignored) is beyond the scope of this dissertation, it is possible to give a sketch of how it could be implemented in SNePS. Throughout this work I have assumed Cassie uses a single belief space. However, SNePS provides the mechanism of contexts as a way of partitioning the belief space. Cassie could create a context of abstract arithmetic *from* a context of embodied arithmetic by copying only those

representations that are needed for the routines in Chapter 4. If, after doing this, Cassie places the embodied context in some form of long-term storage (e.g., a database backend), then Cassie would get the desired performance-boost from her “trimmed down” KL. It is important to emphasize that Cassie should only “temporarily” ignore her embodied representations (i.e., the long-term storage should be accessible), since they allow her to appeal to embodied experiences when explaining herself.

6.3 The Ontogeny of Sortals

In the previous chapter, Cassie was assumed to have a set of sortals she could apply to phenomenal objects and a set of names for these sortals. Language clearly plays a role in categorization. The sortal terms we use are the ones our language gives us. A long game of trial and error in the application of terms may be needed before an agent’s usage conforms to social conventions. Thus, even an agent that is fluent in a language must perpetually calibrate the way in which it applies sortal terms.

However, if the psychological math-cognition literature is accurate (§2.1.1), then humans begin performing embodied arithmetic well *before* they have linguistic names for all of the sortals they need. In this section, I briefly consider the pre-linguistic origin for a sortal. In particular, we need a model of how the base nodes representing sortals come to be in Cassie’s KL.

Xu (2007) suggests that some very basic sortals may be innate. These include “objects” and “persons”. In a way, these are handled innately by Cassie via the “object” and “agent” arc-labels used throughout the implementation. How about more complex sortals? Can they arise before language is acquired? If so, can this acquisition be modeled computationally?

To answer this last question, we need to consider what might turn an otherwise uncategorized phenomenal-object into an instance of a universal. Given a set of distinguishing features for a particular sensory modality, any phenomenal object will exhibit some distribution of these features. Many computational systems store these features as a feature vector in a “feature space”. The metaphor of representing phenomenal objects in a feature space replaces the vague notions of similarity and resemblance with the precise (and computable!) notion of metric distance. A set of examples of a given sort (often called exemplars in the literature) will be more similar to other members of the sort and thus will “cluster” around each other in the feature space.

A cluster of exemplars is likelier to represent a category the closer the exemplars are to each other (i.e., a low within-sort scatter) and the closer the mean value of the cluster in feature space is from the means of all other clusters. The former property increases exemplar resemblance, and the latter increases the discriminability of a given sort from other sorts.

These two properties are captured in the notion of Fisher linear discriminants (Fisher, 1936). This is one of many sophisticated pattern classification techniques (Duda, Hart, and Stork, 2001). The goal of this method is to separate exemplars from two proposed sorts using a linear function that will partition the feature space. Suppose the proposed sorts are apples and bananas. Furthermore, suppose Cassie has a sortal concept for apples but none for bananas. We are interested in the ontogeny of Cassie’s banana concept. If C is the entire set of exemplars and suppose, for simplicity,

that $C = A \cup B$ where A is the set of apple exemplars and B is the set of uncategorized exemplars (these are uncategorized for Cassie; we know them to be bananas). Additionally, suppose that $|A| = |B| = n$ and that A and B are disjoint (i.e., $A \cap B = \emptyset$). Define the *mean* of the cluster of apples as $m_a = \frac{1}{n} \sum_{x \in A} x$ and the *scatter* of the apple exemplars as $s_a = \sum_{x \in A} (x - m_a)^2$. Similarly, define the mean of the cluster of uncategorized exemplars as $m_b = \frac{1}{n} \sum_{x \in B} x$ and the scatter of the uncategorized exemplars as $s_b = \sum_{x \in B} (x - m_b)^2$. Now we can find the linear discriminant by maximizing the function:

$$\frac{|m_a - m_b|^2}{s_a + s_b}$$

This value increases as the denominator decreases and as the numerator increases. A decreasing denominator indicates that the total scatter of apples and uncategorized exemplars is low (i.e., high resemblance). An increasing numerator indicates that the distance between the two means of the clusters is large (i.e., high differentiability). The distribution of exemplars in feature space might look something like Figure 6.4. The apple exemplars congregate around red on the color axis,

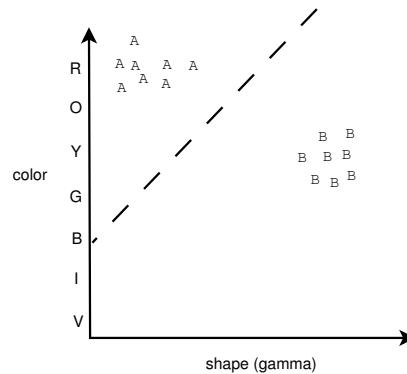


Figure 6.4: Linear discrimination of apple exemplars **A** from uncategorized exemplars **B**.

whereas the uncategorized exemplars congregate around yellow. Along the shape axis, the apple exemplars congregate around a high circularity (γ as specified in the previous chapter), whereas the uncategorized exemplars congregate around a low circularity.

The linear discrimination procedure can be iterated. Cassie can pit the uncategorized exemplars against every set of known-sortal exemplars in her KL. If, after such an iteration, it turns out that the exemplars occupy a region of feature space that is linearly separable from each known sortal, then she should create a symbol representing a new sortal. As a KL base-node, this symbol would initially represent objects in Cassie’s perceptual field that are distinct from apples, oranges, grapes, plums, Later, Cassie could associate the linguistic term “banana” with this sortal, but all of the embodied functionality of the sortal is available pre-linguistically. She has used the cluster’s discriminability in feature space as the impetus for creating a new node.

Of course, answering the question of sortal origin in this way brings up questions regarding the origin of the distinguishing features. Where do the distinguishing features that make up the axes in feature space come from? I believe this is a more fundamental question than the question of sortal origin because it brings us closer to the nature of the sensory apparatus and the mode of presentation for sensory data. Indeed, the fact that shape and color are distinguishing features is one of the fundamental ways in which vision differs from the other sensory modalities. Computationally answering the question of the origin of distinguishing features is beyond the scope of this dissertation.

As we saw in the previous chapter, acts construed as universals can also be sortals. The method of linear discrimination applied to a feature space of sensorimotor features can serve as the impetus for the introduction of act sortals. In fact, such an account can help explain how the equivalence relation G from the previous section works. G relates two embodied acts α_1 and α_2 if they correspond to the same grounding metaphor. An agent might notice that collection acts frequently require the grasping and moving of objects, construction acts require the gluing or snapping together of objects, and measurement acts require the lining up of a measurement object and its target. If these primitives are taken as features, the similarities between, for example, instances of collection acts will yield a dense cluster in feature space.

6.4 Summary

In this chapter, I have attempted to formalize and unify the results of the previous two chapters. The GLAIR architecture used in the implementation of CASSIE was presented abstractly in terms of sets of symbols and partial functions between those sets. Under this characterization, the notions of embodied agent, embodied action, and embodied experience were spelled out. The abstractability of embodied arithmetic is captured in the *BRIDGE* function. An account of sortal origin was sketched using the method of Fisher linear discriminants.

Chapter 7

Conclusion and Further Work

There are two primary results of this dissertation: (1) a theory of early mathematical cognition inspired by the relevant cognitive-science literature, and (2) a computational implementation guided by the theory. In this chapter, I summarize the claims of the theory and describe how each of the claims is addressed in the SNePS-agent implementation. I then consider the significance of these results (from both a cognitive science and an artificial intelligence perspective). Potential objections to the multiple realizability of mathematical cognition are addressed. A series of natural extensions to the computational theory (both representational and operational extensions) are presented as avenues of further study. These extensions are shown to be coherent with the basis of arithmetic and counting presented in this work.

7.1 Summary

The claims made by my theory were first presented in §3.1. The implementation of Cassie, as described in the previous three chapters, addresses the claims as follows:

- **Claim 1: A portion of early mathematical reasoning is multiply realizable.** The next ten claims on this list each can be expressed within the framework of the SNePS KRRRA system and GLAIR architecture. Every mathematical ability granted to Cassie is an ability that does not rely on the human medium for its implementation.
- **Claim 2: Mathematics is embodied.** The GLAIR architecture is deployed as a model of Cassie’s embodiment. In chapter 5, I describe some representations and mechanisms required for embodied activities (e.g., grounded quantities, a list of prototypes, and an alignment table), and an implementation of such an activity (embodied enumeration). This activity demonstrates how mathematical results can flow from embodied perceptions.
- **Claim 3: Mathematics is represented publicly and privately.** Cassie has representations for her private “objects-of-thought” (e.g., numerons, pre-linguistic sortals). These symbols are associated with publicly-accepted names (e.g., numerlogs, sortal names).

- **Claim 4: Mathematical reasoning utilizes extended cognition.** Cassie has access to a Lisp calculator that she can use to obtain results. The results given by this external tool can be integrated seamlessly with Cassie’s belief space (i.e., calculator-addition is treated as a way of adding, and the numeral answers given by the calculator imply the relevant numeron answers).
- **Claim 5: Quantities can be usefully considered as a complex consisting of a number and a sortal.** The formalism of a grounded quantity is developed in chapter 5. The philosophical notion of a sortal is given a precise computational definition.
- **Claim 6: An agent must be able to shift between operational and structural meanings to understand mathematics.** The method of procedural decomposition used in exhaustive explanations requires Cassie to make inferences based on both the “structural” evaluations of her acts (represented as SNeRE `Effects`), and the “operational” procedures for performing those acts (represented as SNeRE `ActPlans`).
- **Claim 7: Numbers obtain meaning through the act of counting.** The four fundamental arithmetic operations can be expressed in terms of count-based routines. Moreover, the technique of procedural decomposition can be applied in understanding arithmetic acts *on the basis* of counting routines. The `GreaterThan` relationship can be inferred from the structure Cassie generates during counting. In chapter 4, the complex act of finding GCDs is shown to “bottom out” at counting routines.
- **Claim 8: Mathematical reasoning utilizes metacognition.** Cassie is able to select between various plans for performing the same act. Using a re-implemented `do-one` primitive act, she can make a deliberative decision in selecting an appropriate plan on the basis of a metacognitive preference (i.e., using the `Prefer` predicate).
- **Claim 9: Understanding can be treated as an analyzable gradient of features.** The kinds of inferences an agent makes are evidence for different kinds of understanding. Understanding, in this sense, is a matter of degree. The techniques of procedural decomposition, conceptual definition, and embodied action each correspond to a different component of Cassie’s understanding.
- **Claim 10: The ability to justify results through the method of exhaustive explanation demonstrates mathematical understanding.** A Turing-test style question-and-answer dialogue is determined to be a useful way of probing an agent’s mathematical understanding. An interrogator can conduct such a dialogue in a restricted subset of English by invoking the `answerQuestion` act. Based on the type of question, this triggers the appropriate form of justification.
- **Claim 11: The ability to apply quantitative reasoning to non-mathematical domains demonstrates mathematical understanding.** The representations Cassie uses for mathematical reasoning are shown to be applicable in service of basic reasoning tasks, including:

self-action enumeration, ordinal assignment to self-action, reasoning about vague relations, and word-sense disambiguation.

7.2 Significance

A theory is only as significant as the evidence that supports it. A computer program is unique in that it can both express a theory and serve as evidence for it. I have sought to provide this kind of support for the theory by applying the methodology of agent building.

The theory and implementation are significant from a computational perspective for several reasons. By computationally modeling human behavior in the SNePS formalism, with its particular language and logic, I was forced to flesh out the details of the representations and routines corresponding to the claims of the theory. In any such effort there are necessarily trade-offs of functionality for faithfulness to the human psychological data. However, the precision required for agent programming is one of the great benefits of developing a computational theory, because it yields a concrete platform for experimental investigation. Johnson-Laird (1981) puts this point as follows:

There is a well established list of advantages that programs bring to a theorist: they concentrate the mind marvelously; they transform mysticism into information processing, forcing the theorist to make intuitions explicit and to translate vague terminology into concrete proposals; they provide a secure test of the consistency of a theory and thereby allow complicated interactive components to be safely assembled; they are "working models" whose behavior can be directly compared with human performance. Yet, many research workers look on the idea of developing their theories in the form of computer programs with considerable suspicion. The reason...[i]n part...derives from the fact that any large-scale program intended to model cognition inevitably incorporates components that lack psychological plausibility The remedy . . . is not to abandon computer programs, but to make a clear distinction between a program and the theory that it is intended to model. For a cognitive scientist, the single most important virtue of programming should come . . . from the business of developing [the program] (pp. 185–186).

Developing computational theories also tests the fundamental mechanisms of an architecture which strives, as SNePS/GLAIR does, to be a general cognitive architecture and KRRA system. SNePS holds up well in this respect, because no great "under the hood" modifications of the system were required. The implementation addresses the extent to which SNePS/GLAIR (in its current state) can be applied to a theory of early mathematical cognition. Having said this, it is also worth mentioning that future versions of SNePS will include a more powerful logic and an improved acting system. Also, the next generation of GLAIR, called MGLAIR, will include a dedicated mechanism for handling asynchronous input from Cassie's various sensory modalities. These developments will be a great benefit to further implementing and expanding the theory.

The theory is significant from a psychological standpoint because it utilizes cognitively plausible representations and broadly applicable reasoning methods. The theory also includes an explicit model of embodiment. Models of mathematical reasoning seldom incorporate explicit representations of both embodied perception and logical deduction under the umbrella of a single system.

The theory is linguistically significant in several respects. A GCD algorithm inspired by the natural-language semantics of “greatest” and “common” was found to be both the most common “back-of-the-envelope” technique subjects used, and the most easily connected with an exhaustive explanation. The method of conceptual definition, traditionally applied in the natural-language setting of contextual-vocabulary acquisition, was found to be sufficient for definitions of mathematical terms. The private language and public language distinctions were made throughout the work.

The implementation of Cassie is philosophically significant in that it serves as an empirical counterexample to the claim made by Lakoff and Núñez (2000) that mathematical understanding cannot be detached from the human embodiment and implemented in a computational agent. Such a claim echoes similar claims made by Searle (1980), Dretske (1985), Fetzer (1990) and Hauser (1993). For various reasons, none of these philosophers is likely to see Cassie as a counterexample to their philosophical positions (see §7.3 below), but I believe putting candidate implementations on the table furthers the (otherwise murky) discourse.

Finally, the theory is significant from an educational perspective because it computationally illustrates the effect of individual differences on a student’s performance and overall understanding. Cassie is very much a product of her “lessons” in that her performance is determined by categories of acts (e.g., syntactic, semantic, and extended). Additionally, the order in which these acts are presented, and the particular problems she is given to solve will impact her performance during explanation.

7.3 Attacks on Multiple Realizability

As mentioned in chapter 6, I have taken the view that an implementation relies on an abstraction and a medium. Figure 7.1 illustrates five kinds of objections that might be raised against the claim of multiple realizability of arithmetic. I will address each of these in turn.

(A) *Some essential feature of human arithmetic is not abstractable.* One could object that some fundamental part of arithmetic cannot be expressed computationally and, thus, is simply uncomputable. The response to this sort of objection may vary depending on the feature in question. However, given that computation seems to be at the heart of mathematical reasoning, another strategy might be to argue that the feature in question is not really essential (and thus need not be part of the abstraction).

Fetzer (personal communication) gives an objection along these lines. He has said:

[The] shapes, sizes, and relative locations [of the marks computers manipulate] exert causal influence upon computers but do not stand for anything for those systems.

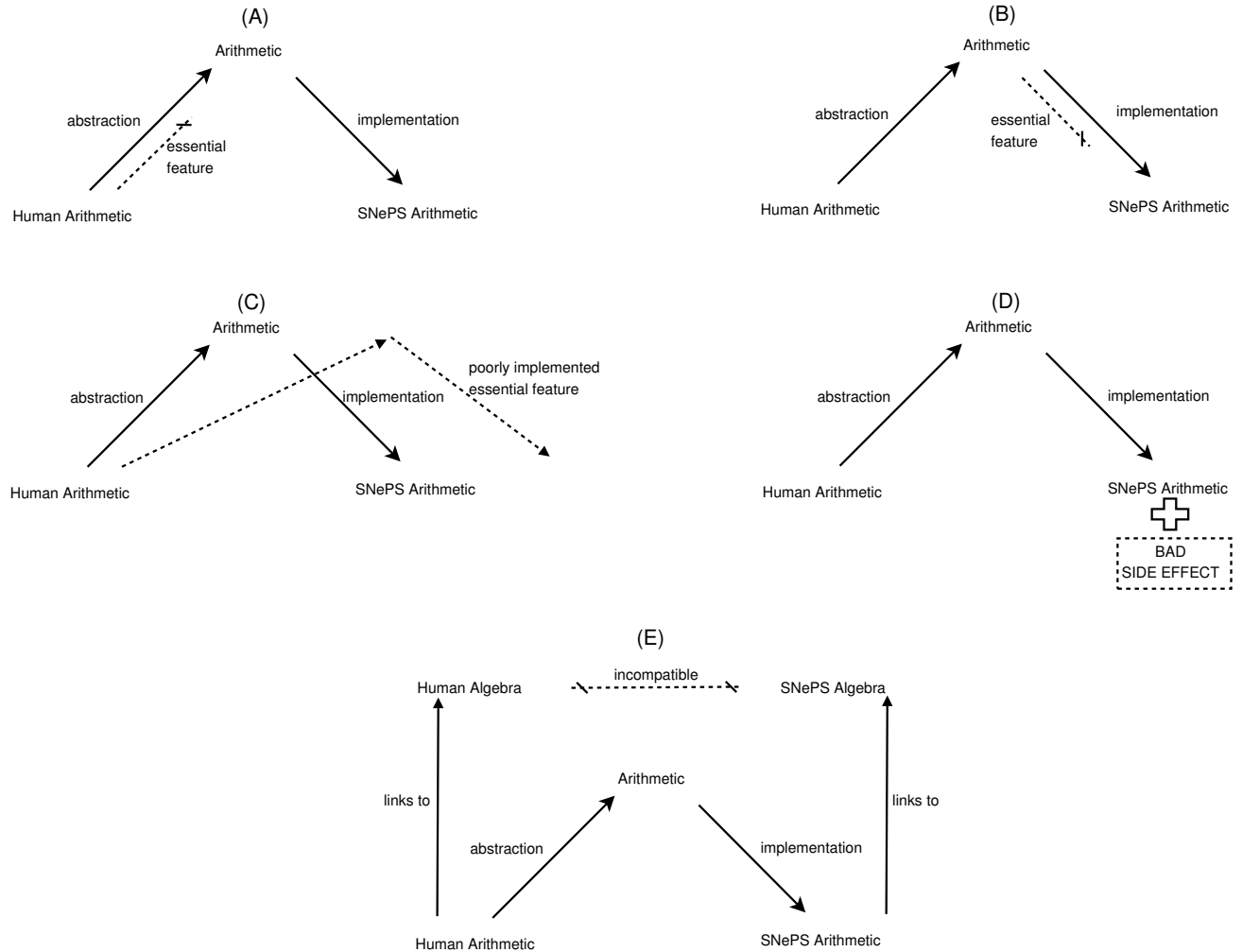


Figure 7.1: Classes of objections to multiple realizability.

This is a conjunction of two claims. I will grant the first part of the conjunction. Computers do function on the basis of the “shapes, sizes, and relative locations” of the marks being manipulated. However, Fetzer attaches a sense of “mere” symbols to such marks and “mere” symbol manipulation to computations over such marks.¹ However, there is nothing “mere” about Cassie’s symbols. They are symbols grounded in action and perception. In this sense, they are symbols *with a history* of how they came to stand for something for Cassie. Which brings us to the second part of Fetzer’s conjunction. I would offer the following counterargument to the implication that Cassie’s symbols do not stand for anything for her:

1. *Structuralist ontology of numbers.* The natural numbers a cognitive agent refers to are denoted by a progression of unique symbols that are ordered by the successor relation. As

¹The concrete evidence suggests that humans essentially function on the basis of the “shapes, sizes, and relative locations” of neuron firings in the brain

such, these symbols exemplify a finite initial segment of the natural-number structure.

2. *Generation through counting.* A finite initial-segment of the natural-number structure can be generated by Cassie during the act of counting.
3. *Meaninglessness for human user.* Each symbol in such a structure may be machine-generated using the Lisp `gensym` function. The resulting symbols (e.g., b4532, b182, b9000) will have no meaning for a human user who does not have access to how they are ordered.
4. *Numeron-Numerlog associability.* Such private computer-generated symbols (numerons) are associable with publicly meaningful symbols (numerlogs) (e.g., b4532 denotes the same number as the numeral '1', b182 denotes the same number as the numeral '2', b9000 denotes the same number as the numeral '3', etc).
5. *Sufficiency of numerons.* Cassie can do mathematics solely on the basis of her private numerons. For example, she can compute $\text{gcd}(3,2)=1$ as $\text{gcd}(b9000,b182)=b4532$.

From these premises, we can derive two conclusions:

1. Cassie's symbols stand for something *for her*.
2. Despite the fact that the results obtained in premise 5 are meaningless to us, premise 4 allows us to check whether they are correct because Cassie associates these symbols with public symbols we can recognize (e.g., with patterns of pixels on a monitor that we can recognize as Arabic numerals).

We now turn to the second kind of objection:

(B) *Some essential abstracted feature is not implementable.* This objection is one against the implementation platform. In the case of my theory, the claim would be that the feature in question cannot be implemented in the SNePS formalism. Given that SNePS (along with SNeRE) is Turing-complete, the objection collapses into a restatement of objection (A). Another way to address this is to "shop around" for a different formalism.

It is worth saying a few words about why my theory is implemented using the symbolic-AI paradigm in general and the SNePS/GLAIR system in particular (as opposed to say, artificial neural-networks). Exhaustive explanation, as I describe it, requires that the agent can traverse its "lattice" of beliefs, specifically, its beliefs about its own acts and beliefs formed during action. In order to do this, these beliefs must persist and be recoverable after action. The neural-network paradigm utilizes a training procedure that, destroys the "memory" of the network's beliefs. Weights in a neural network are adjusted according to a test set of patterns, but there is no "memory" of how the weights arrived at certain values. Even supposing that such a memory mechanism can be devised, it would be hard to recover "answers" from weights or to formulate questions to be asked of a set of weights. Polk et al. (2001) observes:

Proposals regarding conceptual number knowledge are not as explicit about the nature of the underlying representation, but descriptions of the knowledge itself suggests a

representation that is more symbolic than could be supported by a representation of analogue magnitude. For example, it is hard to imagine how an analogue number line representation could represent the fact that multiplication corresponds to repeated addition (p. 548).

It is precisely this sort of knowledge that needs to be accessible during an explanation, and it is precisely the symbolic representations that deliver it.

(C) Some essential feature is incorrectly implemented. This objection is the easiest to address. Either the abstraction of the feature must be corrected (work for the cognitive scientist) or the implementation must be corrected (work for the computer scientist).

(D) There is a bad side-effect in the implementation. If early computational math cognition was only possible at the expense of a highly undesirable (or unnatural) implementation detail (e.g., a very large lookup table for question answering), then the entire theory could be objected to. A possible response is to claim that the side-effect of the implementation is actually found in the human case as well. Indeed, this would be an occasion of further discovery and one of the reasons we build models. Another approach is to bite the bullet and claim that the undesirable feature is simply a necessary by-product of a computational implementation.

(E) The implementation does not scale-up properly. Even with a good implementation of arithmetic, the theory may not scale up to be a useful part of higher mathematics (e.g., in a computational theory of algebra). Lakoff and Núñez (2000) contend that, after the grounding metaphors of arithmetic are established, linking metaphors serve to connect arithmetic with the other branches of mathematics. This would definitely be a serious objection, but it is not clear why the theory would not scale-up properly with the right sort of (learning) algorithms. Or, perhaps, the theory might scale up with some (small) modification (or generalization).

A slightly weaker version of this objection might be that the theory does not scale-”across”, i.e., that the GCD implementation described in chapter 4 was somehow “rigged” or “biased” in some way so as to make the claims of my theory trivially true.

To address this sort of objection, I set Cassie to the task of finding the least common-multiple (LCM) of two natural numbers. LCM is a problem that is (somewhat deceptively) similar to GCD. Also, like GCD, the LCM of x and y can be computed using the following commonsense algorithm.

1. List the multiples of x up to $x \times y$
2. List the multiples of y up to $x \times y$
3. List the common multiples from the lists in step 1 and 2.
4. Circle the least common multiple from the list in step 3.

Interestingly, both GCD and LCM require some notion of a “default” answer if a “better answer” isn’t found. For GCD, the default answer is 1. This represents Cassie’s “initialization step” of assuming that 1 will be the GCD unless something larger is found. Similarly, Cassie assumes $x \times y$ is the LCM unless something smaller is found. The implementation of this commonsense algorithm (called NLLCM for natural language LCM) is given in the Appendix.

In the next section, I will consider a set of extensions to the theory I have developed in this dissertation.

7.4 Extensions and Further Work

Although some people do not pursue mathematics beyond arithmetic, it is reasonable to expect a theory of early mathematical cognition to scale up to higher mathematics. This section describes some representational and operational extensions to the theory.

7.4.1 Integers, Rationals, and Reals

As it stands, the theory is firmly built on the natural-number progression. A very natural extension can be made to introduce the negative numbers into Cassie's numeration system. The resulting numbers are the integers. One way in which Cassie's counting act could generate a finite segment of integers is by having Cassie form the belief that $-x - 1$ is the predecessor of $-x$ every time she forms the belief that $x + 1$ is the successor of x . This creates a "mirror" count of the positives on the negative side of zero.

Importantly, there are embodied analogs of negative numbers. For example, if a dirt pile of height h is considered to represent a positive number h , then a hole of depth h can be considered to represent the number $-h$. This metaphor makes results such as $h + (-h) = 0$ very intuitive (i.e., filling a hole of depth h with a pile of dirt of height h yields no pile or hole).

The next extension could be to introduce the rational numbers. When performing a division operation, Cassie tacitly assumes that the quotient will be a number. In the current implementation, if she is given the command `perform Divide(n8, n3)` she will not form any beliefs about the quotient $\frac{8}{3}$, because this rational number is simply not in her inventory of concepts. She will instead believe that the result is undefined.

A natural way to represent rationals is by reusing the framework of evaluation frames. Under the current implementation, Cassie's expectation of the division producing a quotient is represented in the `Effect` of the general division act:

```
Effect(Divide(n8, n3), Evaluated(Result(Quotient, n8, n3)))
```

After performing `Divide(n8, n3)`, Cassie knows that the result is not a natural number (i.e., the result does not have `Number` predicated of it. However, she could use this as a reason for creating a new base node, say `b1`, to stand in for the value as follows:

```
Evaluation(Result(Quotient, n8, n3), b1)
```

This creates a strong link between rational numbers and quotients. Cassie "invents" the rationals as mental representations playing the quotient role in division acts. More properly, Cassie should have a `NaturalNumber` case-frame that she applies to the numerons generated during the act of counting, an `IntegerNumber` case-frame that she applies to integers conceived of using

the “mirror” counting method described above, and a `RationalNumber` case-frame that she applies as the result of divisions not yielding naturals or integers. The correct subclass relationships among these numbers could be represented using the following rules:

```
all(x)(NaturalNumber(x) => {IntegerNumber(x),RationalNumber(x)}).
all(x)(IntegerNumber(x) => RationalNumber(x)).
all(x)({NaturalNumber(x),IntegerNumber(x),RationalNumber(x)} v=>
    Number(x)).
```

An interesting feature of this method for generating rationals is that a result is presupposed even when dividing by zero. I would speculate that, unless being told, a child would expect that dividing by zero is a defined operation. The intensional concept “result of the division of n_8 by n_0 ” still needs mental representation, even if it has no extensional referent. There is no grounding embodied activity for this division (e.g., there is no way zero people can share eight objects), so this should tip off the agent to the impossibility of this result.

Unfortunately, this is a very piecemeal way of generating rationals. Unlike natural numbers, which are introduced through the systematic routine of counting, rationals introduced on this case-by-case basis do not form a complete number system for the agent. The successor relationship is all there is to the natural numbers, but the rationals (if they are properly understood), introduce part-whole relationships and a “dense” number line (i.e., between any two rationals is another rational).

The irrational numbers make the representation of real numbers, which include both the rationals and irrationals, even more challenging. The computer (like the human brain) has access to a finite memory. Thus, no irrational number is represented digit by digit. Rather, we can represent a number like π by alluding to the *process* that produces it. Consider the following assertions (with the semantics specified in English):

- `Circle(Circ)`. `[[Circ]]` is a circle.
- `DiameterOf(d,Circ)`. The diameter of `[[Circ]]` is `[[d]]`.
- `CircumferenceOf(c,Circ)`. The circumference of `[[Circ]]` is `[[c]]`
- `Effect(Divide(c,d),Evaluated(Result(Quotient,c,d)))`. The effect of dividing `[[c]]` by `[[d]]` is that the procept `[[Result(Quotient,c,d)]]` will have a value.
- `Evaluation(Result(Quotient,c,d),b1)`. `[[b1]]` is the value of `[[Result(Quotient,c,d)]]`.

Here, `b1` is a *precise* and *finite* representation of π that an agent could hold beliefs about and use in reasoning. If the agent ever has the practical need for an approximation of π , there are plenty of algorithms available. Unfortunately, the presence of irrational numbers like π means that the mechanism for generating rationals described above needs modification to distinguish between rationals and irrationals.

In these proposed intensional representations of integers, rationals, and reals, the key is a reference back to the process of generating the number concept. The quotient representation of π also relies on geometric concepts which the current implementation does not provide. I consider this extension next.

7.4.2 Geometry

It would be somewhat strange to call geometric reasoning an “extension” of the purely numerical theory developed thus far. The geometric conception of mathematics develops alongside the numerical conception. Lakoff and Núñez (2000) present the conceptual metaphor of motion along a path as one of the fundamental grounding metaphors. With this metaphor, the framework of embodied spatial-reasoning is imported into an abstract representation of space. This lays one of the foundations for geometric reasoning.

One sense in which geometry can serve as an extension to the theory is by having Cassie “re-interpret” her numerical representations as geometric ones. This was hinted at in §5.1.5, where representations associated with constructions and measures were considered. These representations clearly have geometric analogs. In fact, Euclid conceived of numbers in just this way, phrasing the GCD problem as a problem of “greatest common measure”.

To maintain cognitive plausibility, a geometric extension of the theory and implementation would need to include several features:

- A representation of (and mechanism for) mental imagery.
- A KL symbolic representation of geometric entities (e.g., points, lines, planes, shapes, polygons) and relations between these entities (e.g., intersection, acuteness)
- A PML representation of the agent’s “sensitivity” to figural representations of shape.
- A mechanism for linking geometric representations to topological representations used in commonsense reasoning (e.g., for some physical object X, there is no exact distance at which it is appropriate to use “this X” as opposed to “that X”, but the “schema” for appropriate usage can be spelled out topologically (Talmy, 2000)).

Like arithmetic, geometry has had a variety of axiomatizations (e.g., Hilbert(1899/1980) and Veblen(1904)). This suggests that a technique similar to procedural decomposition might be used to demonstrate geometric understanding. If lines can be *understood as* collections of points satisfying certain properties, an agent can “work back” towards axioms about points in its explanations.

7.4.3 Scaling Up

As indicated above, it should be possible to scale up the theory from arithmetic. Two interesting areas that are in need of cognitive analysis are algebra and probability.

Algebra imports the familiar operations of arithmetic and extends the mathematical vocabulary by introducing variables. At the system level, Cassie already has a native representation for variables (i.e., variable terms in SNePSLOG such as ?x). These are used throughout the implementation to match an “unknown” in rules, *withsomes*, and *withalls*. To give a cognitive account of algebra, however, would require a semantic account for *algebraic* variables. Also, such an account should tell a story about the embodied foundations of variables.

Probability is another branch of mathematics that partly hinges on arithmetic principles. Much like arithmetic, I believe probabilistic intuitions precede by many years the formal probability

theory taught in schools. There are certainly commonsense notions of likelihood, expectation, and uncertainty agents associate with particular outcomes. One avenue of further research might extend the current theory to give embodied semantic accounts of probabilistic reasoning.

7.4.4 Syntactic Semantics

As I mention above, I am sympathetic to the theory of syntactic semantics as given in (Rapaport, 1988; Rapaport, 1995; Rapaport, 2006). An interesting aspect of this theory as a recursive theory of understanding is that it requires a basis domain “understandable in terms of itself (or in terms of ‘innate ideas’ or some other mechanism)”. Moreover, this basis domain must be entirely internalized (i.e., in the mind of an agent):

My position is this: The mind-world gap cannot be bridged by the mind. There are causal links between [the mind and the world], but the only role these links play in semantics is this: The mind’s internal representations of external objects (which internal representations are caused by external objects) *can* serve as “referents” of other internal symbols, but, since they are *all* internal, meaning is in the head and is syntactic (Rapaport, 1995)

But we have seen that, in the case of a procept, a single internal symbol can be usefully ambiguous, denoting either a process or a concept. These are both equally “syntactic”, but in the process-denoting sense, we have left the classical linguistic notion of syntax—we are interested in what a performance of the act denoted by the symbol means. It would be interesting to see if a common feature across basis domains is symbols capable of doing the double duty of meaning something and referring back to themselves.

Procepts in the counting domain are able to do this double duty by referring to numbers and the act of counting. This is represented by the network in Figure 7.2. The SNePSLOG representation and semantics for the propositions m1 and m5 are:

- m1: $\text{Effect}(\text{Count}(x), \text{Evaluated}(\text{Result}(\text{Count}, x)))$. The effect of counting to $\llbracket x \rrbracket$ is that the procept $\llbracket \text{Result}(\text{Count}, x) \rrbracket$ will have a value.
- m5: $\text{Evaluation}(\text{Result}(\text{Count}, x), x)$. The value of the procept $\llbracket \text{Result}(\text{Count}, x) \rrbracket$ is $\llbracket x \rrbracket$.

Nodes m2 and m4 represent both aspects of the procept “a count of x ”. m3 represents the effect of counting to x , namely, that m4 will have a value. In this role, x is a position on the number line that Cassie is trying to reach. m3 implies the existence of m5 (the evaluation of m4) by the rule given in §4.1.1.3. The value given in m5 refers back to x . In this role, x is an abstraction of the cardinality (e.g., a count of x anything).

7.4.5 Quantity Disposition

The cognitive semantics laid out by Talmy (2000) includes a discussion of how natural languages deal with configurational structure. One of the features introduced in this analysis is that of quantity

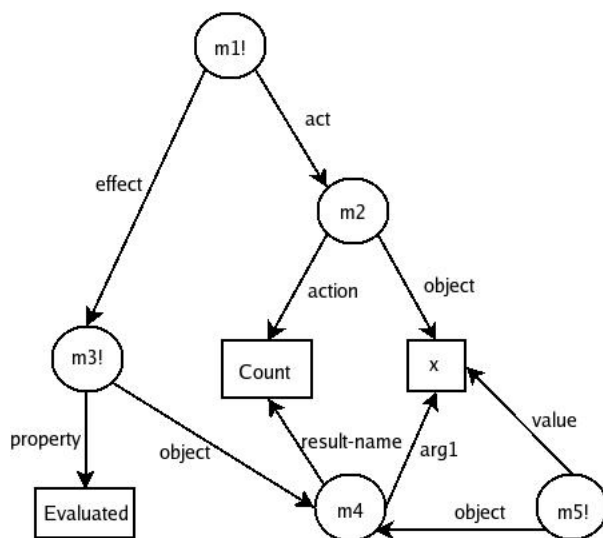


Figure 7.2: The procept “a count of x”.

disposition. This account takes “quantity” to include precise number-sortal pairings as I have (e.g., “three apples”), but also generalized quantifiers (e.g., “some apples”). The focus is on the sortal term, and what it contributes to the way an amount of that sortal is conceived. According to Talmy, the disposition of a quantity includes attributes across the following four categories:

1. *Domain*. Specifies whether the quantity distributes its elements over *space* (manifested as nouns in English), or *time* (manifested as verbs in English).
2. *Plexity*. “A quantity’s state of articulation into equivalent elements. Where the quantity consists of only one such element, it is *uniplex*, and where it consists of more than one, it is *multiplex*” (p. 48). This term is introduced to cover the conception quantities of both matter and actions.
3. *State of boundedness*. “When a quantity is understood as *unbounded*, it is conceived as continuing on indefinitely with no necessary characteristic of finiteness intrinsic to it. When a quantity is understood as *bounded*, it is conceived to be demarcated as an individuated unit entity” (p. 50).
4. *State of dividedness*. “[R]efers to a quantities internal segmentation” (p. 55). This category can be either *discrete* or *continuous*.

Talmy gives several examples to illustrate the various ways in which the four-categories can combine. These are shown in Table 7.1. Along with these different dispositions, Talmy presents a series of cognitive operations for manipulating them. A uniplex quantity can be *multiplexed*. Through multiplexing ‘bird’ becomes ‘birds’, ‘sigh’ becomes ‘kept sighing’. An unbounded quantity can be subjected to *bounding*. Through bounding, ‘furniture’ becomes ‘piece of furniture’, ‘breathe’ becomes ‘breathing for some amount of time’, ‘water’ becomes ‘portion of water’, and ‘sleep’

<i>Disposition</i>	<i>Example</i>
$\langle \textit{space,uniplex} \rangle$	bird
$\langle \textit{time,uniplex} \rangle$	(to) sigh
$\langle \textit{space,multiplex,bounded,discrete} \rangle$	family
$\langle \textit{time,multiplex,bounded,discrete} \rangle$	(to) molt
$\langle \textit{space,multiplex,unbounded,discrete} \rangle$	furniture
$\langle \textit{time,multiplex,unbounded,discrete} \rangle$	(to) breathe
$\langle \textit{space,multiplex,bounded,continuous} \rangle$	(a) sea
$\langle \textit{time,multiplex,bounded,continuous} \rangle$	(to) empty
$\langle \textit{space,multiplex,unbounded,continuous} \rangle$	water
$\langle \textit{time,multiplex,unbounded,continuous} \rangle$	(to) sleep

Table 7.1: Quantity Dispositions.

becomes ‘sleeping for some amount of time’. Boundedness also determines ungrammatical usages (e.g., one cannot say “He slept in eight hours”, but must say “He slept for eight hours”).

Cassie’s natural-language production capacity has not been directly addressed in this work. However, a fuller theory would coordinate the representations I have used for precise numerical-quantities with quantity dispositions. Spelling this out would accommodate an agent that had to reason with both imprecise quantities (when generating natural language), and precise quantities (when reasoning mathematically).

7.4.6 Multi-agent Systems

If I am right in claiming that early mathematics is multiply realizable, then there is something to be said about how communities of agents come to abstract different “private” mathematical mechanisms, in different locations, within the framework of different cultures, and from different particular sense-data, and, nevertheless, end up agreeing upon mathematical propositions.² Computationally, such concerns can be considered in the context of a multi-agent system.

Although SNePS agents are usually deployed in a single-process (single-agent) context, some work has been done on communicating SNePS agents. Campbell and Shapiro (1998) discuss a set of algorithms that would allow a “mediator” agent to resolve communication issues between two different SNePS agents. Rapaport (2003) describes a method by which different agents can “negotiate” the meanings for the expressions they use. Integrating such components with my existing theory, we can begin to examine the following questions:

- How does a community of agents come to agree on the mechanisms of mathematical cognition (e.g., a numeration system and a set of effective algorithms)?
- How can one agent teach another agent to transfer a mathematical skill to a new domain? For example, an agent that knows how to count apples might teach an agent that has no conception of the sortal “apple” what the term means on the basis of color and shape. The

²For a detailed philosophical account on the social construction of mathematical knowledge, see (Kitcher, 1981).

agent can transfer its enumeration skills to a new target just by learning the new sortal term and its properties.

- How are new mathematical ideas constructed through social interaction?

Quine (1969) has pointed out several difficulties one would have in attempting to attribute the use of individuating terms within a community of foreign-language speakers (see pp. 8–10). These difficulties can be sidestepped in a computational multi-agent system since both the behavior and the mental content generating that behavior can be probed, adjusted, and experimented with.

7.4.7 Cognitive Robotics

The implementation of Cassie’s embodiment relied on a simulated sensori-actuator layer. Although simulating the outside world was sufficient to generate the phenomenal objects Cassie needed, a simulation can only go so far in getting at all of the issues associated with embodiment. Fortunately, GLAIR can be utilized in the context of physically embodied agents (i.e., robots). This introduces a whole set of engineering challenges (e.g., the vision system cannot make some of the simplifying assumptions when deployed in the real world), however the resulting agent would be all the more convincing for skeptics of computational math cognition.

7.5 Final Thoughts

As I have indicated in this chapter, the current theory and implementation of Cassie are just a starting point and can be greatly expanded and improved. In the introduction to this work, I characterized mathematical cognition as an emerging field and, indeed, it is still emerging. I have by no means given a complete account of even early mathematical cognition. However, I do believe that both the theory and the implementation support taking a computational perspective towards the subject. Moreover, I believe developing computational theories of mathematical cognition will benefit AI researchers and the broader cognitive science community.

Appendix A

SNePS Implementation

This appendix includes the “core” SNePSLOG source code for the implementation of Cassie. I have not listed here some of the code that is described fully in the body of the dissertation. Currently, the source files are accessible on any of the UB CSE servers under the following directory:

```
/project/rapaport/GOLDFAIN/MathCogCassie/
```

The implementation spans several files. The following two files must be demoed in order to run any of the others.

- **Config.snepslog**: Implements the deliberative do-one primitive act and sets up SNePS system variables.
- **Counting.snepslog**: Provides the `ConceiveInitialElement` and `ConceiveOrdering` primitive acts.

The following commands should be performed by Cassie before any other files are loaded:

```
perform ConceiveInitialElement()  
perform ConceiveOrdering(10)
```

The following files implement abstract arithmetic:

- **SemArith.snepslog**: Implements count addition, count subtraction, iterated-addition multiplication, and iterated-subtraction division.
- **SynArith.snepslog**: Implements syntactic addition.
- **ExtArith.snepslog**: Implements the Lisp calculator.
- **Shortcuts.snepslog**: Implements shortcut arithmetic routines.
- **GreaterThan.snepslog**: Implements path-based inference rules for the greater-than relation.
- **NLGCD.snepslog**: Implements NLGCD commonsense algorithm.

The following files implement embodied arithmetic:

- **EmbEnum.snepslog**: Implements embodied enumeration.
- **Accumulator.snepslog**: Implements `timed-do`.
- **ConsciousCounting.snepslog**: Implements conscious counting mode of self-action enumeration.
- **OrderedAction.snepslog**: Implements ordinal assignment to self-actions.

Finally, the following files implement Cassie's explanation system:

- **Explanation.snepslog**: Implements procedural decomposition.
- **ConceptualDef.snepslog**: Implements conceptual definition.
- **QuestionParser.snepslog**: Implements the recursive descent parser for NL questions.

The source code is given below along with its comments (lines starting with `';` in SNePSLOG), but is not otherwise annotated. Code delimited by the double-carat marks `'g` is the Lisp code implementing Cassie's PML.

For a discussion of SNePSLOG syntax and semantics, see §3.2 or (Shapiro and SNePS Implementation Group, 2008). For a discussion on SNeRE syntax and semantics, see §3.2.1.3 or (Shapiro and SNePS Implementation Group, 2008). For a discussion of the syntax and semantics used for abstract arithmetic, see §4.1.1. For a discussion of the syntax and semantics used for embodied arithmetic, see §5.1.2.

A. Counting

```
; =====
; FILENAME:      Counting.snepslog
; DATE:         3/20/07
; PROGRAMMER:   Albert Goldfain
;
; Lines beginning with a semi-colon are comments.
; Regions delimited by "^^" are Lisp commands.
; All other lines are SNePSLOG commands.
;
; To use this file: run SNePSLOG; at the SNePSLOG prompt (:), type:
;
;     demo "Counting.snepslog"
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; =====

define-frame Number(class member)
define-frame ResultName(class member)
define-frame Numeral(class member)

define-frame NumeralOf(nil numeral number)
define-frame Successor(nil succ pred)

define-frame Say(action utterance)
define-frame SayLine(action utterance)
define-frame SayLine+(action utterance arg1 arg2)
define-frame ConceiveOrdering(action num)
define-frame ConceiveInitialElement(action)
define-frame ConceiveSuccessor(action)

;;;Result Frame
define-frame Result(nil result-name arg1 arg2)

;;;Evaluation Frames
define-frame Evaluation(nil result value)
define-frame Evaluated(property object)

;;;General Acts
define-frame Add(action arg1 arg2)
define-frame Subtract(action arg1 arg2)
define-frame Multiply(action arg1 arg2)
define-frame Divide(action arg1 arg2)

^^
(defparameter *attend-number* 0)
(defparameter *old-number* 0)
(defparameter *attend-numeral* 1)

(define-primaction Say ((utterance))
  "Print the utterance (without linebreak)"
  (format t "~A" utterance))

(define-primaction SayLine ((utterance))
  "Print a line containing the utterance"
  (format t "~A~%" utterance))

(define-primaction SayLine+ ((utterance) (arg1) (arg2))
  "Print a line containing the utterance followed by arg1 and arg2"
  (format t "~A ~A ~A~%" utterance arg1 arg2))

(define-primaction ConceiveInitialElement ())
```

```

"Create an initial element in the number sequence"
(let ((newnum "n0"))
  (tell (concatenate 'string
    "Number(" newnum ")."))
  (setf *attend-number* newnum)
  (tell "Numeral(0).")
  (tell (concatenate 'string
    "NumeralOf(0," newnum ")."))))

(define-primaction ConceiveSuccessor ()
  "Create a base node to succeed the current node being attended to"
  ;;
  ;;;NOTE: gensym can be substituted to create a randomly named newnum
  ;;
  (let ((newnum (concatenate 'string "n" (princ-to-string *attend-numeral*)))
        (*old-number* *attend-number*))
    (tell (concatenate 'string
      "Number(" newnum ")."))
    (setf *attend-number* newnum)
    (tell (concatenate 'string "Numeral(" (princ-to-string
      *attend-numeral*) ")."))
    (tell (concatenate 'string
      "NumeralOf(" (princ-to-string *attend-numeral*) "," newnum ")."))
    (setf *attend-numeral* (cl:++ 1 *attend-numeral*))
    (tell (concatenate 'string
      "Successor(" newnum "," (princ-to-string *old-number*) ")."))))

(define-primaction ConceiveOrdering ((num))
  "DESCRIPTION"
  (loop for i from 1 to (sneps:node-to-lisp-object num)
    do (tell "perform ConceiveSuccessor()")))

(attach-primaction Say Say SayLine SayLine SayLine+ SayLine+
  ConceiveInitialElement ConceiveInitialElement
  ConceiveSuccessor ConceiveSuccessor
  ConceiveOrdering ConceiveOrdering)

^^

;;;Register result names
ResultName(Sum).
ResultName(Difference).
ResultName(Product).
ResultName(Quotient).

;;;Register effects of general acts.
all(x,y)({Number(x),Number(y)} &=>
  {Effect(Add(x,y),Evaluated(Result(Sum,x,y))),
   Effect(Subtract(x,y),Evaluated(Result(Difference,x,y))),
   Effect(Multiply(x,y),Evaluated(Result(Product,x,y))),
   Effect(Divide(x,y),Evaluated(Result(Quotient,x,y)))}).

;;;UVER x for y
all(x)(Number(x) =>
  {Effect(Add(x,x),Evaluated(Result(Sum,x,x))),
   Effect(Subtract(x,x),Evaluated(Result(Difference,x,x))),
   Effect(Multiply(x,x),Evaluated(Result(Product,x,x))),
   Effect(Divide(x,x),Evaluated(Result(Quotient,x,x)))}).

```

B. Semantic Arithmetic

```
; =====
; FILENAME:      SemArith.snepslog
; DATE:         9/17/07
; PROGRAMMER:   Albert Goldfain
;
; Lines beginning with a semi-colon are comments.
; Regions delimited by "^^" are Lisp commands.
; All other lines are SNePSLOG commands.
;
; =====

;;;Register result names
ResultName(CountSum).
ResultName(CountDifference).
ResultName(SumProduct).
ResultName(DifferenceQuotient).

; =====
;                               COUNT-ADDITION
; =====
;;;
;;;[[Zero(x)]] = [[x]] is the number 0
;;;
define-frame Zero(nil zero)

;;;
;;;[[FinalSum(x)]] = [[x]] is the final sum of count addition
;;;
define-frame FinalSum(nil final-sum)
;;;
;;;[[CountFromFor(x,y)]] = Act of counting from [[x]] for [[y]]
;;;numbers.
;;;
define-frame CountFromFor(action object1 object2)

;;;
;;;[[CountAdd(x,y)]] = Act of performing count-addition on [[x]] and
;;;[[y]]
define-frame CountAdd(action object1 object2)

;;;Establish which numeron is zero
perform withsome(?n,NumeralOf(0,?n),believe(Zero(?n)),Say("I don't understand what 0 is"))

;;; To count from x for y numbers, count from x+1 for y-1 numbers.
all(x,y)({Number(x),Number(y)} =>
  ActPlan(CountFromFor(x,y),
    withsome(?xp1,
      Successor(?xp1,x),
      withsome(?yml,
        Successor(y,?yml),
        sniff({if(Zero(?yml),believe(FinalSum(?xp1))),
          else(CountFromFor(?xp1,?yml)})),
        Say("I don't know how to count add these numbers")),
      Say("I don't know how to count add these numbers")))).

;;;UVBR x for y
all(x,y)(Number(x)=>
  ActPlan(CountFromFor(x,x),
    withsome(?xp1,
      Successor(?xp1,x),
```



```

        withsome(?xml,
            Successor(x,?xml),
            snif({if(Zero(?xml),believe(FinalSum(?xpl))),
                else(CountFromFor(?xpl,?xml))}),
            Say("I don't know how to count add these numbers")),
        Say("I don't know how to count add these numbers"))).

;;;
;;; To Count-Add x and y, count from x for y numbers.
;;;
all(x,y)({Number(x),Number(y)} &=>
    {ActPlan(CountAdd(x,y),
        snsequence(SayLine+"Count Adding",x,y),
        snsequence(CountFromFor(x,y),
            withsome(?z,
                FinalSum(?z),
                snsequence(believe(Evaluation(Result(CountSum,x,y),?z)),
                    disbelieve(FinalSum(?z))),
                SayLine("I could not determine the sum."))}),
        Effect(CountAdd(x,y),Evaluated(Result(CountSum,x,y)))}).

;;;UVBR x for y
all(x,y)(Number(x) =>
    {ActPlan(CountAdd(x,x),
        snsequence(SayLine+"Count Adding",x,x),
        snsequence(CountFromFor(x,x),
            withsome(?z,
                FinalSum(?z),
                snsequence(believe(Evaluation(Result(CountSum,x,x),?z)),
                    disbelieve(FinalSum(?z))),
                SayLine("I could not determine the sum."))}),
        Effect(CountAdd(x,x),Evaluated(Result(CountSum,x,x)))}).

;;;
;;;Every counted sum obtained by this method is a sum per se.
;;;
all(x,y,z)(Evaluation(Result(CountSum,x,y),z) => Evaluation(Result(Sum,x,y),z)).

;;;UVBR x for y
all(x,z)(Evaluation(Result(CountSum,x,x),z) => Evaluation(Result(Sum,x,x),z)).

;;;UVBR y for z
all(x,y)(Evaluation(Result(CountSum,x,y),y) => Evaluation(Result(Sum,x,y),y)).

;;;UVBR x for z
all(x,y)(Evaluation(Result(CountSum,x,y),x) => Evaluation(Result(Sum,x,y),x)).

;;;UVBR x for y and z
all(x)(Evaluation(Result(CountSum,x,x),x) => Evaluation(Result(Sum,x,x),x)).

;;;
;;;CountAdd is a plan for adding per se.
;;;
all(x,y)({Number(x),Number(y)} &=> ActPlan(Add(x,y),CountAdd(x,y))).

;;;UVBR x for y
all(x)(Number(x) => ActPlan(Add(x,x),CountAdd(x,x))).
=====
;
;          COUNT-SUBTRACTION
;
=====
define-frame FinalDifference(nil final-difference)
define-frame CountDownFromFor(action object1 object2)
define-frame CountSubtract(action object1 object2)

;;;To count down from x for y numbers, count down from x-1 for y-1 numbers.

```

```

all(x,y)({Number(x),Number(y)} &=>
    ActPlan(CountDownFromFor(x,y),
        withsome(?xml,
            Successor(x,?xml),
            withsome(?yml,
                Successor(y,?yml),
                sniff({if(Zero(?yml),believe(FinalDifference(?xml))),
                    else(CountDownFromFor(?xml,?yml)})),
                believe(FinalDifference(x)))))).

;;;UVBR: x for y
all(x)(Number(x) =>
    ActPlan(CountDownFromFor(x,x),
        withsome(?xml,
            Successor(x,?xml),
            sniff({if(Zero(?xml),believe(FinalDifference(?xml))),
                else(CountDownFromFor(?xml,?xml)})),
            believe(FinalDifference(x)))).

;;;
;;; To Count-Subtract y from x, count down from x for y numbers.
;;;
all(x,y)({Number(x),Number(y)} &=>
    {ActPlan(CountSubtract(x,y),
        sniff({if(GreaterThan(y,x),
            snsequence(SayLine+"I cannot subtract a
                larger number from a smaller number.",x,y),
            believe(Evaluation(Result(CountDifference,x,y),undefined))),
        else(snsequence(SayLine+"Count Subtracting",x,y),
            snsequence(CountDownFromFor(x,y),
                withsome(?z,
                    FinalDifference(?z),
                    snsequence(believe(Evaluation(Result(CountDifference,x,y),?z)),
                        disbelieve(FinalDifference(?z))),
                    SayLine("I could not determine the difference. ")))))),
        Effect(CountSubtract(x,y),Evaluated(Result(CountDifference,x,y)))).

;;;UVBR x for y
all(x)(Number(x) =>
    {ActPlan(CountSubtract(x,x),
        snsequence(SayLine+"Count Subtracting",x,x),
        snsequence(CountDownFromFor(x,x),
            withsome(?z,
                FinalDifference(?z),
                snsequence(believe(Evaluation(Result(CountDifference,x,x),?z)),
                    disbelieve(FinalDifference(?z))),
                SayLine("I could not determine the difference. "))))),
        Effect(CountSubtract(x,x),Evaluated(Result(CountDifference,x,x)))).

;;;
;;;Every counted difference obtained by this method is a difference per se.
;;;
all(x,y,z)(Evaluation(Result(CountDifference,x,y),z) => Evaluation(Result(Difference,x,y),z)).

;;;UVBR x for y
all(x,z)(Evaluation(Result(CountDifference,x,x),z) => Evaluation(Result(Difference,x,x),z)).

;;;UVBR y for z
all(x,y)(Evaluation(Result(CountDifference,x,y),y) => Evaluation(Result(Difference,x,y),y)).

;;;UVBR x for z
all(x,y)(Evaluation(Result(CountDifference,x,y),x) => Evaluation(Result(Difference,x,y),x)).

```

```

;;;UVBR x for y and z
all(x)(Evaluation(Result(CountDifference,x,x),x) => Evaluation(Result(Difference,x,x),x)).

;;;
;;;CountSubtract is a plan for adding per se.
;;;
all(x,y)({Number(x),Number(y)} &=> {ActPlan(Subtract(x,y),CountSubtract(x,y))}).

;;;UVBR x for y
all(x)(Number(x) => ActPlan(Subtract(x,x),CountSubtract(x,x))).

;=====
;
;                               ADD-MULTIPLICATION
;=====

;;;
;;;[[NumToAdd(x)]] = [[x]] is the number subject to repeated
;;; additions.
;;;
define-frame NumToAdd(nil num-to-add)

;;;
;;;[[ProductResultSoFar(x)]] = [[x]] is the temporary result of the
;;;product being computed.
;;;
define-frame ProductResultSoFar(nil product-result-so-far)

;;;
;;;[[AddFromFor(x,y)]] = Act of adding [[x]] to itself the predecessor
;;;of [[y]] times.
;;;
define-frame AddFromFor(action object1 object2)

;;;
;;;[[AddMultiply(x,y)]] = Act of performing add-multiplication on
;;;[[x]] and [[y]].
define-frame AddMultiply(action object1 object2)

;;;Base case of add multiplication. To add from x for 0 iterations
;;;just stay at x. Also, believe appropriate SumProduct for unit
;;;multiplication
all(x,y)({ProductResultSoFar(x),Zero(y)} &=>
  ActPlan(AddFromFor(x,y),
    withsome(?one,
      Successor(?one,y),
      believe(Evaluation(Result(SumProduct,x,?one),x)),
      Say("Problem asserting the unit multiplication")))).

;;;Recursive case of add multiplication. To repeat the addition of w
;;;for z iterations, add w to itself and repeat the process for z-1
;;;iterations.
all(w,x,y,z)({NumToAdd(w), Successor(z,y),ProductResultSoFar(x)} &=>
  ActPlan(AddFromFor(x,z),
    snsequence(disbelieve(ProductResultSoFar(x)),
      snsequence(Add(x,w),
        withsome(?p,
          Evaluation(Result(Sum,x,w),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR x for w
all(x,y,z)({NumToAdd(x), Successor(z,y),ProductResultSoFar(x)} &=>
  ActPlan(AddFromFor(x,z),
    snsequence(disbelieve(ProductResultSoFar(x)),
      snsequence(Add(x,x),
        withsome(?p,
          Evaluation(Result(Sum,x,x),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR y for w
all(x,y,z)({NumToAdd(y), Successor(z,y),ProductResultSoFar(x)} &=>
  ActPlan(AddFromFor(x,z),
    snsequence(disbelieve(ProductResultSoFar(x)),
      snsequence(Add(x,y),
        withsome(?p,
          Evaluation(Result(Sum,x,y),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR z for w
all(x,y,z)({NumToAdd(z), Successor(z,y),ProductResultSoFar(x)} &=>
  ActPlan(AddFromFor(x,z),
    snsequence(disbelieve(ProductResultSoFar(x)),
      snsequence(Add(x,z),
        withsome(?p,
          Evaluation(Result(Sum,x,z),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR y for x
all(w,y,z)({NumToAdd(w), Successor(z,y),ProductResultSoFar(y)} &=>
  ActPlan(AddFromFor(y,z),
    snsequence(disbelieve(ProductResultSoFar(y)),
      snsequence(Add(y,w),
        withsome(?p,
          Evaluation(Result(Sum,y,w),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR z for x
all(w,y,z)({NumToAdd(w), Successor(z,y),ProductResultSoFar(z)} &=>
  ActPlan(AddFromFor(z,z),
    snsequence(disbelieve(ProductResultSoFar(z)),
      snsequence(Add(z,w),
        withsome(?p,
          Evaluation(Result(Sum,z,w),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

```

```

;;;UVBR z for x and z for w
all(y,z)({NumToAdd(z), Successor(z,y),ProductResultSoFar(z)} &=>
  ActPlan(AddFromFor(z,z),
    snsequence(disbelieve(ProductResultSoFar(z)),
      snsequence(Add(z,z),
        withsome(?p,
          Evaluation(Result(Sum,z,z),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,y)),
          SayLine(`I could not determine the sum`)))))).

;;;UVBR w for x and w for y
all(w,z)({NumToAdd(w), Successor(z,w),ProductResultSoFar(w)} &=>
  ActPlan(AddFromFor(w,z),
    snsequence(disbelieve(ProductResultSoFar(w)),
      snsequence(Add(w,w),
        withsome(?p,
          Evaluation(Result(Sum,w,w),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,w)),
          SayLine(`I could not determine the sum`)))))).

;;;UVBR w for y and x for z
all(w,x)({NumToAdd(w), Successor(x,w),ProductResultSoFar(x)} &=>
  ActPlan(AddFromFor(x,x),
    snsequence(disbelieve(ProductResultSoFar(x)),
      snsequence(Add(x,w),
        withsome(?p,
          Evaluation(Result(Sum,x,w),?p),
          snsequence(believe(ProductResultSoFar(?p)),
            AddFromFor(?p,w)),
          SayLine(`I could not determine the sum`)))))).

;;;
;;; To Add-Multiply x and y, add x to itself y-1 times
;;;
all(x,y)({Number(x),Number(y)} &=>
  {ActPlan(AddMultiply(x,y),
    snsequence(SayLine+"Add Multiplying",x,y),
    snsequence(believe(ProductResultSoFar(x)),
      snsequence(believe(NumToAdd(x)),
        snsequence(withsome(?p,Successor(y,?p),AddFromFor(x,?p),
          SayLine("I do not know how to add-multiply these numbers")),
        snsequence(withsome(?z,
          ProductResultSoFar(?z),
          snsequence(believe(Evaluation(Result(SumProduct,x,y),?z)),
            disbelieve(ProductResultSoFar(?z))),
          SayLine("I could not determine the product."),
          disbelieve(NumToAdd(x))))))),
    Effect(AddMultiply(x,y),Evaluated(Result(SumProduct,x,y))))}).

;;;UVBR x for y
all(x)(Number(x) =>
  {ActPlan(AddMultiply(x,x),
    snsequence(SayLine+"Add Multiplying",x,x),
    snsequence(believe(ProductResultSoFar(x)),
      snsequence(believe(NumToAdd(x)),
        snsequence(withsome(?p,Successor(x,?p),AddFromFor(x,?p),
          Say("I do not know how to add-multiply these numbers")),

```

```

snsequence(withsome(?z,
                ProductResultSoFar(?z),
                snsequence(believe(Evaluation(Result(SumProduct,x,x),?z)),
                            disbelieve(ProductResultSoFar(?z))),
                SayLine("I could not determine the product.")),
            disbelieve(NumToAdd(x))))),
Effect(AddMultiply(x,x),Evaluated(Result(SumProduct,x,x)))).

;;;
;;;Every added product obtained by this method is a product per se.
;;;
all(x,y,z)(Evaluation(Result(SumProduct,x,y),z) => Evaluation(Result(Product,x,y),z)).

;;;UVBR x for y
all(x,z)(Evaluation(Result(SumProduct,x,x),z) => Evaluation(Result(Product,x,x),z)).

;;;UVBR x for z
all(x,y)(Evaluation(Result(SumProduct,x,y),x) => Evaluation(Result(Product,x,y),x)).

;;;UVBR y for z
all(x,y)(Evaluation(Result(SumProduct,x,y),y) => Evaluation(Result(Product,x,y),y)).

;;;UVBR x for y and x for z
all(x)(Evaluation(Result(SumProduct,x,x),x) => Evaluation(Result(Product,x,x),x)).

;;;
;;;Add multiplication is a plan for multiplication per se.
;;;
all(x,y)({Number(x),Number(y)} &=> {ActPlan(Multiply(x,y),AddMultiply(x,y))}).

;;;UVBR x for y
all(x)(Number(x) => ActPlan(Multiply(x,x),AddMultiply(x,x))).

;=====
;
; SUBTRACT-DIVISION
;=====
define-frame NumToSubtract(nil num-to-subtract)
define-frame QuotientSoFar(nil quotient-so-far)
define-frame SubtractUntilZero(action object1 object2)
define-frame SubtractDivide(action object1 object2)

;;; Problem case of subtracting until zero
all(x,z)({NumToSubtract(x),QuotientSoFar(z)} &=>
    ActPlan(SubtractUntilZero(undefined,x),
            snsequence(believe(QuotientSoFar(undefined)),
                        disbelieve(QuotientSoFar(z)))).

;;; UVBR x for z
all(x)({NumToSubtract(x),QuotientSoFar(x)} &=>
    ActPlan(SubtractUntilZero(undefined,x),
            snsequence(believe(QuotientSoFar(undefined)),
                        disbelieve(QuotientSoFar(x)))).

;;; Base case of subtracting until zero
all(x,z)({Number(x),NumToSubtract(x),QuotientSoFar(z)} &=>
    ActPlan(SubtractUntilZero(x,x),
            snsequence(Subtract(x,x),
                        withsome(?s,
                                Successor(?s,z),
                                snsequence(believe(QuotientSoFar(?s)),
                                            disbelieve(QuotientSoFar(z))),
                                SayLine("I could not determine the successor")))).

;;;UVBR x for z

```

```

all(x,z)({Number(x),NumToSubtract(x),QuotientSoFar(x)} &=>
    ActPlan(SubtractUntilZero(x,x),
            snsequence(Subtract(x,x),
                withsome(?s,
                    Successor(?s,x),
                    snsequence(believe(QuotientSoFar(?s)),
                                disbelieve(QuotientSoFar(x))),
                    SayLine("I could not determine the successor")))).

;;;
;;; Recursive case of subtracting until zero
;;;
all(x,y,z)({Number(x),NumToSubtract(y),QuotientSoFar(z)} &=>
    ActPlan(SubtractUntilZero(x,y),
            snsequence(Subtract(x,y),
                snsequence(withsome(?s,
                    Successor(?s,z),
                    snsequence(believe(QuotientSoFar(?s)),
                                disbelieve(QuotientSoFar(z))),
                    SayLine("I could not determine the successor")),
                withsome(?r,
                    Evaluation(Result(Difference,x,y),?r),
                    SubtractUntilZero(?r,y),
                    SayLine("I could not determine the difference")))).

;;; UVBR x for z
all(x,y)({Number(x),NumToSubtract(y),QuotientSoFar(x)} &=>
    ActPlan(SubtractUntilZero(x,y),
            snsequence(Subtract(x,y),
                snsequence(withsome(?s,
                    Successor(?s,x),
                    snsequence(believe(QuotientSoFar(?s)),
                                disbelieve(QuotientSoFar(x))),
                    SayLine("I could not determine the successor")),
                withsome(?r,
                    Evaluation(Result(Difference,x,y),?r),
                    SubtractUntilZero(?r,y),
                    SayLine("I could not determine the difference")))).

;;; UVBR y for z
all(x,y)({Number(x),NumToSubtract(y),QuotientSoFar(y)} &=>
    ActPlan(SubtractUntilZero(x,y),
            snsequence(Subtract(x,y),
                snsequence(withsome(?s,
                    Successor(?s,y),
                    snsequence(believe(QuotientSoFar(?s)),
                                disbelieve(QuotientSoFar(y))),
                    SayLine("I could not determine the successor")),
                withsome(?r,
                    Evaluation(Result(Difference,x,y),?r),
                    SubtractUntilZero(?r,y),
                    SayLine("I could not determine the difference")))).

;;;
;;; To Subtract-Divide x by y, subtract y from x until the difference
;;; is 0.
;;;
all(x,y)({Number(x),Number(y)} &=>
    {ActPlan(SubtractDivide(x,y),
            snif({if(Zero(y),snsequence(Say("I cannot divide by zero."),
                believe(Evaluation(DifferenceQuotient,x,y),undefined))),
            else(snsequence(SayLine+("Subtract Dividing",x,y),

```

```

withsome(?z,Zero(?z),
         snsequence(believe(QuotientSoFar(?z)),
                    snsequence(believe(NumToSubtract(y)),
                                snsequence(SubtractUntilZero(x,y),
                                            snsequence(withsome(?q,
                                                            QuotientSoFar(?q),
                                                            snsequence(believe(Evaluation(Result(DifferenceQuotient,x,y),?q)),
                                                                disbelief(QuotientSoFar(?q))),
                                                                SayLine("I could not determine the difference.")),
                                                                disbelief(NumToSubtract(y))))),
                                                                SayLine(" I could not determine the difference."))))),
         Effect(SubtractDivide(x,y), Evaluated(Result(DifferenceQuotient,x,y))}).

;;;
;;;UVBR x for y
;;;
all(x)(Number(x) =>
      {ActPlan(SubtractDivide(x,x),
               snif({if(Zero(x),snsequence(Say("I cannot divide by zero."),
                                             believe(Evaluation(DifferenceQuotient,x,y),undefined))),
                    else(snsequence(SayLine+("Subtract Dividing",x,x),
                                       withsome(?z,Zero(?z),
                                               snsequence(believe(QuotientSoFar(?z)),
                                                           snsequence(believe(NumToSubtract(x)),
                                                                       snsequence(SubtractUntilZero(x,x),
                                                                                   snsequence(withsome(?q,
                                                                                               QuotientSoFar(?q),
                                                                                               snsequence(believe(Evaluation(Result(DifferenceQuotient,x,x),?q)),
                                                                                                   disbelief(QuotientSoFar(?q))),
                                                                                                   SayLine("I could not determine the difference.")),
                                                                                                   disbelief(NumToSubtract(x))))),
                                                                                                   SayLine(" I could not determine the difference."))))))),
                    Effect(SubtractDivide(x,x), Evaluated(Result(DifferenceQuotient,x,x))}).

;;;
;;;Every difference quotient obtained by this method is a
;;;quotient per se.
;;;
all(x,y,z)(Evaluation(Result(DifferenceQuotient,x,y),z) => Evaluation(Result(Quotient,x,y),z)).

;;;
;;;UVBR x for y
;;;
all(x,z)(Evaluation(Result(DifferenceQuotient,x,x),z) => Evaluation(Result(Quotient,x,x),z)).

;;;
;;;UVBR x for z
;;;
all(x,y)(Evaluation(Result(DifferenceQuotient,x,y),x) => Evaluation(Result(Quotient,x,y),x)).

;;;
;;;UVBR y for z
;;;
all(x,y)(Evaluation(Result(DifferenceQuotient,x,y),y) => Evaluation(Result(Quotient,x,y),y)).

;;;
;;;UVBR x for y and z
;;;
all(x)(Evaluation(Result(DifferenceQuotient,x,x),x) => Evaluation(Result(Quotient,x,x),x)).

;;;
;;;Subtract division is a plan for division per se
;;;

```



```
all(x,y)({Number(x),Number(y)} &=> ActPlan(Divide(x,y),SubtractDivide(x,y))).  
  
;;UVBR x for y  
all(x)(Number(x) => ActPlan(Divide(x,x),SubtractDivide(x,x))).
```

C. Extended Arithmetic

```
; =====
; FILENAME:      ExtArith.snepslog
; DATE:         3/10/08
; PROGRAMMER:   Albert Goldfain
;
; Lines beginning with a semi-colon are comments.
; Regions delimited by "^^" are Lisp commands.
; All other lines are SNePSLOG commands.
;
; To use this file: run SNePSLOG; at the SNePSLOG prompt (:), type:
;
;     demo "ExtArith.snepslog"
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; =====

define-frame CalcAdd(action arg1 arg2)
define-frame CalcSubtract(action arg1 arg2)
define-frame CalcMultiply(action arg1 arg2)
define-frame CalcDivide(action arg1 arg2)

^^
(define-primaction CalcAdd((arg1) (arg2))
  (format t "Calc Adding ~A ~A~%" arg1 arg2)
  (let* ((add1 (sneps:node-to-lisp-object arg1))
        (add2 (sneps:node-to-lisp-object arg2))
        (num1 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                                "NumeralOf("
                                                (princ-to-string add1)
                                                ",?x)"))))))))
        (num2 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                                "NumeralOf("
                                                (princ-to-string add2)
                                                ",?x)"))))))))
        (num3 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                                "NumeralOf("
                                                (princ-to-string (cl:+ add1 add2))
                                                ",?x)"))))))))
    (tell (concatenate 'string
      "Evaluation(Result(CalcSum,"
      (princ-to-string add1) ", " (princ-to-string add2) "),"
      (princ-to-string (cl:+ add1 add2)) ")."))
    (tell (concatenate 'string
      "Evaluation(Result(Sum,"
      (princ-to-string num1) ", " (princ-to-string num2) "),"
      (princ-to-string num3) ")."))))

(define-primaction CalcSubtract((arg1) (arg2))
  (format t "Calc Subtracting ~A ~A~%" arg1 arg2)
  (let* ((sub1 (sneps:node-to-lisp-object arg1))
        (sub2 (sneps:node-to-lisp-object arg2))
        (num1 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                                "NumeralOf("
                                                (princ-to-string sub1)
```

```

                                ",?x")))))))
(num2 (sneps:node-to-lisp-object
      (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string sub2)
                                           ",?x")))))))
(num3 (sneps:node-to-lisp-object
      (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string (cl:- subl sub2))
                                           ",?x")))))))

(tell (concatenate 'string
  "Evaluation(Result(CalcDifference,"
  (princ-to-string subl) ", " (princ-to-string sub2) "),"
  (princ-to-string (cl:- subl sub2)) ")."))

(tell (concatenate 'string
  "Evaluation(Result(Difference,"
  (princ-to-string num1) ", " (princ-to-string num2) "),"
  (princ-to-string num3) ")."))

(define-primaction CalcMultiply((arg1) (arg2))
  (format t "Calc Multiplying ~A ~A~%" arg1 arg2)
  (let* ((mult1 (sneps:node-to-lisp-object arg1))
        (mult2 (sneps:node-to-lisp-object arg2))
        (num1 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string mult1)
                                           ",?x")))))))
        (num2 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string mult2)
                                           ",?x")))))))
        (num3 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string (cl:* mult1 mult2))
                                           ",?x")))))))

(tell (concatenate 'string
  "Evaluation(Result(CalcProduct,"
  (princ-to-string mult1) ", " (princ-to-string mult2) "),"
  (princ-to-string (cl:* mult1 mult2)) ")."))

(tell (concatenate 'string
  "Evaluation(Result(Product,"
  (princ-to-string num1) ", " (princ-to-string num2) "),"
  (princ-to-string num3) ")."))

(define-primaction CalcDivide((arg1) (arg2))
  (format t "Calc Dividing ~A ~A~%" arg1 arg2)
  (let* ((div1 (sneps:node-to-lisp-object arg1))
        (div2 (sneps:node-to-lisp-object arg2))
        (num1 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string div1)
                                           ",?x")))))))
        (num2 (sneps:node-to-lisp-object
              (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("

```

```

                (princ-to-string div2)
                ",?x"")))))))
(num3 (sneps:node-to-lisp-object
      (cdr (first (first (askwh (concatenate 'string
                                           "NumeralOf("
                                           (princ-to-string (cl:/ div1 div2))
                                           ",?x"")))))))

(tell (concatenate 'string
  "Evaluation(Result(CalcQuotient,"
  (princ-to-string div1) ", " (princ-to-string div2) ")","
  (princ-to-string (cl:/ div1 div2)) ")."))

(tell (concatenate 'string
  "Evaluation(Result(Quotient,"
  (princ-to-string num1) ", " (princ-to-string num2) ")","
  (princ-to-string num3) ")."))))

(attach-primaction CalcAdd CalcAdd CalcSubtract CalcSubtract
                  CalcMultiply CalcMultiply CalcDivide CalcDivide)
^^

ResultName(CalcSum).
ResultName(CalcDifference).
ResultName(CalcProduct).
ResultName(CalcQuotient).

all(x,nx,y,ny)({NumeralOf(x,nx),NumeralOf(y,ny)} &=>
  {ActPlan(Add(nx,ny),CalcAdd(x,y)),
   Effect(CalcAdd(x,y),Evaluated(Result(CalcSum,x,y))),
   ActPlan(Subtract(nx,ny),CalcSubtract(x,y)),
   Effect(CalcSubtract(x,y),Evaluated(Result(CalcDifference,x,y))),
   ActPlan(Multiply(nx,ny),CalcMultiply(x,y)),
   Effect(CalcMultiply(x,y),Evaluated(Result(CalcProduct,x,y))),
   ActPlan(Divide(nx,ny),CalcDivide(x,y)),
   Effect(CalcDivide(x,y),Evaluated(Result(CalcQuotient,x,y)))}).

all(x,nx)(NumeralOf(x,nx) =>
  {ActPlan(Add(nx,nx),CalcAdd(x,x)),
   Effect(CalcAdd(x,x), Evaluated(Result(CalcSum,x,x))),
   ActPlan(Subtract(nx,nx),CalcSubtract(x,x)),
   Effect(CalcSubtract(x,x), Evaluated(Result(CalcDifference,x,x))),
   ActPlan(Multiply(nx,nx),CalcMultiply(x,x)),
   Effect(CalcMultiply(x,x), Evaluated(Result(CalcProduct,x,x))),
   ActPlan(Divide(nx,nx),CalcDivide(x,x)),
   Effect(CalcDivide(x,x), Evaluated(Result(CalcQuotient,x,x)))}).

```

D. Syntactic Arithmetic

```
; =====
; FILENAME:      SynArith.snepslog
; DATE:         3/10/08
; PROGRAMMER:   Albert Goldfain
;
; Lines beginning with a semi-colon are comments.
; Regions delimited by "^^" are Lisp commands.
; All other lines are SNePSLOG commands.
;
; To use this file: run SNePSLOG; at the SNePSLOG prompt (:), type:
;
;     demo "SynArith.snepslog"
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; =====
;;;
;;;[[DigitString(x,y)]] = [[x]] is the digit string representation of [[y]]
;;;
define-frame DigitString(nil digit-string number)

;;;
;;;[[S(x,y)]] = A structured individual used to recursively build up the
;;;           left-recursive digit string [[x]] [[y]]. NOTE: The
;;;           code below only works for two-digit arithmetic.
;;;
define-frame S(nil head tail)

;;;
;;;[[AddInCols(x,y)]] = Act of adding digits [[x]] and [[y]] in the
;;; current column.
;;;
define-frame AddInCols(action arg1 arg2)

;;;
;;;[[AddDS(x,y)]] = Act of adding digit strings [[x]] and [[y]].
define-frame AddDS(action arg1 arg2)

;;;To add a and b in the current column, use generic Add routine
;;;and then probe for the evaluation.
all(a,b)({Number(a),Number(b)} &=>
  {ActPlan(AddInCols(a,b),
    snsequence(SayLine+("Adding Column: ",a,b),
      snsequence(Add(a,b),
        withsome(?r,
          Evaluation(Result(Sum,a,b),?r),
          snsequence(SayLine+("Column", "Result:",?r),
            snif({if(GreaterThan(?r,n9),
              SayLine("Carry for next column")),
              else(SayLine("No carry for next column"))})),
          SayLine("Unable to determine sum")))),
    ActPlan(AddInCols(a,a),
      snsequence(SayLine+("Adding Column: ",a,a),
        snsequence(Add(a,a),
          withsome(?r,
            Evaluation(Result(Sum,a,a),?r),
            snsequence(SayLine+("Column", "Result:",?r),
              snif({if(GreaterThan(?r,n9),
                SayLine("Carry for next column")),
                else(SayLine("No carry for next column"))})),
              SayLine("Unable to determine sum")))))).
```

```

;;;All digit-string permutations of AddDS act.
all(a,b,c,d)({Number(a),Number(b),Number(c),Number(d)} &=>
  ActPlan(AddDS(S(a,b),S(c,d)),
    snsequence(SayLine("Addition of the form S(a,b) + S(c,d)"),
      snsequence(AddInCols(b,d),
        AddInCols(a,c))))).

all(a,b,c)({Number(a),Number(b),Number(c)} &=>
  {ActPlan(AddDS(S(a,a),S(b,c)),
    snsequence(SayLine("Addition of the form S(a,a) + S(b,c)"),
      snsequence(AddInCols(a,c),
        AddInCols(a,b))),
  ActPlan(AddDS(S(a,b),S(a,c)),
    snsequence(SayLine("Addition of the form S(a,b) + S(a,c)"),
      snsequence(AddInCols(b,c),
        AddInCols(a,a))),
  ActPlan(AddDS(S(a,b),S(c,a)),
    snsequence(SayLine("Addition of the form S(a,b) + S(c,a)"),
      snsequence(AddInCols(b,a),
        AddInCols(a,c))),
  ActPlan(AddDS(S(a,b),S(b,c)),
    snsequence(SayLine("Addition of the form S(a,b) + S(b,c)"),
      snsequence(AddInCols(b,c),
        AddInCols(a,b))),
  ActPlan(AddDS(S(a,b),S(c,b)),
    snsequence(SayLine("Addition of the form S(a,b) + S(c,b)"),
      snsequence(AddInCols(b,b),
        AddInCols(a,c))),
  ActPlan(AddDS(S(a,b),S(c,c)),
    snsequence(SayLine("Addition of the form S(a,b) + S(c,c)"),
      snsequence(AddInCols(b,c),
        AddInCols(a,c))))}).

all(a,b)({Number(a),Number(b)} &=>
  {ActPlan(AddDS(S(a,a),S(b,b)),
    snsequence(SayLine("Addition of the form S(a,a) + S(b,b)"),
      snsequence(AddInCols(a,b),
        AddInCols(a,b))),
  ActPlan(AddDS(S(a,b),S(a,b)),
    snsequence(SayLine("Addition of the form S(a,b) + S(a,b)"),
      snsequence(AddInCols(b,b),
        AddInCols(a,a))),
  ActPlan(AddDS(S(a,b),S(b,a)),
    snsequence(SayLine("Addition of the form S(a,b) + S(b,a)"),
      snsequence(AddInCols(b,a),
        AddInCols(a,b))),
  ActPlan(AddDS(S(a,a),S(a,b)),
    snsequence(SayLine("Addition of the form S(a,a) + S(a,b)"),
      snsequence(AddInCols(a,b),
        AddInCols(a,a))),
  ActPlan(AddDS(S(a,b),S(a,a)),
    snsequence(SayLine("Addition of the form S(a,b) + S(a,a)"),
      snsequence(AddInCols(b,a),
        AddInCols(a,a))),
  ActPlan(AddDS(S(a,b),S(b,b)),
    snsequence(SayLine("Addition of the form S(a,b) + S(b,b)"),
      snsequence(AddInCols(b,b),
        AddInCols(a,b))))}).

all(a)(Number(a) =>
  ActPlan(AddDS(S(a,a),S(a,a)),
    snsequence(SayLine("Addition of the form S(a,a) + S(a,a)"),
      snsequence(AddInCols(a,a),
        AddInCols(a,a)))).

```

```
;;;Hook into the rest of the system.  
all(x,y,dx,dy)({DigitString(dx,x),DigitString(dy,y)} &=>  
  ActPlan(Add(x,y),AddDS(dx,dy))).
```

E. NLGCD

```
define-frame DivisorOf(rel arg1 arg2)
define-frame CommonDivisorOf(rel arg1 arg2 arg3)
define-frame DivisorCandidate(nil divisor-candidate)
define-frame Dividend(nil dividend)

define-frame UpdateDivisorCandidate(action number)
define-frame CreateDivisorList(action number)
define-frame CreateCommonList(action arg1 arg2)
define-frame FindGreatestCommonDivisor(action arg1 arg2)
define-frame NLGCD(action arg1 arg2)

all(d)(DivisorCandidate(d) =>
  ActPlan(UpdateDivisorCandidate(d),
    withsome(?dpl,
      Successor(?dpl,d),
      snsequence(disbelieve(DivisorCandidate(d)),
        believe(DivisorCandidate(?dpl))),
      SayLine("I don't know the successor")))).

all(nx)({Dividend(nx),DivisorCandidate(nx)} &=>
  ActPlan(CreateDivisorList(nx),
    snsequence(believe(DivisorOf(nx,nx)),
      snsequence(SayLine("Done with divisor list"),
        snsequence(disbelieve(DivisorCandidate(nx)),
          disbelieve(Dividend(nx)))))).

all(nx,d)({Dividend(nx),DivisorCandidate(d)} &=>
  ActPlan(CreateDivisorList(nx),
    snsequence(Divide(nx,d),
      withsome(?q,
        (Number(?q) and Evaluation(Result(Quotient,nx,d),?q)),
        snsequence(Say(d),
          snsequence(SayLine(" is a divisor "),
            snsequence(believe(DivisorOf(d,nx)),
              snsequence(UpdateDivisorCandidate(d),
                CreateDivisorList(nx))))),
          snsequence(Say(d),
            snsequence(SayLine(" is not a divisor"),
              snsequence(UpdateDivisorCandidate(d),
                CreateDivisorList(nx)))))).

all(nx,ny)({Number(nx),Number(ny)} &=>
  ActPlan(CreateCommonList(nx,ny),
    withall(?div,
      (DivisorOf(?div,nx) and DivisorOf(?div,ny)),
      believe(CommonDivisorOf(?div,nx,ny)),
      SayLine("problems creating common list")))).

all(nx,ny)({Number(nx),Number(ny)} &=>
  ActPlan(FindGreatestCommonDivisor(nx,ny),
    snsequence(SayLine("Now finding GCD"),
      snsequence(believe(Evaluation(Result(GCD,nx,ny),n1)),
        withall(?div,
          CommonDivisorOf(?div,nx,ny),
          withsome(?g,
            Evaluation(Result(GCD,nx,ny),?g),
            snif({if(GreaterThan(?div,?g),
              snsequence(disbelieve(Evaluation(Result(GCD,nx,ny),?g)),
                believe(Evaluation(Result(GCD,nx,ny),?div))),
```



```

        else(believe(Evaluation(Result(GCD,nx,ny),?div))))},
    SayLine("I can't find the GCD"),
    SayLine("GCD found")))).

```

```

;;;ActPlan for NLGCD
all(nx,ny,z)({Number(nx),Number(ny),Zero(z)} &=>
    ActPlan(NLGCD(nx,ny),
        withsome(?one,
            Successor(?one,z),
            snsequence(believe(DivisorCandidate(?one)),
                snsequence(believe(Dividend(nx)),
                    snsequence(CreateDivisorList(nx),
                        snsequence(believe(DivisorCandidate(?one)),
                            snsequence(believe(Dividend(ny)),
                                snsequence(CreateDivisorList(ny),
                                    snsequence(CreateCommonList(nx,ny),
                                        FindGreatestCommonDivisor(nx,ny)))))))))},
        SayLine("Problem with NLGCD.")))).

```

F. NLLCM

```
define-frame MultipleOf(rel arg1 arg2)
define-frame CommonMultipleOf(rel arg1 arg2 arg3)
define-frame CurrentBase(nil current-base)
define-frame CurrentMultiple(nil current-multiple)
define-frame GreatestCandidateLCM(nil greatest-candidate-lcm)
define-frame UpdateCurrentMultiple(action arg1 arg2)
define-frame CreateMultipleList(action number)
define-frame CreateCommonMultipleList(action arg1 arg2)
define-frame FindLeastCommonMultiple(action arg1 arg2)
define-frame NLLCM(action arg1 arg2)

;Plan for updating the current multiple
all(nx,m)({CurrentBase(nx),CurrentMultiple(m)} &=>
  ActPlan(UpdateCurrentMultiple(m,nx),
    ssequence(SayLine+"Updating with rule 1", m, nx),
    ssequence(Add(m,nx),
      withsome(?s,
        Evaluation(Result(Sum,m,nx),?s),
        ssequence(disbelieve(CurrentMultiple(m)),
          believe(CurrentMultiple(?s))),
        SayLine("I don't know the sum")))).

;UVBR nx for m
all(nx)(CurrentMultiple(nx)=>
  ActPlan(UpdateCurrentMultiple(nx,nx),
    ssequence(SayLine+"Updating with rule 2",nx,nx),
    ssequence(Add(nx,nx),
      withsome(?s,
        Evaluation(Result(Sum,nx,nx),?s),
        ssequence(disbelieve(CurrentMultiple(nx)),
          believe(CurrentMultiple(?s))),
        SayLine("I don't know the sum")))).

;Plan for creating a multiple list for nx
all(nx,m,gclcm)({CurrentBase(nx),CurrentMultiple(m),GreatestCandidateLCM(gclcm)} &=>
  ActPlan(CreateMultipleList(nx),
    ssequence(SayLine("UsingRule1"),
      snif({if(GreaterThan(gclcm,m),
        ssequence(believe(MultipleOf(m,nx)),
          ssequence(UpdateCurrentMultiple(m,nx),
            CreateMultipleList(nx))),
        else(SayLine("Done with multiple list"))}))).

;UVBR nx for m
all(nx,gclcm)({CurrentBase(nx),CurrentMultiple(nx),GreatestCandidateLCM(gclcm)} &=>
  ActPlan(CreateMultipleList(nx),
    ssequence(SayLine("UsingRule2"),
      snif({if(GreaterThan(gclcm,nx),
        ssequence(believe(MultipleOf(nx,nx)),
          ssequence(UpdateCurrentMultiple(nx,nx),
            CreateMultipleList(nx))),
        else(SayLine("Done with multiple list"))}))).

;UVBR nx for gclcm
all(nx,m)({CurrentBase(nx),CurrentMultiple(m),GreatestCandidateLCM(nx)} &=>
  ActPlan(CreateMultipleList(nx),
    snif({if(GreaterThan(nx,m),
      ssequence(believe(MultipleOf(m,nx)),
        ssequence(UpdateCurrentMultiple(m,nx),
```

```

        CreateMultipleList(nx))),
        else(SayLine("Done with multiple list")))))).

;UVBR nx for m and nx for gclcm
all(nx)({CurrentBase(nx),CurrentMultiple(nx),GreatestCandidateLCM(nx)} &=>
    ActPlan(CreateMultipleList(nx),
        sniff({if(GreaterThan(nx,nx),
            ssequence(believe(MultipleOf(nx,nx))),
            ssequence(UpdateCurrentMultiple(nx,nx),
                CreateMultipleList(nx))),
            else(SayLine("Done with multiple list")))))).

;UVBR m for gclcm
all(nx,m)({CurrentBase(nx),CurrentMultiple(m),GreatestCandidateLCM(m)} &=>
    ActPlan(CreateMultipleList(nx),
        ssequence(believe(MultipleOf(m,nx)),
            SayLine("Done with multiple list")))).

;Plan for creating common multiple list
all(nx,ny)({Number(nx),Number(ny)} &=>
    ActPlan(CreateCommonMultipleList(nx,ny),
        withall(?m,
            (MultipleOf(?m,nx) and MultipleOf(?m,ny)),
            believe(CommonMultipleOf(?m,nx,ny)),
            SayLine("problems creating common multiple list")))).

;Plan for finding the LCM
all(nx,ny,gclcm)({Number(nx),Number(ny),GreatestCandidateLCM(gclcm)} &=>
    ActPlan(FindLeastCommonMultiple(nx,ny),
        ssequence(SayLine("Now finding LCM"),
            ssequence(believe(Evaluation(Result(LCM,nx,ny),gclcm)),
                withall(?m,
                    CommonMultipleOf(?m,nx,ny),
                    withsome(?l,
                        Evaluation(Result(LCM,nx,ny),?l),
                        sniff({if(GreaterThan(?l,?m),
                            ssequence(disbelieve(Evaluation(Result(LCM,nx,ny),?l)),
                                believe(Evaluation(Result(LCM,nx,ny),?m))),
                            else(believe(Evaluation(Result(LCM,nx,ny),?l)))}),
                        SayLine("I can't find the LCM")),
                    SayLine("LCM found")))))).

;;;ActPlan for NLLCM
all(nx,ny)({Number(nx),Number(ny)} &=>
    ActPlan(NLLCM(nx,ny),
        ssequence(Multiply(nx,ny),
            ssequence(withsome(?p,
                Evaluation(Result(Product,nx,ny),?p),
                believe(GreatestCandidateLCM(?p)),
                SayLine("I don't know the product")),
            ssequence(believe(CurrentMultiple(nx)),
                ssequence(believe(CurrentBase(nx)),
                    ssequence(CreateMultipleList(nx),
                        ssequence(disbelieve(CurrentBase(nx)),
                            ssequence(withsome(?c,
                                CurrentMultiple(?c),
                                disbelieve(CurrentMultiple(?c)),
                                SayLine("There's a problem")),
                            ssequence(believe(CurrentMultiple(ny)),
                                ssequence(believe(CurrentBase(ny)),
                                    ssequence(CreateMultipleList(ny),
                                        ssequence(disbelieve(CurrentBase(ny)),
                                            ssequence(withsome(?c,

```

```

CurrentMultiple(?c),
disbelieve(CurrentMultiple(?c)),
SayLine("There's a problem"),
snsequence(CreateCommonMultipleList(nx,ny),
snsequence(FindLeastCommonMultiple(nx,ny),
withsome(?p,
Evaluation(Result(Product,nx,ny),?p),
believe(GreatestCandidateLCM(?p)),
SayLine("I don't know the product")))))))))))))).

```

G. Question Parser

```
define-frame answerQuestion(action question)

^^
;;PMLb

;;;<QUESTION> := <ISDOES> <EVAL> | <WHY> <EVAL> | <WHAT> <RES> | <HOW><RES> | <DEFINE> <CONCEPT> ;
;;;<ISDOES> := is | does ;
;;;<WHY> := why <ISDOES>;
;;;<WHAT> := what is;
;;;<HOW> := how <DOCAN> you <COMPUTE>;
;;;<DEFINE> := define
;;;<DOCAN> := do | can;
;;;<COMPUTE> := ((compute | find) <RES>) | <GENERALACT>;
;;;<GENERALACT> := ((add | multiply) <NUMBER> and <NUMBER>) |
;;;                (subtract <NUMBER> from <NUMBER>) |
;;;                (divide <NUMBER> by <NUMBER>);
;;;<EVAL> := <RES> = <NUMBER>;
;;;<RES> := <NUMBER> <OP> <NUMBER>;
;;;<OP> := + | - | * | / ;

(defparameter *WhiteSpace* '(#\Space #\Newline #\Tab #\Page #\Return)
  "Characters to be ignored")

(defparameter *QType* 'unknown)

(defun parseQuestion (question)
  (format t "~%Parsing <QUESTION>")
  (setf pos 0)
  (setf *Input* question)
  (setf *QType* 'unknown)
  (setf *RType* 'unknown)
  (cond ((string= (subseq *Input* pos (+ pos 2)) "is") (parseIsDoes (+ pos 3)))
        ((string= (subseq *Input* pos (+ pos 4)) "does") (parseIsDoes (+ pos 5)))
        ((string= (subseq *Input* pos (+ pos 3)) "why") (parseWhy (+ pos 4)))
        ((string= (subseq *Input* pos (+ pos 4)) "what") (parseWhat (+ pos 5)))
        ((string= (subseq *Input* pos (+ pos 3)) "how") (parseHow (+ pos 4)))
        ((string= (subseq *Input* pos (+ pos 6)) "define") (parseDefine (+ pos 7)))
        (t (format t "Question Parse Error"))))

(defun parseIsDoes (pos)
  (format t "~%Parsing <ISDOES>")
  (when (equal *QType* 'unknown) (setf *QType* 'is-does-question))
  (parseEval pos)
  (cond ((equal *RType* '+) (setf *ResultName* "Sum"))
        ((equal *RType* '-') (setf *ResultName* "Difference"))
        ((equal *RType* '*') (setf *ResultName* "Product"))
        ((equal *RType* '/') (setf *ResultName* "Quotient")))
  (setf *SNePSLOG-Q* (concatenate 'string "Evaluation(Result("
                                     *ResultName* ",n" *Arg1* ",n" *Arg2* "),n" *Number* ")?"))))

(defun parseWhy (pos)
  (format t "~%Parsing <WHY>")
  (when (equal *QType* 'unknown) (setf *QType* 'why-question))
  (cond ((string= (subseq *Input* pos (+ pos 2)) "is") (parseEval (+ pos 3)))
        ((string= (subseq *Input* pos (+ pos 4)) "does") (parseEval (+ pos 5))))
  (cond ((equal *RType* '+) (setf *ResultName* "Sum"))
        ((equal *RType* '-') (setf *ResultName* "Difference"))
        ((equal *RType* '*') (setf *ResultName* "Product"))
        ((equal *RType* '/') (setf *ResultName* "Quotient")))
  (setf *SNePSLOG-Q*
    (concatenate 'string "perform Explain(Evaluation(Result("
                                                              *ResultName* ",n" *Arg1* ",n" *Arg2* "),n" *Number* "))"))))
```

```

(defun parseWhat (pos)
  (format t "~%Parsing <WHAT>")
  (when (equal *QType* 'unknown) (setf *QType* 'what-question))
  (cond ((string= (subseq *Input* pos (+ pos 2)) "is") (parseRes (+ pos 3)))
        (t (format t "Not handling yet")))
  (cond ((equal *RType* '+) (setf *ResultName* "Sum"))
        ((equal *RType* '-') (setf *ResultName* "Difference"))
        ((equal *RType* '*') (setf *ResultName* "Product"))
        ((equal *RType* '/') (setf *ResultName* "Quotient")))
  (setf *SNePSLOG-Q*
    (concatenate 'string "Evaluation(Result("
      *ResultName* ",n" *Arg1* ",n" *Arg2* " ),?x)?"))))

(defun parseHow (pos)
  (format t "~%Parsing <HOW>")
  (setf *ActType* nil)
  (when (equal *QType* 'unknown) (setf *QType* 'how-question))
  (cond ((string= (subseq *Input* pos (+ pos 6)) "do you") (parseCompute (+ pos 7)))
        ((string= (subseq *Input* pos (+ pos 7)) "can you") (parseCompute (+ pos 8))))
  (cond ((equal *RType* '+) (setf *ResultName* "Sum"))
        ((equal *RType* '-') (setf *ResultName* "Difference"))
        ((equal *RType* '*') (setf *ResultName* "Product"))
        ((equal *RType* '/') (setf *ResultName* "Quotient")))
  (if *ActType*
    (setf *SNePSLOG-Q*
      (concatenate 'string "ActPlan(" *ActType* "(n" *Arg1* ",n"
        *Arg2* " ),?x)?"))
    (setf *SNePSLOG-Q*
      (concatenate 'string "Effect(?x,Evaluated(Result("
        *ResultName* ",n" *Arg1* ",n" *Arg2* " )))?"))))

(defun parseDefine (pos)
  (format t "~%Parsing <DEFINE>")
  (when (equal *QType* 'unknown) (setf *QType* 'definition))
  (setf *concept-to-define* (subseq *Input* pos (- (length *Input*) 1)))
  (setf *SNePSLOG-Q*
    (concatenate 'string "perform ssequence(believe(ConceptToDefine("
      (princ-to-string *concept-to-define*
        ")),Define("
      (princ-to-string *concept-to-define*
        ")))"))))

(defun parseCompute (pos)
  (format t "~%Parsing <COMPUTE>")
  (cond ((string= (subseq *Input* pos (+ pos 3)) "add")
        (parseGeneralAct "Add" (+ pos 4)))
        ((string= (subseq *Input* pos (+ pos 6)) "divide")
        (parseGeneralAct "Divide" (+ pos 7)))
        ((string= (subseq *Input* pos (+ pos 8)) "subtract")
        (parseGeneralAct "Subtract" (+ pos 9)))
        ((string= (subseq *Input* pos (+ pos 8)) "multiply")
        (parseGeneralAct "Multiply" (+ pos 9)))
        ((string= (subseq *Input* pos (+ pos 4)) "find") (parseRes (+ pos 5)))
        ((string= (subseq *Input* pos (+ pos 7)) "compute") (parseRes (+ pos 8)))))

(defun parseGeneralAct (generalact pos)
  (format t "~%Parsing <GENERALACT>")
  (setf *ActType* generalact)
  (cond ((or (string= *ActType* "Add") (string= *ActType* "Multiply"))
        (parseArg2 (parseAnd (parseArg1 pos))))
        ((string= *ActType* "Subtract")
        (parseArg1 (parseFrom (parseArg2 pos))))
        ((string= *ActType* "Divide")
        (parseArg2 (parseBy (parseArg1 pos)))))

```

```

(defun parseEval (pos)
  (format t "~%Parsing <EVAL>")
  (parseNumber (parseEquals (parseRes pos))))

(defun parseRes (pos)
  (format t "~%Parsing <RES>")
  (parseArg2 (parseOp (parseArg1 pos))))

(defun parseOp (pos)
  (format t "~%Parsing <OP>")
  (cond ((string= (subseq *Input* pos (+ pos 1)) "+") (setf *RType* '+))
        ((string= (subseq *Input* pos (+ pos 1)) "-") (setf *RType* '-))
        ((string= (subseq *Input* pos (+ pos 1)) "*") (setf *RType* '*))
        ((string= (subseq *Input* pos (+ pos 1)) "/") (setf *RType* '/))
        (t (format t "Error parsing <OP>")))
  (+ pos 2))

(defun parseArg1 (pos)
  (format t "~%Parsing <ARG1>")
  (setf *Arg1* (subseq *Input* pos (+ pos 1)))
  (+ pos 2))

(defun parseArg2 (pos)
  (format t "~%Parsing <ARG2>")
  (setf *Arg2* (subseq *Input* pos (+ pos 1)))
  (+ pos 2))

(defun parseEquals (pos)
  (format t "~%Parsing =")
  (+ pos 2))

(defun parseAnd (pos)
  (format t "~%Parsing and")
  (+ pos 4))

(defun parseFrom (pos)
  (format t "~%Parsing from")
  (+ pos 5))

(defun parseBy (pos)
  (format t "~%Parsing by")
  (+ pos 3))

(defun parseNumber (pos)
  ;;make this better
  (format t "~%Parsing <NUMBER>")
  (setf *Number* (subseq *Input* pos (+ pos 1)))
  (+ pos 2))

;;PMLa
(define-primaction answerQuestion ((question))
  (let ((q (parseQuestion (princ-to-string question))))
    (format t "~%~%Question in SNePSLOG:~A~%~%" q)
    (cond ((equal *QType* 'why-question) (snepslog:tell q))
          ((equal *QType* 'is-does-question) (snepslog:ask q :verbose t))
          ((equal *QType* 'what-question)
           (snepslog:askwh q :verbose t))
          ((equal *QType* 'how-question)
           (snepslog:askwh q :verbose t))
          ((equal *QType* 'definition)
           (snepslog:tell q))))))

```

(attach-primaction answerQuestion answerQuestion)
^^

H. Exhaustive Explanation

```
define-frame Explain(action evaluation)
define-frame IsFactual(action evaluation)
define-frame ActExplain(action result)
define-frame SayDescrip(action utterance)

^^
(define-primaction SayDescrip (utterance)
  "DESCRIPTION"
  (snepslog:snepslog-print utterance))

(attach-primaction SayDescrip SayDescrip)
^^

all(rn,a1,a2)({ResultName(rn),Number(a1),Number(a2)} &=>
  ActPlan(ActExplain(Result(rn,a1,a2)),
    withsome(?a,
      Effect(?a,Evaluated(Result(rn,a1,a2))),
      snsequence(SayDescrip(?a),
        snsequence(SayLine("And I can do this by performing"),
          withsome(?p,
            ActPlan(?a,?p),
            snsequence(SayDescrip(?p),
              snsequence(Say("Which has the effect(s) of"),
                withsome(?e,
                  Effect(?p,?e),
                  SayDescrip(?e),
                  SayLine("I don't knowthe effect(s) of that")))),
                SayLine("I don't know any plans for that")))),
            SayLine("I don't know how to evaluate this result")))).

all(rn,a1,a2,r)({ResultName(rn),Number(a1),Number(a2),Number(r)} &=>
  {ActPlan(IsFactual(Evaluation(Result(rn,a1,a2),r)),
    snif({if(Evaluation(Result(rn,a1,a2),r),
      SayLine("I know that the result is correct")),
      else(SayLine("I don't yet know that the result is correct"))})),
  ActPlan(Explain(Evaluation(Result(rn,a1,a2),r)),
    snsequence(IsFactual(Evaluation(Result(rn,a1,a2),r)),
      snsequence(SayLine("Here's HOW I can find the result"),
        ActExplain(Result(rn,a1,a2)))))).

;;;UVBR a1 for a2
all(rn,a1)({ResultName(rn),Number(a1)} &=>
  ActPlan(ActExplain(Result(rn,a1,a1)),
    withsome(?a,
      Effect(?a,Evaluated(Result(rn,a1,a1))),
      snsequence(SayDescrip(?a),
        snsequence(SayLine("And I can do this by performing"),
          withsome(?p,
            ActPlan(?a,?p),
            snsequence(SayDescrip(?p),
              snsequence(Say("Which has the effect(s) of"),
                withsome(?e,
                  Effect(?p,?e),
                  SayDescrip(?e),
                  SayLine("I don't knowthe effect(s) of that")))),
                SayLine("I don't know any plans for that")))),
            SayLine("I don't know how to evaluate this result")))).
```

```

all(rn,a1,r)({ResultName(rn),Number(a1),Number(r)} &=>
  {ActPlan(IsFactual(Evaluation(Result(rn,a1,a1),r)),
    sniff({if(Evaluation(Result(rn,a1,a1),r),
      SayLine("I know that the result is correct")),
      else(SayLine("I don't yet know that the result is correct"))})),
  ActPlan(Explain(Evaluation(Result(rn,a1,a1),r)),
    snsequence(IsFactual(Evaluation(Result(rn,a1,a1),r)),
    snsequence(SayLine("Here's HOW I can find the result"),
      ActExplain(Result(rn,a1,a1)))))).

;;;UVBR a1 for r
all(rn,a1,a2)({ResultName(rn),Number(a1),Number(a2)} &=>
  {ActPlan(IsFactual(Evaluation(Result(rn,a1,a2),a1)),
    sniff({if(Evaluation(Result(rn,a1,a2),a1),
      SayLine("I know that the result is correct")),
      else(SayLine("I don't yet know that the result is correct"))})),
  ActPlan(Explain(Evaluation(Result(rn,a1,a2),a1)),
    snsequence(IsFactual(Evaluation(Result(rn,a1,a2),a1)),
    snsequence(SayLine("Here's HOW I can find the result"),
      ActExplain(Result(rn,a1,a2)))))).

;;;UVBR a2 for r
all(rn,a1,a2)({ResultName(rn),Number(a1),Number(a2)} &=>
  {ActPlan(IsFactual(Evaluation(Result(rn,a1,a2),a2)),
    sniff({if(Evaluation(Result(rn,a1,a2),a2),
      SayLine("I know that the result is correct")),
      else(SayLine("I don't yet know that the result is correct"))})),
  ActPlan(Explain(Evaluation(Result(rn,a1,a2),a2)),
    snsequence(IsFactual(Evaluation(Result(rn,a1,a2),a2)),
    snsequence(SayLine("Here's HOW I can find the result"),
      ActExplain(Result(rn,a1,a2)))))).

```

I. Conceptual Definition

```
define-frame ConceptToDefine(nil concept-to-define)

define-frame GetClassMembers(action classtype)
define-frame GetClassContainment(action member-inst)
define-frame GetActsWithArgumentType(action argtype)
define-frame GetOperationsWithResultType(action resulttype)
define-frame GetMannerOfRelationships(action procedure)
define-frame GetGeneralResultType(action resulttype)
define-frame ProvideExampleOfResult(action result-inst)
define-frame ProvideExampleOfRel(action relation)
define-frame Define(action entity)
^^

(define-primaction GetClassMembers (classtype)
"Determines the members of the given class type"
  (setf result #!((find (member- ! class) ~classtype)))
  (when result (format t "~&~A has the following class membership~%~A~%"
    (sneps:choose.ns classtype) result))
)

(define-primaction GetClassContainment (member-inst)
"Determines the classes of the given member instance"
  (setf result #!((find (class- ! member) ~member-inst)))
  (when result (format t "~&~A is a ~%~A~%"
    (sneps:choose.ns member-inst) result))
)

(define-primaction GetActsWithArgumentType (argtype)
"Determines which acts have the given argument type"
  (setf result #!((find (action- act- cq- ant class) ~argtype)))
  (when result (format t "~&~A is an argument for the following acts~%~A~%"
    (sneps:choose.ns argtype) result))
)

(define-primaction GetOperationsWithResultType (resulttype)
"Determines which operations have at least one instance of the given
result type"
  (setf result #!((find (result-name- ! value member- class) ~resulttype)))
  (when result (format t "~&~A is the result type of the following
operations I have already performed~%~A~%"
    (sneps:choose.ns resulttype) result))
)

(define-primaction GetMannerOfRelationships (procedure)
"Determines the ways in which the given procedure can be executed"
  (setf result #!((find (action- plan- act action) ~procedure)))
  (when result (format t "~&~A can be performed via the following acts~%~A~%"
    (sneps:choose.ns procedure) result))
)

(define-primaction GetGeneralResultType (resulttype)
"Determines the general type of result for the given particular result
type"
  (setf result #!((find (result-name- cq- ant result-name) ~resulttype)))
  (when result (format t "~&~A is a specific instance of the following kinds of
result ~%~A~%"
    (sneps:choose.ns resulttype) result))
)

(define-primaction ProvideExampleOfResult (result-inst)
```

```

"Finds one grounded example for the given result"

;(format t "Some examples of ~A as a result ~%" result-inst)

#!((perform (build action withsome
  vars ($x $y $z)
  suchthat (build arg1 *x arg2 *y result ~result-inst
  op *z)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence
    object1 (build action Say object *z)
    object2
    (build action snsequence
    object1 (build action Say object " of ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " and ")
    object2
    (build action snsequence
    object1 (build action Say object *y)
    object2
    (build action snsequence
    object1 (build action Say object " is ")
    object2 (build action SayLine object ~result-inst)))))))))
  else (build action Say object " ")))

;;;UVBR x for y
#!((perform (build action withsome
  vars ($x $z)
  suchthat (build arg1 *x arg2 *x result ~result-inst
  op *z)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence
    object1 (build action Say object *z)
    object2
    (build action snsequence
    object1 (build action Say object " of ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " and ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " is ")
    object2 (build action SayLine object ~result-inst)))))))))
  else (build action Say object " ")))

)

(define-primaction ProvideExampleOfRel (relation)
"Finds one grounded example for the given relation"

```

```

;(format t "Some examples of the ~A relation~%" relation)

#l((perform (build action withsome
  vars ($x $y $z)
  suchthat (build arg1 *x arg2 *y result *z
    op ~relation)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence
    object1 (build action Say object ~relation)
    object2
    (build action snsequence
    object1 (build action Say object " of ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " and ")
    object2
    (build action snsequence
    object1 (build action Say object *y)
    object2
    (build action snsequence
    object1 (build action Say object " is ")
    object2 (build action SayLine object *z)))))))))
  else (build action Say object " "))))

;;;UVBR x for z
#l((perform (build action withsome
  vars ($x $y)
  suchthat (build arg1 *x arg2 *y result *x
    op ~relation)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence
    object1 (build action Say object ~relation)
    object2
    (build action snsequence
    object1 (build action Say object " of ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " and ")
    object2
    (build action snsequence
    object1 (build action Say object *y)
    object2
    (build action snsequence
    object1 (build action Say object " is ")
    object2 (build action SayLine object *x)))))))))
  else (build action Say object " "))))

;;;UVBR x for y
#l((perform (build action withsome
  vars ($x $z)
  suchthat (build arg1 *x arg2 *x result *z
    op ~relation)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence

```

```

        object1 (build action Say object ~relation)
        object2
        (build action snsequence
        object1 (build action Say object " of ")
        object2
        (build action snsequence
        object1 (build action Say object *x)
        object2
        (build action snsequence
        object1 (build action Say object " and ")
        object2
        (build action snsequence
        object1 (build action Say object *x)
        object2
        (build action snsequence
        object1 (build action Say object " is ")
        object2 (build action SayLine object *z)))))))))
    else (build action Say object " "))))
;;; UVBR x for y and x for z
#!((perform (build action withsome
  vars $x
  suchthat (build arg1 *x arg2 *x result *x
  op ~relation)
  do (build action snsequence
    object1 (build action Say object "The ")
    object2
    (build action snsequence
    object1 (build action Say object ~relation)
    object2
    (build action snsequence
    object1 (build action Say object " of ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " and ")
    object2
    (build action snsequence
    object1 (build action Say object *x)
    object2
    (build action snsequence
    object1 (build action Say object " is ")
    object2 (build action SayLine object *x)))))))))
  else (build action Say object " ")))
)
(attach-primaction
  GetActsWithArgumentType GetActsWithArgumentType
  GetClassMembers GetClassMembers
  GetClassContainment GetClassContainment
  GetOperationsWithResultType GetOperationsWithResultType
  GetMannerOfRelationships GetMannerOfRelationships
  GetGeneralResultType GetGeneralResultType
  ProvideExampleOfResult ProvideExampleOfResult
  ProvideExampleOfRel ProvideExampleOfRel)
^^
;;;
;;;Define(x): Provides a CVA-style definition by determining class-membership,
;;; argument and result relationships, and manner-of realtionships
;;; between x and other concepts in the network. If x is a relation,
;;; this routine also tries to give an example of the relation.
;;;
;;;

```

```
all(x)(ConceptToDefine(x) => ActPlan(Define(x),
    ssequence(GetClassMembers(x),
    ssequence(GetClassContainment(x),
    ssequence(GetActsWithArgumentType(x),
    ssequence(GetOperationsWithResultType(x),
    ssequence(GetMannerOfRelationships(x),
    GetGeneralResultType(x)))))).

;;;ssequence(ProvideExampleOfResult(x),
;;;ProvideExampleOfRel(x)))))).
```

J. Embodied Enumeration

```
define-frame NumSort(nil number sortal)
define-frame Collection(nil collection quantity)
define-frame Construction(nil construction quantity)
define-frame Measure(nil measure quantity)

define-frame Name(nil object name)
define-frame InstanceOf(nil particular universal)

;;; Implementation detail: use arcs embobject embproperty to
;;; avoid redefinition from Evaluated frame
define-frame HasProperty(nil embobject embproperty)
define-frame ConstituentOf(nil constituent collection)

define-frame TargetColor(nil target-color)
define-frame TargetShape(nil target-shape)
define-frame Color(class member)
define-frame Shape(class member)

define-frame Image(class member)
define-frame ObjectsLeft(status image)

define-frame BuildEnumFrame(action sortal)
define-frame Perceive(action image)
define-frame DistinguishByColor(action color thresh)
define-frame DistinguishByShape(action shape thresh)
define-frame AddToCollection(action sortal collection)
define-frame AttendToNewObject(action image)
define-frame CheckIfClear(action image)
define-frame UpdateCollection(action object)

define-frame EnumerateApples(action image)

^^
(defclass BinaryImg ()
  ((width :accessor img-width :initarg :width)
   (height :accessor img-height :initarg :height)
   (pixels :accessor img-pixels :initarg :pixels)))

(defclass RGB ()
  ((width :accessor img-width :initarg :width)
   (height :accessor img-height :initarg :height)
   (red-pixels :accessor img-red-pixels :initarg :red-pixels)
   (green-pixels :accessor img-green-pixels :initarg :green-pixels)
   (blue-pixels :accessor img-blue-pixels :initarg :blue-pixels)))

;;;
;;; Default prototype color pallete.
;;;
(defconstant *COLOR-PALLETE* (make-hash-table :test #'equal)
  "Key is color name; Value is list of component values (R G B)" )

(setf (gethash 'Red *COLOR-PALLETE*) '(255 0 0))
(setf (gethash 'Orange *COLOR-PALLETE*) '(255 128 0))
(setf (gethash 'Yellow *COLOR-PALLETE*) '(255 255 0))
(setf (gethash 'Green *COLOR-PALLETE*) '(0 255 0))
(setf (gethash 'Blue *COLOR-PALLETE*) '(0 0 255))
(setf (gethash 'Purple *COLOR-PALLETE*) '(255 0 128))

(defun pixel-error (candidate-val target-val)
  "Computes absolute single pixel error"
  (abs (cl:- candidate-val target-val)))
```



```

(defun component-color-error (img target-color)
  "Computes component color pixel error for the given img and target-color"
  (let ((pix-err-red nil) (pix-err-green nil) (pix-err-blue nil))
    (make-instance 'RGB
      :width (img-width img)
      :height (img-height img)
      :red-pixels (dolist (pix (img-red-pixels img) pix-err-red)
        (setf pix-err-red
              (append pix-err-red
                      (list (pixel-error pix
                                       (first (gethash target-color *COLOR-PALLETE*)))))))
      :green-pixels (dolist (pix (img-green-pixels img) pix-err-green)
        (setf pix-err-green
              (append pix-err-green
                      (list (pixel-error pix
                                       (second (gethash target-color *COLOR-PALLETE*)))))))
      :blue-pixels (dolist (pix (img-blue-pixels img) pix-err-blue)
        (setf pix-err-blue
              (append pix-err-blue
                      (list (pixel-error pix
                                       (third (gethash target-color *COLOR-PALLETE*))))))))))

(defun pixel-err-sum (red-err green-err blue-err)
  (cl:+ red-err green-err blue-err))

(defun pixel-classify (err-sum thresh)
  (if (> thresh err-sum) 1 0))

(defun threshold-classify (img thresh)
  "Performs a binary classification of each pixel in img whose total component error is less than thresh"
  (let ((binary-classif nil))
    (make-instance 'BinaryImg
      :width (img-width img)
      :height (img-height img)
      :pixels (dolist (err-sum
                      (mapcar #'pixel-err-sum
                              (img-red-pixels img)
                              (img-green-pixels img)
                              (img-blue-pixels img)))
      (binary-classif)
      (setf binary-classif
            (append binary-classif
                    (list (pixel-classify err-sum thresh)))))))
    ;;*****
    ;;*****Lisp Matrix Representation*****
    ;;*****

(defun firstn (n l)
  "Returns the first n elements of list l"
  (if (cl:= n 0)
      nil
      (append (list (first l)) (firstn (cl:- n 1) (rest l)))))

(defun val (m i j)
  "Returns the value in matrix M at index I,J"
  (nth j (nth i m)))

(defun assign (m i j v)
  "Assigns value v in matrix m at position i j"
  (let ((row (nth i m)))
    (append (firstn i m)
            (list (append (firstn j row) (list v) (nthcdr (cl:+ j 1) row)))
            (nthcdr (cl:+ i 1) m))))

;;list to array conversion
(defun list-to-array (l w h)

```

```

"Returns a W x H array filled with the elements of L"
(let ((matrix nil))
  (dotimes (i h matrix)
    (setf matrix (append matrix (list (firstn w (nthcdr (cl:* i w) 1)))))))

(defun print-matrix (m)
  (if (endp m)
      (format nil "")
      (format nil "~A~%" (first m) (print-matrix (rest m)))))

(defun check-if-clear-helper (m i j maxrow)
  (if (cl:= i maxrow)
      t
      (if (cl:= 1 (val m i j))
          nil
          (if (cl:< j (cl:- (length (first m)) 1))
              (check-if-clear-helper m i (cl:+ j 1) maxrow)
              (check-if-clear-helper m (cl:+ i 1) 0 maxrow)))))

(defun check-if-clear (m maxrow)
  (check-if-clear-helper m 0 0 maxrow))

;;*****
;;*****CHAIN CODING*****
;;*****

(defun top-left-object-pixel-helper (m i j)
  (if (cl:= 1 (val m i j))
      (list i j)
      (if (cl:< j (cl:- (length (first m)) 1))
          (top-left-object-pixel-helper m i (cl:+ j 1))
          (top-left-object-pixel-helper m (cl:+ i 1) 0))))

(defun top-left-object-pixel (m)
  "Returns the coordinates of the top leftmost object pixel in m"
  (top-left-object-pixel-helper m 0 0))

(defun clockwise-chain-code (m f si sj i j dir)
  "Add description"
  (if (and (cl:= i si) (cl:= j sj) (cl:= f 1))
      nil
      (cond
       ((equal dir 'E)
        (cond((cl:= 1 (val m (cl:- i 1) (cl:- j 1))) (append (list 'NW) (clockwise-chain-code m 1
                                                                    si sj (cl:- i 1) (cl:- j 1) 'NW)))
              ((cl:= 1 (val m (cl:- i 1) j)) (append (list 'N) (clockwise-chain-code m 1
                                                                    si sj (cl:- i 1) j 'N)))
              ((cl:= 1 (val m (cl:- i 1) (cl:+ j 1))) (append (list 'NE) (clockwise-chain-code m 1
                                                                    si sj (cl:- i 1) (cl:+ j 1) 'NE)))
              ((cl:= 1 (val m i (cl:+ j 1))) (append (list 'E) (clockwise-chain-code m 1
                                                                    si sj i (cl:+ j 1) 'E)))
              ((cl:= 1 (val m (cl:+ i 1) (cl:+ j 1))) (append (list 'SE) (clockwise-chain-code m 1
                                                                    si sj (cl:+ i 1) (cl:+ j 1) 'SE)))
              ((cl:= 1 (val m (cl:+ i 1) j)) (append (list 'S) (clockwise-chain-code m 1
                                                                    si sj (cl:+ i 1) j 'S)))
              ((cl:= 1 (val m (cl:+ i 1) (cl:- j 1))) (append (list 'SW) (clockwise-chain-code m 1
                                                                    si sj (cl:+ i 1) (cl:- j 1) 'SW))))))
       ((equal dir 'SE)
        (cond((cl:= 1 (val m (cl:- i 1) j)) (append (list 'N) (clockwise-chain-code m 1
                                                                    si sj (cl:- i 1) j 'N)))
              ((cl:= 1 (val m (cl:- i 1) (cl:+ j 1))) (append (list 'NE) (clockwise-chain-code m 1
                                                                    si sj (cl:- i 1) (cl:+ j 1) 'NE)))
              ((cl:= 1 (val m i (cl:+ j 1))) (append (list 'E) (clockwise-chain-code m 1
                                                                    si sj i (cl:+ j 1) 'E)))
              ((cl:= 1 (val m (cl:+ i 1) (cl:+ j 1))) (append (list 'SE) (clockwise-chain-code m 1
                                                                    si sj (cl:+ i 1) (cl:+ j 1) 'SE)))))))))

```



```

      ((cl:= 1 (val m i (cl:+ j 1))) (append (list 'E) (clockwise-chain-code m 1
                                                    si sj i (cl:+ j 1) 'E))))
((equal dir 'N)
 (cond((cl:= 1 (val m (cl:+ i 1) (cl:- j 1))) (append (list 'SW) (clockwise-chain-code m 1
                                                         si sj (cl:+ i 1) (cl:- j 1) 'SW)))
      ((cl:= 1 (val m i (cl:- j 1))) (append (list 'W) (clockwise-chain-code m 1
                                                         si sj i (cl:- j 1) 'W)))
      ((cl:= 1 (val m (cl:- i 1) (cl:- j 1))) (append (list 'NW) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) (cl:- j 1) 'NW)))
      ((cl:= 1 (val m (cl:- i 1) j)) (append (list 'N) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) j 'N)))
      ((cl:= 1 (val m (cl:- i 1) (cl:+ j 1))) (append (list 'NE) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) (cl:+ j 1) 'NE)))
      ((cl:= 1 (val m i (cl:+ j 1))) (append (list 'E) (clockwise-chain-code m 1
                                                         si sj i (cl:+ j 1) 'E)))
      ((cl:= 1 (val m (cl:+ i 1) (cl:+ j 1))) (append (list 'SE) (clockwise-chain-code m 1
                                                         si sj (cl:+ i 1) (cl:+ j 1) 'SE))))))
((equal dir 'NE)
 (cond((cl:= 1 (val m i (cl:- j 1))) (append (list 'W) (clockwise-chain-code m 1
                                                         si sj i (cl:- j 1) 'W)))
      ((cl:= 1 (val m (cl:- i 1) (cl:- j 1))) (append (list 'NW) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) (cl:- j 1) 'NW)))
      ((cl:= 1 (val m (cl:- i 1) j)) (append (list 'N) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) j 'N)))
      ((cl:= 1 (val m (cl:- i 1) (cl:+ j 1))) (append (list 'NE) (clockwise-chain-code m 1
                                                         si sj (cl:- i 1) (cl:+ j 1) 'NE)))
      ((cl:= 1 (val m i (cl:+ j 1))) (append (list 'E) (clockwise-chain-code m 1
                                                         si sj i (cl:+ j 1) 'E)))
      ((cl:= 1 (val m (cl:+ i 1) (cl:+ j 1))) (append (list 'SE) (clockwise-chain-code m 1
                                                         si sj (cl:+ i 1) (cl:+ j 1) 'SE)))
      ((cl:= 1 (val m (cl:+ i 1) j)) (append (list 'S) (clockwise-chain-code m 1
                                                         si sj (cl:+ i 1) j 'S)))))))))

(defun get-deltas (m cc)
  (let ((tli (first (top-left-object-pixel m)))
        (tlj (second (top-left-object-pixel m)))
        (delta_i 0)
        (delta_j 0)
        (delta_negj 0)
        (first-half (firstn (cl:/ (cl:- (length cc) (mod (length cc) 2)) 2) cc))
        (second-half (firstn (cl:/ (cl:- (length cc) (mod (length cc) 2)) 2)
                             (nthcdr (cl:/ (cl:- (length cc) (mod (length cc) 2)) 2) cc))))
    (dolist (dir first-half)
      (when (equal dir 'E) (setf delta_j (cl:+ delta_j 1)))
      (when (equal dir 'SE) (progl (setf delta_j (cl:+ delta_j 1))
                                   (setf delta_i (cl:+ delta_i 1))))
      (when (equal dir 'NE) (setf delta_j (cl:+ delta_j 1)))
      (when (equal dir 'S) (setf delta_i (cl:+ delta_i 1)))
      (when (equal dir 'SW) (progl (setf delta_negj (cl:+ delta_negj 1))
                                   (setf delta_i (cl:+ delta_i 1))))))
    (dolist (dir second-half)
      (when (equal dir 'W) (setf delta_negj (cl:+ delta_negj 1)))
      (when (equal dir 'NW) (setf delta_negj (cl:+ delta_negj 1)))
      (when (equal dir 'SW) (progl (setf delta_negj (cl:+ delta_negj 1))
                                   (setf delta_i (cl:+ delta_i 1))))))
    (setf delta_negj (abs (cl:- delta_negj delta_j)))
    (list delta_i delta_j delta_negj)))

;;;
;;; wipe-object visually "marks" the object as having been counted
;;;

```

```

(defun wipe-object (m cc)
  "Removes top left object from matrix m, given the chain code for that object"
  (let ((tli (first (top-left-object-pixel m)))
        (tlj (second (top-left-object-pixel m)))
        (delta_i (first (get-deltas m cc)))
        (delta_j (second (get-deltas m cc)))
        (delta_negj (third (get-deltas m cc))))
    ;now wipe the object
    (dotimes (i (cl:+ delta_i 1) m)
      (dotimes (j (cl:+ delta_j delta_negj 1))
        (setf m (assgn m (cl:+ tli i) (cl:- tlj delta_negj j) 0))))))

;;;
;;;object-gamma is a better circularity measure
;;;
(defun object-perimiter (cc)
  "perimeter determined by adding 1 for N E S W and sqrt(2) for diagonals"
  (let ((p 0))
    (dotimes (i (length cc) p)
      (if (or (equal (nth i cc) 'NE) (equal (nth i cc) 'NW)
              (equal (nth i cc) 'SW) (equal (nth i cc) 'SE))
          (setf p (cl:+ p (sqrt 2)))
          (setf p (cl:+ p 1))))))

(defun object-area (m cc)
  "area determined by counting object pixels"
  (let ((tli (first (top-left-object-pixel m)))
        (tlj (second (top-left-object-pixel m)))
        (delta_i (first (get-deltas m cc)))
        (delta_j (second (get-deltas m cc)))
        (delta_negj (third (get-deltas m cc)))
        (area 0))
    (dotimes (i (cl:+ delta_i 1) area)
      (dotimes (j (cl:+ delta_j delta_negj 1))
        (when (cl:= (val m (cl:+ tli i) (cl:- tlj delta_negj j)) 1)
          (setf area (cl:+ area 1))))))

(defun object-gamma (m cc)
  "gamma_n cl:= 1 - (4*pi*Area)/(Perimeter^2) (from Sonka)"
  (cl:- 1 (cl:/ (cl:* 4 pi (object-area m cc))
                (cl:* (object-perimiter cc) (object-perimiter cc)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ALIGNMENT TABLE;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconstant *ALIGNMENT-TABLE* (make-hash-table :test #'equal)
  "Key is base node; Value is list of attribute-value pairs" )

;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; PMLA;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-primaction BuildEnumFrame ((sortal))
  (let* ((sortalname (princ-to-string sortal))
        (prototype (concatenate 'string "prot" sortalname)))
    (tell (concatenate 'string
                      "InstanceOf(" prototype "," sortalname ").")
          (setf targcolor (princ-to-string (cdr (first (first (askwh (concatenate 'string
                                                                                "Color(?c) and HasProperty(" prototype ",?c)"))))))))
          (setf targshape (princ-to-string (cdr (first (first (askwh (concatenate 'string
                                                                                "Shape(?s) and HasProperty(" prototype ",?s)"))))))))
          (tell (concatenate 'string "TargetColor(" targcolor ").")
                (tell (concatenate 'string "TargetShape(" targshape ").")))))

```

```

(define-primaction Perceive ((image))
  (let ((imagename (princ-to-string image)))
    (format t "Acquiring image ~A~%" imagename)
    (cond ((string= imagename "img1") (setf curimg img1)))
    (format t "RED PIXELS:~%~A~%" (print-matrix (list-to-array
                                                (img-red-pixels curimg) 16 16)))
    (format t "GREEN PIXELS:~%~A~%" (print-matrix (list-to-array
                                                (img-green-pixels curimg) 16 16)))
    (format t "BLUE PIXELS:~%~A~%" (print-matrix (list-to-array
                                                (img-blue-pixels curimg) 16 16))))))

(define-primaction DistinguishByColor ((color) (thresh))
  (let* ((colorname (princ-to-string color))
        (threshval (sneps:node-to-lisp-object thresh)))
    (format t "Distinguishing by color ~A with threshold ~A~%" colorname
            threshval)
    (cond ((string= colorname "Red") (setf curcolor 'Red)))
    (format t "Prototype for color is: ~A~%" (gethash curcolor *COLOR-PALLETE*))
    (setf binimg
      (list-to-array
        (img-pixels
          (threshold-classify
            (component-color-error curimg curcolor)
            threshval))
          16 16))
    (format t "Binary Image:~%~A~%" (print-matrix binimg))))

(define-primaction DistinguishByShape ((shape) (thresh))
  (let* ((shapename (princ-to-string shape))
        (threshval (sneps:node-to-lisp-object thresh))
        (tlopx (first (top-left-object-pixel binimg)))
        (tlopy (second (top-left-object-pixel binimg))))
    (format t "Distinguishing by shape ~A with threshold ~A~%"
            shapename threshval)
    (cond ((string= shapename "Round") (setf curshape 'Round)))
    (format t "Prototype for shape is compactness = 1.0 ~%")
    (setf cc (clockwise-chain-code binimg 0 tlopx tlopy tlopx tlopy 'E))
    (setf circularity (object-gamma binimg cc))
    (format t "Chain code for top-left object is ~A~%
              Circularity for top-left object is ~A~%" cc circularity)))

(define-primaction AddToCollection ((sortal) (collection))
  (let* ((sortalname (princ-to-string sortal))
        (colname (princ-to-string collection))
        (tlopx (first (top-left-object-pixel binimg)))
        (tlopy (second (top-left-object-pixel binimg))))
    (setf newbasenode (princ-to-string (gensym sortalname)))
    (format t "Adding KL constituent to collection.~%")
    (tell (concatenate 'string
                      "ConstituentOf(" newbasenode "," colname ")."))
    (tell (concatenate 'string
                      "InstanceOf(" newbasenode ",u"
                      sortalname ")."))
    (format t "Updating alignment table.~%")
    (setf (gethash newbasenode *ALIGNMENT-TABLE*) (list 'location (list tlopx tlopy)))
    (format t "KLSym: ~A PMLSym: ~A Modality Vision~%" newbasenode
            (gethash newbasenode *ALIGNMENT-TABLE*))))

(define-primaction AttendToNewObject ((image))
  (setf binimg (wipe-object binimg cc))
  (format t "Clearing top-left object~%")
  (format t "New Binary Image:~%~A~%" (print-matrix binimg)))

(define-primaction CheckIfClear ((image))
  (if (check-if-clear binimg 16)

```

```

    (tell (concatenate 'string
      "perform disbelieve(ObjectsLeft(" (princ-to-string image) ")"))
      (format t "I detect more objects to enumerate~%"))

(attach-primaction BuildEnumFrame BuildEnumFrame Perceive Perceive DistinguishByColor
  DistinguishByColor DistinguishByShape
  DistinguishByShape AttendToNewObject
  AttendToNewObject AddToCollection AddToCollection
  CheckIfClear CheckIfClear)
^^
Name(uRed,Red).
Name(uRound,Round).
Name(uApple,Apple).

Color(uRed).
Shape(uRound).
all(x)(InstanceOf(x,uApple) => {HasProperty(x,uRed),HasProperty(x,uRound)}).

ActPlan(UpdateCollection(q),
  withsome({?n,?m},
    (Collection(q,NumSort(?n,uApple)) and Successor(?m,?n)),
    ssequence(disbelieve(Collection(q,NumSort(?n,uApple))),
      believe(Collection(q,NumSort(?m,uApple)))),
    Say("Something is wrong.))).

all(i)(Image(i)=>
  ActPlan(EnumerateApples(i),
    ssequence(BuildEnumFrame(uApple),
      ssequence(believe(Collection(q,NumSort(n0,uApple))),
        ssequence(Perceive(i),
          ssequence(withsome({?c,?cn},
            (TargetColor(?c) and Name(?c,?cn)),
            DistinguishByColor(?cn,100),
            SayLine("I don't know what color apples are"))),
          sniterate({if(ObjectsLeft(i),
            withsome({?s,?sn},
              (TargetShape(?s) and Name(?s,?sn)),
              ssequence(DistinguishByShape(?sn,0.4),
                ssequence(AddToCollection(Apple,q),
                  ssequence(UpdateCollection(q),
                    ssequence(AttendToNewObject(i),CheckIfClear(i))))),
                SayLine("I don't know what shape apples are"))),
            else(SayLine("Enumeration Complete"))})))))))).

Image(img1).
ObjectsLeft(img1).

```

K. Deliberative do-one

```
(defun m-to-wff (mstr)
  "Converts string mN to string wffN"
  (concatenate 'string "wff" (subseq mstr 1)))

(define-primaction do-one (object1)
  "A more deliberative do-one"
  (let ((act-list (sneps:ns-to-lisp-list object1))
        (preferred-acts nil))
    (when (cl:> (length object1) 1)
      (progn (when *show-deliberation* (format t "DELIBERATING BETWEEN ~A ACTS:~%" (length object1)))
             (when *show-deliberation* (snepslog:snepslog-print object1))
             (when *show-deliberation* (format t "DETERMINING PREFERRED ACTS~%"))
             (setf preferred-acts
                   (loop for a in act-list
                        do (tell "clear-infer")          ; context of question
                            if (snepslog:ask (concatenate 'string
                                                           "Prefer("
                                                           (snepslog:m-to-wff (princ-to-string a))
                                                           ")"))
                                collect (snepslog:m-to-wff (princ-to-string a))))
             (when *show-deliberation* (format t "PREFERRED ACTS ARE: ~A~%" preferred-acts))))
    (if preferred-acts
        (tell (concatenate 'string "perform " (princ-to-string (first preferred-acts))))
        (snip:schedule-act
         (sneps:lisp-object-to-node (nth (random (sneps:cardinality.ns object1))
                                         (sneps:ns-to-lisp-list object1)))))))
```


K. Accumulator `timed-do`

```
define-frame Duration (nil act duration)
define-frame timed-do (action timedact)
^^
(define-primaction timed-do ((timedact))
  "performs a and times the duration of the performance"
  (let ((starttime (cl:get-internal-run-time))
        (actstring (princ-to-string (sneps:node-to-lisp-object timedact))))
    ;(format t "~A ~A~%" (type-of timedact) timedact)
    (snepslog:tell (concatenate 'string "perform " (snepslog:m-to-wff
                                                    actstring)))
    (let ((cpu-time (/ (float (cl:- (cl:get-internal-run-time) starttime))
                      cl:internal-time-units-per-second))
          (snepslog:tell (concatenate 'string "Duration("
                                      (snepslog:m-to-wff actstring)
                                      ", "
                                      (princ-to-string cpu-time)
                                      ")."))))))

(attach-primaction timed-do timed-do)
^^
```

References

- Anderson, John R. 2007. *How Can the Human Mind Occur in the Physical Universe?* New York: Oxford University Press.
- Aristotle. 1963. *Categories*. J. L. Ackrill (trans.). Oxford: Clarendon.
- Artemov, Sergei and Elena Nogina. 2005. Introducing Justification into Epistemic Logic. *Journal of Logic and Computation*, 15(6):1059–1073.
- Ashcraft, Mark H. 1992. Cognitive Arithmetic: A Review of Data and Theory. *Cognition*, 44:75–106.
- Baier, Annette. 1971. The Search for Basic Actions. *American Philosophical Quarterly*, 8(2):161–170.
- Barsalou, Lawrence W. 1999. Perceptual Symbol Systems. *Behavioral and Brain Sciences*, 22:577–660.
- Belnap, Nuel D. and Thomas B. Steel. 1976. *The Logic of Questions and Answers*. New Haven: Yale University Press.
- Bermúdez, José Luis. 2003. *Thinking Without Words*. Oxford: Oxford University Press.
- Beth, Evert W. and Jean Piaget. 1966. *Mathematical Epistemology and Psychology*. Dordrecht: Reidel.
- Biever, Celeste. 2004. Language May Shape Human Thought. *Science Express*, 19:1.
- Bisanz, Jeffrey and Jo-Anne LeFevre. 1992. Understanding Elementary Mathematics. In Jamie I. D. Campbell (ed), *The Nature and Origins of Mathematical Skills*. Elsevier, Amsterdam, pages 113–136.
- Blackburn, Patrick and Johan Bos. 2005. *Representation and Inference for Natural Language*. Stanford, CA: CSLI.
- Bobrow, Daniel. 1968. Natural Language Input for a Computer Problem Solving System. In Marvin Minsky (ed), *Semantic Information Processing*. MIT Press, Cambridge, MA, pages 133–216.
- Bringsjord, Selmer, Clarke Caporale, and Ron Noel. 2000. Animals, Zombanimals, and the Total Turing Test: The Essence of Artificial Intelligence. *Journal of Logic, Language and Information*, 9(4):397–418.
- Bromberger, Sylvain. 1992. Why Questions. In S. Bromberger (ed), *On What We Know We Don't Know*. University of Chicago Press, Chicago, pages 75–100.

- Brown, John Seely and Kurt VanLehn. 1982. Towards a Generative Theory of Bugs. In Thomas P. Carpenter, James M. Moser, and Thomas A. Romberg (eds), *Addition and Subtraction: A Cognitive Perspective*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 117–135.
- Butterworth, Brian. 1995. Editorial. In Brian Butterworth (ed), *Mathematical Cognition: Volume I*. Psychology Press, New York, pages 1–2.
- Butterworth, Brian. 1999. *What Counts: How Every Brain Is Hardwired for Math*. New York: Free Press.
- Butterworth, Brian, Marco Zorzi, Luisa Girelli, and A.R. Jonckheere. 2001. Storage and Retrieval of Addition Facts: The Role of Number Comparison. *Quarterly Journal of Experimental Psychology*, 54A(4):1005–1029.
- Campbell, Alistair E. and Stuart C. Shapiro. 1998. Algorithms for Ontological Mediation. In S. Harabagiu (ed), *Usages of Wordnet in Natural Language Processing Systems: Proceedings of the Workshop*. COLING-ACL, New Brunswick, NJ, pages 102–107.
- Campbell, Jamie I. D. and Qilin Xue. 2001. Cognitive Arithmetic Across Cultures. *Journal of Experimental Psychology: General*, 130(2):299–315.
- Campbell, Jamie I.D. 2004. *Handbook of Mathematical Cognition*. New York: Psychology Press.
- Carey, Susan. 2002. Evidence for Numerical Abilities in Young Infants: A Fatal Flaw? *Developmental Science*, 5(2):202–205.
- Carnap, Rudolf. 1958. *Introduction to Symbolic Logic and Its Applications*. New York: Dover.
- Carr, Martha and Hillary Hettinger. 2003. Perspectives on Mathematics Strategy Development. In James M. Royer (ed), *Mathematical Cognition*. Information Age Publishing, Greenwich, CT, pages 33–68.
- Chalupsky, Hans and Stuart C. Shapiro. 1994. SL: A Subjective, Intensional Logic of Belief. *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 165–170.
- Cho, Sung-Hye. 1992. Representations of collections in a propositional semantic network. *Working Notes of the AAAI 1992 Spring Symposium on Propositional Knowledge Representation*, pages 43–50.
- Clark, Andy and David Chalmers. 1998. The Extended Mind. *Analysis*, 58:10–23.
- Clements, Douglas H. 1999. Subitizing: What is it? Why Teach it? *Teaching Children Mathematics*, 5(7):400–405.
- Clements, Douglas H. and Julie Sarama. 2007. Early Childhood Mathematics Learning. In F. K. Lester Jr. (ed), *Second Handbook of Research on Mathematics Teaching and Learning*. Information Age Publishing, New York, pages 461–555.

- Corcoran, John. in press. An Essay on Knowledge and Belief. *International Journal of Decision Ethics*.
- Cowan, Richard and Margaret Renton. 1996. Do They Know What They Are Doing? Children's Use of Economical Addition Strategies and Knowledge of Commutativity. *Educational Psychology*, 16(4):407–420.
- Crump, Thomas. 1990. *The Anthropology of Numbers*. Cambridge, UK: Cambridge University Press.
- Davis, Randall and Doug Lenat. 1982. AM: Discovery in Mathematics as Heuristic Search. In *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill, New York, pages 3–225.
- Dehaene, Stanislas. 1992. Varieties of Numerical Abilities. *Cognition*, 44:1–42.
- Dehaene, Stanislas. 1997. *The Number Sense: How the Mind Creates Mathematics*. Oxford: Oxford University Press.
- Dellarosa, Denise. 1985. SOLUTION: A Computer Simulation of Children's Recall of Arithmetic Word Problem Solving. *University of Colorado Technical Report*, 85-148.
- Dijkstra, Edsger W. 1974. Programming as a Discipline of Mathematical Nature. *American Mathematical Monthly*, 81(6):608–612.
- Dretske, Fred. 1985. Machines and the Mental. *Proceedings and Addresses of the American Philosophical Association*, 59(1):23–33.
- Duda, Richard O., Peter E. Hart, and David G. Stork. 2001. *Pattern Classification, Second Edition*. New York: Wiley Interscience.
- Euclid. 2002. *Elements*. Santa Fe, NM: Green Lion Press.
- Feigenson, Lisa, Stanislas Dehaene, and Elizabeth Spelke. 2004. Core Systems of Number. *Trends in Cognitive Science*, 8:307–314.
- Fetzer, James H. 1990. *Artificial Intelligence: Its Scope and Limits*. New York: Springer.
- Fisher, Ronald A. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(2):179–188.
- Fletcher, Charles R. 1985. Understanding and Solving Arithmetic Word Problems: A Computer Simulation. *Behavioral Research Methods, Instruments, and Computers*, 17:365–371.
- Fodor, Jerry A. 1983. *Modularity of Mind*. Cambridge, MA: MIT Press.
- Frege, Gottlob. 1884/1974. *The Foundations of Arithmetic: A Logico-Mathematical Enquiry into the Concept of Number*. J. L. Austin (trans.). Evanston: Northwestern University Press.

- Fuson, Karen C. 1982. An Analysis of the Counting-On Solution Procedure in Addition. In Thomas P. Carpenter, James M. Moser, and Thomas A. Romberg (eds), *Addition and Subtraction: A Cognitive Perspective*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 67–81.
- Fuson, Karen C. 1998. Pedagogical, Mathematical, and Real-World Conceptual-Support Nets: A Model for Building Children’s Multidigit Domain Knowledge. *Mathematical Cognition*, 4(2):147–186.
- Gallistel, Charles R. 2005. Mathematical Cognition. In K. Holyoak and R. Morrison (eds), *The Cambridge Handbook of Thinking and Reasoning*. Cambridge University Press, Cambridge, UK, pages 559–588.
- Gelman, Rochel and Charles R. Gallistel. 1978. *The Child’s Understanding of Number*. Cambridge: Harvard University Press.
- Gettier, Edmund. 1963. Is Justified True Belief Knowledge? *Analysis*, 23:121–123.
- Glaserfeld, Ernst von. 2006. A Constructivist Approach to Experiential Foundations of Mathematical Concepts Revisited. *Constructivist Foundations*, 1(2):61–72.
- Goldfain, Albert. 2006. A Computational Theory of Inference for Arithmetic Explanation. *Proceedings of ICoS-5*, pages 145–150.
- Grandy, Richard E. 2006. Sortals. In Edward N. Zalta (ed), *Stanford Encyclopedia of Philosophy*, <http://plato.stanford.edu/entries/sortals/>.
- Gray, Eddie M. and David O. Tall. 1994. Duality, Ambiguity, and Flexibility: A ‘Proceptual’ View of Simple Arithmetic. *Journal for Research in Mathematics Education*, 25(2):116–140.
- Greeno, James G. 1987. Instructional Representations Based on Research about Understanding. In Alan H. Schoenfeld (ed), *Cognitive Science and Mathematics Education*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 61–88.
- Greeno, James G. 1991. Number Sense as Situated Knowing in a Conceptual Domain. *Journal for Research in Mathematics Education*, 22(3):170–218.
- Groen, Guy and Lauren B. Resnick. 1977. Can Preschool Children Invent Addition Algorithms? *Journal of Educational Psychology*, 69(6).
- Haller, Susan. 1996. Planning Text about Plans Interactively. *International Journal of Expert Systems*, 9(1):85–112.
- Hamm, Cornel M. 1989. *Philosophical Issues in Education: An Introduction*. New York: Falmer Press.
- Harnad, Stevan. 1990. The Symbol Grounding Problem. *Physica D*, 42:335–346.

- Hauser, Larry. 1993. Why Isn't My Pocket Calculator a Thinking Thing. *Minds and Machines*, 3(1):3–10.
- Hexmoor, Henry, Johan Lammens, Guido Caicedo, and Stuart C. Shapiro. 1993. Behaviour Based AI, Cognitive Processes, and Emergent Behaviors in Autonomous Agents. In G. Rzevski, J. Pastor, and R. Adey (eds), *Applications of Artificial Intelligence in Engineering VIII, Vol. 2 Applications and Techniques*. Computational Mechanics-Elsevier, New York, pages 447–461.
- Hexmoor, Henry and Stuart C. Shapiro. 1997. Integrating Skill and Knowledge in Expert Agents. In P. J. Feltovich, K. M. Ford, and R. R. Hoffman (eds), *Expertise in Context*. AAAI Press, Menlo Park, CA, pages 383–404.
- Hilbert, David. 1980. *The Foundations of Geometry, Second Edition*. Chicago: Open Court.
- Hintikka, Jaakko and Ilpo Halonen. 1995. Semantics and Pragmatics for Why-Questions. *Journal of Philosophy*, 92(12):636–657.
- Hoffman, Joshua and Gary S. Rosenkrantz. 1994. *Substance among other Categories*. Cambridge, UK: Cambridge University Press.
- Hofstadter, Douglas. 1995. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. New York: Basic Books.
- Holden, Constance. 2004. Life Without Numbers in the Amazon. *Science*, 305:1093.
- Husserl, Edmund. 1913/2000. *Logical Investigations*. London: Routledge.
- Hutchins, Edwin. 1995. *Cognition in the Wild*. Cambridge, MA: MIT Press.
- Isaacson, Daniel. 1994. Mathematical Intuition and Objectivity. In Alexander George (ed), *Mathematics and Mind*. Oxford University Press, Oxford, pages 118–140.
- Ismail, Haythem O. and Stuart C. Shapiro. 2000. Two Problems with Reasoning and Acting in Time. In A.G. Cohn and F. Giunchinglia and B. Selman (ed), *Proceedings of KR2000*. Morgan Kaufmann, San Fransisco, CA, pages 355–365.
- Ismail, Haythem O. and Stuart C. Shapiro. 2001. The Cognitive Clock: A Formal Investigation of the Epistemology of Time. Technical Report Technical Report 2001-08, University at Buffalo. Department of Computer Science and Engineering.
- Jackendoff, Ray. 1987. On Beyond Zebra: The Relation of Linguistic and Visual Information. *Cognition*, 26(2):89–114.
- Johnson-Laird, Philip. 1981. Mental Models in Cognitive Science. In Donald A. Norman (ed), *Perspectives in Cognitive Science*. Ablex, Norwood, NJ, pages 147–191.
- Johnson-Laird, Philip. 2006. *How We Reason*. Oxford: Oxford University Press.

- Jones, Randolph M. and Kurt VanLehn. 1994. Acquisition of Children's Addition Strategies: A Model of Impasse-Free, Knowledge Level Learning. *Machine Learning*, 16:11–36.
- Kant, Immanuel. 1787/1958. *Critique of Pure Reason*. Paul Guyer and Allen W. Wood (trans.). London: Macmillan.
- Katz, Victor J. 1998. *A History of Mathematics: An Introduction*. Boston: Addison Wesley.
- Kintsch, Walter and James G. Greeno. 1985. Understanding and Solving Word Arithmetic Problems. *Psychological Review*, 92(1):109–129.
- Kitcher, Philip. 1981. Explanatory Unification. *Philosophy of Science*, 48(4):507–531.
- Kitcher, Philip. 1984. *The Nature of Mathematical Knowledge*. Oxford: Oxford University Press.
- Klahr, David. 1973. A Production System for Counting, Subitizing and Adding. In W. G. Chase (ed), *Visual Information Processing*. Academic Press, New York, pages 527–546.
- Lakoff, George and Rafael Núñez. 2000. *Where Mathematics Comes From*. New York: Basic Books.
- LeBlanc, Mark D. 1993. From Natural Language to Mathematical Representations: A Model of 'Mathematical Reading'. In Hyacinth Sama Nwana (ed), *Mathematical Intelligent Learning Environments*. Intellect Books, Oxford, pages 126–144.
- Lenat, Doug. 1979. On Automated Scientific Theory Formation: A case study using the AM program. In J. Hayes, D. Michie, and L. I. Mikulich (eds), *Machine Intelligence 9*. Halstead Press, New York, pages 251–283.
- Lenat, Doug and John Seely Brown. 1984. Why AM and EURISKO appear to work. *Artificial Intelligence*, 23(3):269–294.
- Mac Lane, Saunders. 1981. Mathematical Models. *American Mathematical Monthly*, 88:462–472.
- Mancosu, Paolo. 2001. Mathematical Explanations: Problems and Prospects. *Topoi*, 20:97–117.
- Meck, Warren H. and Russell M. Church. 1983. A Mode Control Model of Counting and Timing Processes. *Journal of Experimental Psychology: Animal Behavior Processes*, 9:320–334.
- Mill, John Stuart. 1843/2002. *A System of Logic: Ratiocinative and Inductive*. Honolulu: University Press of the Pacific.
- Moles, Jerry A. 1977. Standardization and Measurement in Cultural Anthropology: A Neglected Area. *Current Anthropology*, 18(2):235–258.
- Moser, Paul K. 1999. Epistemology. In Robert Audi (ed), *The Cambridge Dictionary of Philosophy: Second Edition*. Cambridge University Press, Cambridge, UK, pages 273–278.

- Neches, Robert. 1987. Learning Through Incremental Refinement of Procedures. In David Klahr, Pat Langley, and Robert Neches (eds), *Production System Models of Learning and Development*. MIT Press, Cambridge, MA, pages 163–219.
- Nwana, Hyacinth Sama. 1993. *Mathematical Intelligent Learning Environments*. Oxford: Intellect Books.
- Ohlsson, Stellan and Ernest Rees. 1991. The Function of Conceptual Understanding in the Learning of Arithmetic Procedures. *Cognition and Instruction*, 8(2):103–179.
- Piaget, Jean. 1955. *The Child's Construction of Reality*. London: Routledge.
- Piaget, Jean. 1965. *The Child's Conception of Number*. New York: Norton Library.
- Polk, Thad A., Catherine L. Reed, Janice M. Keenan, Penelope Hogarth, and C. Alan Anderson. 2001. A Dissociation between Symbolic Number Knowledge and Analogue Magnitude Information. *Brain and Cognition*, 47:545–563.
- Polya, George. 1945. *How to Solve It: A New Aspect of Mathematical Method*. Princeton, NJ: Princeton University Press.
- Quillian, M. R. 1968. Semantic Memory. In *Semantic Information Processing*. MIT Press, Cambridge, MA, pages 216–270.
- Quine, W. V. 1969. *Ontological Relativity and Other Essays*. New York: Columbia University Press.
- Rapaport, William J. 1988. Syntactic Semantics: Foundations of Computational Natural-Language Understanding. In James H. Fetzer (ed), *Aspects of Artificial Intelligence*. Kluwer Academic Publishers, Dordrecht, The Netherlands, pages 81–131.
- Rapaport, William J. 1993. Cognitive Science. In Anthony Ralston and Edwin D. Reilly (ed), *Encyclopedia of Computer Science, Third Edition*. Van Nostrand Reinhold, New York, pages 185–189.
- Rapaport, William J. 1995. Understanding Understanding. In J. Tomberlin (ed), *AI, Connectionism, and Philosophical Psychology*. Ridgeview, Atascadero, CA, pages 49–88.
- Rapaport, William J. 1999. Implementation Is Semantic Interpretation. *The Monist*, 82:109–130.
- Rapaport, William J. 2002. Holism, Conceptual-Role Semantics, and Syntactic Semantics. *Minds and Machines*, 12(1):3–59.
- Rapaport, William J. 2003. What Did You Mean by That?: Misunderstanding, Negotiation, and Syntactic Semantics. *Minds and Machines*, 13(3):397–427.
- Rapaport, William J. 2005. Implementation Is Semantic Interpretation: Further Thoughts. *Journal of Experimental and Theoretical Artificial Intelligence*, 17(4):385–417.

- Rapaport, William J. 2006. How Helen Keller Used Syntactic Semantics to Escape from a Chinese Room. *Minds and Machines*, 16(4):381–436.
- Rapaport, William J. and Karen Ehrlich. 2000. A Computational Theory of Vocabulary Acquisition. In Lucja M. Iwanska and Stuart C. Shapiro (ed), *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language*. AAAI Press-MIT Press, Menlo Park, CA-Cambridge, MA, pages 347–375.
- Rapaport, William J., Stuart C. Shapiro, and Janyce M. Wiebe. 1997. Quasi-Indexicals and Knowledge Reports. *Cognitive Science*, 21(1):63–107.
- Resnik, Michael D. 1997. *Mathematics as a Science of Patterns*. Oxford: Clarendon Press.
- Rips, Lance J. 1983. Cognitive Processes in Propositional Reasoning. *Psychological Review*, 90(1):38–71.
- Ritchie, Graeme D. and F. K. Hanna. 1984. AM: A Case Study in AI Methodology. *Artificial Intelligence*, 23(3):249–268.
- Rosch, Eleanor. 1978. Principles of Categorization. In Eleanor Rosch and Barbara B. Lloyd (eds), *Cognition and Categorization*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 27–48.
- Roy, Deb. 2005. Semiotic Schemas. *Artificial Intelligence*, 167(1-2):170–205.
- Russell, Bertrand. 1907/1973. The Regressive Method of Discovering the Premises of Mathematics. In Douglas Lackey (ed), *Essays in Analysis*. George Allen and Unwin, London, pages 272–283.
- Russell, Bertrand. 1918/1983. The Philosophy of Logical Atomism. In J. G. Slater (ed), *The Philosophy of Logical Atomism and Other Essays*. George Allen and Unwin, London, pages 157–244.
- Santore, John F. and Stuart C. Shapiro. 2004. A Cognitive Robotics Approach to Identifying Perceptually Indistinguishable Objects. In Alan Schultz (ed), *The Intersection of Cognitive Science and Robotics: From Interfaces to Intelligence, Papers from the 2004 AAAI Fall Symposium*. AAAI Press, Menlo Park, CA, pages 47–54.
- Sapir, Edward. 1921. *Language: An Introduction to the Study of Speech*. New York: Harcourt Brace.
- Schoenfeld, Alan H. 1987. Cognitive Science and Mathematics Education: An Overview. In Alan H. Schoenfeld (ed), *Cognitive Science and Mathematics Education*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 1–31.
- Schwitzgebel, Eric. 2006. Belief. In Edward N. Zalta (ed), *Stanford Encyclopedia of Philosophy*, <http://plato.stanford.edu/entries/belief/>.

- Searle, John R. 1980. Minds, Brains, and Programs. *Behavioral and Brain Sciences*, 3(3):417–457.
- Secada, Walter G., Karen C. Fuson, and James W. Hall. 1983. The Transition from Counting-All to Counting-On in Addition. *Journal for Research in Mathematics Education*, 14(1):47–57.
- Sfard, Anna. 1991. On the Dual Nature of Mathematical Conceptions. *Educational Studies in Mathematics*, 22:1–36.
- Shanahan, Murray P. 1999. The Event Calculus Explained. In Michael J. Wooldridge and Manuela Veloso (eds), *Artificial Intelligence Today*. Springer, New York, pages 409–430.
- Shapiro, Stewart. 1997. *Philosophy of Mathematics: Structure and Ontology*. Oxford: Oxford University Press.
- Shapiro, Stuart C. 1977. Representing Numbers in Semantic Networks. In *Proceedings of the 5th IJCAI*, page 284, Los Altos. Morgan Kaufmann.
- Shapiro, Stuart C. 1978. Path-based and node-based inference in semantic networks. In D. Waltz (ed), *TINLAP-2: Theoretical Issues in Natural Language Processing*. ACM, New York, pages 219–225.
- Shapiro, Stuart C. 1979. The SNePS Semantic Network Processing System. In Nicholas V. Findler (ed), *Associative Networks: The Representation and Use of Knowledge by Computers*. Academic Press, New York, pages 179–203.
- Shapiro, Stuart C. 1986. Symmetric Relations, Intensional Individuals, and Variable Binding. *Proceedings of the IEEE*, 74(10):1354–1363.
- Shapiro, Stuart C. 1989. The CASSIE Projects: An Approach to Natural Language Competence. In *Proceedings of the Fourth Portuguese Conference on Artificial Intelligence*. Springer Verlag, Lisbon, Portugal, pages 362–380.
- Shapiro, Stuart C. 1991. Cables, Paths and “Subconscious” Reasoning in Propositional Semantic Networks. In John F. Sowa (ed), *Principles of Semantic Networks*. Morgan Kaufmann, San Mateo, CA, pages 137–156.
- Shapiro, Stuart C. 1992. Artificial Intelligence. In Stuart C. Shapiro (ed), *Encyclopedia of Artificial Intelligence, Second Edition*. John Wiley and Sons, New York, pages 54–57.
- Shapiro, Stuart C. 2000. SNePS: A Logic for Natural Language Understanding and Commonsense Reasoning. In Lucja M. Iwanska and Stuart C. Shapiro (eds), *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language*. AAAI Press, Menlo Park, CA, pages 379–395.
- Shapiro, Stuart C. and Haythem O. Ismail. 2001. Symbol-Anchoring in Cassie. In Silvia Coradeschi and Alessandro Saffioti (eds), *Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems*. AAAI Press, Menlo Park, CA, pages 2–8.

- Shapiro, Stuart C. and Haythem O. Ismail. 2003. Anchoring in a Grounded Layered Architecture with Integrated Reasoning. *Robotics and Autonomous Systems*, 43:97–108.
- Shapiro, Stuart C. and William J. Rapaport. 1987. SNePS Considered as a Fully Intensional Propositional Semantic Network. In Nick Cercone and Gordon McCalla (eds), *The Knowledge Frontier*. Springer Verlag, New York, pages 262–315.
- Shapiro, Stuart C. and William J. Rapaport. 1995. An Introduction to a Computational Reader of Narrative. In Judith F. Duchan, Gail A. Bruder, and Lynne E. Hewitt (eds), *Deixis in Narrative*. Lawrence Erlbaum Associates, Hillsdale, NJ, pages 79–105.
- Shapiro, Stuart C., William J. Rapaport, Michael Kandefor, Frances L. Johnson, and Albert Goldfain. 2007. Metacognition in SNePS. *AI Magazine*, 28(1):17–31.
- Shapiro, Stuart C. and SNePS Implementation Group. 2008. SNePS 2.7 User’s Manual.
- Sharon, Tanya and Karen Wynn. 1998. Individuation of Actions from Continuous Motion. *Psychological Science*, 9(5):357–362.
- Sierpiska, Anna. 1994. *Understanding in Mathematics*. London: Falmer Press.
- Slovan, Aaron. 1985. What Enables a Machine to Understand? In *Proceedings of the Ninth International Joint Conference on AI*. pages 995–1001.
- Smith, Brian Cantwell. 1996. *On the Origin of Objects*. Cambridge, MA: MIT Press.
- Sonka, Milan, Vaclav Hlavac, and Roger Boyle. 1999. *Image Processing, Analysis, and Machine Vision, Second Edition*. New York: PWS.
- Steffe, Leslie P., Erns von Glasersfeld, John Richards, and Paul Cobb. 1983. *Children’s Counting Types: Philosophy, Theory, and Application*. New York: Praeger.
- Steup, Matthias. 2006. The Analysis of Knowledge. In Edward N. Zalta (ed), *Stanford Encyclopedia of Philosophy* <http://plato.stanford.edu/entries/knowledge-analysis/>.
- Talmy, Leonard. 2000. *Towards a Cognitive Semantics*. Cambridge, MA: MIT Press.
- Tarski, Alfred. 1944. The Semantic Conception of Truth and the Foundations of Semantics. *Philosophy and Phenomenological Research*, 4:341–376.
- Thomasson, Amie. 2004. Categories. In Edward N. Zalta (ed), *Stanford Encyclopedia of Philosophy*, <http://plato.stanford.edu/entries/categories/>.
- Turing, Alan M. 1950. Computing Machinery and Intelligence. *Mind*, 59(236):433–460.
- Veblen, Oswald. 1904. A System of Axioms for Geometry. *Transactions of the American Mathematical Society*, 5:343–384.

- Vera, Alonso H. and Herbert A. Simon. 1993. Situated Action: A Symbolic Interpretation. *Cognitive Science*, 17:7–48.
- Vitay, Julien. 2005. Towards Teaching a Robot to Count Objects. *Fifth International Workshop on Epigenetic Robotics*, pages 125–128.
- Westerstahl, Dag. 2005. Generalized Quantifiers. In Edward N. Zalta (ed), *Stanford Encyclopedia of Philosophy*, <http://plato.stanford.edu/entries/generalized-quantifiers/>.
- Whorf, Benjamin. 1956. *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf*. Cambridge, MA: MIT Press.
- Wiese, Heike. 2003. *Numbers, Language, and the Human Mind*. Cambridge, UK: Cambridge University Press.
- Wittgenstein, Ludwig. 1922/2003. *Tractatus Logico-Philosophicus*. C. K. Ogden (trans.). New York: Barnes and Noble.
- Wittgenstein, Ludwig. 1939/1976. *Wittgenstein's Lectures on the Foundations of Mathematics, Cambridge 1939*. Chicago: University of Chicago Press.
- Wynn, Karen. 1992. Evidence against Empiricist Accounts of the Origins of Numerical Knowledge. *Mind and Language*, 7(4):315–332.
- Wynn, Karen. 1995. Origins of Numerical Knowledge. *Mathematical Cognition*, 1(1):35–60.
- Wynn, Karen. 1996. Infants' Individuation and Enumeration of Actions. *Psychological Science*, 7(3):164–169.
- Xu, Fei. 2007. Sortal Concepts, Object Individuation, and Language. *TRENDS in Cognitive Sciences*, 11(9):400–406.
- Zorzi, Marco, Ivilin Stoianov, and Carlo Umiltà. 2005. Computational Modeling of Numerical Cognition. In Jamie I. D. Campbell (ed), *Handbook of Mathematical Cognition*. Psychology Press, New York, pages 67–84.