Course Information

Description

The class will emphasize both mathematical thinking and understanding mathematical tools, with some commentary on programming and implementation issues. The goals of this class are

1) to become familiar with mathematical tools in general, and with the types of mathematical tools used in software engineering for specification, formal modeling, and programming in specific.

2) to gain multiple perspectives on the primary mathematical techniques of logic, induction, proof, functions, relations, and graphs.

Homework will consist of substantial *thinking and reflection* on mathematical ideas and content, plus short *selected exercises* on each topic. Problems and readings from the text will accompany each class topic, however the lectures and class discussions will not necessarily duplicate or refer to textual materials. In-class exercises will not be graded. No graded tests, no final exam. The final assignment will be to outline the field of mathematics as you understand it.

My general attitude is that each student is a mature adult responsible for his/her own learning and motivation. As well, each student is an individual who will learn best with individually tailored content and experiences. I will provide all standard educational structures for students to choose between (assignments by the instructor, structured environments, multiple resources and references, and self-motivated exploration). I expect each student to know the style of both teaching and learning which best encourages his/her own positive educational experience. I also expect each student to be aware of his/her own goals and motivations for being here, and his/her own needs and expectations for success. Of course, you will need to let me know your goals and needs for me to effectively address them.

Text

Edgar G. Goodaire and Michael M. Parmenter, *Discrete Mathematics with Graph Theory*, Prentice-Hall, 1998.

This undergraduate text covers the elementary concepts of discrete mathematics. All students are expected to read or to skim *all* material in the book, and to do selected exercises on topics you do not fully undertstand.

The text is a supplement to the lectures and class time, not the primary content. Students are expected to do text book exercises as part of reading, to the extent needed for individual understanding. Class participation is *in addition to* the text readings and problems.

Unless specifically requested, the following chapters and sections of the text will *not* be covered in depth in class, although I will mention each briefly:

Chapter	*Topic*	*Pages*
3.3	Prime Numbers	118-133
5	Principles of Counting	211-239
6	Permutations and Combinations	240-274

Lecture notes and special readings for each class will be distributed in hardcopy prior to class meetings.

Evaluation

Available grades:

non-completion: Incomplete, Withdraw, etc.

completion: A A- B+ B B- C

- A: reserved for superior performance
- A- or B+: expected grade for conscientious performance
- B: adequate work
- B-: barely adequate
- C: equivalent to failing

Grading Options:

1. Grading Contract: specify a set of behaviors and an associated grade.
2. Performance Quality: attendance, participation, assigned exercises
3. Self-determined: negotiate with instructor

Discussion:

If you already understand the subject, if you plan to excel, or if you need clear performance goals for motivation, then **Option 1** is a good idea. If you prefer a clearly defined agenda, if you do well with concrete task assignments, or if you need a schedule of activities for motivation, then **Option 2** is a good idea. If you are not concerned about grades, if you intend to do what you choose anyway, or if you are self-motivated, then **Option 3** is a good idea.

I will notify any student who is not on a trajectory for personal success.

Course Syllabus

Meeting	Topic	Text	Exercises
1)	introduction, overview of mathematics		ch0 exercises
2)	formal systems, theories of computation		
3)	propositional logic		ch0 due
4)	history of logic		
5)	proof strategies	Ch 0	proof exercises
6)	predicate calculus		
7)	boundary logic		
8)	induction and recursion	Ch 4	induction exercises
9)	set theory	Ch 1	
10)	relational structure	Ch 1	
11)	functional structure	Ch 2	algebraic exercises
12)	algebraic systems		
13)	exotic numbers	Ch 3,5	
14)	algorithms	Ch 7	
15)	graphs and trees	Ch 8	
16)	paths and circuits	Ch 9,10	graph exercises
17)	graph algorithms	Ch 12,13,14	final project
18)	grammars, FSMs		
19)	review		final project due
20)	closure		

Chapter 0 Exercises

Almost all mathematical textbooks begin their story in the middle, as if you already knew the assumptions, critical choices, and relevance of each topic. For example, your text begins to teach about proof theory on page 1.

The following questions come prior to studying the content of mathematics. Answer each question in *one sentence or less*. These questions identify what you believe about mathematics, therefore there are no right or wrong answers, just answers that are less or more thoughtful.

HAND IN YOUR ANSWERS AT THE BEGINNING OF CLASS.

Science

1. Which of the following *physically exist*?
    a. electrons
    b. the temperature of the center of the Sun
    c. the cosmological big bang
    d. the state of the internet

2. Is *length* an objective concept;
    that is, is the length of an object independent of a particular observer?

3. Can you know something about the world without changing the world?

Mathematics

4. Are mathematical ideas invented, discovered, or something else?

5. What is mathematical Truth?

6. In what sense does π exist? Where might it exist?
    Is π a constant? Is it eternal or might it change over time?

7. Are there any mathematical concepts that have only one property
    (i.e., are there pure concepts independent of other concepts)?

8. When is the following equation True?: $7 + 8 = 3$

9. Is every number either even or odd? How do you know?

10. Prove the *Pigeonhole Principle*, that N+1 pigeons cannot fit into N holes without sharing.

11. Imagine two points as close together as possible. Is there another point in between them?
    Are there an infinite number of points in between them?

12. Is the following statement True or False or something else?:
    Somewhere in the decimal expansion of π, there are exactly 34 sevens in a row.

13. What kinds of mathematics can be beautiful?

Computation

14. What is the shortest program which will produce a *random number*?

15. Is a *bit* the simplest computational object?

16. Binary computation uses two states, 0 and 1. Is *unary* computation possible?

17. Name three mathematical concepts which cannot be computed.

Attitude

18. How many of the above questions have you thought about before today?

19. Look at the Chapter 0 quotes which follow. Make three lists of names:
    people that you agree with,
    people that you disagree with, and
    people that you do not understand well enough to have an opinion about.

20. *Essay:* Write less than one page in response to this famous question:

    Why is mathematics so unreasonably effective in describing and predicting reality?

Chapter 0 Quotes

Attitude

"Multiplicity ought not be posited without necessity."
-- William of Occam (1340)

"The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:
1) Combining several simple ideas into one compound one, and thus all complex ideas are made.
2) The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.
3) The third is separating them from all other ideas that accompany them in their real existence; this is called abstraction, and thus all its general ideas are made."
-- John Locke (1690)

"The language of all art forms, such as cookery, drawing, programming, research, mathematics, and music, is a set of instructions which, if followed, will lead the reader to the same ecstacies as those experienced by the original artist."
-- G. Spencer-Brown

Science

"As far as the propositions of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality."
-- A. Einstein

"The axiomatization and algebraization of mathematics, after more than fifty years, has led to the illegibility of such a large number of mathematical texts that the threat of complete loss of contact with physics and the natural sciences has been realized."
-- V. I. Arnold

"I believe there are exactly
15,747,724,136,275,002,577,605,653,961,181,555,468,044,717,914,527,116,709,366,231,425,076,185,631,031,296 protons in the universe, and the same number of electrons."
-- Sir Arthur Eddington

Mathematics

"Mathematics has always skirted dangerously close to the shores of metaphysics."
-- S. G. Shanker

"We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. ... There are many operations which [the mathematician] may carry out with these symbols, without ever having to look at the things they stand for."
-- H. Weyl

"Of all escapes from reality, mathematics is the most successful ever. It is a fantasy that becomes all the more addictive because it works back to improve the same reality we are trying to evade. All other escapes -- sex, drugs, hobbies, whatever -- are ephemeral by comparison."
-- G.C. Rota

"Mathematics, this product of human activity, 'alienates itself' from the human activity which has been producing it. It becomes a living, growing organism, that *acquires a certain autonomy* from the activity which has produced it; it develops its own autonomous laws of growth, its own dialect."
-- I. Lakatos

Computation

"Computer science also differs from physics in that it is not actually a science. It does not study natural objects. Neither is it, as you might think, mathematics."
-- R. Feynman

"There is no need for infinities; there are quite enough finite numbers to serve any purpose."
-- D. Knuth

"Calculation is a method of getting rid of information in which you are not interested. You throw away what is not relevant."
-- T. Norretranders

Formal Symbol Systems

"We obscure our knowledge with technical details in order to take a voyage of discovery that will eventually lead us back to what we already know."

-- Arthur "Arty" Fischell

Computing and Mathematics

-- David Parnas

Types of mathematical training needed by computer professionals:

calculus
discrete mathematics
logic
linear algebra
graph theory
differential equations
probability and applied statistics
optimization
numerical analysis

Software Engineering and Formal Methods

-- J. B. Wordsworth, in *Formal Methods of Software Engineering*

The roles in which software engineers need formal methods:

specification engineer
design engineer
programmer
documentation engineer
test engineer
service engineer

Approaches to Specification

-- Dean and Hinchey, in *Teaching and Learning Formal Methods*

"Natural language...is hopelessly inadequate when we have to deal unambiguously with situations of great intricacy...in such activities as legislation, arbitration, mathematics or programming."

Three Approaches

Only do what natural language can handle. *Exclude and redefine* intracacy.
Bend natural language to fit the purpose.
Design a completely new language (mathematics, programming).

Beauty

Scientific beauty consists of

1. simplicity (completeness, economy)
2. harmony (symmetry)
3. brilliance (clarity, connectedness)

"Beauty is the primary standard for scientific truth."
-- Augos and Staneiv, *The New Story of Science*, p.39

"You can recognize truth by its beauty and simplicity"
-- Richard Feynman

"Frequently a theorist will throw out a lot of data on the grounds that if they don't fit an elegant scheme, they are wrong."
-- Murray Gell-Mann

"A theory is more impressive the greater the simplicity of its premises is, the more different kinds of things it relates, and the more extended is its area of applicability."
-- Albret Einstein

The Modeling Hierarchy

Conceptualization	(imaginary, perceptual, ideal, cognitive)
Mathematical Model	(formal, symbolic, abstract, mathematical)
Data Structure and Algorithms	(representation, computational, software)
Machine Implementation	(actual, structured, physical hardware)

Formal Symbol Systems

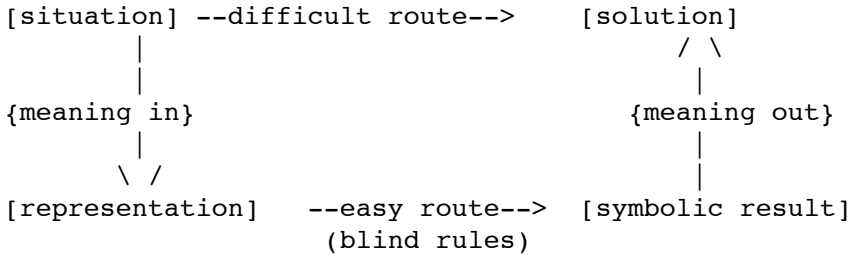
- a *universe* of discrete, stable, unique, disjoint, localized objects
- stable *maps* between objects. A territory with paths that can be traversed. A graph.
- *State Space*: the graph of states and transforms between states.
- an *interpretation* which maps symbols/tokens one-to-one onto objects/meanings.
- a defined and stable notion of *truth*.

Formal Modeling

A **formal system** consists of

- several sets of labels (for objects, functions, relations) called *constants*,
- rules for building *compound sentences* (or equations or expressions), and
- rules for *evaluating and simplifying* compound expressions.

Using a Formal System



Numeric or Symbolic Computation

Compute symbolically, unless no efficient symbolic technique is known;
then use optimized numeric techniques.

SYMBOLIC:
meaning
-- written as -->
symbol structures
-- reduced by -->
symbolic transformation rules
-- turning into -->
simpler symbol structures
-- read for -->
meaning

NUMERIC:
meaning
-- exemplified by -->
selected instances
-- substituted into -->
symbol structures
-- reduced by -->
numeric simplification rules
-- turning into -->
approximate results
-- read for -->
meaning

Computation

"A computational process is indeed much like a sorcerer's idea of spirit.
It cannot be seen or touched.
It is not composed of matter at all.
However, it is very real."
-- Abelson and Sussman,
Structure and Interpretation of Computer Programs, p.1

Computation is that which can be done by an *effective procedure*, or by a *Universal machine*.

EFFECTIVE PROCEDURE
well-founded recursive algorithm

UNIVERSAL MACHINE
Turing tape
programming language + instruction sequence
transistor network + timing
cellular automata
game of Life
FSM + stack
spreadsheet (like Excel)
database (like Access)

Models of computer

Levels of computational architecture

hierarchy of abstraction specification languages
machine language specification
vonNeumann tradeoff
circuit behavioral specification
hierarchy of realization specification languages

design model	abstract behavior
architecture model	abstract structure
performance model	abstract efficiency
correctness of behavior	functionality
efficiency of behavior	performance
actual behavior of physical circuit	reality

bit, word, instruction, program, message, application, user interface

Programming hierarchy

- *User interface*: metaphoric system which makes design interface accessible to non-experts.

- *Design interface*: hidden symbolic system which provide conceptual language for non-expert human to specify design abstractions.

- *Design abstraction*: pure symbolic system which expresses a human objective

- *High-level programming language*: symbolic system which closely models expert human models (math) and hides machine needs [Often math and algorithm are confused.]

- *Programming language*: symbolic system which expresses assembly steps in human writable form. Does not cleanly differentiate between requirements of the human and those of the machine.

- *Assembly language*: symbolic system which expresses machine language in process steps over specific logic function systems

- *Machine language*: symbolic system which transfers low level machine instructions into processes within designed physical logic function systems

- *Hardware design language:* symbolic system which specifies parallel operation of gate arrays

- *Logic function systems:* physical system, integrated networks of gates

- *Gates:* abstract physical system of logic operations and connecting wires

- *Transistors:* ignored physical system, assumed to be bundled in gates

Theories of Computation

All are formal, all are distinctly different.

- Formal Symbol Manipulation
  The result of a century of work in metamathematics.
  The mechanical manipulation of symbolic structures, without regard to meaning.

- Effective Computability
  What can be done mechanically by an abstract mathematical model of a machine.
  How difficult such a computation is to perform.

- Rule-base Reasoning, algorithm execution
  The behavior produced by following an explicit set of rules or transformations.
  How to construct rule following machines.

- Digital State Machines
  Circuits and machines with a finite, disjoint set of internal, homogeneous states.
  A state is an array of values or configurations when time is stopped.

- Information Processing
  Storing, manipulating, displaying, and transferring "information"
  Encoding utility in vast arrays of bits.

- Physical Symbol Systems
  Computers are made of, and interact with, symbols, in a way which depends upon

the physical embodiment of the symbols.
Physical embodiment occurs in bit arrays and streams, in instruction sets, in cellular arrays, and in minds.

- Interactive Agents
  Agents embodied in an environment interact and communicate with each other.
  Atomic programs with specific functionality moving in an emergent network.

- Non-linear Dynamic Systems
  Fractals, chaos theory.
  Non-linear phenomena and computations,
  modelled by solutions to differential equations and iterative systems.

- Complex Adaptive Systems
  Artificial life, complex systems.
  Entities and programs in array-based environments which respond to changes in the environment by survival adjustments.
  Learning through evolution

- Clockwork Universe
  Dated: 17th and 18th centuries.
  The Universe and everything in it
  is like the mechanism of a large, perfectly tuned clock.

- Quantum Computing
  All observation and behavior is generated by the "collapse"
  of a superposed wave function.
  Probabilistic indistinguishability yields reality through measurement.

- Molecular Computing
  Massive numbers (~10^23) of molecules transact chemical and physical exchanges.
  Results are the statistical average of dominant configurations (i.e. types of molecules)

- Biological Computing
  DNA and RNA structures perform replicate pattern-matching.
  Results are functional systems whcih compute behavior in an environment.

- Agoric Computing
  Economic agents and programs transact tokens of value to accumulate processing resources.
  Competitive survival of function in a free market.

The Virtues of Mathematical Models

-- Gries and Schneider, *A Logical Approach to Discrete Math*

- A mathematical model may be more *understandable, concise, precise, or rigorous*

than an informal description in natural language.

- Answers to questions about an object or phenomenon can often be *computed directly* using a mathematical model of the object or phenomenon.

- Mathematics provides *methods for reasoning*:
  for *manipulating* expressions,
  for *proving* properties, and
  for *obtaining new results* from known facts.
  This reasoning can be done without knowing or caring what the symbols being manipulated mean.

The Trouble with Mathematical Models

- Only a *small portion of the world* and of our experience can be discretely objectified.

- *Abstraction discards information*.

- Modeling does not reflect *human processes*.
  (Students taught how to think in models generally make poor programmers.)

- Modeling dictates a worldview which, at times, *may be dysfunctional*.
  "Present mathematical and scientific education is a hotbed of authoritarianism and is the worst enemy of independent and critical thought." -- Lakatos

- Human reasoning is *physiologically mediated* by human emotion.
  Discovery is intuitive and involves guesswork.

The Pattern of Growth of Theories

-- Lakatos, *Proofs and Refutations*

1. *Primitive conjecture*
2. *Proof* (a rough thought experiment)
3. *Global counterexamples* emerge, questioning the proof.
4. *Proof reexamined*. The incorrect portion of the proof is made explicit, and either the conjecture is limited or the definitions are broadened.

Steps 1-4 are the basic cycle, also

5. The *proofs of other theorems are examined* to see if the newly corrected portion (from 4) is relevant.
6. *Consequences* of the proof are examined.
7. *Counterexamples* are turned into new examples, and new theories or fields are created.

Chapter 0 Responses

Science

1. Which of the following *physically exist*?	*EXIST*	*NOT*
a. electrons	15	2
b. the temperature of the center of the Sun	6	11
c. the cosmological big bang	3	14
d. the state of the internet	3	14

2. Is *length* an objective concept;
that is, is the length of an object independent of a particular observer?

OBJECTIVE: 8 *NOT: 9*

3. Can you know something about the world without changing the world?

YES: 7 *NO: 10*

Mathematics

4. Are mathematical ideas invented, discovered, or something else?

INVENTED: 2 *DISCOVERED: 8* *OTHER: 7*

5. What is mathematical Truth?

Universally true, follows the rules, anything is possible, logically/deductively true, consistency of results, laws that can't be disproved, reality in numbers, true or false, validity

6. In what sense does π exist? Where might it exist?
Is π a constant? Is it eternal or might it change over time?

EXISTS	*yes: 9*	*no: 1*	*as a concept: 3*
CONSTANT	*yes: 14*	*no: 0*	
ETERNAL	*yes: 10*	*no: 2*	

7. Are there any mathematical concepts that have only one property
(i.e., are there pure concepts independent of other concepts)?

YES: 9 *(zero, void, axioms)* *NO: 8*

8. When is the following equation True?: 7 + 8 = 3

watch-time, mod12, based on false premise, mod10, = means >,
change the definition of the symbols
NEVER: 4

9. Is every number either even or odd? How do you know?

YES: 11 *not the other, every integer, by generalization, theory of divisors, years of learning, whole numbers, by induction*
NO: 5 *zero, infinity-1, Pi, fractions, irrationals,*

10. Prove the *Pigeonhole Principle*, that N+1 pigeons cannot fit into N holes without sharing.

By contradiction: assume N+1 pigeons can fit into N holes. There are N holes, therefore N pigeons, (one-to-one correspondence). Thus N+1 = N contradiction. This is an example of something that cannot be proved by an algorithm.

11. Imagine two points as close together as possible. Is there another point in between them? Are there an infinite number of points in between them?

YES, one and infinite: 13 *NO: 4* *(finite geometry)*

12. Is the following statement True or False or something else?:
Somewhere in the decimal expansion of π, there are exactly 34 sevens in a row.

TRUE: 3 *non-repeating*
FALSE: 9 *non-terminating algorithm, non-repeating*
OTHER: 5 *can't determine*

13. What kinds of mathematics can be beautiful?

subjective choice, any kind, all, elegant and abstract, following rules, Mandelbrot set, language of science, gives explanations, analytical, fractal, integral calculus, simple clear and correct.

Computation

14. What is the shortest program which will produce a *random number*?

NONE: 7
EGS: sampling noise, pop a number 1, pick a number, non-trivial algorithm, rand() %n

15. Is a *bit* the simplest computational object?

YES: 15 *NO: 2*

16. Binary computation uses two states, 0 and 1. Is *unary* computation possible?

YES: 11 *(stroke arithmetic, change sensitive systems)* *NO: 6*

17. Name three mathematical concepts which cannot be computed.

Pi , infinity, i, point, circle, irrational numbers, void, chaos theory, empty, set, space, division by zero, black holes, real numbers, trisecting an angle, doubling a cube, squaring a circle, 1/3, set of natural numbers.

Attitude

18. How many of the above questions have you thought about before today?

0: 8 1-2: 4 3: 3 >3: 2

19. Look at the Chapter 0 quotes which follow. Make three lists of names:
people that you agree with,
people that you disagree with, and
people that you do not understand well enough to have an opinion about.

	AGREE	*DISAGREE*	*HUH?*
William of Occam	*7*	*2*	*5*
John Locke	*12*	*2*	*0*
G. Spencer-Brown	*8*	*8*	*0*
A. Einstein	*12*	*0*	*4*
V.I. Arnold	*4*	*5*	*5*
A. Eddington	*2*	*7*	*7*
S.G. Shankar	*4*	*3*	*8*
H. Weyl	*13*	*2*	*1*
G. Rota	*9*	*4*	*3*
I. Lakatos	*9*	*3*	*3*
R. Feynman	*8*	*3*	*3*
D. Knuth	*8*	*8*	*1*
T. Norretrandres	*10*	*1*	*4*

20. *Essay:* Write less than one page in response to this famous question:

Why is mathematics so unreasonably effective in describing and predicting reality?

Math concepts come from reality.
From years of discovery and quantification, well defined rules.
Objective and precise, can prove anything that is true, builds upon itself.
Simple, clear and correct. reasonable, applicable to real-life.
Build upon known principles, symbols and abstraction, what leads to wisdom?
The way to understand the Universe, sea of interrelated details, successful prediction, laws.
Makes a lot of assumptions and not precise, rules of the way reality works.
Humans avoid analysis and reasoning, reality comes through visionary senses.
An invention of human minds, doesn't apply to feelings, may not apply outside our experience.
Determines patterns, simplicity of concepts, describes what we perceive.
Math truncates information, simulates world, based on probability.
Explains natural mechanisms, prediction helps survival, identifies structural stabilities.
Ineffective, misses non-linear relations and rapid changes and irregular shapes.
Encompasses objective information, misses intangibles and complexities, simplified view.
Removes emotion, rigid rule sets, scientific beauty (simplicity, harmony, brilliance).

Models of Computation

What is Mathematical Computation?

At the turn of the 20th century, mathematicians believed that proof required skill and ingenuity. Logic had just been formalized, so the natural questions were

What is proof? What is a mathematical system?

In particular, mathematicians wanted to know if all mathematical proofs could be verified by an algorithm. This is very similar to the constructivist philosophy that any mathematical object must come with a means to generate it. The trouble was with infinities, like the set of real numbers or the set of points on a plane, neither of which can be constructed, or shown to actually exist.

The initial program was simply to show that the *arithmetic of natural numbers*, a very small portion of mathematical knowledge, could be placed on a firm, unambiguous basis.

In 1931, Kurt Godel's incompleteness theorem ended all hopes that mathematics generated certain knowledge or Truth.

Incompleteness theorem:

Any formal system as complex as *the arithmetic of positive integers* is either

incomplete: there are some statements which cannot be proven to be either True or False

or

inconsistent: there are provable statements which are contradictory

Until very recently, inconsistent theories were felt to be intolerable, therefore the idea of incompleteness was necessarily accepted.

1936

In 1936 four independent mathematical models of the meaning of "computation" appeared. This was prior to the invention of silicon computers but inspired by large mechanical computers, the equivalent of today's one dollar calculators.

partial recursive functions Kurt Godel and Stephen Kleene

functions which can be defined through recursion on well-formed sequences

equational general recursive functions Jacques Herbrand

recursive functions combined with equality

lambda calculus — Alonzo Church

a very small functional system based on substitution and abstraction

Turing machines — Alan Turing

a mechanistic model of problem solving

Interestingly, each of these unrelated models defines *the same set of functions*, thus they are functionally identical.

Computer Science Models of Computation

Since the Turing machine model is mechanical, it appealed to computer scientists, who in the 1940s were dealing with large machines rather than mathematical systems. The Turing machine model is the most clumsy and difficult of the alternatives.

In 1963, a model of computation was introduced which was closely based on the structure of modern computers:

Universal register machines — Sheppardson and Sturgis

The *hardware* point of view is

> An algorithm is a piece of machinery which realizes a desired computation.
> The set of instructions for the algorithm is defined by the hardware architecture.

The *software* point of view is

> An algorithm is a sequence of textual instructions.

Programming Language

In 1955, John McCarthy built the programming language LISP based on lambda calculus. LISP is the most elegant and abstract of all popular programming languages. However, the important point is that no matter what software language you use on what hardware architecture, and no matter how poorly an algorithm is implemented, all implementations are the same with regard to what can be expressed or computed.

> *A programming language is a formal language for specifying effective procedures.*

Programming languages define a class of mathematical functions, those functions which can be stated in the language. At best a programming language specifies the set of recursive functions.

Church-Turing Thesis

All reasonable formulations of the intuitive notion of computability are equivalent.

Wolfram's Fundamental Theorem (2000)

All nontrivial interactions have effective computability as a minimal model.

Effective Procedures

All algorithms (and models of computation) have these properties:

1. finite set of instructions from a finite set of types of instruction
2. discrete stepwise process
3. deterministic (no random elements)
4. finite time and space for the process
5. each step involves a finite amount of data.

Algorithms are Not Functions

An *algorithm* computes a recursive *function*.

Algorithm	Function
a text	an idea
a string of tokens	a set of pairs
instructions	no instructions
input --> output	domain and range

Algorithms are effective procedures which solve a problem.

The types of problem that programming can solve are

1. deciding set membership
2. computing a recursive function

It is therefore desirable to specify problems for computers in one or both of the above ways.

Non-computational Mathematics

Here are some mathematical objects and operations which cannot be expressed as an algorithm or within an effective procedure.

real numbers	(absolute precision arithmetic)
transcendental numbers	(such as Pi and e)
infinity	(in any variety)
void	(non-symbolic grounds)
sets	(requires a parallel processor, one for each set member)
existence proof	(demonstration without producing a specific object)

Non-computational Functions

Here are some computational machine behaviors which cannot be expressed as recursive functions, and thus cannot be decided through computation.

Halting Problem

Does machine model M halt when given input string S?

Empty String Acceptance

Does machine model M accept the empty string?

Empty Language Acceptance

Does machine model M accept an empty language?

Regular Machine Recognition

Does a machine exist which can determine
if another machine accepts a regular language?

Rice's Theorem

No algorithm exists which accepts a machine description M
and determines if M accepts an effective procedure.

Self-terminating Machine

Does a machine halt when given a description of itself?

The Age of Mathematical Concepts and Symbols

Our clarity of understanding of mathematical concepts corresponds to the time evolution of these concepts. That is, *older is simpler*. As well, the sequence of math concepts taught in schools pretty much follows the historical evolution of mathematical ideas. Here is a rough road map of the time evolution of various mathematical concepts. Asterisks, *, mark content covered in class.

8000 BC*		one-to-one correspondence
4000 BC*		counting
1000 BC	.	zero (as dot)
400 BC*		zero as blank space
300 BC*	0	zero
300 BC*		syllogistic logic
1050	__	horizontal fraction bar
1417	+	plus
1425	%	percent
1432*		mathematician
1484		exponent
1484		billion, trillion,...
1530	0.0	decimal fractions
1544		division
1549		parallel
1551		irrational numbers
1551		theorem
1556*	( )	parentheses
1557*	=	equals
1570		equation
1570		prime number
1575	x	variables as letters
1583	sin	sine function
1618	*	times (X in 1618, * in 1659)
1624	log	logarithm function
1631	>	greater/less than
1634		angle
1637		imaginary, real (Descartes)
1647	π	pi
1655	A,B,C	lettering for triangles
1655	∞	infinity
1672		“math” (Newton)
1674	cos	cosine function
1675	d/dx	derivative, integral
1690	e	base of natural logs

1718		probability
1734	f(x)	function symbol
1763		natural number
1770	∂	partial derivative
1777	i	imaginary unit
1786	lim	limit
1808	!	factorial
1816	ax = bx+c	linear equation
1827		long division
1839		“Fermat’s last theorem”
1840		pencil
1841	| |	absolute value
1843	[]	matrices
1848		factor
1851*		set
1882*		isomorphism
1883		eigenvalue
1887		tensor
1888*	U	union, intersection
1891		histogram
1892		standard deviation
1902*	e	identity element
1910*	~, V	not, or, and symbols
1921*		truth table
1931		spinor
1935*		homomorphism
1938		googol, googolplex
1940*	Ø	null set
1940*		onto
1975		fractal
1975		chaos
1989*		boundary mathematics

Propositional Logic

What is Logic?

the laws of thought	(Boole, c1850, mathematics)
principles of right reasoning	(religion)
methodology of valid argumentation	(law)
study of grammar	(linguistics)
stages of cognitive development	(Piaget, developmental psychology)
truths based solely on the meaning of the terms	(math)
the most abstract and general description of reality	(philosophy)
force of reason rather than dogma	(politics)
science or history of the human mind	(Encyclopedia Britannica, 1771)
technique for design of	(computer science)
circuitry	
program control	
process description	
structured programming	
deductive computation	
programming connectivity	
decision making in algorithms	

The Elements of Programming

"A powerful programming language...serves as a framework
within which we organize our ideas about processes."
-- Abelson and Sussman, "Structure and Interpretation of Computer Programs"

Three Mechanisms

1. *Primitive expressions*
   the simplest entities that the language addresses

2. *Means of combination*
   compound elements are built from simpler ones

3. *Means of abstraction*
   compound elements are named and manipulated as units.

Representation

Lexicon: the typographical forms which represent statements

Syntax: the rules of composition, making compound forms out of objects and functions.
Atomic objects are propositions, functions and relations.
Sentences are atomic objects + logical connectives.

Semantics: the rules of meaning, connecting statements to values

Boolean Algebra = Propositional Logic

Boolean algebra is the algebraic approach (match and substitute using equations), and Propositional calculus is the logical approach (inference using conjunction of facts)

to the *same mathematical structure*

(even though the fields developed independently, and don't talk to each other)

Both address the **easiest and simplest** useful formal system,
which poses the **hardest and most important** technical issues for computation.

Propositional Calculus

The simplest formal system with great utility. A proposition is the simplest complete unit of thought. Any statement or decision with a Yes/No or True/False result.

Value Domain: {True, False}

Object Domain: statements (propositions) that are either True or False
{p, q, r, s...}

Operator Domain: connectives {not, or, and, if-then, if-and-only-if}

Constructing Sentences

The **logical connectives** allow construction of compound ideas, which can include several propositions. for example: (if (A and B) then (C or (not D)))

The **truth value** of a compound sentence is the truth value of its component parts.

(A and B)	isTrue exactly when both A isTrue and B isTrue.
(not A)	isTrue exactly when A is False.
(A or B)	isTrue exactly when either A isTrue or B isTrue.
(A implies B)	isTrue exactly when either (not A) isTrue or B isTrue
(A iff B)	isTrue exactly when either A and B are both True or A and B are both False

There are 16 unique Boolean connectives of two variables, but only five are common {and, or, not, if-then, if-and-only-if}. All connectives can be expressed using only one {nor}.

Tautologies, Contradictions, and Indeterminate Sentences

Sentences that are always true regardless of the values of the atoms are called *tautologies*. A tautology conveys no information about its components.
Sentences that are always false are *contradictions*.
Sentences which do depend on (at least one of) their component atoms are *indeterminate*.

History of Logic

Ancient Party Games

Logic has confused, perplexed, and challenged philosophers and scholars from the beginning of culture. It was built into our language (and presumably our thinking) from the beginning of language. However, philosophers did not (and still do not) understand the subtleties of the simple words {true, false, and, or, not, if, equal, some, all, therefore}

Some men are barbarians. Some barbarians are kind. Thus, some men are kind.	Is this a proper conclusion?
If it is raining, then I am happy. If I am dead, then I am happy	Is this necessarily True when I am in the rain? Is this "if" the same as the above "if"?
He or me. Watch or listen.	Are there two types of "or"? (exclusive and inclusive)
If you say that you are lying and that is the truth, then you are lying.	What do paradoxes mean? (Cicero)
Is. Not is. Not not is.	Does “not not” mean nothing at all?

Aristotle

Aristotle was the first person to classify declarative language. He used three polar categories:

single vs compound	Socrates is happy vs Man is happy.
universal vs particular	Everyone vs someone.
affirm vs deny	Everyone vs no one.

The latter two categories form the **Square of Opposition.**

		AFFIRM		*DENY*
UNIVERSAL	A	Every__is__.. (Everyone is happy.)	E	No__is__. (No one is happy.)
PARTICULAR	I	Some__is__. (Someone is happy.)	O	Some__is not__. (Someone is not happy.)

The Syllogism

According to Aristotle, the fundamental unit of reasoning is the *syllogism*. He defined it as

> "discussion in which, when things are posited, other things necessarily follow."

All men are mortal.
Socrates is a man.
Thus, Socrates is mortal.

The syllogistic form of logic was developed into the first ever **Axiomatic System** with variables.

The Figures of the Syllogism

The ancients discovered these four figures or forms of syllogism, which applied no matter what proposition was substituted for each of the three terms. Here,

C stands for the major term,
M stands for the middle term, and
B stands for the minor term of a syllogism.

The figures (or forms) of reasoning:

```
      C _ M       M _ C       C _ M       M _ C
      M _ B       M _ B       B _ M       B _ M
Thus  B _ C       B _ C       B _ C       B _ C
```

The blanks can be any one of the "AEIO" forms from the square of opposition.

```
A:    Every _ is _.
E:    No    _ is _.
I:    Some  _ is _.
O:    Some  _ is not _.
```

Scholastic Logic

The syllogism survived the Dark Ages in the form of the **rules of theological debate**. During the 13th century, Pope John XXI wrote a book on logic which dominated logical thought for the next 300 years. He observed that:

Nouns and *Verbs* form *Subjects* and *Predicates*

These subjects and predicates are CATEGORMATA; they have a **referent** in the real world.

The logical connectives are SYNCATEGOREMATA; they are **without** a referent in the real world.

Theological debates noticed the **use/mention** distinction:

Man is mortal. versus Man is a noun.

and the **paradoxes** generated by the absence of articles in Latin:

The man is mortal. versus Man is mortal.

The Categorical Syllogism Chant

There were only 19 syllogisms (rules of logic) in the middle ages; no one had figured out the mathematical symmetries (or the logic) which generate 24 balanced syllogisms formed by taking three pairs of four things. No one was bothered by the contradictions in the naming scheme either, since logic was to be *memorized* rather than deduced. The vowels in each of the Latin names for the *moods* of the syllogism are a mnemonic for the AEIO form, which had become associated with the logical connectives.

```
A:    ->          (implies)
E:    ->~         (implies not)
I:    &           (and)
O:    &~          (and not)
```

Quantification also began to show up in unprincipled ways (Q in some forms is "there exists").

```
Figure I
      Barbara:    M ->  C     and   B ->  M     thus  B ->  C          (AAA)
      Celarent:   M -> ~C     and   B ->  M     thus  B -> ~C          (EAE)
      Darii:      M ->  C     and   B  &  M     thus  B  &  C          (AII)
      Ferio:      M -> ~C     and   B  &  M     thus  B  & ~C          (EIO)

Figure II
      Cesare:     C -> ~M     and   B ->  M     thus  B -> ~C          (EAE)
      Camestres:  C ->  M     and   B -> ~M     thus  B -> ~C          (AEE)
      Festino:    C -> ~M     and   B ->  M     thus  B  & ~C          (EIO)
      Baroco:     C ->  M     and   B -> ~M     thus  B  & ~C          (AOO)

Figure III
      Darapti:    M ->  C     and  QM ->  B     thus  B  &  C          (AAI)
      Disamis:    M  &  C     and   M ->  B     thus  B  &  C          (IAI)
      Datisi:     M ->  C     and   M  &  B     thus  B  &  C          (AII)
      Felapton:   M -> ~C     and  QM ->  B     thus  B  & ~C          (EAO)
      Bocardo:    M  & ~C     and   M ->  B     thus  B  & ~C          (OAO)
      Feriso:     M -> ~C     and   M  &  B     thus  B  & ~C          (EIO)

Figure IV
      Bamalip:    C ->  M     and   M ->  B     thus  B  & QC          (AAI)
      Calemes:    C ->  M     and   M -> ~B     thus  B -> ~C          (AEE)
      Dimatis:    C  &  M     and   M ->  B     thus  B  &  C          (IAI)
      Fesapo:     C -> ~M     and  QM ->  B     thus  B  & ~C          (EAO)
      Fresison:   C -> ~M     and   M  &  B     thus  B  & ~C          (EIO)
```

Meanwhile in the Non-European World

In 10th century Baghdad, the Nestorian **Abu Bishr Matta ibn Yunus** refined Aristotle's logic, but his work was lost in the passage of time.

In India, logic was hotly debated in a form which differed only slightly from the syllogism:

The mountain is fiery	that is the Proposition
Because smoky	that is the Reason
All that is smoky is fiery	that is the Example
So here	that is the Application
Therefore it is so.	that is the Conclusion

The use of *negation* caused debate:

why should the same words in different order have different meanings?

He shall-not look.
He shall not-look.
Not-he shall look.

"Absence of constant absence of pot is essentially identical with pot"
-- Mathuranatha c. 1700

In the West, Logic Evolved into Formal Systems

Renaissance: Logic was ignored (experience was in vogue)

Enlightenment: Leibniz sought a Universal Calculus of Reason, and studied Indistinguishability.

1850 **Boole**: expressed sentences and noun expressions as algebra

`x + y = y + x`	associativity of OR
`x (y + z) = x y + x z`	distribution of AND over OR
`if x = y then x + z = y + z`	algebraic substitution

1880 **Venn**: logical diagrams

1885 **Peirce**: truth tables

1900 **Russell**: logical foundations of mathematics

1920 **Post**: metalogic (just what are we doing?)

Crisis in the Twentieth Century

Oh No! There is **no consistency in mathematics**, there are paradoxes in every system.

Logicism Bertrand Russell

Mathematics is identical to logic.
(We'll patch the holes.)

Intuitionism L.E. Brouwer

Mathematics presupposes concepts.
Concepts rest on natural numbers.
(We'll construct what is known, and not admit infinity.)

Formalism David Hilbert

Mathematics is a set of syntactic transformations.
(We'll refuse to interpret it.)

Logic and Computation

From Conceptualization to Formalization

Here's what we do when we build a ***formal model*** (or ***do a computation***):

0. Identify a **collection of objects**/events in the real world. This is the *semantic mapping*, how math is linked with reality. The objects/events must have these properties:

unique	not confused with different objects/events
stable and permanent	not in flux or changing too rapidly to identify
discrete	not lacking well defined borders
comprehendible	not confusing or too ambiguous
relevant	not outside of what we consider to be the objects in question
permitted	not in violation of tacit understandings about how things are

1. Use **unique labels** to identify each of the things in the semantic mapping. The *value* of a label is the thing it identifies.

2. Limit our interest in the types of things in the real world to an **abstract mathematical property**, such as Truth or Count or Membership.

3. Use different labels to **name** different *abstract* things:

labels for things	*object labels*
labels for an entire set of things	*property labels*
labels for an arbitrary thing in the set	*variable labels*
labels to name properties of things indirectly	*function labels*
labels to name combinations of things	*relation labels*

4. Follow the *rules of symbol transformation* to **manipulate the labels** as if they were the things. However, the labels do not have to share any of the real world properties of the physical things.

In summary, we convert from physical to virtual, ignore the physical aspects of reality, manipulate the virtual (or digital) aspects using the rules of virtuality, and then return to physical reality with new knowledge. The rules of virtuality are listed in the handout *The Canons of Formal Symbol Systems*.

Virtuality has become so prevalent in the current Information Age that much of what we do is never part of physical reality in the first place. Computer Science, for example, is a discipline in which the only connection to physical reality is silicon hardware. Study of the physicality of hardware is a different discipline, Electrical Engineering.

Computational hardware is engineered (constructed) to behave as if it were a mathematical system called *Timed Boolean Logic*. "Timed" simply means that some parts of the physical hardware are used more than once to do the Boolean logic task for which they were constructed.

Formal Simplicity

The simplest interesting formal domain is **Propositional Calculus**, also known as **Boolean Logic**, and also known as **Boolean Algebra**.

In Boolean algebra, the labels for real world objects are sentences in human language which can be evaluated as either True or False. These sentences are composed of atoms (propositions) and logical connectives. The atoms are labels for things which exist and are in some way actual, or real, or True. The logical connectives are those parts of language which do *not* refer to something in physical reality.

More generally, Boolean algebra includes all of *decision theory*, which phrases a problem space as a sequence of binary decisions. (Note that more complex decisions can usually be phrased as a collection of binary decisions.)

The truth of a sentence, the direction of a decision, and the voltage in a digital circuit wire are all binary properties. Every object label in a Boolean problem has the *binary property*, either 0 or 1, and that is the only property of interest. Logic is the set of rules which maintains the invariance of the value of binary properties.

Labeling Reality

Let {a, b, c, ...} be simple Boolean variables identifying a set of decision events. Encode the value of each decision (yes/no, true/false, act/pass) as 1/0.

Combinations of Boolean operators and Boolean variables are called *sentences* or *expressions*.

Let {A, B, C, ...} be names of Boolean expressions.

Valid Boolean expressions are defined recursively,

a	simple variables are expressions
not A	negated expressions are expressions
A or B	expressions joined by disjunction are expressions

Since all other Boolean operators can be expressed in terms of **not** and **or**, they are all included in the above definition. This definition means that Boolean expressions are composable and decomposable; the recursive rules provide Constructors and Accessors.

Symbolic Complexity

The truth table (or Boolean property table) of a Boolean operator specifies how that operator combines the Boolean properties, or the values, joined by the operator. But here things start to get complex. Mainly, we can form a huge variety of expressions, and to find the Boolean value of any expression, we must examine a table with a size exponential in the number of variables (2^n entries). So a logic circuit with 200 input variables requires an impractical 2^200 rows in the truth table.

So all decision problems which require a large number of decisions become intractably inconvenient to examine in the whole. Boolean algebra provides *transformation rules* which hold for all expressions, regardless of which Boolean value an expression has. For example,

not (not A) = A

is valid regardless of the expression A.

Axioms and Theorems

Transformation rules can be decomposed into axioms and theorems. The axioms are a minimal set of rules which preserve the Boolean property and serve as a basis for all other possible theorems. All the axioms of an axiomatic formal system can be compressed into one single, usually complex axiom. For Boolean logic, the simplest single axiom which supports all other Boolean transformations is the Kauffman/Flagg formalization of the Robbins problem:

(A or B) and (A or (not B)) = A EXTENSION

Similarly, the Resolution rule is a single rule which supports all Boolean transformation. The many rules of Natural Deduction provide another, more complex set of axioms. One particularly simple set of supporting axioms (Bricken's formalization of Boundary Logic) is the following:

A or True = True	DOMINION
not (not A) = A	INVOLUTION
A or (not (A or B)) = A or (not B)	PERVASION

This set is particularly useful for pattern-matching and automated deduction, since the patterns to be matched are relatively simple, and the right-hand-side of each equation is simpler than the left-hand-side solely through erasing something from the right-hand-side.

Proof

Boolean algebra axioms and theorems (valid transformations) provide a way to explore decision spaces without making the actual decisions. This is called *logical or algebraic proof*. However, the situation remains complex because now we must select which theorem to apply and where to apply it. Although the search space is more abstract, it is still intractable and inconvenient. Although Boolean algebra abstracts the physical properties of decisions, it is still a *real world* problem to use Boolean algebra efficiently.

The Equivalence of Deduction and Computation

From 1920 to 1970, a central issue for theoretical computer science was whether a computation maintained the meaning of a logical (deductive) process. Fortunately the answer is yes, given that the programming language follows the rules of logic. These are the essential concepts:

P |- Q Single turnstile: Q is ***computed*** from P

P |= Q Double turnstile: Q is ***deducible*** from P

Soundness: **If P |- Q, then P |= Q**

A sound computation always maintains the deductive model. The computation never generates a False model, it never produces an incorrect fact.

Completeness: **If P |= Q, then P |- Q**

A complete computation never varies from the deductive model. The computation generates all True models. If something can be deduced, the computation will be able to deduce it.

S*ound and Complete*: **P |- Q = P |= Q**

The deductive model and the computation represent the same Universe. The computation produces all that can be deduced, and only what can be deduced.

Decidability

Universal:

If it can be computed formally, then it can be computed using First Order Logic (Turing equivalence).

Decidable:

The computational procedure will terminate with a Yes/No result.

Semi-decidable:

The computation might halt, but you don't know when. It may never halt if you ask the wrong kind of question. What we can't do is ask questions which depend on the **failure** to prove something:

No: "Check to see if nothing is wrong"
No: "Prove that this search will fail to find X"

SAT and TAUT

The ***most important theoretical question*** in Computer Science (with the most critical practical applications) is

Can we write an algorithm which runs
in ***polynomial space and time*** with regard to the number of variables
that shows that an arbitrary expression in Propositional Calculus
has a *non-trivial variable* in it?

SAT: Is a Boolean expression SATisfiable?

That is, is there an assignment of variable values (True or False) which results in different outcomes for the entire expression (again either True or False)?

TAUT: Is a Boolean expression a TAUTology

That is, is every variable in the expression irrelevant? Is the expression always True regardless of the values assigned to the variables in it?

If TAUT or SAT is solvable within polynomial bounds, then **P = NP**

P =?= NP

Almost all complex but common computational problems can be mapped onto the SAT problem. If you can show propositional satisfiability within polynomial bounds, then most computational problems have an efficient solution. In particular, programs and circuits can be verified efficiently.

Polynomial complexity: O[n^some-power]

some polynomials: n^2 (n+1)(n - 1) 14n^6 + 3n^3 + n + 104

Non-deterministic polynomial complexity: O[2^n] or worse

some non-polynomials: 2^n 7^(n + 3) n! 2^(2^n)

Polynomial algorithms are called ***tractable***: they all run in "reasonable" time. Non-polynomial algorithms are intractable: none of them are efficient enough to be useful for problems of arbitrary size.

Logical Proof

Ways of Expressing the Mathematics of Logic

Boolean connectives (and, or, not, if-then, if-and-only-if)
function tables (truth tables)
Boolean algebra
Venn diagrams
switching circuits
transistor arrays (silicon chips)
Boolean lattice
Boolean cubes (blocks in space)
matrix logic
boundary logic

Ways of Computing the Mathematics of Logic

exhaustive listing of possibilities	(truth tables)
deduction/inference	(Boolean connectives)
algebra	(Boolean algebra)
spatial overlap	(Venn diagrams)
current through transistors	(circuitry)
partial orderings	(lattices)
spatial conjunction	(cubes)
operators	(matrix logic)
containment	(boundary logic)

Mechanisms of Proof

Truth tables
Natural deduction
Resolution (not covered in class)
Boundary logic
Induction

Truth Table Analysis

Examining all possibilities is exponential in number of cases: there are 2^n cases to evaluate for n variables even in the simplest case of propositional logic without functions or relations. However, lookup tables are a brute force algorithm that is easy to understand and to implement. The technique is to list all possible combinations of values for each variable, and use simple definitions of the logical connectives to evaluate compound sub-expressions.

Example: `if (P and Q) then (R = (not S))`

P	Q	R	S	(not S)	(P and Q)	(R = (not S))	(if P&Q then R=~S)
T	T	T	T	F	T	F	F
T	T	T	F	T	T	T	T
T	T	F	T	F	T	T	T
T	T	F	F	T	T	F	F
T	F	T	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	T	F	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	F	F	F	T
F	T	T	F	T	F	T	T
F	T	F	T	F	F	T	T
F	T	F	F	T	F	F	T
F	F	T	T	F	F	F	T
F	F	T	F	T	F	T	T
F	F	F	T	F	F	T	T
F	F	F	F	T	F	F	T

Deduction

The rules of inference, or natural deduction, apply at three different **levels of abstraction**: individual propositions, individual sentences, and collections of sentences. Modus Ponens serves as an example.

Atoms: `(p and (p -> q)) -> q`

Sentences: `(A and (A -> B)) -> B`

Sets of sentences: `({A,B...} and ({A,B...} -> {C,D...})) -> {C,D...}`

Deductive Steps

There are three separate concepts of **proof step** (written above as "implies") which have been shown to be equivalent:

material implication, logical implication, and computation.

Material implication: **p -> q**

is defined by the Truth Table of values. Notice that "(if False then True) is True", (the second row) does not make sense in language structures, it is True by definition.

p	q	(p -> q)
0	0	1
0	1	1
1	0	0
1	1	1

Logical Implication: **p |= q**
is defined by common sense and by the rules of deduction.
"if (p is logically True) then (q is logically True)"

Formal Proof: **p |- q**
is defined by taking logical implication steps from p to q
"if (p is True) then a sequence of implications shows (q is True)"

The Rules of Natural Deduction

Natural deduction evolved from natural language and from human intuition, so it is relatively easy to understand. It is very difficult to find the right rules to apply at the right time (exponential in difficulty of use). Recall that humankind has had an extremely difficult time coming to understand logic, and logic itself is still undergoing extreme revision. "**|=**" means "logically implies" while "**->**" is simply a symbol referring to a particular truth table. The same subtle difference exists between "**and**" and "**&**".

Modus Ponens:	A	and	A -> B	|=	B
Modus Tollens:	~B	and	A -> B	|=	~A
Double negation:	A			|=	~~A
	~~A			|=	A
Conjunction:	A	and	B	|=	A & B
Simplification:	A & B			|=	A
	A & B			|=	B
Addition:	A			|=	A v B

Natural Deduction Proof Techniques

Modus Ponens:	A	and	A |- B	|=	B
Modus Tollens:	~B	and	A |- B	|=	~A
Conditional proof:	A |- B			|=	A -> B
Dilemma:	(A or B)	and (A |- C)			
		and (B |- C)		|=	C
Contradiction:	(A |- B)	and	~B	|=	~A
Cases:	(A is True |- B) and				
	(A is False |- B)			|=	B

Note that "|-" is a sequence of formal steps, while "|=" is assurance of logical truth.

Natural Deduction Example

Premise 1: If he is lying, then (if we can't find the gun, then he'll get away).
Premise 2: If he gets away,
then (if he is drunk or not careful, then we can find the gun).
Premise 3 It is not the case that (if he has a car, then we can find the gun).
Conclusion: It is not the case that he is both lying and drunk.

Encode the propositions as letters:

L = he is lying
G = we can find the gun
A = he will get away
D = he is drunk
C = he is careful
H = he has a car

Premise 1:	If L then (if (not G) A)
Premise 2:	If A then (if (D or not C) then G)
Premise 3:	Not (if H then G)
Conclusion:	Not (L and D)

Encode the logical connectives:

```
P1:   L ->  (~G ->  A)
P2:   A ->  ((D v ~C) -> G)
P3:   ~(H -> G)
C:    ~(L & D)
```

Figure out a good proof strategy. This step is not algorithmic, and is the source of difficulty in natural deduction approaches. Here the Contradiction strategy works:

1.	(L & D)	assume the negation of the conclusion, and plan to show a contradiction
2.	L	simplification of 1
3.	D	simplification of 1
4.	~G -> A	modus ponens with 2 and P1
5.	~(~H v G)	rewrite P3 with conditional exchange: X -> Y = ~X v Y
6.	~(~H v ~~G)	double negation of part of 5
7.	H & ~G	rewrite 6 with DeMorgan: ~(~X v ~Y) = X & Y
8.	~G	simplification of 8
9.	A	modus ponens with 8 and 4
10.	((D v ~C) -> G)	modus ponens with 9 and P2
11.	(D v ~C)	addition of ~C to 3
12.	G	modus ponens with 11 and 10
13.	G & ~G	conjunction of 8 and 12
14.	~(L & D)	steps from 1 to 13 have created a contradiction: G & ~G = False, therefore the assumption on line 1 is false. But that assumption is the negation of the conclusion. Therefore the negation of the negation of the conclusion is True. That is, the conclusion is True.

Deduction Exercises

Here are some theorems to prove using each of the different Boolean computational techniques. Prove that each sentence is a tautology.

Double negation: `(not (not A)) = A`

Modus Ponens: `(A and (A -> B)) -> B`

Subsumption: `(A and (A or B)) = A`

Resolution: `((A -> B) and ((not A) -> C)) -> (B or C)`

Drunken Liars/Fruit:

```
((L -> ((not G) -> A)) and
 (A -> ((D or (not C)) -> G)) and
 (not (H -> G))) -> (not (L and D))
```

To make the deduction exercises more relevant, here are some pseudocode fragments (using infix notation). Capital letters represent calls to existing subprograms. You should transform each into a more efficient code fragment using any of the deductive tools we have discussed.

1. `(not ((not A) or (not B)))`

2. `(if ((if test then A else True) and test) then A else True)`

3. `((not A) and (B or (not A)))`

4. `(block (if test1 then A else B); (if test2 then B else A); A)`

5.
```
(if (test1[A D E F] and (not test2[A B C D]))
 then ((not A) or (not E)) else True)

  test1[A D E F] =def= (if D then (A or (not (E or (not F)))) else True)
  test2[A B C D] =def= ((not (A or (not B))) and
                        (if C then (A or D) else True)
```

6.
```
(if test then A else B)

  A[] =def= (if test then C else True)
  B[] =def= (if test then D else True)
```

7. You are working on a Y2K problem, and you know that

```
  (if ((X has-two-digits) or (X isa-name-of-a-month))
   then (X isa-two-digit-month))
and
  (if ((X isa-name-of-a-month) or (not (X isa-two-digit-month)))
   then (if (not (X has-three-digits)) then (X isa-date)))
and
  (not ((X isa-two-digit-month) or (X has-three-digits)))

Prove that (X isa-date)
```

Determine which of these sentences is a tautology. For those that are not tautologies, determine a set of values for the relevant variables that satisfy the sentence.

1. `(or (if t t t) (if f f f))`

2. `(iff (and t f) (or t f))`

3. `(if (or a b) (or b a))`

4. `(if (not a) (or a a))`

5. `(and p p)`

6. `(and (or a b) (or (nor a b) (nor a b)))`

7. `(iff (if a a a) a)`

8. `(iff (or p (and q r)) (and (or p q) (or p r)))`

9. `(not (iff (not (and a (or a b))) (not a)))`

10. `(not (if (and a b) a b))`

11. `(if (if a b) (if a (or b c)))`

12. `(if (and p q) (if (and p q) p))`

13. `(or (and (or a b) (or a c)) (and (or a b) (or b e)))`

14. `(if (and (and p (not p)) (if r u)) (or q s) )`

15. `(if (and (if d (or a c)) (and d (not a))) c)`

16. `(if (and (if a b) (if b c) (if c d) (not d)) (not a))`

17. `(iff (if (if a b c) d e) (if a (if b d e) (if c d e)))`

18.
```
(and (or (not a) (not c)) (or (not a) c)
      (or a b) (or a c) (or a (not b) (not c)))
```

19.
```
(if (and (if (or a b) (or c d)) (if (or c f) h)
          (and e (not d)) (if e a)) (or h i) )
```

20.
```
(iff (nor (and a c) (and b (or a (not c)) (or c (not a))))
      (or (nor a (nor c (not b))) (nor c (nor a (not b)))))
```

21.
```
(and (or a c) (or (not a) (not c))
      (or (and (not a) (not b)) (and a b (not c)))
      (or (and (not b) (not c)) (and b c (not a))))
```

22.
```
(if (and (if (or b c) a) (if a (or s u))
          (and b (not s)) (if (and u a) (if w s))) (not w))
```

23.
```
(iff (or (and a b c) (and a (not b) c)
      (and a (not b) (not c)) (and (not a) (not b) c)
      (and (not a) (not b) (not c)))
   (or (not b) (and a c)))
```

Logical Tautologies

1.	P v ~P	excluded middle
2.	~(P & ~P)	noncontradiction
3.	~~P = P	double negation
4.	(P & Q) -> P	simplification
5.	P -> (P v Q)	simplification
6.	(P & P) = P	idempotence
7.	(P v P) = P	idempotence
8.	(P & (P -> Q)) -> Q	modus ponens
9.	((P -> Q) & (Q -> R)) -> (P -> R)	syllogism
10.	(P -> Q) = (~Q -> ~P)	contraposition
11.	((P -> Q) & ~Q) -> ~P	modus tollens
12.	((P v Q) & ~P) -> Q	disjunctive syllogism
13.	(P -> Q) = (~P v Q)	conditional disjunction
14.	(~P -> (Q & ~Q)) -> P	reductio ad absurdum
15.	(((P -> R) & (Q -> S)) & (P v Q)) -> (R v S)	dilemma
16.	(P -> (Q -> R)) = ((P & Q) -> R)	exportation
17.	(P = Q) = ((P -> Q) & (Q -> P))	biconditional
18.	~(P v Q) = (~P & ~Q)	DeMorgan
19.	~(P & Q) = (~P V ~Q)	DeMorgan
20.	~(P -> Q) = (P & ~Q)	negation of conditional
21.	~(P = Q) = (~P = Q)	negation of biconditional

22.	(P v Q) = (Q v P)	commutativity
23.	(P & Q) = (Q & P)	commutativity
24.	(P = Q) = (Q = P)	commutativity
25.	((P v Q) v R) = (P v (Q v R))	associativity
26.	((P & Q) & R) = (P & (Q & R))	associativity
27.	((P = Q) = R) = (P = (Q = R))	associativity
28.	(P & (Q v R)) = ((P & Q) v (P & R))	distribution
29.	(P v (Q & R)) = ((P v Q) & (P v R))	distribution
30.	(P -> (Q v R)) = ((P -> Q) v (P -> R))	distribution
31.	(P -> (Q & R)) = ((P -> Q) & (P -> R))	distribution
32.	((P v Q) -> R) = ((P -> R) & (Q -> R))	disjunction/conditional
33.	((P & Q) -> R) = ((P -> R) v (Q -> R))	conjunction/conditional
34.	(P -> Q) -> ((R & P) -> (R & Q))	factorization
35.	(P -> Q) -> ((R v P) -> (R v Q))	summation

Algebraic Logic

Equational Logic A=B

Logical expressions joined by 'equals'.

Axioms of Equations

1. Equality (and Truth) is preserved
   whenever an expression is substituted for its equal.

   ```
   If A = B and B = C, then A = C
   ```

2. Functions of equals are equal.

   ```
   If A = B,  then F(A) = F(B).
   ```

Axioms of Equals

1. ***Identity***: `A = A`

2. ***Commutative***: `A = B  iff  B = A`

3. ***Transitive***: `if  A = B  and  B = C,  then  A = C`

Axioms of Substitution

0. A[X/Y] means "substitute Y for every X in A"

1. Substituting one expression for another in an equation
   preserves the equality.

   ```
   If A = B,  then  A[C/E] = B[C/E]
   ```

2. Substituting equal expressions for any subexpressions in an expression
   preserves the equality.

   ```
   If A = B,  then  C[A/E] = C[B/E]
   ```

Rule of Standardization

A = B iff (A iff B) = True

Algebraic Proof Techniques

Standard Form:

```
A = B     iff    ((A -> B) and (B -> A))  =  True
```

Direct Transformation:

```
A = B     iff     A => B   or   B => A
```

Mutual Transformation:

```
A = B     iff     A => C   and    B  =>  C
```

Case Analysis:

```
A = B     iff           A[T/E]  =  B[T/E]
                  and   A[F/E]  =  B[F/E]
```

Linear Algebra:

```
 A = B     iff         A => T  and  B => T
                  or   A => F  and  B => F
```

Lattice Theory

Lattice theory is the study of a single binary relation $\geq$
to be read as “is contained in”.

A *lattice* is a partially-ordered set (poset), and two elements of which have a
greater lower bound (glb, meet) and a **least upper bound** (lub, join)

A *boolean lattice* is a complemented, distributed lattice, and forms a boolean algebra.

Techniques for Logical Deduction

Approaches to Deduction

Asterisks indicate primary features of approach

Truth Tables

* easy to understand
exhaustive listing of all cases (doesn't work for infinite domains)
* brute force, little thinking
exponential (2^n) in number of variables

Natural Deduction

relatively easy to follow, hard to understand
flexible input form
* many one-directional inference rules
requires insight and cleverness
stored intermediate facts grow exponentially

Resolution

hard to understand
standardized input (CNF), grows exponentially
* single one-directional inference rule (good for algorithms)
stored intermediate facts grow exponentially

Algebraic Logic

easy to understand
flexible input form
* few bidirectional simplification rules
requires some insight
stored intermediate facts not used

Matrix Logic

relatively easy to understand
* every object is an operator
* standard matrix addition and multiplication
brute force
exponential (effectively the same as truth tables)

Boundary Logic (void-based reasoning)

hard to understand
flexible input form (any logical form)
* few easy to apply rules
requires little thinking
* facts shrink instead of growing

Logic Gates

When numbers are expressed in binary, addition can be expressed in terms of logical gates.

```
              32    16    8     4     2     1          powers of 2

 13                         .     .           .       summand
+22                   .           .     .             summand

=35           .                          .     .       sum
```

Rules of combination:

```
A     B     sum   carry

0     0     0     0
0     1     1     0
1     0     1     0
1     1     0     1

            XOR   AND
```

N-variable Boolean Functions

Different Boolean functions have different binary values associated with each combination of values of variables. For `N` variables, there are `2^N` combinations of values (all the rows in a truth table). For the `2^N` combinations, there are two ways to assign a truth value, resulting in (`2^2^N`) Boolean functions of `N` variables.

Number of variables	*Number of functions*	
0	2	2^2^0
1	4	2^2^1
2	16	2^2^2
3	256	2^2^3
4	65536	2^2^4
5	very large	2^2^5

These functions can be arranged at the nodes of an N-dimensional hypercube, which is also a *binary, complemented, distributed lattice*. Here are the listing for 0,1, and 2 variables:

0 variables

functions

```
0     1
```

function names `True  False`

1 variable

a	*functions*			
0	0	0	1	1
1	0	1	0	1
function names	False	a	~a	True

2 variables

a	*b*	*functions*														
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
names	F	&		a		b	xor	v	nor	=	~b		~a	if	nand	T
			nif		nfi							fi				

```
nif = ~(a -> b)          fi = (b -> a)          nfi = ~(b -> a)
```

Some of the 16 two-variable functions have common names, some have technical names from Electrical Engineering, and others are not named. Note that the above columns for the functions are the truth tables for that function.

Normal Forms

Any Boolean function can be expressed as a *conjunction of disjunctive clauses:*

CNF: Conjugate Normal Form

E.g.: `(A v B v ~C) & (A v C) & (B v ~C v D)`

Groupings of forms joined by OR are called **clauses**. Clauses are joined by AND.

CNF has minimal depth (2 layers deep) and
a maximal number of variable references (up to 2^n)

CNF is a normal form in that a specific Boolean function will have a single CNF form.

Any Boolean function can be expressed as a *nesting of implications:*

INF: Implicate Normal Form

E.g.: `((((A -> ~B) -> C) -> (D -> E)) -> ~G)`

INF has maximal nesting, or depth and a minimal number of variable references.

There are many INF forms for a given Boolean function (so it's not truly a normal form)

It is always possible to express a Boolean function
with only two occurrences of a selected variable.

Minimal Bases

It is possible to express many the Boolean functions in terms of other functions. The ***basis set*** is the set of functions which are taken to be non-decomposable.

Common Basis Sets:	`{and, or, not, T}` `{not, if, T}`	
Small Basis Sets	`{nor, F}` `{nand, T}`	
Minimal Basis Set:	`{nor}`	(this requires an innovative notation)

Resolution

Resolution expresses Boolean functions as sets of literals. This is a different way to express CNF. The disjunctive forms in each clause form a set with implicit disjunction. Each clause forms a different set.

Literals: atoms and negated atoms

Clauses: sets of literals joined by OR

The Resolution Rule

Let S1 and S2 be sets of clauses, and U be the set Union operator:

```
({a,b,...} U S1) & ({~a,b,...} U S2)  ==>  {b,...} U S1 U S2
```

E.g.: `{a,b,~c} & {~a,b,d}  ==>  {b,~c,d}` resolve on a

Termination

```
{a} & {~a}  ==>  { }  ==>  False

{a,~a}  ==>  True
```

Not complete

```
~{~a,~b} & { }  ==>  no action
```

Resolution and Natural Deduction

	Resolution	*Natural deduction*
End Case:	`{a} & {~a} = { }`	`(a & ~a) = False`
Modus Ponens:	`{a} & {~a,b} ==> {b}`	`a & (~a v b) ==> b` `a & (a -> b) ==> b`
Chaining:	`{a,b} & {~a,c} ==> {b,c}`	`(a v b) & (~a v c) ==> (b v c)`

Lattices

A lattice is a *directed graph* with links representing an ordering relation. Lattices can have a maximal and a minimal element

```
( )   maximal
 |
( )
 |                > greater than
( )
 |
( )   minimal
```

A ***partial ordering*** uses the ordering relation *greater-than-or-equal-to*.

```
  ( )          maximal
 /   \
(1)   (2)                  1 = 2
 \   /
  ( )          minimal
```

Hasse Diagrams (aka lattices)

A *set* and an *ordering relation* {S,>}, such that

- each object is a *vertex*
- if `(a > b)`, then a is *higher than* `b`.
- if there is no c such that `(a > c > b)`, then a *is connected to* `b`.

```
(a)   (b)      no maximal element
 \   /
  (c)          (a>c) & (b>c) & (c>d)
   |
  (d)          minimal
```

Matrix Logic

By arranging the truth table of a Boolean function in a matrix form, the rules of logic can be converted into the rules of matrix algebra. The general format is:

```
        B
      T   F
  T   .   .
A
  F   .   .
```

Some examples:

A & B	A v B	A = B	A->B	A	~A	T
1 0	1 1	1 0	1 0	1 1	0 0	1 1
0 0	1 0	0 1	1 1	0 0	1 1	1 1

Each Boolean matrix is an ***operator***. That is, in this formulation, there are no objects. When using binary operations, matrix addition is `xor`; matrix multiplication is `and`.

a + b = c	a * b = c
0 + 0 = 0	0 * 0 = 0
0 + 1 = 1	0 * 1 = 0
1 + 0 = 1	1 * 0 = 0
1 + 1 = 0	1 * 1 = 1
xor	and

Note that these relations are the same ones that apply to computational addition.

As well, some matrix combinations result in matrices which are not Boolean functions. This then extends Boolean operations into generally unexplored territory, *imaginary Boolean operations*. Some examples of translating between operators:

a + ~a = T	1 1	+	0 0	=	1 1
	0 0		1 1		1 1
a * ~a = a	1 1	*	0 0	=	1 1
	0 0		1 1		0 0
xor + and = or	0 1	+	1 0	=	1 1
	1 0		0 0		1 0
nor^2 = nor	0 0	*	0 0	=	0 0
	0 1		0 1		0 1
xor^2 = equal	0 1	*	0 1	=	1 0
(square-root of equal)	1 0		1 0		0 1
and + or = ?	1 0	+	1 1	=	2 1
	0 0		1 0		1 0

Boolean Cubes

A Boolean function can be expressed in terms of a collection of vertices of a hypercube (*this is not the same use as the lattice hypercube*). The set of all Boolean functions of N variables is defined by all the possible collections (the power set) of vertices (called **cubes**).

Each cube is the *conjunction of unique literals*, one from each variable. The whole is formed by the disjunction of all cubes.

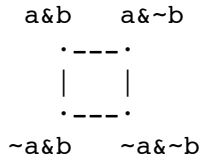
Examples:

1 variable

All possible combinations:

`void = F`	no cubes
`a`	single cube
`~a`	single cube
`a v ~a = T`	both cubes

2 variables



All possible combinations:

`void`	`= F`	no cubes
`a&b`	`= and`	single cubes
`a&~b`	`= nif`	
`~a&b`	`= nfi`	
`~a&~b`	`= nor`	
`a&b v a&~b`	`= a`	two cubes
`a&b v ~a&b`	`= b`	
`a&b v ~a&~b`	`= equal`	
`a&~b v ~a&b`	`= xor`	
`a&~b v ~a&~b`	`= ~b`	
`~a&b v ~a&~b`	`= ~a`	
`a&b v a&~b v ~a&b`	`= or`	three cubes
`a&b v a&~b v ~a&~b`	`= fi`	
`a&b v ~a&b v ~a&~b`	`= if`	
`a&~b v ~a&b v ~a&~b`	`= nand`	
`a&b v a&~b v ~a&b v ~a&~b`	`= T`	all cubes

Boolean Cube Operations

Cubes can be used for computation, either symbolically or physically.

function = set of cubes

not function = set of cubes not in function

f or g = overlay the cubes of f and the cubes of g

f and g = intersect the cubes of f and the cubes of g

Perspective as an Operator

By removing the orientation of a Boolean cube (or a Boolean lattice), varieties of Boolean function collapse into the same form. For example, all single cube functions are the same (i.e. composed of one cube) with orientation is ignored. Another example, expressed in matrix notation:

```
1 1  =  0 1  =  0 0  =  1 0
0 0     0 1     1 1     1 0

 a       ~b      ~a      b
```

These four functions are the same when the matrix is free to rotate.

Boundary Logic

Advances in knowledge must necessarily appear to be unintelligible *before* their discovery and simple or obvious *after* their discovery.
-- Arthur Fischell

Challenge

Computation and logic (Boolean algebra) are universally built on binary representations.

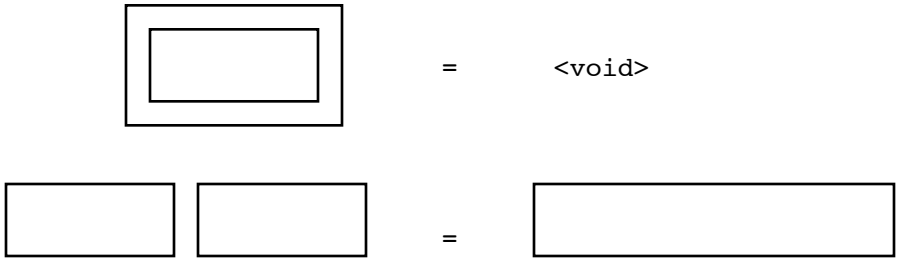
```
0     1                 True  False            Yes   No
```

Is there a simpler approach? Can logic be expressed in a *unary* notation?

Boundary Mathematics

The use of delimiting tokens, or *containers*, as both constants and functions.

Pure math example: *Common boundaries cancel*.



Concepts

Boundary token	an enclosure
Representational Space	the bounded space

The Simplest Virtual World

<this space is intentionally left blank>

<the above contradicts itself>

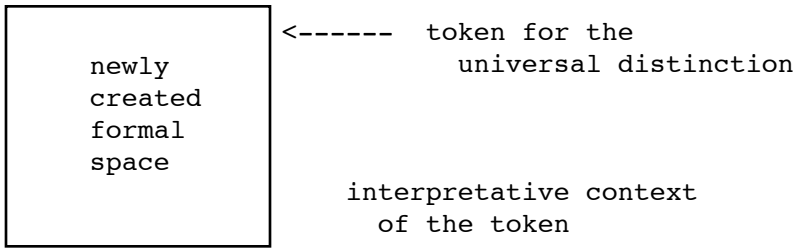
Two Voids

Absolute void:	that which cannot be referred to without contradiction
Relative void:	emptiness enclosed within a boundary

Constructing a Distinction

A **Universal Distinction** is first boundary we agree upon. In forming a universal distinction, we construct three things simultaneously:

a formal space (inside)
an interpretative context (outside)
a token representing the distinction (boundary)

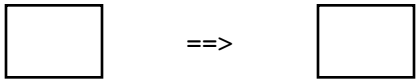


Calling

Focus your attention on the outside,
where you see the mark (the usual viewing point).

Call the boundary that you see a “symbol”.

To call is to maintain perspective.



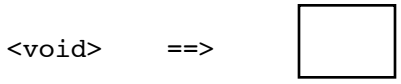
Calling is the rule of **invariance**. It is also is the rule of **naming**. Thus the relationship between an object and its name is invariant.

Crossing

Focus your attention on the inside,
where you cannot see a mark (there is no mark inside).

Cross the boundary to the outside. Now you can see a mark.

To cross is to change perspective.



Crossing is the rule of **variance**. It is also a process of changing.

The Arithmetic of Boundaries

CALLING	( ) ( ) = ( )
CROSSING	(( )) =

Moving to Algebra

The ground, or carrier set, of boundary logic is one token { ( ) } and one absence of a token. If an equation holds for all ground values, it holds in general. Using this, we can construct algebraic truths from the cases of the arithmetic:

DOMINION	INVOLUTION	PERVASION
( ) ( ) = ( )	((( )) = ( )	(( )) ( ) = ( )
( ) = ( )	(( )) =	( ) = ( )
thus		
( ) A = ( )	((A)) = A	(A) A = ( )

Boundary Logic Algebraic Axioms

The transformation axioms of boundary logic:

Dominion (the halting condition, when to stop)

```
( ) A = ( )                    REIFY  <==>  ABSORB
```

Involution (double negation, how to remove excess boundaries)

```
((A)) = A                      ENFOLD  <==>  CLARIFY
```

Pervasion (how to remove excess variable mentions)

```
A (A B) = A (B)                INSERT  <==>  EXTRACT
```

Each axiom suggests a definite reduction strategy:
erase irrelevant structure
to convert the left side of the equation to the right side.

Algebra provides the useful tool of **substitution independence**. Any transform can be applied at any time and at any place in the expression without changing the value of the expression. Thus, all transformation paths do not change the value of an expression. It doesn't matter how you get to a simpler expression (an answer). Some paths may be longer and less efficient, but all lead to equivalent results.

Boundary Logic

Boundary logic uses a **spatial representation** of the logical connectives. Since CALLING provides an object-centered interpretation, and CROSSING provides a process-centered interpretation of the same mark, boundary forms can be evaluated using either an algebraic (match and substitute) process or a functional (input converted to output) process.

Representation of logic and proof in spatial boundaries is new, and quite unfamiliar. Boundary logic is not based on language or on typographical strings, nor is it based on sequential steps. Boundary techniques are inherently **parallel and positional**. The meaning, or interpretation, of a boundary form depends on where the observer is situated. From the outside, boundaries are objects. From the inside, you cross a boundary to get to the outside; boundaries then are processes. This dramatically different approach (that is, permitting the observer to be an operator in the system) does not change the logical consequences or the deductive validity of a logical process.

Spatial representations have built-in associativity and commutativity. The base case is no representation at all, that is, **the void has meaning** in boundary logic. Simplification of logical expressions occurs by **erasure of irrelevancies** rather than by accumulation of facts.

Boundary Logic Representation

logic	*boundary*	*comments*
False	<void>	no representation. Note: (( )) = <void>
True	( )	the empty boundary
A	A	objects are labeled by tokens
not A	(A)	
A or B	A B	Disjunction is sharing the same space
A and B	((A)(B))	
if A then B	(A) B	Implication is separation by a boundary
A iff B	(A B)((A)(B))	

In the above map from conventional logic to boundaries, the many textual forms of logical connectives condense into one boundary form. Note that the parens, ( ), is a linear, or one-dimensional, representation of a boundary. Circles and spheres are expressions of boundaries in higher dimensional representations.

Multiple Readings of the Same Form

A single expression in the simpler notation of boundary logic can express (infinitely) many different forms in a more complex notation.

For example:

`( (A) (B) )`	A and B (not ((not A) or (not B))) (not (A implies (not B))) ((not A) or (not B)) implies False

Comparative Axiomatic Basis

An **axiomatic basis** is a minimal set of transformations from which all other transforms can be derived. The basis of conventional logic:

```
P -> (Q -> P)                                   isTrue
((P -> False) -> False) -> P                    isTrue
(P -> (Q -> R)) -> ((P -> Q) -> (P -> R))       isTrue
```

Transcribing the conventional basis of logic to boundary logic:

```
(P) (Q) P                          =  ( )
(((P))) P                          =  ( )
((P) (Q) R) ((P) Q) (P) R          =  ( )
```

The basis of boundary logic is (mathematically) beautiful:

```
( )  A  =  ( )
((A))   =   A
A (A B) =  A (B)
```

Boundary Logic Examples of Proof

To Prove:	*Transcribe*	*Steps*
A implies A	(A) A	( ) A ( )
not not A = A	((A)) = A	A = A
A or A = A	A A = A	A ((A)) = A A (( )) = A A = A
A and B = not (not A or not B))	((A)(B)) = ((A)(B))	identity

```
(and (A implies B) A) implies B     (( (A) ((A) B) ))  B

                  (( (A) ((A) B) ))  B
                     (A) ((A) B)     B          involution
                     (A) ((A)  )     B          pervasion of B
                     (A) (     )     B          pervasion of (A)
                         (     )                dominion
```

A Constructive Proof

SUBSUME A and (A or B) = A

```
( (A) (    A B) )  =  A       transcribe
( (A) ((A) A B) )  =  A       insert (A)
( (A) (( ) A B) )  =  A       extract A
( (A) (( )    ) )  =  A       absorb A B
( (A)          )  =  A       clarify
   A              =  A       identity, qed.
```

Truth Table Example in Boundary Logic

Example: `if (P and Q) then (R iff (not S))`

Transcribe into boundaries:

```
(P  and  Q)        ((P) (Q))
(R iff (not S))    (R (S)) ((R)((S)))  =  (R (S)) ((R) S)
if... then...      (((P) (Q))) (R (S)) ((R) S) = (P) (Q) (R (S)) ((R) S)
```

The expression is True whenever Dominion applies. Erasing variables sets them to False:

When P is False, it is erased: `( ) (Q) (R (S)) ((R) S)  =  ( )` dominion
When Q is False: `(P) ( ) (R (S)) ((R) S)  =  ( )` dominion

Note that the form `(X (Y)) (Y (X))` is True when `X` is not the same as `Y`. Substituting:

```
(P) (Q) ( (( ))) (( ) ( ))  =  (P) (Q)  ( )        =  ( )
```

and when R is True and S is False

```
(P) (Q) (( ) ( )) ( (( )))  =  (P) (Q)        ( )  =  ( )
```

These four cases identify all the True forms of the expression.

Conversely, the expression is False only when everything vanishes, that is, when

(P is True) and (Q is True) and ((R is True, S is free) or (S is True, R is free))

```
 (( ))          (( ))    ( ( ) (( ))) ((( )) ( ))      (( )) (( ))
```

Natural Deduction Example in Boundary Logic

Premise 1:	If A then (if (not P) C)	the Fruit problem
Premise 2:	If C then (if (O or not K) then P)	
Premise 3:	Not (if B then P)	
Conclusion:	Not (A and O)	

Encode the logical connectives as boundaries, and simplify:

P1:	(A) ((P)) C	=	(A) P C	involution
P2:	(C) (O (K)) P			
P3:	((B) P)			
C:	(((A) (O)))	=	(A) (O)	involution

Join all premises and conclusions into one form, using the logical structure:

(P1 and P2 and P3) -> C

The proof structure of "conjunction of premises imply the conclusion" as boundaries:

(((P1) (P2) (P3))) C => (P1) (P2) (P3) C involution

Substituting the forms of the premises and conclusion, and reducing:

((A) P C)	((C) (O (K)) P)	(((B) P))	(A) (O)				
((A) P C)	((C) (O (K)) P)		(B) P		(A) (O)		involution
((A) C)	((C) (O (K)))		(B) P		(A) (O)		pervasion of P
(C)	((C) (O (K)))		(B) P		(A) (O)		pervasion of (A)
(C)	((O (K)))		(B) P		(A) (O)		pervasion of (C)
(C)	O (K)		(B) P		(A) (O)		involution
(C)	O (K)		(B) P		(A) ( )		pervasion of O
					( )		dominion

Interpret the final form: ( ) = True

Boundary Quantification

All x. P(x)	(x) Px	x -> Px isTrue
Exists x. P(x)	((x) (Px))	x and Px isTrue

Quantifier relations:

All x. P(x) iff (not (Exists x. (not P(x))))	(x) Px	= (((x) ((Px))))
All x. (not P(x)) iff (not (Exists x. P(x)))	(x)(Px)	= (((x) (Px)))
(not (All x. P(x))) iff Exists x. (not P(x))	((x) Px)	= ((x) ((Px)))
(not (All x. (not P(x)))) iff Exists x. P(x)	((x)(Px))	= ((x) (Px))

Predicate Calculus and Sets

Predicate Calculus

A **general purpose** language for describing objects, facts, and transformations for particular domains. Also called **First Order Logic**. It consists of

connective logic	{and, or, if, not, iff} inference, proof
object domains	{<unique atoms>}
quantification	{all.x, exists.x}
predicates	classes and properties
relations	True associations between objects
functions	indirect names, maps from one object to another

Predicate logic differs from propositional logic in two ways:

- Ground objects are more complex. Instead of being concepts which are either True of False, objects can consist of discrete elements from infinite sets. Functions are indirect names for particular objects. Relations are propositions about these complex objects.

- Quantification permits direct reference to sets of objects.

Compound Objects

propositions (Boolean variables)

The simplest objects. Has values in the carrier set {0,1}.

properties/domains (set variables)

Simple collections of objects.
Can have *finite* sets (computer science) or *infinite* sets (mathematics).

relations (pairs of objects)

Propositions between objects, relating one object to another.
Compound objects.

functions/procedures (structured pairs of objects)

Alternative names for objects ("2+3" is another name for 5).
Ways to move from one object to another.

Sets

Sets are unordered collections of unique objects.

`S = {x | <statement about x> }` *intensional*, set defined by a common property
`S = {a,b,c,...}` *extensional*, set defined by naming the members

characteristic function[x, S]:
A function which takes the value 1 exactly when `x` is in `S`.

membership: `x inS =def= x=s1 or x=s2 or x=s3 or ...`

empty set: `x not inS`

subset: `if x inS1, then x inS2`

union: `x inS1 or x inS2`

intersection: `x inS1 and x inS2`

difference: `x inS1 and not x inS2`

power set: `the set of all subsets of S`

recursive set membership:

```
x inS =def= not[x = empty-set] and
            x = first[S] or x in rest[S]
```

Set Axioms

Extent: `A = B iff x inA <--> x inB`

Specification: `Exists A. x inA iff x inB and P(x)`

Empty Set: The empty set is a member of all sets.

Specification means that a subset `A` of a set `A` can always be identified by specifying a property that uniquely identifies members of `A`.

Set Equivalence

Two finite sets are equal when they both contain exactly the same members.

```
S1 = S2 iff x inS1 -> x inS2 and x inS2 -> x inS1
```

Logic Plus Sets

Sets identify collections of objects.

Properties, or attributes, identify collections of objects.

The ***extrinsic description*** of a collection is an enumeration, or listing, of the objects in the set.

The ***intrinsic description*** of a collection is the name of the property which uniquely identifies the objects in a set.

```
S  =  {1,3,5,7,9}  =  OddDigit[x]
```

extrinsic *intrinsic*

Monadic Predicate Calculus

Adding properties to propositional logic makes *predicate calculus without relations*.

Monadic refers to operators with one argument only. I.e. Unary functions.

Quine (c. 1950) showed that, for finite sets,

monadic predicate calculus = propositional logic

One method to demonstrate this:

For each x in S, make (x in S) a proposition (i.e. True or False)

```
S  =  {a,b,c}  =  (a in S) & (b in S) & (c in S)
```

Quantification

Quantifiers introduce sets into logic, and serve to define the scope of variables in a logical expression.

Universal quantification: `All x. P(x)`

The statement `All x. P(x)` is True exactly when the predicate `P` (or the characteristic function for the set `P`) is True for all objects in the set `U` for which `x` is an arbitrary member.

For finite domains `U`, `All x. P(x) iff (x1 and x2 and...and xn)`

Existential quantification: `Exists x. P(x)`

The statement `Exists x. P(x)` is True exactly when the predicate `P` is True for at least one object in the set `U` for which `x` is an arbitrary member.

For finite domains `U`, `Exists x. P(x) iff (x1 or x2 or...or xn)`

Relationships between Quantifiers

All true = none false:

```
All x. P(x)  iff  (not (Exists x. (not P(x))))
```

All false = none true:

```
All x. (not P(x))  iff  (not (Exists x. P(x)))
```

Not all true = at least one false:

```
(not (All x. P(x)))  iff  Exists x. (not P(x))
```

Not all false = at least one true:

```
(not (All x. (not P(x))))  iff  Exists x. P(x)
```

Induction and Recursion

Induction is a mathematical proof technique. When this technique is used in programming, it is called **recursion**. The difference is that induction applies to infinite sets and cannot be implemented, while recursion applies to finite sets and can be implemented.
Induction/recursion is the fundamental mechanism for

- extending logical proof techniques into object domains and data structures,
- defining and building mathematical and programming objects,
- decomposing objects so that functions can be applied to the elementary units, and
- robust programming style and program verification.

Most practical computational problems are succinctly expressed in a recursive form (for instance, tree and graph traversal, spatial decomposition, divide-and-conquer algorithms, sorting, searching, and large classes of mathematical functions). As well, recursive function theory defines what can and cannot be computed.

Optimizing compilers usually convert recursion into do-loops. This is because the overhead of making multiple function calls during recursion is greater than a single do-loop. During the 1980s, programmers learned to write loops rather than recursions so that programs would run faster. This type of optimization is *irrelevant* today. It is far more important to enhance code readability, maintenance, and extendibility than to do an obsolete runtime optimization.

Recursive programming is mathematical programming. Once learned, almost all data structures and algorithms become both simpler and more comprehendible using recursion. The essential difficulty is that programmers need to be trained to think recursively.

Recursive Programming

Inductive definitions build up from the base case to any degree of complexity. Recursive programs reduce any degree of complexity one step at a time until the base case is reached. A recursion must be *well-founded*, that is, the steps must eventually terminate at the base. In most cases, the step increment is monotonically decreasing.

Recursive programs can be expressed in two forms, *mathematical* and *accumulating*. The mathematical form accumulates unevaluated operators on the outside and evaluates them after the base is reached. The accumulating form evaluates operators as they accumulate; when the base is reached, the result is returned. Accumulating functions have better performance than mathematical recursions, and easily compile into do-loops.

Mathematical:

```
if (base-case is true) then base-value else F[recursive-step]
```

Accumulating:

```
if (base-case is true) then accum else F[recursive-step, (accum + step)]
```

Mathematical Induction

Induction depends on a order relation over a domain U. The idea is to demonstrate truth for the base case (the simplest member of the ordered set), and then to demonstrate the truth for an arbitrary member of the set, assuming the truth of the member next to it in the order relation.

```
If N is an ordered set and property P isTrue for
      1) the minimal member of N, and
      2) if P(x) then P(next(x))
then P isTrue for all members x of N.
```

Using the natural numbers, N = {1, 2,...}:

```
If P(1) isTrue, and
      assuming P(x) we can show that P(x+1) isTrue, then
P(x) isTrue for all members of N.
```

Some Inductive Definitions

Base case: the value of *the most elementary case*

Examples:

`zero`	the additive identity
`one`	the multiplicative identity
`Phi`	the empty set
`nil`	the empty list, the empty tree
`false`	the logical ground

Generating rule: the transform which *defines the next case*, given an arbitrary case

Examples:

```
successor[n]      =     current[n] + 1
power-of-2[n]     =     2 * current[n]
summation[n]      =     n + current[n]
last[list]        =     rest[list] = nil
length[list]      =     length[rest[list]] + 1
member[x,S]       =     x=select[S] or member[x,rest[S]]
power-set[S]      =     current[S] * S
cardinality[S]    =     cardinality[rest[S]] + 1
node[btree]       =     left[btree] + right[btree]
logic-form[lf]    =     current[lf] implies next[lf]
parenthesis[pf]   =     "(" + current[pf] + ")" or
                              current[pf] + next[pf]
```

Recursion

Recursion is a form of induction in which we demonstrate two truths via computation. For the boolean property P,

```
1) P(minimal-member) isTrue

2)  If P(arbitrary-member) isTrue
      then P(arbitrary-member-one-step-closer-to-minimal-member) isTrue
```

Without algebraic processing, the second step requires an *instance* of the arbitrary member in order to perform computation. For example, say you are trying to find the product of N integers (the factorial function):

```
factorial[n] =def=

            (if n=1 then 1 else (n * factorial[n-1]))
```

This code requires a value of N in order to compute a value for the factorial of N. However, writing and proving the code itself requires mathematical induction. Above, the base case is when N=1. The *recursive invariant*, which is true for all N>1, is:

```
        factorial[n] = (n * factorial[n-1])
```

The recursive invariant is the general definition of the function. It states what remains the same when the value of N changes. When a function is written as a do-loop, the recursive invariant is called the *loop invariant*. Again, it defines what is always true each time through the loop. By verifying that a loop or a recursion maintains its invariant, it is possible to prove that code performs correctly.

The accumulating version of the factorial function is

```
factorial-acc[n, acc] =def=

                  (if n=1 then acc else factorial-acc[n-1, n*acc])
```

Recursive Function Exercises

Write recursive procedures (actual or pseudo-code) for the following functions. Where possible, write both the mathematical and the accumulating versions. Example:

```
+[i,j] =def= (if j=0 then i else (+[i,j-1] + 1))

+[i,j] =def= +acc[j,i]

    +acc[j,acc] =def= (if j=0 then acc else +acc[j-1,acc+1])
```

Integer Domain: `{i,j,k,n}` are positive integers. Note that the definitions below use mathematical induction; a recursive procedure must phrase the ordering relation in a descending form.

```
i+j     =def=  i+0 = i
               i+next[j] = (i+j) + 1

i*j     =def=  i*0 = 0
               i*next[j] = (i*j) + i

i^j     =def=  i^0 = 1
               i^next[j] = (i^j) * i

sum[n] =def=   sum[0] = 0
               sum[i+1] = sum[i] + (i+1)

fac[n] =def=   fac[0] = 1
               fac[i+1] = fac[i] * (i+1)

fib[n] =def=   fib[1] = fib[2] = 1
               fib[i+2] = fib[i+1] + fib[n]
```

Using *mathematical induction*, prove the following for integers:

```
(i*i) = (i^2)

(i*j) + (i*k) = i*(j+k)

(2*sum[n]) = n*(n+1)

(n^2) = (2*sum[n-1]) + n

(3*sum[n^2]) = (2*n + 1) * sum[n]

fib[n^2] = fib[n+1] * fib[n]

sum[n^3] = (sum[n]^2)
```

List Domain: `{x,y}` are elements, `list` is a list. Write these recursive functions:

`last[list]`

returns the last element of `list`.

`length[list]`

returns the length of `list`.

`member[x,list]`

Boolean, returns `T` iff `x` is a member of `list`.

`copy[list]`

returns a copy of `list`.

`alternating-elements[list]`

returns a new list of every other element of `list`.

`reverse[list]`

returns a new list with the elements of `list` in reverse order.

`samelength[list1,list2]`

Boolean, returns `T` iff both lists have the same length
(Do not use any integer arithmetic for this.)

`intersection[list1,list2]`

returns a new list which contains elements in both `list1` and `list2`.

`list-equal[list1,list2]`

Boolean, returns `T` iff `list1=list2` (elements in same order).

`set-equal[list1,list2]`

Boolean, returns `T` iff both `list1` and `list2` have the same elements,
not necessarily in order.

`append[list1,list2]`

returns a new list which is `list1` appended to `list2`.

`substitute[x,y,list]`

returns a new list with `x` substituted for every occurrence of `y` in `list`.

Using *mathematical induction*, prove the following properties about list functions (harder):

```
length[append[list1,list2]] = length[list1] + length[list2]

last[list] = first[reverse[list]]

samelength[list, reverse[list]]

reverse[reverse[list]] = list

substitute[x,y,substitute[y,x,list]] =/= list
```

Tree Domain: `tree` is an arbitrary tree.

`depth[tree]`

returns the maximum depth of `tree`.

`flatten[tree]`

returns a list of all nodes of `tree`

`fringe[tree]`

returns a list of the leaf nodes of `tree`

Function Domain: `f` is a function; `p` is a Boolean function

`andlist[list]`

returns the Boolean `AND` of a list of Boolean elements

`map[f,list]`

returns a new list with the function `f` applied to each element of `list`

`mapchoose[p,list]`

returns a new list with every element from `list` which satisfies test `p`

Challenge: Implement the following recursive functions over integers and trace their execution. What do you observe? (Try `fm` for `i= {341,96,104,336,133}`; try `ackermann` and `takeuchi` for small integers only.)

```
fm[i] =def=

            if i=1 then stop else
                  if even[i] then fm[i/2] else
                        fm[(3i+1)/2]

ackermann[i,j] =def=

            if i=0 then j+1 else
                  if j=0 then ackermann[i-1,j] else
                        ackermann[i,j-1]

ackermann2[i,j] =def=

            if i=0 then j+1 else
                  if j=0 then ackermann[i-1,1] else
                              ackermann[i-1,ackermann[i,j-1]]
```

```
takeuchi[i,j,k] =def=

  if (i =< j) then k else
    takeuchi[takeuchi[i-1,j,k],takeuchi[j-1,k,i],takeuchi[k-1,i,j]]
```

Domain Theories

A **domain** is the collection of simple objects which are of (mathematical) interest. Generally the labels of objects in a domain refer, or point, to concrete objects in reality. A **domain theory** consists of a collection of objects, together with a particular set of functions and relations which define and constrain the generic behavior of both simple and complex objects in the domain.

Domain theories have a specific mathematical form which not only identifies how the objects in that domain behave, but also provides *all the information needed to write processing algorithms* for the domain objects. The prototypical components of a domain theory are

- a **representation** of the elementary unit or constants, the **base** of the structure (also called the *carrier set*)
- **recognizer** predicates which identify the particular types of structure
- a **constructor** function which builds compound structures from simple units
- an **accessor** function which gets parts of a compound structure
- a collection of **functions** which transfer between domain objects
- a collection of **invariants,** or equations, which define the structure's behavior
- an **induction principle** which specifies how to verify correct manipulations

In an algebraic theory, you usually also have **variables**, names which are *generic*, standing in place of an *arbitrary member* of the domain base.

Propositional Calculus as an Example

Propositional calculus consists of a collection of simple objects, called **propositions**, and logical connectives which join propositions into more complex forms. A proposition is an object with a binary value from the set {True,False} and no internal structure. Propositional calculus is the domain theory for propositions. It consists of

a collection of *value labels*	`{T,F}`
a collection of *object labels*	`{a,b,c,...},`
the *logical connectives*	`{and,or,not,if,...},`
the *rules of inference* (deduction)	`{modus ponens,...}`

Recognizer predicates differentiate these components:

`isTrue[X]`	identifies the ground value True
`proposition[X]`	identifies when `X` is a proposition
`compound[X]`	identifies when `X` contains any logical connective.

Constructors tell us how to build compound objects. These are always defined inductively. This definition is called a **generation axiom**.

> Ground values are objects.
> Propositions are objects.
> If `X` and `Y` are objects, then so is `(X -> Y)`.
> There are no other valid objects.

Since all other Boolean functions can be constructed from the basis `{if,F}`, they do not need to be mentioned as part of the definition of compound objects. Rather they can be treated as syntactic variants. For example, define

```
not X       =     X -> F

X or Y      =     (not X) -> Y

X and Y     =     (not ((not X) or (not Y)))

X iff Y     =     (X -> Y) and (Y -> X)
```

Accessors are constructors used in reverse, to decompose compound objects. More importantly, accessors define how value is maintained across compound forms. Value is asserted by the predicate `isTrue`. Accessors is expressed mathematically as a **uniqueness axiom**. Here, "uniqueness" means that a compound form can be uniquely decomposed while maintaining its value. The uniqueness axiom for our only Boolean operator is:

`(X -> Y) isTrue`, when either `X isnotTrue` or `Y isTrue`.

We can substitute the syntactic variants to obtain the other decomposition rules. For example, substitute the definition of `not X` into the uniqueness axiom for implication, giving:

```
not X = X -> F
```

`(X -> F) isTrue` when either `X isnotTrue` or `F isTrue`.

Since `F` is never `True`, this simplifies to:

`(X -> F) isTrue` when `X isnotTrue`.

`(not X) isTrue` when `X isnotTrue`.

Other semantic definitions are derived in a similar fashion:

`(X or Y) isTrue` when either `X isTrue` or `Y isTrue`.

`(X and Y) isTrue` when both `X isTrue` and `Y isTrue`.

`(X iff Y) isTrue` when the value of `X` and the value of `Y` are the same.

Notice that the expression of these relationships between value and structure in English appears to sound rather obvious. This is because the logical connectives are so deeply built into the language that we cannot define them without using them in the definition.

In propositional calculus, there are ***no functions*** which compute properties, since propositions have no properties other than their value (they have no internal structure).

The ***invariants*** which describe and constrain the structure and behavior of compound propositional forms are the **axioms of deduction**. The choice of an axiom set depends on the goals of transformation. One single (algebraic) axiom from which all others can be derived is:

`(A -> B) and (not A -> B) = B`	Single axiom basis

Another set of propositional axioms is that of Boolean algebra. These axioms use three Boolean connectives and are therefore highly redundant:

`X and (Y and Z) = (X and Y) and Z`	Associativity
`X or (Y or Z) = (X or Y) or Z`	Associativity
`X and Y = Y and X`	Commutativity
`X or Y = Y or X`	Commutativity
`X and (Y or Z) = (X and Y) or (X and Z)`	Distributivity
`X or (Y and Z) = (X or Y) and (X or Z)`	Distributivity
`X or F = X`	Zero element
`X and T = X`	One element
`X and (not X) = F`	Complement
`X or (not X) = T`	Complement

Yet another set of axioms are the rules of natural deduction. Again these are highly redundant, the usual list includes about 35 rules. This system is typified by the axiom of *modus ponens*:

`(X and (X -> Y)) -> Y`	Modus Ponens

A fourth example of a possible axiomatic basis is the erasure axioms of boundary logic:

`X or T = T`	Dominion
`not (not X) = X`	Involution
`(X or Y) -> X = Y -> X`	Pervasion

Finally, the **induction principle** for propositional calculus might be loosely stated as:

Base case: `T isTrue`

Inductive case: If an arbitrary form `X` is assumed to be true, `X isTrue`, and if we can demonstrate that `(X -> Y) isTrue`, then for any `Y`, `Y isTrue`.

This of course is just a restatement of *modus ponens*. This induction principle is rather degenerate, since the *logic of induction* itself is expressed using the same logical connectives that define the operations in propositional calculus. The well-founded ordering is not a countable structure like the natural integers, rather it is the potential nesting of implications. When a premise implies a conclusion, then the premise, in a very general sense, is *less-than* the conclusion. This ordering is clearly demonstrated both in the Boolean lattice and in the deep Pervasion rule of boundary logic.

Domains with Internal Structure

To add descriptive complexity, we add internal structure to propositions. There are two general classes of structure: **relations** and **functions**. Relations are connections, or structures, holding together pairs of simple objects. Functions are a restricted type of relation, one that permits functions to stand in place of object names. Functions are relations which name objects in a domain by using other object names.

In computer science, we refer to complex objects as *data structures*, and the set of relational constraints on these objects as *abstract data types*.

The important idea is that all data structures, all domains, have the same organizational structure. All domains and data types consist of a collection of these axiomatic principles:

Labels
Recognizers
Constructors
Accessors
Functions
Invariants (relations)
Induction Principle

In *object-oriented approaches*, the abstract data type includes all algorithmic functionality. That is, using oo-techniques, the above principles define the entirety of an object, and thus the entirety of a program.

Three examples of domain theories follow. These examples are schematic outlines, intended to suggest both mathematical approach and coding technique. Each domain has additional functions and relations which are not included here.

The Domain Theory of Non-negative Integers

Base

```
0
```

Objects

```
{positive integers}
```

Recognizer

```
integer[n]
```

Constructor

```
+1[n], inc[n]
```

Accessor

```
-1[n], dec[n]
```

Decomposition axioms

```
(integer[n] and not[n=0]) -> (+1[-1[n]] = n)
integer[n] -> (-1[+1[n]] = n)
```

Uniqueness axiom

```
(+1[n1] = +1[n2]) iff n1=n2
```

Functions

```
+:    (associative, commutative, identity=0)
    n+0 = n
    n1 + +1[n2] = +1[n1+n2]
    (n1=n2) -> (n1+n3) = (n2+n3)

-:
    n-0 = n
    +1[n1] - +1[n2] = n1-n2

*:    (associative, commutative, identity=1)
    n*0 = 0
    n1*(n2+1) = n1*n2 + n1

^:
    n^0 = 1
    n1^(n2+1) = (n1^n2)*n1
```

Some invariants

```
integer[n] or not[integer[n]]
integer[+1[n]]
integer[0]
not[+1[n] = 0]
integer[n1+n2]
+1[0] = 1
n+1 = +1[n]
n^1 = n
if not[n=0] then 0^n = 0
```

Induction

```
if (F[0] and (F[n] -> F[+1[n]])) then F[n]
```

Decomposition Induction

```
if (F[0] and (F[-1[n]] -> F[n]) then F[n]
```

The Domain Theory of Lists

Note that nested lists are *trees*.

Base

```
[] nil
```

Objects

```
{x,y,z,...} lists
{u,v,...}   atoms
```

Recognizers

```
atom[x]
list[x]
atom-or-list[x]
```

Constructor

```
x·y, insert x into list y
```

Accessors

```
first[x], rest[x]
first[x·y] = x
rest[x·y] = y
```

Decomposition axiom

```
(not[x=nil]) -> (x = (first[x] · rest[x]))
```

Uniqueness axiom

```
((x1·x2) = (y1·y2)) -> (x1=y1 and x2=y2)
```

Functions

```
append:     (associative, identity=nil)
    append[nil,x] = x
    append[u·x,y] = u · append[x,y]

member:
    not[member[x,nil]]
    member[u,v·x] iff u=v or member[u,x]

flat:
    flat[nil] = []
    flat[u·y] = u · flat[y]
    flat[x·y] = append[flat[x],flat[y]]
```

Some invariants

```
not[x·y] = nil
not[atom[x·y]]
(atom[x] and list[x]) -> x=nil
list[append[x,y]]
member[u,append[x,y]] iff member[u,x] or member[u,y]
flat[append[x,y]] = append[flat[x],flat[y]]
```

Induction

```
if (F[nil] and (F[x] -> F[u·x])
           and ((F[y] and F[x]) -> F[y·x]))
      then F[x]
```

This Inductive Principle has three conditions: the base case, the atom case, and the list case. The Induction Principle for lists can also be stated as a decomposition rule:

Decomposition Induction

```
if (F[nil] and
   (when (not[x=nil] and atom[first[x]])
         F[rest[x]] -> F[x]) and
   (when (not[x=nil] and list[head[x]])
         (F[first[x]] and F[rest[x]]) -> F[x]))
  then F[x]
```

The Domain Theory of Sets

Set Base `{} Phi (the empty set)`

Element Base `{a,b,c,...} from some domain`

Objects `{S1,S2,...}` Universe = PowerSet[elements]

Recognizers `atom[a]` `set[S]`

Constructor `a·S, insert atom a into set S`

Accessor

```
member[a,S]
member[choice[S],S]
not[member[choice[S],rest[S]]]
```

Decomposition axiom `(not[S=Phi]) -> (S = (choice[S] · rest[S]))`

Uniqueness axiom `(member[a,b·S] iff (a=b) or member[a,S]`

Functions

Equality:

```
S1=S2 iff (choice[S1]=choice[S2] and
  member[choice[S1]] and member[choice[S2]])
```

Intersection: (associative, commutative, idempotent, identity=Universe)

```
intersect[Phi,S] = Phi
intersect[a·S1,S2] = if member[a,S2]
          then (a·intersect[S1,S2]) else intersect[S1,S2]
```

Symmetric-difference: (associative, commutative, identity=Phi)

```
sym-diff[S1,Phi] = Phi
sym-diff[S1,S2] = if (member[a,S1] and not[member[a,S2]])
            or (member[a,S2] and not[member[a,S1]])
            then member[a,sym-diff[S1,S2]]
```

Cardinality, #:

```
#[Phi] = 0
if not[member[a,S]] then #[a·S] = #[S] + 1
```

Some invariants

```
set[Phi]
set[a·S]
not[member[a,Phi]]
(intersect[{a},{b}] = Phi) -> (not[a=b])
intersect[S,Phi] = Phi
member[a,intersect[S1,S2]] iff
    member[a,S1] and member[a,S2]
S1 intersect (S2 sym-diff S3) =
    (S1 intersect S2) sym-diff (s1 intersect S3)
```

Induction

```
if F[Phi] and if not[member[a,S]] then (F[a] -> F[a·S])
   then F[S]
```

Recursive Definitions

Almost all mathematical structures are defined by induction (recursion). The *composition* of valid compound objects and the *decomposition* of compound objects into elementary components is a recursive process. The definitions of *domain functions* are recursive. Most *proof systems* use induction for verification. Recursion is also necessary to write *elegant programs*.

An inductive definition consists of three components:

- a ***base case***, the simplest possible application of the induction
- an ***inductive case*** which assumes an arbitrary member of the domain, and constructs the adjacent member.
- an ***ordering principle*** which provides a structure for inferring that when one member can be constructed from adjacent member, then all members can be constructed.

Learning to Think Recursively

Recursive programming requires a different way of thinking about and writing computer programs. Procedural thinking was necessary twenty years ago when the art of programming was much closer to the construction of the OS and its hardware infrastructure. In this century, the programmer must *address the problem rather than the mechanism*. John McCarthy, the father of LISP and symbolic programming, says it nicely:

> "In order to write recursive function definitions, one must think about programming differently than is customary when writing programs in languages like FORTRAN or ALGOL [or C] or in machine language. In these languages, one has in mind the state of the computation as represented by the values of certain variables or locations in the memory of the machine, and then one writes statements or machine instructions in order to make the state change in an appropriate way. When writing recursive function definitions one takes a different approach. Namely, one thinks about the value of the function, asks for what values of the arguments the value of the function is immediate [the base case], and, given arbitrary values of the arguments, for what simpler arguments must the function be known in order to give the value of the function for the given arguments [the inductive case]."
>
> -- John McCarthy, LISP Programming and Proving

Induction Principles

When the three components of an inductive definition are combined, they produce an ***Induction Principle*** for the particular domain. Inductive principles are *second order functions, or functionals*. We are most familiar with first-order functions, which vary over domain objects (ie variables). Second-order functions vary over other functions; the domain is a set of functions rather than a set of objects. Sometimes second-order functions are called *functional*

schema, they are patterns, or schema, which specify relations between objects which hold for a *set* of functions. Each schema can be instantiated to many (often infinite) specific induction rules for specific functions.

Different object domains have different induction principles. However, across most domains, the form of the inductive principles is quite similar. Thus we can even consider abstraction of inductive schema over domains. For example,

Domain	*Decomposition*	*Base*	*Inductive Step*
integers	`-1`	`0`	`f[n+1] -> f[n]`
sets	`choose, rest`	`{ }`	`f[choose·rest] -> f[rest]`
logic	`antecedent, consequent`	`F`	`(p->q) -> q`
pairs	`first,second`	`<nil,nil>`	`f[first] -> f[second]`
lists	`first, rest`	`( )`	`f[first·rest] -> f[rest]`
strings	`prefix, rest`	`Empty char`	`f[prefix·rest] -> f[rest]`
binary trees	`left,right`	`root`	`f[left·right]->f[left]&f[right]`
trees	`left, butleft`	`root`	`f[left·butleft]->f[left]&f[butleft]`

Generic Induction Schema

Primitive recursive schema without parameters for the ***integer domain***:

```
f[n] =def=
            if n=0 then k else h[n-1,f[n-1]]
```

A function acting on an integer argument `n` can be generically thought of as follows:

If `n` is `0`, return some constant `k`,
otherwise apply function `h` with the arguments one step closer to `0`
[that is, with arguments `(n-1)` and `f` applied to `(n-1)`].

Example: the factorial function

```
f = fac     k = 1       h = *       h[p,q] = (p+1)*q

fac[n] =def=
            if n=0 then 1 else n*fac[n-1]
```

Primitive recursive schema with parameters for the ***integer domain***
(for simplicity, only one parameter is shown here):

```
f[n,m] =def=
            if n=k then g[m] else h[n-1,m,f[n-1,m]]
```

Example: the times function

```
f = *       k = 1       g[m] = m    h = +       h[p,q,r] = q*(r+1)

*[n,m] =def=
            if n=1 then m else +[m,*[n-1,m]]
```

Primitive recursive schema without parameters for the ***list domain***:

```
f[x] =def=
            if null[x] then k else h[first[x],rest[x],f[rest[x]]]
```

If the list is empty, return constant `k`,
otherwise apply `h` with arguments one step closer to the empty list.

Example: the last function

```
f = last    k = first[x]      h = identity      h[p,q,r] = r

last[x] =def=
            if null[rest[x]] then first[x] else last[rest[x]]
```

Example: the length function

```
f = len     k = 0             h = +1            h[p,q,r] = r

len[x] =def=
            if null[x] then 0 else +1[len[rest[x]]]
```

Example: the reverse function (`*` is the append function)

```
f = reverse       k =()       h = append        h[p,q,r] = r*(p)

reverse[x] =def=
   if null[x] then () else (reverse[rest[x]] * list[first[x]])
```

Primitive recursive schema with parameters for the ***list domain***:
Here, the parameter `y` used in the definition, but not decomposed by the induction.

```
f[x,y] =def=
        if null[x] then g[y] else h[first[x],rest[x],y,f[rest[x],y]]
```

If the list is empty, return `g` applied to parameter `y`
otherwise apply `h` to the decomposed list, to the parameter,
and to `f` applied to the rest of the list.

Example: the member function (`u` is the parameter)

```
f = member  g = False   h = or            h[p,q,r,s] = r=p or s

member[u,x] =def=
            if empty[x] then False
                  else (u=first[x] or member[u,rest[x]])
```

Primitive recursive schema without parameters for the ***tree domain***:

```
f[x] =def=
      if leaf[x] then g[x] else h[first[x],rest[x],f[first[x]],f[rest[x]]]
```

If the tree is an leaf, return `g[x]`
else apply `h` to the decomposed tree and to `f` applied to the decomposed tree.

Example: the size function (number of nodes in the tree, including leaves)

```
f = size    g[x] = 1    h = +       h[p,q,r,s] = r + s

size[x] =def=
            if leaf[x] then 1
                  else (size[first[x]] + size[rest[x]])
```

Primitive recursive schema with parameters for the ***tree domain***:

```
f[x,y] =def=
        if leaf[x] then g[x,y] else
              h[first[x],rest[x],y,f[first[x],f[first[x],first[y]]],
                                   f[rest[x],f[rest[x],rest[y]]]]
```

If the tree is a leaf, return `g[x,y]`
otherwise apply `h` with arguments of the decomposed tree,
(`f` applied to the first of the tree and to `f` applied to the first of each argument `x,y`),
and (`f` applied to the rest of the tree and to `f` applied to the rest of each argument `x,y`)

Example: the tree-equal function

```
f = tree-equal    g = =       h[p,q,r,s,t] =

tree-equal[x,y] =def=
            x=y or (not[leaf[x]] and not[leaf[y]]
                        and tree-equal[first[x],first[y]]
                        and tree-equal[rest[x],rest[y]])
```

Recursive Mathematical Structures

Below isa partial listing of the recursive definitions of some common mathematical structures and their functions. In the following, I have used the token "•" to stand for a generic constructor.

Propositional Logic

Notation:
truth symbols = {T,F}
propositional symbols = {a,b,...}
sentence symbols = {A,B,...}

Sentences:
Truth symbols are sentences.
Propositional symbols are sentences.
If A and B are sentences, so is A->B.

Truth-values:
The truth-values of T and F are respectively True and False.
The truth-value of a is the value assigned to it by the semantics.
The truth-value of A->B is True when either A is False
or when both A and B are True.

Predicate Logic

Notation:
truth symbols = {T,F}
constant symbols = {a,b,...}
variable symbols = {x,y,...}
function sysmbols = {f,g,...}
relation symbols = {p,q,...}

Terms:
Constants and variables are terms.
If {t1,t2,...} are terms, and f has arity n, then f[t1,...,tn] is a term.

Relations:
Truth symbols are relations.
If {t1,t2,...} are terms, and p has arity n, then p[t1,...,tn] is a relation.

Sentences:
Terms are sentences.
Relations are sentences.
If S1 and S2 are sentences, then so is S1->S2.
If S is a sentence, then so is (all x.S)

Well-formed Parentheses (WFP)

() is a WFP
If x and y are WFP, so is (x) and xy

Pairs

If x and y are both atoms, then <x,y> is a pair.

Strings

E is a string. (E is the empty character)
u is a string. (u is a character)
If x is a string, then so is u·x

Binary Strings

E, 0, 1 are each binary strings.
If x is a binary string, so is x·0 and x·1.

Lists

() is a list.
If x is a list, so is u·x (u is an atom)

```
last[()] = ()
last[(u)] = u
last[u·x] = last[x]

length[()] = 0
length [u·x] = length[x] + 1

member[u,()] = F
member[u,v·x] = u=v or member[u,x]

copy[()] = ()
copy[u·x] = u·copy[x]

insert[u,()] = (u)
insert[u,v·x] = if u =< v then u·(v·x) else v·insert[u,x]

alternating-elements[()] = ()
alternating-elements[(u)] = (u)
alternating-elements[u·v·x] = u·alternating-elements[x]

reverse[()] = ()
reverse[u·x] = reverse[x]*(u)       (* is append)

same-length[(),()] = T
same-length[(),v·y] = F
same-length[u·x,()] = F
same-length[u·x,v·y] = same-length[x,y]
```

```
lintersection[(),()] = ()
lintersection[(),v·y] = ()
lintersection[u·x,()] = ()
lintersection[u·x,v·y] =
      if u=v then u·lintersection[x,y] else
            (lintersection[x,v·y] or lintersection[u·x,y]

lequal[(),()] = T
lequal[u·x,()] = F
lequal[(),v·y] = F
lequal[u·x,v·y] = u=v and lequal[x,y]

append[(),y] = y
append[v·x,y] = v·[append[x,y]

substitute[u,v,()] = ()
substitute[u,v,w·x] = if v=w then u·[substitute[u,v,x]]
      else [substitute[u,v,x]
```

Integers

Notation:

```
Successor[n] = n'
```

```
integer[1]
if integer[x] and integer[y] then integer[x+y]
```

Counting:

```
not[x'=0]
1      = 0'
n + 1  = n'
```

Addition:

```
m + 0  = m
m + n' = (m + n)'
```

Multiplication:

```
m * 0  = 0
m * n' = (m * n) + n
```

Exponentiation:

```
m ^ 0  = 1
m ^ n' = (m ^ n) * n
```

```
sum[0] = 0
sum[i'] = sum[i] + i'

fac[0] = 1
fac[i'] = fac[i] * i'

sumfac[0] = 1
sumfac[i'] = fac[i'] + sumfac[i]

fib[1] = fib[2] = 1
fib[i''] = fib[i'] + fib[i]
```

```
power-of-2[0] = 1
power-of-2[n'] = 2*power-of-2[n]

0 = 0
m' = n' iff m = n
```

Trees

Leaves are trees.
If `x` and `y` are trees, then so is `x·y`

```
depth[()] = 0
depth[u·x] = depth[x]
depth[(x)] = 1 + depth[x]

flatten[()] = ()
flatten[u] = (u)
flatten[x·y] = flatten[x]*flatten[y]

fringe[()] = ()
fringe[u] = u
fringe[u·y] = u·fringe[x]
fringe[x·y] = fringe[x]*fringe[y]

tsubstitute[x,y,()] = ()
tsubstitute[x,y,u] = if u=y then x else u
tsubstitute[x,y,z] =
tsubstitute[x,y,first[z]]*tsubstitute[x,y,rest[z]]

treverse[()] = ()
treverse[u·y] = treverse[y]*(u)
treverse[x·y] = treverse[y]*treverse[(x)]
```

Sets

`{}` is a set.
If `S` is a set, then so is `u·S`

```
powerset[{}] = {{}}
powerset[S+{e}] = powerset[S]*(S+e)

member[u,{}] = F
member[u,v·S] iff u=v or member[u,S]

add-to-set[u,{}] = {u}
add-to-set[u,v·S] = if u=v then S else add-to-set[u,S]

{} = {}
u·S1 = v·S2 iff u=v and S1=S2
```

```
union[{},S] = S
union[u·S1,S2] = if member[u,S2] then union[S1,S2]
                        else u·union[S1,S2]

intersection[{},S] = {}
intersection[S,{}] = {}
intersection[u·S1,S2] = if member[u,S2] then u·intersection[S1,S2]
                              else intersection[S1,S2]

subset[{},S2] = T
subset[S1,{}] = F
subset[u·S1,S2] = member[u,S2] and subset[S1,S2]

proper-subset[S1,S2] iff subset[S1,S2] and not[S1=S2]

cardinality[{}] = 0
cardinality[u·S] = if [member[u,S]] then cardinality[S] + 1
                              else cardinality[S]
```

Functions

```
andlist[()] = ()
andlist[u·x] = u and andlist[x]

map[f,()] = f[()]
map[f,u·x] = f[u]·map[f,x]

mapfilter[p,()] = ()
mapfilter[p,u·x] = if p[u] then u·mapfilter[p,x] else mapfilter[p,x]

every[p,()] = F
every[p,u·x] = p[u] and every[p,x]

apply[f,()] = f[()]
apply[f,u·x] =

count-if[p,()] = 0
count-if[p,u] = if p[u] then 1 else 0
count-if[p,u·x] = if p[u] then count-if[p,x] + 1 else count-if[p,x]

delete-if[p,()] = ()
delete-if[p,u·x] = if p[u] then delete-if[p,x] else u·delete-if[p,x]

ordered-by[p,()] = T
ordered-by[p,(u) = T
ordered-by[p,(u v)] = p[u,v]
ordered-by[p,u·v·x] = p[u,v] and ordered-by[p,v·x]

map-if[f,p,()] = if p[] then f[] else ()
map-if[f,p,u·x] = if p[u] then f[u]·map-if[f,p,x] else map-if[f,p,x]
```

Primitive Recursive Functions (PRF)

The successor function is a PRF. `s[i] = next[i]`
The zero function is a PRF. `z[i] = k`
The selector functions are PRFs. `first[i], rest[i]`
If `f` and `g` are PRF, then so is `(f o g)`. `function composition`
If `f` and `h` are PRF, then so is any function which fits a primitive recursion schema,
EG: `(if n=0 then z[n] else h[n-1,f[n-1]])`

Miscellaneous Applications

marking a ruler:

```
rule[left,right,0] = no-op
rule[left,right,height] = let m= (left+right)/2
                          mark[m,height];
                          rule[left,m,height-left]
                          rule[m,right,height-left
```

binary tree traversal

```
traverse[()] = no-op
traverse[leaf[node]] = no-op
traverse[node] = traverse[left[node]];
                 visit[node];
                 traverse[right[node]]
```

depth-first binary tree search

```
depthsearch[u,()] = F
depthsearch[u,v] = u=v
depthsearch[u,v·y] = if u=v then T else depthsearch[u,y]
depthsearch[u,x·y] = depthsearch[u,x] or depthsearch[u,y]
```

insertion sort

```
insertion-sort[()] = ()
insertion-sort[u·x] = insert[u, sort[x]]
```

symbolic differentiation (all in reference to `dx`)

```
dc = 0
dx = 1
d(-U) = -dU
d(U+V) = dU + dV
d(c*U) = c* dU
d(U*V) = u*dV + V*dU
d(U/V) = d(U*(V^-1))
d[U^c] = c*(U^(c-1))*dU
d(ln U) = (U^-1)*dU
```

Final Project

HAND IN AT THE BEGINNING OF CLASS.

Make of a map of the territory of discrete mathematics.

- Include each of the topics we have covered in class, and each topic in the text.

- Include the defining characteristics of each separate topic: the domain, the axioms, the essential idea.

- Pay particular attention to the relationships between topics.

- Order your map so that it clearly displays which topics are subsumed by (or are subsets of, or "inherit the characteristics of") other topics.

- Distinguish between old and new topics, between topics that are well understood by the mathematical community and those that are still evolving rapidly.

- Next, assign to each topic and to each collection of defining characteristics, three rating values:

    1. *your understanding* of the topic or characteristic
       (0= no understanding at all, 10 = total and complete understanding)

    2. *your confidence* in the above understanding rating
       (0 = no confidence, 10 = complete confidence)

    3. *the importance* of the topic to you
       (0 = completely irrelevant, 10 = completely relevant and important)

- You may want to extend your map with topics that you consider to be part of discrete math, but were not covered in class.

- *ONE PAGE* only please.

Relations

Relations

A *relation* is an ordered set of tuples. A *binary relation* is a set of ordered pairs.

Domain:	the set of first elements in the ordered pair
Range:	the set of second elements in the ordered pair
Cartesian Product:	the set of all possible ordered pairs (domain X range)
Empty Relation:	the set of no ordered pairs
Inverse:	the relation formed by exchanging the range and the domain
Relation on a Set:	the domain and the range are the same set

Identity Relation:	`(x,x)	x inR`
Equivalence of ordered pairs:	`(a,b) = (c,d)  iff  a = c  and  b = d`

Relations as Graphs

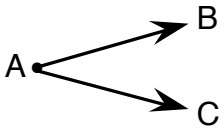
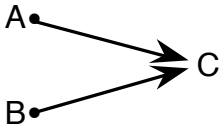
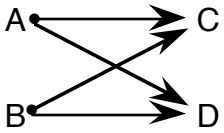
Relations are a way of pointing from one set to another. Relations establish directional pointers between elements of the *domain* and elements of the *range*. Thus, every relation is isomorphic to a directed graph with elements as nodes and concrete relations as arcs.

Types of Existence

Some elements in the domain do not map onto a element in the range.
Some elements in the range do not correspond to an element in the domain.

Types of Connectivity

Relations support any type of connectivity
between elements in the domain and those in the range.

GRAPH	*NAME*	*Example*
	*one-to-one*	`{(a,1),(b,2),(c,3)}`
	*one-to-many*	`{(a,1),(a,2),(b,3)}`
	*many-to-one*	`{(a,1),(b,1),(c,2)}`
	*many-to-many*	`{(a,1),(a,2),(b,1),(c,2)}`

Types of Relational Structure

Relations on a Set

reflexive	`all x | (x,x) inR`
symmetric	`if (x,y) inR, then (y,x) inR`
transitive	`if (x,y) inR and (y,z) inR, then (x,z) inR`
antisymmetric	`if (x,y) inR and (y,x) inR, then x=y`
trichotomy	`(x,y) inR xor (y,x) inR  xor x=y`
irreflexive	`not reflexive`
asymmetric	`not symmetric`

Functions

identity	`Id op A = A op Id = A`
inverse	`A op iA = iA op A = Id`
associative	`(A op B) op C = A op (B op C)`
commutative	`A op B = B op A`
distributive	`A op1 (B op2 C) = (A op1 B) op2 (A op1 C)`
idempotent	`A op A = A`

Equivalences

An *equivalence set* is a relation which is

reflexive	`xRx`
symmetric	`xRy -> yRx`
transitive	`xRy and yRz -> xRz`

Partitions

A *partition* of a set (or a relation) is a collection of disjoint subsets of the set. The union of partitions is the entire set.

The *equivalence relation* determines a partition, and each partition of a set defines an equivalence relation.

Orderings

A *partial order* is a relation which is

reflexive	`xRx`
antisymmetric	`xRy and yRx -> x=y`
transitive	`xRy and yRz -> xRz`

A *total order* is a relation which is

trichotomous	`xRy xor x=y xor yRx`
transitive	`xRy and yRz -> xRz`

Cartesian Product

A relation is between two sets. The Cartesian Product of two sets is the set of ordered pairs consisting of all possible combinations of elements from each set. *Example:*

```
S1 = {1,2,3}            S2 = {a,b}

S1xS2 = {(1,a),(2,a),(3,a),(1,b),(2,b),(3,b)}
```

The set of *all possible relations* between S1 and S2 is defined by all the possible combinations of the product elements. In the example above, there are six product pairs, so the total number of possible relations is

6 things taken	0	at a time	=	1	
"	1	"	=	6	
"	2	"	=	15	
"	3	"	=	20	
"	4	"	=	15	
"	5	"	=	6	
"	6	"	=	1	
Total number of relations			=	64	= 2^6

There are two ways of analyzing the structure of relations illustrated above.

1. Using the coefficients of the binomial expansion

```
sum[i choose n] for i=0..n
```

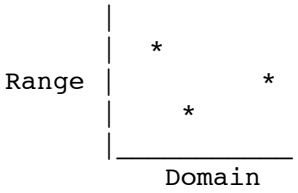
2. Using the power set of the relational pairs

```
2^n
```

Ways to View Relations

I. A set of *ordered pairs*: {(a,b),(a,c),(b,d)…}

II. A *set of points* matching elements in the Domain set with elements in the Range set.



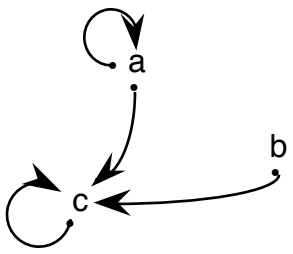
III. A *lookup table* between two sets

R	a	b	c
a	x		x
b		x	
c			x

IV. A *matrix*:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

V. A *connection graph*:



VI. A *relational database*:

R	a	a
R	a	c
R	b	b
R	c	c

VII. A *link table* (a possibility database):

A	B	C	n
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Relations on a Set

When both the Domain and the Range of a relation are the same Set, the relation is *on a set*.

The lookup table and matrix representations can contain *common patterns*, which define the concepts associated with relations.

REFLEXIVE

_R_	_a_	_b_	_c_
a	x		
b		x	
c			x

SYMMETRIC

_R_	_a_	_b_	_c_
a		x	
b	x		
c			x

TRANSITIVE

_R_	_a_	_b_	_c_
a		x	
b			x
c	x		

ANTISYMMETRIC

_R_	_a_	_b_	_c_
a		x	
b	x		
c			

iff a = b

Composition of Relations

Relational composition is very similar to functional composition.

(R o S) =def= all pairs (x,z) exists y | (x,y) inS and (y,z) inR

Note that the range of S is a subset of the domain of R

```
(R o S)(A) = R(S(A))
```

associative: `(R o S) o T = R o (S o T)`

not commutative: `R o S =/= S o R`

inverse of a composition: `(R o S)^-1 = S^-1 o R^-1`

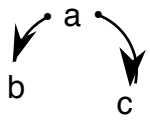
Transitive Closure

A relation is *transitive* when it is possible to follow relations for one pair to another in a cycle.

The transitive closure is computed by multiplying the matrix of a relation by itself N times. A matrix will return to its original configuration after N multiplies if there is transitive path of N steps between the elements.

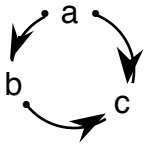
Example: non-transitive, no path between `b` and `c`

```
  a b c
a 0 1 1     0 1 1     0 0 0
b 0 0 0  *  0 0 0  =  0 0 0      no paths
c 0 0 0     0 0 0     0 0 0
```



Example: non-transitive, no path to `a`

```
                A^1       A^2                 A^3
  a b c
a 0 1 1     0 1 1     0 0 1       0 1 1     0 0 0
b 0 0 1  *  0 0 1  =  0 0 0    *  0 0 1  =  0 0 0
c 0 0 0     0 0 0     0 0 0       0 0 0     0 0 0
```

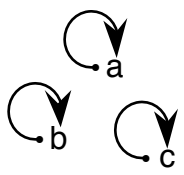


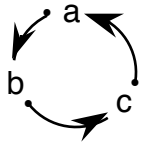
Example: transitive with cycle length 3

```
                A^1       A^2                 A^3                 A^4
  a b c
a 0 1 0     0 1 0     0 0 1       0 1 0     1 0 0       0 1 0     0 1 0
b 0 0 1  *  0 0 1  =  1 0 0    *  0 0 1  =  0 1 0    *  0 0 1  =  0 0 1
c 1 0 0     1 0 0     0 1 0       1 0 0     0 0 1       1 0 0     1 0 0
```

Example: identity, degenerate transitive with cycle length 1

```
                A^1       A^2
  a b c
a 1 0 0     1 0 0     1 0 0
b 0 1 0  *  0 1 0  =  0 1 0
c 0 0 1     0 0 1     0 0 1
```





Relations or Functions?

Functions are a subset of relations. Using functions provides the advantages that functions place on relational structures: *existence* and *uniqueness*. However, since functions are less general, these same constraints make generalization impossible. Thus functional encoding is often brittle and difficult to modify. For example:

Domain D = students `{s1,s2,s3}`

Range R = chairs `{c1,c2,c3,c4,c5}`

When every student sits in a chair, the uniqueness (no student sits in two chairs) and existence (a chair for every student) constraints on a function are met.

F[students] = `{(s1,c1),(s2,c2),(s3,c5)}`

Note that it is still permitted for two students to sit in one chair.

If the situation changes, the functional constraints may be violated. For example, if one student lounges across two chairs, uniqueness is violated. If some student stands rather than sits in a chair, then existence is violated.

Most generally, whenever a specification may change (almost always the case), more general data structures achieve better code modularity, portability, and maintenance.

Many concepts do not permit a functional approach. For example, consider:

Proof-of[theorem] = <proof-sequence>

Domain = logical formulas as theorems
Range = proof-sequences

The above assertion has *relational semantics*. There is no assurance that a proof does exist for any formula (existence violated), and certainly there are many different proof sequences for any formula (uniqueness violated). In contrast:

Theorem-of[proof-sequence] = <theorem>

Domain = proof-sequences
Range = theorems as formulas

This assertion is (or can be) functional, since any proof leads to a unique theorem, and every proof-sequence leads to some theorem. The assertion, of course, can also be seen as relational.

Relational Examples and Exercises

Patterns

A relation is a pattern. Naturally, we have a free choice for the syntax of the relation, given a consistent notation. Some structural alternatives include:

```
aRb
R[a,b]
(R a b)
(a R b)
```

The advantage of seeing relations as patterns is that we can then use a pattern-matching engine for abstract manipulation of patterns, that is, for computation. Pattern-matching is algebraic, permitting us to compute with unbound variables, i.e. with abstractions.

Let the `?` operator identify unbound variables, such that `?x` represents an arbitrary member of a domain (or range). Now we can express relational patterns directly in the pattern language:

reflexive	`(?x R ?x)`
symmetric	`(?x R ?y) -> (?y R ?x)`
transitive	`(?x R ?y) & (?y R ?z) -> (?x R ?z)`
antisymmetric	`(?x R ?y) & (?y R ?x) -> x=y`
trichotomy	`(?x R ?y) xor (?y R ?x) xor x=y`
irreflexive	`not (?x R ?x)`
asymmetric	`not ((?x R ?y) -> (?y R ?x))`

Binary Arithmetic Example

Relations without variables are called *facts*. The facts of binary arithmetic are:

```
(0 + 0 = 0)
(0 + 1 = 1)
(1 + 0 = 1)
(1 + 1 = 0)

(0 * 0 = 0)
(0 * 1 = 0)
(1 * 0 = 0)
(1 * 1 = 1)
```

{+,*} are functions which map pairs onto singles. This relational database has one relation, {=}. Facts are simply those relations which we assert to match our model (i.e. we consider them to be True). They define the semantics of the relation =. Specifically, the following assertions are not in the database, each is not True:

```
(0 + 0 = 1)
(0 + 1 = 0)
(1 + 0 = 0)
(1 + 1 = 1)

(0 * 0 = 1)
(0 * 1 = 1)
(1 * 0 = 1)
(1 * 1 = 0)
```

For each operator in {+,*}, the Cartesian product has eight members, four of which we distinguish as valid. The eight possible relations come from two possible values in each of three different places:

```
(A + B = C)              A,B,C = {0,1}
```

More generally,

```
(A op B R C)             op = {+,*}        R = {=}
```

A *closed world* is one in which we know that if a fact is asserted, then its negation is also asserted to be False. Binary arithmetic is a closed world. An *open world* is one in which if we know a fact, we do not necessarily know its negation. For example, if we know `(Mary isa-student)` and `(John isa-student)`, we cannot also assert `(No-one-else isa-student)`.

Pattern Generators for Binary Arithmetic

The transformational rules which define arithmetic functions can be expressed as relational patterns:

```
(?x + ?y = ?z) -> (?y + ?x = ?z)

(?x + ?y = ?z) & (?x = ?y) -> (?z = 0)

(?x + ?y = ?z) & not(?x = ?y) -> (?z = 1)
```

Pattern abstraction is available for operators as well:

```
(?x ?op ?y = ?z) -> (?y ?op ?x = ?z)
```

The above pattern generator defines a symmetry structure for both operators in our world of simple arithmetic. The operators (i.e. functions) differ by their pattern definition. Compare the constraints for + above with those for * below:

```
(?x * ?y = ?z) & (?x = 1) & (?y = 1) -> (?z = 1)

(?x * ?y = ?z) & not(?x = 1) & not(?y = 1) -> (?z = 0)
```

Pattern generators provide an alternative way to record facts into a relational database.

Option 1: explicit listing

```
(1 + 0 = 1)
(0 + 1 = 1)
```

Option 2: generator

```
(1 + 0 = 1)
(?x + ?y = ?z) -> (?y + ?x = ?z)
```

For large databases, pattern generators exchange time for space. For example, if the arithmetic example were expressed in decimal notation, we would need to assert 100 facts for each operator, out of a Cartesian product of 1000 possible facts. A symmetry rule would reduce the number of facts by half (50 here), at the cost of generating them dynamically when needed.

Using Pattern Generators for Computation

The following pattern-matching computational regime is how logic engines and theorem provers (e.g. Prolog) manage computation:

```
(0 + 0 = ?x)
```
matches
```
(0 + 0 = 0)
```
by binding `?x=0`.

Thus we have computed this Boolean sum.

Since relational structures are non-directional, we can compute inverses as well as answer more exotic queries:

```
(1 * ?x = 0)
```
matches
```
(1 * 0 = 0)
```
by binding `?x=0`.

```
(1 ?op 1 = 1)
```
matches
```
(1 * 1 = 1)
```
by binding `?op=*`

```
(1 ?op 1 = ?x)
```
matches
```
(1 * 1 = 1)
```
by binding `?op=*` and `?x=1`
```
(1 + 1 = 0)
```
by binding `?op=+` and `?x=0`

Algorithm Calling Example

Let `S = {a,b,...n}` be a set of algorithms. Define the relation `R` on `S` as

```
(a R b) iff (a CALLS b)
```

Suppose we have the following code structure:

```
(a R b)
(a R c)
(b R c)
(c R e)
(d R c)
(e R b)
```

The transitive closure of `R` identifies all algorithms which may eventually call another algorithm. Make a listing of this transitive closure. Express the closure as a pattern rule rather than an exhaustive list.

An algorithm is recursive if it calls itself, that is, if `(a R a)` is asserted. Are any of the above five algorithms `{a,b,c,d,e}` recursive?

Graph Example

Here is an airline database of costs for flying between cities

```
(San Diego to LA       $100)
(San Diego to SF       $150)
(San Diego to Portland $200)
(San Diego to Seattle  $300)
(LA to SF              $100)
(LA to Portland        $150)
(LA to Seattle         $300)
(SF to Seattle         $200)
(Portland to Seattle   $150)
```

Assume this relation is symmetric for all flights within California.

- Is it reflexive?
- Is it transitive?
- Are all combinations of flights for less than $260 transitive?
- What is the type of connectivity (1-to-1, 1-to-many, many-to-1, many-to-many) within each State?
- If the relation were antisymmetric instead of symmetric, where could you travel to?
- Which flights (could be more than one stop) are in the same cost equivalence class?
- What is the ordering of flights by cost?
- What is the transitive closure of flights?

Define a new relation `S` over the above airports to mean `in-the-same-state`. Show that S is an equivalence relation.

Text Example

Consider the relations `left-prefix` and `alphabetical-order`, for the set of alphabetic characters `{blank,a,b,...,z}`.

Build separate relational databases for each of these words:

```
dot
level
pepper
bookkeeper
```

Characterize the relational structure of each database.

Relational Database Example

Here is a relational database of students:

ID	Name	Age	Enrolled-in
29	Bob	32	512
48	John	29	514
89	Sally	28	512
59	Ignatz	34	512
30	Joan	27	510
58	Tom	24	510
45	Jane	24	514
22	Keri	28	510
97	Sam	27	512

The SELECT operation chooses row entries based on a pattern.

The PROJECT operation chooses column entries based on a pattern.

The JOIN operation creates a new entry by combining two or more relations.

Construct queries which gather the following information:

All students taking 512.
The ages of all students.
All classes with more than two students.
All students older than Joan.
Id numbers of students in each class.
Id numbers of twenty-something students in 512.

Relational Algebra

A *relation* is a set of tuples. In CS, this is called a database or file. The items of the relation can be seen as **attribute value pairs**, with the values being atomic ground forms (ie not composite terms and not pointers). Other terms:

Domain:	the set of possible values for a relation
Relation:	a subset of the cartesian product of domains
Attribute:	column of a relation
Item:	row of a relation, a tuple
Key:	a minimal set of attributes which identify a unique tuple

Since a relational database contains the same information as a relation table, the database must:

1. have no duplicates
2. have values from the same domain
3. have simple attribute structures (not composite)
4. an attribute must be accessed by a single key

It is always possible to express multiple argument relations using only binary relations. E.g.:

```
PERSON[name, age, sex]  =  PERSON[name, age] and PERSON[name, sex]
```

Operators in a Relational Algebra:

Selection: Reduce the number of rows in a table (horizontal cut)

Projection: Reduce the number of columns (vertical cut), and remove duplicates.

Restriction: Make a relation consisting of all rows which meet a functional test

Union: Combine rows of two tables (same attributes). Same as file-merge.
aka: Or, Append

Difference: The rows of relation 1 with duplicates in relation2 removed
aka: -, remove, minus

Join: Make a table with the items common to both relations.
aka: intersection

Generalized Join: For relations with unequal number of attributes, carry along the extra attributes in the new table. If the relations have no common attributes, form the cartesian product.

Hierarchy is formed in two ways:

1. *generalization* of subtypes (standard oo inheritance)
2. *aggregation* of components, making new relations of existing fields

Relational Knowledge-base Example

Vocabulary:

```
(father X Y)
(mother X Y)
(male Y)
(female Y)
(parent X Y)
(sibling X Y)
(brother X Y)
(sister X Y)
(uncle X Y)
(aunt X Y)
(gfather X Y)
(gmother X Y)
(ancestor X Y)
(cousin X Y)
```

Knowledge Base:

```
(if (father A B) (parent A B))
(if (mother A B) (parent A B))
(if (and (parent A C) (parent A B) (not (= B C))) (sibling B C))
(if (and (sibling A B) (male A)) (brother A B))
(if (and (sibling A B) (female A)) (sister A B))
(if (and (parent B C) (brother A B)) (uncle  A C))
(if (and (parent B C) (sister A B)) (aunt  A C))
(if (and (parent B C) (father A B)) (gfather  A C))
(if (and (parent B C) (mother A B)) (gmother  A C))
(if (parent A B) (ancestor A B))
(if (and (parent A B) (ancestor B C)) (ancestor  A C))
(if (and (parent A C) (parent B D) (sibling A B)) (cousin C D))
(if (father A B) (male A))
(if (mother A B) (female A))
```

Facts:

```
(father arthur bertram)
(father arthur bailey)
(father bertram cornish)
(father bertram carey)
(mother beatrice cornish)
(mother beatrice carey)
(father bailey carleton)
(father bailey cassandra)
(mother bessie carleton)
(mother bessie cassandra)
(male cornish)
(male carey)
(male carleton)
(female cassandra)
```

Example questions:

```
(gfather arthur ?)
(cousin ? cassandra)
```

Functions

Ordered Pairs

We have seen elements, `a`, and sets of elements `{a,b}`. Adding an ordering relation creates a lattice of ordered functions. Each function is specified by a collection of ordered pairs, `(a,b)`.

Example:

> The logical function `(if a then b)` is defined by a collection of three ordered pairs of the form `(a,b)`, where the values of `a,b` are in the set `{0,1}`:
>
> ```
> if a then b =def= {(0,0),(0,1),(1,1)}
> ```
>
> The sixteen different ways of collecting the four possible ordered pairs, `N` at a time, `N=0..4`, define the sixteen different Boolean functions of two variables.

Functions and Relations

relation: `xRy isTrue` *function:* `f(x)=y isTrue`

> The set of all first values of a set of ordered pairs is called the ***Domain***.
>
> The set of all second values of a set of ordered pairs is called the ***Range***.

A ***relation*** is a collection of ordered pairs over two sets, the domain set and the range set.

A ***function*** is a relation `(x,f(x))`, such that

1. Every member of the domain is associated with a member of the range, and
2. No element in the domain is associated with more than one element in the range.

Perspectives on Functions

1. *Formal constraints on a relation*

   existence: `all x inDomain . exists y inRange`

   uniqueness: `all pairs (x,f(x)) . if x1=x2 then f(x1)=f(x2)`

2. *Graph*

   Domain on x-axis, Range on y-axis
   uniqueness permits the graph to cross any vertical line (i.e. x-value) *only once*.

3. *Lookup table*

x	f(x)
1	1
2	4
3	9

4. *Static relation between variables*

`x = y + 5` "=" is an equivalence relation

5. *Dynamic relation between variables*

`f(x) = y` `x` is the independent variable (controlled measurement)
`y` is the dependent variable (observed measurement)

6. *Pure operation*

```
(lambda (#)  #^2 + # + 1)
```

`#` is the formal parameter of the function which binds to any value

7. *Sequence of combinators*

```
fac = (Y) lf.ln.(((0)n)1) ((*)n) (f) (-1) n
```

A tree of substitution instructions

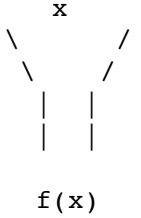
8. *Rule of correspondence/algorithm*

take a number	`x`
double it	`2*x`
add 3	`2*x + 3`

9. *Set transformation*

```
Domain                Range
  a   ----------->    b
  b   ----------->    c
  c   ----------->    d
  d   ----------->    d
```

10. *Input-output machine*



11. *Way of finding and assigning names to unnamed objects*

    `2^100` is the short name of a large number

12. *Digraph*

```
(1) ---> (3) ---> (5)
```

Types of Functions

Surjective, Onto, Epic	all `y` inRange, exists `x` inDomain . `f(x) = y`
Injective, 1-to-1, Monic	`if f(x1) = f(x2) then x1 = x2`
Bijective	1-to-1 and Onto

Bijective functions have an **inverse**, since every element in both the Domain and the Range are in correspondence:

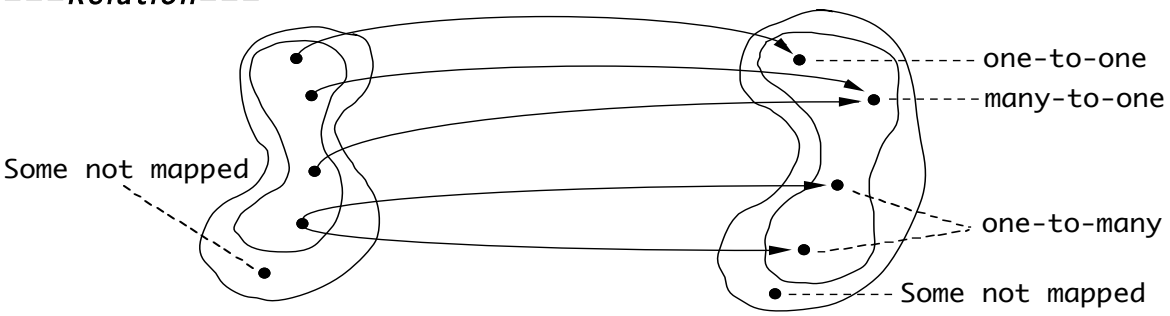
two-way existence	`all x inD, exists y inR . f(x) = y` `all y inR, exists x inD . f(x) = y`
two-way uniqueness	`all (x,f(x)) . x1 = x2 iff f(x1) = f(x2)`
inverse:	Exists `f-inverse` iff `f` is onto and one-to-one

Special Functions

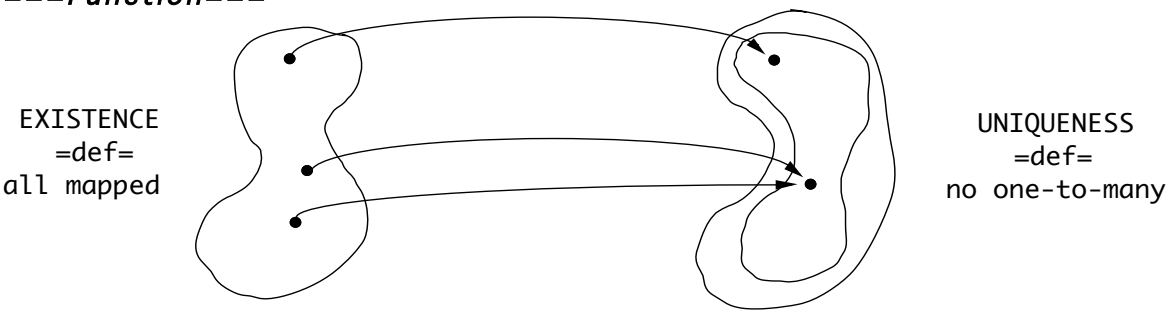
Identity	`f(x) = x`
Characteristic	`f(x) = 1 if x inA` `= 0 if x not inA`
Permutations	`(1,2,3) <--> (3,1,2) <--> (2,3,1)`
Sequences	`1 .. n <--> 1/1 .. 1/n`

Mappings

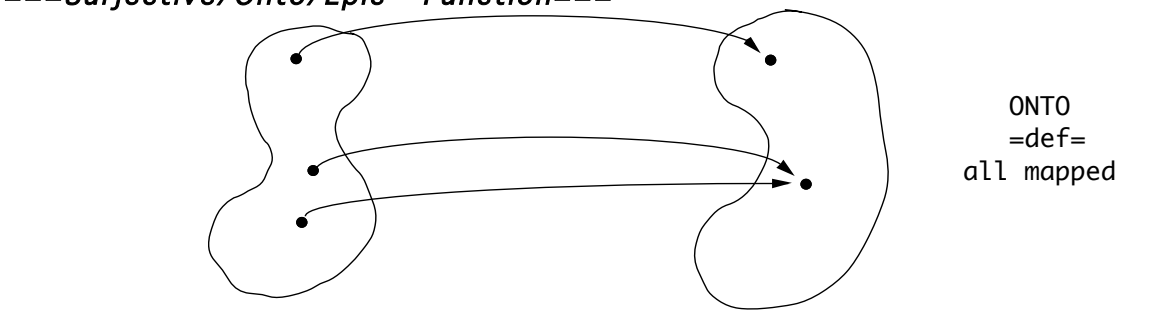
===Relation===



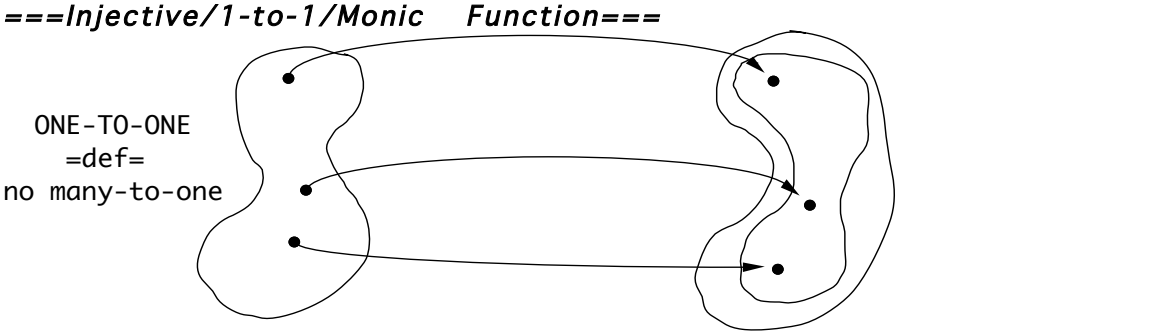
===Function===



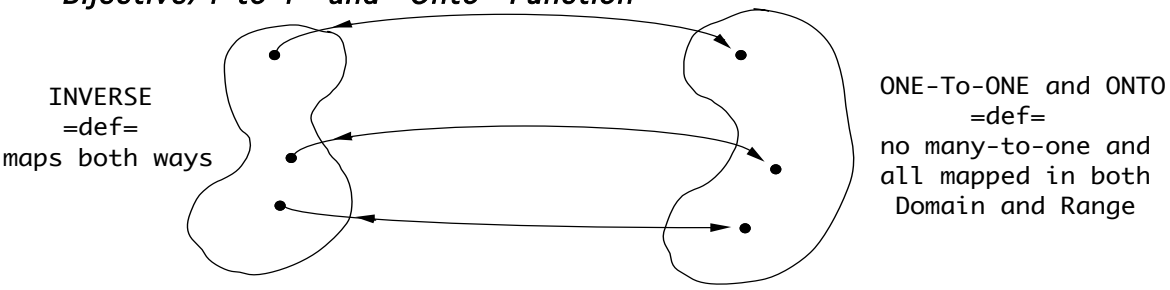
===Surjective/Onto/Epic Function===



===Injective/1-to-1/Monic Function===



===Bijective/1-to-1 and Onto Function===



Function Composition

`(f o g)` = All pairs `(x,z)` Exists `y` such that `(x,y)` in `g` and `(y,z)` in `f`
Note that the Range of g is a subset of the Domain of f

`(f o g)(x) = f( g(x) )`

Associative: `(f o g) o h = f o (g o h)`

Not commutative: `f o g =/= g o f`

Maintains the type of the function:

if `f` and `g` are functions, then `(f o g)` is a function
if `f` and `g` are onto, then `(f o g)` is onto
if `f` and `g` are one-to-one, then `(f o g)` is one-to-one

Composition of a function with its inverse:

`f o f-inverse = identity I` on Range of `f`
`f-inverse o f = identity I` on Domain of `f`

Inverse of a composition: `(f o g)-inverse = g-inverse o f-inverse`

Binary Functions

Binary functions are a mapping of ordered pairs onto elements: `((a,b) c)`
e.g.: `a + b = c` `+ = {((a,b),c) such that (a,b) in S X S and c inS}`

The domain consists of ordered pairs rather than single elements.

If `a`,`b`, and `c` are in the Domain,
then the Domain is closed with regard to the function:

`All x1,x2 inD such that f(x1,x2) inD`

Algebraic Systems

Formal Modeling (refrain)

Formal = Atoms + Formations + Transformations + Axioms

A ***formal system*** (a mathematical system) consists of

1. several sets of labels (for objects, functions, relations) called constants,
2. rules for building compound sentences (or equations or expressions), and
3. rules for evaluating and simplifying compound expressions.
4. some axioms or assumptions which assert equivalence sets

A ***calculus*** is a formal transformation system with variables.

Mathematical Data Structures

truth values	`0,1`	arithmetic of logic
propositions	`a,b,c`	algebra of logic
sets	`{},{a},{a,b}`	set theory
ordered pairs	`(a,b),(a,c)`	functions, relations
nested pairs	`((a,b),c),((a,c),d)`	binary functions and relations
nested pairs	`(a,(a,b)),(b,(b,c))`	graphs

Morphism Functions

A ***function*** is a constrained relation between two sets, the Domain and the Range.

An ***algebraic system*** is a Set (the Domain of a function) and at least one binary function on that Set: `(S,f)` where `S` is the Domain, and `f` is a binary function.

A ***homomorphism*** is a special type of function which maps one algebraic system onto another. Given a system `(S,f)` and a system `(T,g)`, the homomorphic function `h` is:

```
All s1, s2 inS .  h(f(s1,s2)) = g(h(s1),h(s2))
```

The morphism function **h** preserves the structure of the two systems. When it exists, we know that the two systems are in some way *functionally identical*. Isomorphic systems are algebraically indistinguishable.

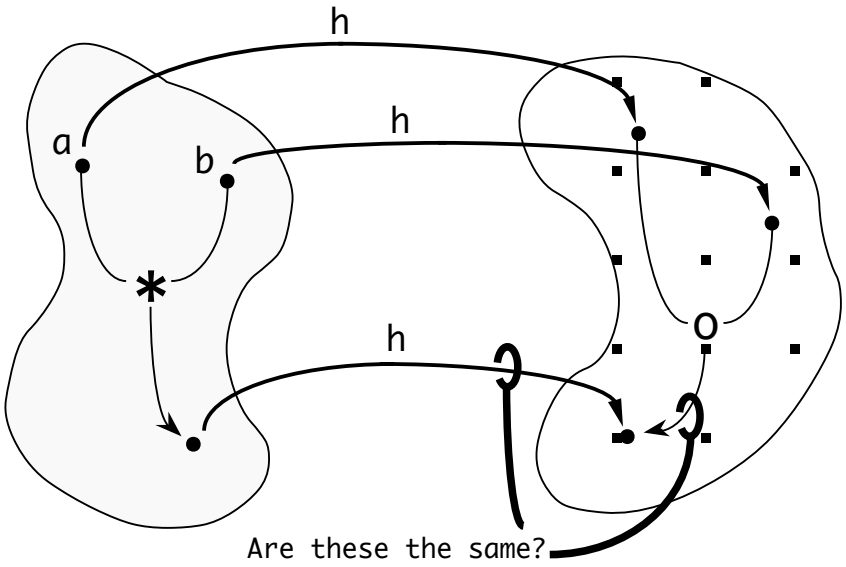
Other types of morphism functions preserve other types of functional structure.

Epimorphic: h preserves the onto characteristic.

Monomorphic: h preserves the one-to-one characteristic

Isomorphic: h preserves one-to-one correspondence

Morphism Diagram



Examples of Morphic Systems

$$h[a*b] = h[a] \bullet h[b]$$

Affine

System 1: (integers,+)
System 2: (integers,+)

Morphism h[x] = 2x

Proof: h[a+b] = 2(a+b) = 2a + 2b = h[a] + h[b]

Logarithm

System 1: (integers,+)
System 2: (reals,*)

Morphism h[x] = e^x

Proof: h[a+b] = e^(a+b) = e^a * e^b = h[a] * h[b]

Signs in Multiplication

System 1: (integers,+)
System 2: ({1,-1},*)

Morphism
```
h[x] =  1 if x is even
     = -1 if x is odd
```

Proof:
```
case a,b even:    h[a+b] =  1    h[a]*h[b] = even*even = 1
case a,b odd:     h[a+b] =  1    h[a]*h[b] = odd*odd = even = 1
case a,b differ:  h[a+b] = -1    h[a]*h[b] = odd*even = odd = -1
```

Group Theory

Algebraic systems (`(S,f)`, where `S` is a set and `f` is a binary function on that set) can be classified into groups having similar structural characteristics. This additional level of abstraction is called **group theory**, or modern algebra.

The essential distinguishing characteristics of algebraic systems `(S,f)`:

Let `a,b,c inS` and `e`, the identity element, `inS`

Closed binary operation:	`f(a,b) = c`
Associativity:	`f(f(a,b),c) = f(a,f(b,c))`
Identity element:	`Exists e inS. f(e,a) = f(a,e) = a`
Inverse element:	`Exists y inS. f(a,y) = f(y,a) = e`
Commutativity:	`f(a,b) = f(b,a)`

Types of Algebraic Systems

Groupoid: `(S,f)` such that `S =/= { }`

Loop: Groupoid and

```
All a,b,c in S. if f(a,b) = f(a,c) then b=c
                if f(a,c) = f(b,c) then a=b
```

Semigroup: Groupoid and
`S` is closed under `f`
`f` is associative on `S`

Monoid: Semigroup and
`(S,f)` has an identity element

Group: Monoid and
every element in `S` has an inverse.

Each type can be combined with the commutative property, to give

commutative loop
commutative groupoid
commutative semigroup
commutative monoid
commutative group (boolean algebra is an example in this category)

Boolean Algebra

Boolean algebra is an algebraic system, {K,A,V,'} consisting of

K	a set of elements
A	the *meet* operation
V	the *join* operation
'	the *complement* operation

Boolean Algebra Axioms

Let • be either AND or OR:

associative	a·(b·c) = (a·b)·c
commutative	a·b = b·a
distributive	a·(b*c) = (a·b)*(a·c)
zero element	a V 0 = a
one element	a A 1 = a
complement	a V a' = 0 a A a' = 1

Boolean Algebra Morphisms

Domain	*meet*	*join*	*complement*	*zero*	*one*	*less-than*
Boolean algebra	meet	join	complement	0	1	<
algebra of sets	union	intersection	complement	Phi	Universe	subset
switching circuits	series	parallel	opposite	open	closed	if-then
propositional logic	and	or	not	false	true	if-then
integer divisors	gcd	lcm	largest/x	1	largest	divides

Exotic Number Systems

History of Integers

Number systems evolve in abstraction and expressibility.

Assyrian tokens `IIIIII + IIIIIIIIII = IIIIIIIIIIIIIII`

very easy to add, hard to multiply, very hard to read

Roman numerals `VI + XI = XVII`

easy to add, hard to multiply, moderate to read

Arabic decimal numbers `6 + 11 = 17`

hard to add and to multiply, easy to read

Binary numbers `110 + 1011 = 10001`

easy to add and to multiply, moderate to read

Boundary numbers `((*)*) + (((*))*)* = ((*)*)(((*))*)*`

very easy to add and to multiply, hard to read

Types of numbers

existential	(*indicative*)	exists or not	
categorical	(*nominal*)	share some property	+ attribute
ranking	(*ordinal*)	put in order	+ less than
discrete	(*interval*)	relate to integers	+ equal steps
comparative	(*ratio*)	relate to fractions	+ parts and zero
continuous	(*real*)	relate to infinite	+ compactness
complex	(*imaginary*)	relate to model	+ i, other unit bases

The Numerical/Measurement Hierarchy

i	*Imaginary*	(real1, real2)	relation of reals
delta	*Real*	dense packing between rationals	relation of rationals
divide	*Rational*	integer1 / integer2	relation of integers
equal	*Interval*	ordinal1 - ordinal2 = constant	relation of ordinals
less-than	*Ordinal*	nominal1 less-than nominal2	relation of nominals
member-of	*Nominal*	indicative1 member-of indicative2	relation of indicatives
<void>	*Indicative*	( )	relation of existence

Integers as Sets

	1	2	3	4	5
Cardinality:	{}	{}{}	{}{}{}	{}{}{}{}	{}{}{}{}{}
Ordinality:	{}	{{}}	{{{}}}	{{{{}}}}	{{{{{}}}}}
Uniqueness:	{}	{{}}	{{},{{}}}	{{},{{}},{{},{{}}}}	

Some Varieties of Numbers

Conway numbers (surreals) provide a single coherent framework for defining all types of numbers, and provide ways to manipulate infinite forms.

Spencer-Brown arithmetic is a boundary representation in which each form is both a numerical object and an operator.

Kauffman arithmetic uses a boundary form of place notation to provide a more efficient computational representation while maintaining operations which are both parallel and insensitive to the magnitude of the number.

The James Calculus uses three boundaries to shift the representation of numbers between exponential and logarithmic forms. This mechanism generalizes the ideas of cardinality and inverse operations, and constructs a new imaginary which imparts phase structure on numbers.

Conway Numbers (Surreal Numbers)

A ***number*** is a partitioned set of prior numbers, {L|G},
such that no member of L is greater than or equal to any member of G.

The set L contains *lesser* numbers, while the set G contains *greater* numbers.

Let xL be an arbitrary member of L, and xG be an arbitrary member of G.

```
x = {xL|xG}        such that no xL >= any xG

                         i.e. every xL < every xG
```

Two Conway numbers are ***ordered***

```
x >= y       when        no xG <= y and no yL >= x

                         i.e. every xG > y and every yL < x
```

Two Conway numbers are ***strictly ordered***

```
x > y        when        x >= y and not y >= x

                  i.e.   all xG > y, all yL < x, some xL < y, some yG > x
```

Two Conway numbers are ***equal***

```
x = y        when        x >= y and y >= x

                  i.e.   all xG > y, all xL < y, all yL < x, all yG > x
```

Integers from Ordinals

"Before we have any numbers, we have a certain set of numbers, namely *the empty set*, {}."
-- John H. Conway

Base: { | } empty partitions of the empty set

Generator: every partition of the set of prior numbers

The base of this system is the *act of partitioning*, not the empty set. Partitioning creates the first distinction, which serves as sufficient structure to build all numerical computation.

The conventional names of numbers can be assigned to Conway numbers:

Is { | } a number? { | } = 0

every xL < every xG? yes since there are no xL

Is { | } >= { | } is 0 >= 0?

every xG > 0 and every yL < 0? yes since there are no xG or yL

By symmetry y >= x, thus 0 = 0

Building from Zero

0 is a Conway number, making the set of numbers currently known = {0}. This generates three new number partitions:

{0| } { |0} {0|0}

{0|0} is not a number, since there is an xL >= xG, namely xL = 0

{0| } is a number, call it 1

{ |0} is a number, call it -1

What is the ordering of these new numbers? For illustration, we'll test 0 against -1:

Ordered:

Is { | } >= { |0}? i.e. is 0 >= -1? x=0, y=-1

every xG > -1 and every yL < 0? yes since there are no xG or yL

Thus 0 >= -1.

Strictly ordered:

Is { | } > { |0}? i.e. not-1 >= 0?

every -1G > 0 and every -1L < 0? no, since -1G = 0

Thus 0 > -1. Similarly (tests omitted) 1 > 0.

Building from One

Now, the current set of prior numbers = {-1,0,1}, with a strict ordering, 1 > 0 > -1.

Three prior numbers generate 8 (2^3, the powerset) possible sets to partition. The definition of a number constrains the forms generated from these sets to 21 new number forms:

{-1|0} {-1|0,1} {-1|1} {0|1} {-1,0|1} { |R} {L| }

where R and L stand for any of the eight sets in the powerset of prior numbers.

Conway numbers have multiple representations (just like another representation of 7 is 3+4). A closer analogy would be to have the number three written in different languages (three, trois, drei,...). For example:

```
0 = { | } = {-1| } = { |1} = {-1|1}
```

In general: *the smallest* `xG` *defines* `G`, *the largest* `xL` *defines* `L`.

This is easy to see since the tests for numbership and ordering compare small `xG`s and large `xL`s.

The new numbers are:

```
{1| } = 2          { |-1} = -2        {0|1} = 1/2        {-1|0} = -1/2
```

This gives a hint about how to think about Conway representations: the new number is "in between" the largest `xL` and the smallest `xG`. When one side of the partition is void, a new integer is formed.

Number Form Rules

A contribution of Conway numbers is that they incorporate all types of numbers in one consistent system. Given a number `{a,b,c,...|d,e,f,...}`, the interpretation of that form is the simplest conventional number which is strictly greater than `max[a,b,c,...]` and strictly less than `min[d,e,f,...]`. In general:

If there's any number that fits, then use the simplest number that fits.

`x` is an ***ordinal*** number when

```
x = {L| }

{n| } = n+1
```

`x` is a ***negative integer*** when

```
x = { |G}

{ |-n} = -(n+1)
```

`x` is a ***fraction*** when

```
{n|n+1} = n + 1/2

{0|1/2^(n-1)} = 1/2^n

{p/2^n | (p+1)/2^n} = (2p+1)/2^(n+1)
```

x is a ***real*** number when

```
x = {x - 1/n|x + 1/n}   for n > 0
```

Infinities and Infinitessimals

Conway numbers allow computation with a diversity of infinities and infinitessimals. ***Infinite*** numbers are generated when an infinity of ordinals is included in xL:

```
w = {0,1,2,...| }        w is infinite
```

Unlike conventional numbers, operations on varieties of infinite numbers are defined:

```
w + 1   = {0,1,2,...,w| }

w - 1   = {0,1,2,...|w}

w/2     = {0,1,2,...|w,w-1,w-2,...}

w^(1/2) = {0,1,2,...|w,w/2,w/4,w/8,...}
```

Conway Operators

For a representation to be useful, it must be accompanied with a set of transformation rules. Here is how the standard numerical operations are defined recursively for Conway numbers:

Addition

Base: `0 + 0 = { | }`
Generator: `x + y = {xL+y, x+yL | xG+y, x+yG}`

Example: `2 + (-1) = {1| } + { |0}`

```
xL+y = 1 + (-1)    = 0        this sum is computed recursively
x+yL = 2 + void    = void
xG+y = void + (-1) = void
x+yG = 2 + 0       = 2        this sum is computed recursively

x + y = {0|2} = 1
```

To show that `{0|2}` is a representation of `{0| } = 1`, show equality:

```
x={0|2} =?= y={0| }

every xG > y                  2>1  true
every xL < y                  0<1  true
every yL < x                  0<1  true
every yG > x                  none true
```

Negation

Base: `-0 = { | }`
Generator: `-x = {-xG|-xL}`

Multiplication

Base: `0*0 = { | }`
Generator:
```
x*y = {xL*y+x*yL-xL*yL, xG*y+x*yG-xG*yG |
       xL*y+x*yG-xL*yG, xG*y+x*yL-xG*yL}
```

Multiplication recurs on each partition of each variable.

Division

`y` is a number and `x*y = 1`

Base: `y = {0| }`
Generator:
```
y = {0, (1 + (xG-x)*yL/xG, (1 + (xL-x)*yG/xL |
        (1 + (xL-x)*yL/xL, (1 + (xG-x)*yG/xG}
```

Conway Star

The form `{0|0}` is not a number. However, it can be treated as an *imaginary* number, `*`, such that

```
* + * = 0          * =/= 0
```

Star is its own inverse. Generally,

```
n + * = {n|n}  for any n
```

```
n + * = {0+*,1+*,...(n-1)+*|0+*,1+*,...(n-1)+*}
```

Consider `{0|*}`, which is less than or equal to `{0|1}, {0|1/2},{0|1/4},...`

`{0|*}` is a positive number which is smaller than all other positive numbers, call it `d+`.

`{*|0}` is a negative number which is larger than all other negative numbers, call it `d-`.

```
{d+|d-} = {d+|0} = {0|d-} = {0|0} = *
```

```
d+ + * = {0,*|0}
```

```
d- + * = {0|0,*}
```

```
{0|d+} = d+ + d+ + *
```

Spencer-Brown Numbers

Spencer-Brown Arithmetic (Parenthesis Version)

In Spencer-Brown arithmetic, each number is both an object and an operator.

Integers: (Stroke arithmetic in a container)

```
0          (  )
1          (())
2          (()())
3          ((()()))
```

Operations:

```
a+b        ((a)(b))
a*b          a  b
a^b        ((a) b )
```

Reduction Rules (implicit commutativity and associativity):

```
((a)) = a                      Involution

(()()) a = ((a)(a))            Distribution
```

Examples: (Brackets are for highlighting, they are identical to parentheses.)

```
2+3 = 5          (()()) + ((()())) =?= ((()()()()))

      [[ (()()) ][ ((()())) ]]                  sum
      [   ()()     (()()()   ]                  involution
                 5                             interpret

2*3 = 6          (()()) * ((()())) =?= ((()()()()()))

      [[][]] (()()())                           product
      [[ (()()()) ][ ((()())) ]]                distribute 3 into 2
      [   ()()()     (()()()   ]                involution

2^3 = 8          (()()) ^ ((()())) =?= ((()()()()()()()))

      [[ (()()) ] ((()())) ]                    power
      [   ()()   ((()())) ]                     involution
      [( (()()) (()()) (()()) )]                distribute 2 into 3
         (()()) (()()) [[][]]                   involution   (2*2*2)
         (()()) [[(()())][(()())]]              distribute 2 into 2
         (()()) [  ][]    [][]  ]               involution
      [[(()())] [(()())] [(()())] [(()())]]     distribute 2 into 4
      [  ()()    ()()    ()()    ()()  ]        involution
```

Kauffman Numbers

Kauffman Arithmetic (String Version)

Integers

```
1     *
2     **     -->  (*)
3     ***    -->  (*)*
4     ****   -->  ((*))
```

Operations

```
+     juxtapose              a + b       is    a   b
x     substitute for *       a x b       is    a   b
                                                \_/
```

Canonical Transformations

Instance	Conventional arithmetic	Algebraic
* a = a *	1 + a = a + 1	a b = b a
** = (*)	1 + 1 = 2 x 1	a a = 2a = (a)
(a)(b) = (a b)	2a + 2b = 2(a + b)	a)(b = a b

Example 2 x (4 + 3) --> 14

```
(*)   ((*)) (*)*   =   (  ((*))(*)*  )  -->  (  ((*)*)*  )
  \_____/____/_/
```

Kauffman Arithmetic (Molecular Version)

Addition is *physical mixing*. 4 + 3

```
  *    *            (*)       (**)             ((*))
 *  *      -->   (*)  *    -->       *   -->          *
  *  *  *          (*)           (*)               (*)
```

Multiplication is *chemical mixing*. 2 x 7

```
     ((*))          (((*)))
   ___/                                 (((*)))
  /         (*)  -->       ((*))   -->              -->  (((*)*)*)
 (*)    *   /          (*)                  ((*)*)
  \____/___/
```

James Numbers

James calculus uses three types of containers/boundaries to represent all types of numbers. Several unique numerical concepts arise from this approach. *Generalized cardinality* applies to negative and fractional counts, as well as to integer counts. The *generalized inverse* unifies subtraction, division, and roots into a single concept and operation. The *James imaginary*, **J**, embeds inverse operations into numbers with *phase* as well as magnitude. J can be used for numerical computation as an alternative to using numbers.

James Form	Interpretation
Boundary Operators	(swapping between exponential and logarithmic spaces)
(a)	e^a
[a]	ln a
<a>	inverse a (generalized)
Boundary Units	(every boundary is both an object and an operator)
( )	e^0 = 1
[ ]	ln 0 = negative infinity
< >	negative 0 = 0
Integers	(stroke arithmetic using containers)
void	0
( )	1
( ) ( )	2 ...

Since stroke representation is rather clumsy. I will use decimal numbers to abbreviate stroke numbers throughout this section.

Operations

a b	a+b	(shared space)
([a] [b])	a*b	
((([a]][b]))	a^b	

Boundaries can be read as exponents and natural logs:

```
a*b = ([a][b]) = e^(ln a + ln b) = e^ln a * e^ln b = a*b

a^b = ((([a]][b])) = e^(e^(lnln a + ln b))
            =      = e^(e^lnln a * e^ln b)
                   = e^(ln a * b) = (e^ln a)^b = a^b
```

Reduction Rules (Axiomatic basis)

Computation is achieved through application of three reduction rules:

`([a]) = [(a)] = a`	*Involution*
`(a [b]) (a [c]) = (a [b c])`	*Distribution*
`a <a> = void`	*Inversion*

The Form of Numbers

Type	Standard form	James form
zero	`0`	`void`
natural	`n`	`()()..n = ([n][()])`
negative integer	`-n`	`<()()..n> = ([n][<()>]) = <([n][()]>`
rational	`m/n`	`([m]<[n]>)`
irrational	`m^-n`	`(([[m]]<[n]>))`
complex	`i`	`(([[<()>]]<[2]>))`
transcendental	`e`	`(())`

The Form of Numerical Computation

In the container representation, the relationships between numerical operations becomes overt. Essentially, any operation is applying the pair `(...[...]...)` to a particular part of the existing form. Addition begins with no boundaries. Like stroke arithmetic, addition (and its inverse subtraction) is putting things in the same space. Multiplication (and its inverse division) involves converting to logs with `[...]` and then back to powers of `e` with `(...)`. Power (and its inverse root) is another application of the `(...[...]...)` form.

addition	`A B`
multiplication	`( [A] [B] )`
power	`(([[A]] [B] ))`
subtraction	`A < B >`
division	`( [A] < [B] >)`
root	`(([[A]]< [B] >))`

These forms are spread out to show how each operator is an `(...[...]...)` elaboration of the previous form:

addition	A B
multiplication	([ ] [ ])
power	([ ])
subtraction	A < B >
division	([ ] [ ])
root	([ ])

The placement of containers reflects the properties of each operator. Both forms are free of containment for commutative addition. Both forms are enclosed for commutative multiplication. One form is enclosed for power, it is not commutative. Inversion is generic, the second form is simply inverted in all cases, creating the non-commutative inverse operations.

Logarithms

log base e	ln n	[n]
antilog base e	antiln n	(n)
log base b	logb n	([[n]] <[[b]]>)
antilog base b	antilogb n	(([n] [[b]]))

Setting the logarithmic base to e results in the appropriate reduction:

```
ln n = ([[n]] <[[(())]]>)
     = ([[n]] <[  ()  ]>)
     = ([[n]] <        >)
     = ([[n]]           )
     =   [n]
```

Converting between bases:

```
ln n = log10 n * ln 10

[n] = (([[n]]<[[10]]>)) times [10]        hybrid

    = ([([[n]]<[[10]]>)][[10]])
    = (  [[n]]<[[10]]>  [[10]])
    = (  [[n]]                )
    =     [n]
```

Using the spread out form, we can see the relationship between logs and the other operations:

subtraction	A < B >
division	([ ] [ ])
logarithm base B	[ ] [ ]
addition	A B
multiplication	([ ] [ ])
antilog base B	([ ])

Generalized Inverse

The ***generalized inverse*** treats subtraction, division, and roots as the same operation in different contexts. Below, the spacing between characters is used for emphasis.

```
-1                      < ()  >
-B                      <  B  >
A-B                   A  <  B  >

1/2                (     <[ 2 ]>)
1/B                (     <[ B ]>)
A/B                ( [A] <[ B ]>)

A^2               ((([A]] [ 2 ] ))
A^B               ((([A]] [ B ] ))
A^-B              ((([A]] [<B>] ))
A^(1/B)           ((([A]]<[ B ]>))

ln B                        [ B ]
logA B             ([[A]]<[[B]]>)
antilogA B        (( [A]  [[B]] ))
```

Some James Calculus Theorems

Name	Form	Interpretation
Cardinality	`A..n..A = ([A][n])`	`A+..n..+A = A*n`
Dominion	`[ ] A  = [ ]` `([ ] A) = void`	`-inf + A = -inf` `e^(A + -inf) = 0`
Inverse Collection	`<A><B> = <A B>`	`(-A)+(-B) = -(A+B)`
Inverse Cancellation	`<<A>> = A`	`--A = A`
Inverse Promotion	`([A][<B>]) = <([A][B])>`	`A*-B = -(A*B)`

Some examples of proof:

```
--A = A                          <<A>>
                                 <<A>><A>A          inversion
                                        A           inversion

-ln(e^A) = -A = ln(e^-A)         <[(A)]>
                                 <  A  >            involution
                                 [(<A>)]            involution

A/A = 1                          ([A] <[A]>)
                                 (         )        inversion
```

```
e^A * e^-A = 1          ([(A)][(<A>)])
                        (  A    <A>  )      involution
                        (            )      inversion
```

Generalized Cardinality

Multiple reference can be explicit (a listing) or implicit (a counting). `n` references to `A` can be abstracted to `n` times a single `A`, in both the additive and the multiplicative contexts. The form of cardinality is:

```
([A][n])
```

Adding `A` to itself `n` times is the same as multiplying `A` by `n`:

```
A ..n.. A   =  ( [A] [n])
```

Multiplying `A` by itself `n` times is the same as raising `A` to the power `n`:

```
([A]..n..[A]) = ((([A]][n]))
```

Negative cardinality cancels or suppresses positive occurrences. The form of negative cardinality is

```
([A][<n>])
```

Adding `A` to itself `-n` times is the same as multiplying `A` by `-n`, and is also the same as adding `-A` to itself `n` times:

```
A..<n>..A = ([A][<n>]) = <([A][n])> = ([<A>][n]) = <A>..n..<A>
```

Adding `ln A` to itself `n` times and then raising `e` to that power is the same as multiplying `A` by itself `-n` times.

```
(<[A]>..n..<[A]>) = (([<[A]>][n])) = (<([[A]][n])>)
                  = (([[A]][<n>])) = ([A]..<n>..[A])
```

Multiplying `-A` by itself `n` times is the same as raising `-A` to the `n`th power:

```
([<A>]..n..[<A>]) = ((([<A>]][n]))
```

Here is a proof that negative cardinality cancels positive cardinality:

```
([A][n]) ([A][<n>])                 (n*A) + (-n*A) = 0

      ([A][n <n>])                  distribution
      ([A][ ])                      inversion
      ([ ])                         dominion
      void                          inversion
```

Fractional cardinality constructs fractions and roots. The form of fractional cardinality is:

```
([A]<[n]>)
```

Adding the fraction A/n to itself n times yields A. Here is a proof that fractional cardinality accumulates into a single form:

```
([A]<[n]>)..n..([A]<[n]>)          (A/n) +..n..+ (A/n) = A

      ([([A]<[n]>)][n])            cardinality
      (  [A]<[n]>  [n])            involution
      (  [A]          )            inversion
          A                        involution
```

Multiplying the fraction n/A by itself 1/n times yields 1/A:

```
([n]<[A]>)..1/n..([n]<[A]>)        (n/A)*(1/n)= 1/A

      ([([n]<[A]>)][(<[n]>)])      cardinality
      (  [n]<[A]>    <[n]>  )      involution
      (     <[A]>           )      inversion
```

James Calculus Unit Combinations

The two-unit combinations generate {0, e, inf}. The only three unit combination which does not reduce has an imaginary interpretation.

Two unit combinations

```
(<>) = () = 1                 e^-0
(()) = e                      e^1
([]) = void = 0               e^(ln 0) = e^(-inf)

[<>] = [] = -inf              ln -0 = -inf
[()] = void = 0               ln e^0 = ln 1 = 0
[[]] = [] = J inf             lnln 0 = ln -inf = ln -1 + ln inf

<<>> = <> = void              --0 = 0
<()> = -1                     -e^0
<[]> = inf                    --inf = inf
```

Three unit combinations

```
<([])> = <[()]> = ([<>]) = [(<>)] = 0

(<[]>) = <[]>           e^--inf = e^inf = inf

[<()>]                  J, the James imaginary

      J = [<()>] = ln -1
```

The James Imaginary

Independence

```
[<(A)>] = A [<()>] = A J
```

Interpretation:

```
ln(-e^A) = A + ln-1 = A + J
```

Proof:

```
[<(A)>] =    [<(A)>][ () ]              add 0
        = [ ([<(A)>][ () ]) ]           involution
        = [<([ (A) ][ () ])>]           promote
        = [ ([ (A) ][<()>]) ]           promote
        =       A   [<()>]              involution
```

Imaginary Cancellation

```
[<()>] [<()>] = JJ = void
```

Interpretation:

```
J + J = 0
```

Proof:

```
[<()>][<()>] = [  ([<()>][<()>])  ]     involution
             = [<<([ () ][ () ])>>]     promote
             = [<<(            )>>]     involution
             = [  (            )  ]     cancel
             = void                     involution
```

Own Inverse (only `0` has this property in conventional number systems)

```
J = <J>
```

Interpretation:

```
J = -J          and J =/= 0
```

Proof:

```
      J = J <>                          add 0
        = J <JJ>                        J cancel
        = J <J><J>                      collect
        = <J>                           inversion
```

Phase

The phase of J is determined by its cardinality.

```
void = JJ = JJJJ = ...                period 2
```

Rules of J

Here are some common patterns which involve `J`.

```
J = <J>
JJ = void

 J  = [<()>]
(J) =  <()>

  A   = <(J  [A] )>                   A (J [A]) = void
 <A>  =  (J  [A] )

 (A)  = <(J   A  )>                   (A) (J A) = void
<(A)> =  (J   A  )

 [A]  = <(J [[A]])>                   [A] (J [[A]]) = void
<[A]> =  (J [[A]])

(  A  [J]) = <( A [J])>
([<A>][J]) =  ([A][J])

void = () <()> = () (J)
```

Inverse Operations as J Operations

`J` is intimately connected with the act of inversion. Its definition contains `-1`; as well, it is implicated in the definition of a reciprocal since `1/A = A^(-1)`. All occurrences of the generalized inverse can be converted to `J` forms:

subtraction	`A-B`	`A <B> = A (J [B])`
reciprocal	`1/B`	`(<[B]>) = ((J [[B]]))`
division	`A/B`	`([A]<[B]>) = ([A](J [[B]]))`
root	`A^(1/B)`	`((([A]]<[B]>)) = ((([A]] (J [[B]])))`
negative power	`A^-B`	`((([A]][<B>])) = ((([A]][(J [B])])) = ((J [B][[A]]))`
log base A	`logA B`	`([[A]]<[[B]]>) = ([[A]] (J [[[B]]]))`

J in Action

J provides an alternative technique for numerical computation. Consider the two versions of this proof:

```
(-1)*(-1) = 1       ([<()>][<()>])
                     <([ () ][<()>])>        promote
                    <<([ () ][ () ])>>       promote
                      ([ () ][ () ])         cancel
                      (            )         involution

(-1)*(-1) = 1       ([<()>][<()>])
                      (  J     J   )         J
                      (            )         J cancel
```

Finding and creating Js in a form usually offers a short cut for reduction.

```
(-1)/(-1) = 1       ([J] <[J]>) = ()        inversion


A^(-1) = 1/A      (([[A]] [<()>])) =?= (<[A]>)

                  (([[A]]   J   ))
                  ( <[A]>        )


(a+1)(a-1) = a^2 - 1

([a ()][a <()>]) = ([a ()][a (J)])
                 = ([a ()][a]) ([a ()][(J)])
                 = ([a][a]) ([()][a]) ([a] J) ([()] J)
                 = ([a][a]) (    [a]) ([a] J) (     J)
                 = ([a][a])       a   ([a] J) (     J)
                 = ([a][a])                   (     J)

                 = (([[a]][2])) (J) = a^2 - 1
```

Transcendentals and Complex Functions

```
      (()) = e^(e^0) = e
```

Since no rules reduce (()) to any other form, e is incommensurable with other numbers.

i, the square root of -1

```
i = (-1)^(1/2)    ((([ -1 ]] [  1/2  ]))        hybrid

                  (([[<()>]] [(<[2]>)]))
                  (([  J   ]   <[2]>  ))
                  (([  J   ](J [[2]]) ))
```

```
i = (([J](J [[2]])))
```

This leads to the interesting interpretation:

```
i = (([J]<[2]>)) = e^(J/2)
```

Squaring:

```
i^2 = e^J = -1
```

This can be derived directly:

```
i^2 = -1          ((([i]] [2]))  =  <()>
                 [((([i]] [2]))] = [<()>]
                   ([[i]] [2])   =   J
```

This yields an interpretation which is consistent with the derivation from `i`:

```
J = 2 ln i
```

The `J` form of complex numbers is:

```
a+bi = a ([b][i]) = a ([b][(([J](J [[2]])))])
                  = a ([b]  ([J](J [[2]]))  )
```

P i

Using Euler's formula we can find another interesting result:

```
e^(i*Pi) = -1
e^J      = -1

J = i*Pi
```

Now we can express `Pi` in terms of `J`:

```
Pi = J/i = ([J]<[i]>) = ([J]<[(([J](J [[2]])))]>)
                      = ([J]<  ([J](J [[2]]))  >)
                      = ([J]   ([J](J [[2]]))   )

Pi = ([J] ([J] (J [[2]])))
```

Interpreting:

```
Pi = ([J] ([J] (J [[2]])))
   = ([J] ([J] <[2]>))
   = ([J]    J/2     )              hybrid
   = ([J]  [(J/2)]   )
   = J * e^(J/2)

Pi = Je^(J/2)
```

Yet another relationship:

```
Pi = 2i ln i = J*i

2i ln i = ([2] [(([J]<[2]>))] [[(([J]<[2]>))]])
        = ([2]   ([J]<[2]>)       [J]<[2]>    )
        = (      ([J]<[2]>)       [J]         ) = Pi
```

Combining results from above, we get the straightforward result:

```
J*i = J/i
```

cos x = (e^ix + e^-ix)/2

Trigonometric functions can be expressed as imaginary powers of e:

```
cos x = ([(ix)(<ix>)]<[2]>)                hybrid

            where  i = (([J] <[2]>))

                  ix = ([i][x]) = (([J]<[2]>)[x])

      cos x = ( [(([J]<[2]>)[x])) (<(([J]<[2]>)[x])>)] <[2]>)

            = ([(([J]<[2]>)[x]))] <[2]>) ([(<(([J]<[2]>)[x])>)] <[2]>)
            = (  (([J]<[2]>)[x])   <[2]>) (  <(([J]<[2]>)[x])>   <[2]>)

      let c = <[2]> = (J [[2]])

            cos x = ((([J] c)[x]) c) (<(([J] c)[x])> c)

      let d = [x] (c [J])

            cos x = ((d) c) (<(d)> c)
                  = ((d) c) ((J d) c)

                  = (c   (d))   (c   (J d)  )
                  = (c [((d))]) (c [((J d))])
                  = (c [((d))((J d))])
```

J in Standard Notation

i is the additive imaginary. J is the multiplicative imaginary.

```
J + J = 0

J = ln -1

e^J = e^(ln -1) = -1
```

Bricken Star, *

When the value -1 is accessed by going through the imaginary J, call it *.

```
* = -1            *^2 = 1          * = -*           * + * = 0

      - A = A times *          = Ae^J

      1/A = A^*                = A^e^J

      n^(1/A) = n^A^*          = n^A^e^J

      i = *^(1/2)              = *^2^*
```

Base-free

In going through imaginary hyperbolic space and then returning, the base is arbitrary.

```
n^J = n(logn -1) = -1
```

Thus the base can be chosen to be the same as the inverse number, i.e.:

```
      - A = A times *          = AA^J = A^(J+1)

      1/A = A^*                = A^A^J

      A^(1/A) = A^A^*          = A^A^A^J
```

** algebra*

```
a^* + b^* = (a+b) times (ab)^*

((([a]][*])) ((([b]][*])) = ([a b] ([[a]][*]) ([[b]][*]))
```

Infinities

This calculus has a natural representation of infinity, <[]> which we can use computationally:

```
[] <[]> =/= void

<[]> = (J [[]])

<[]> X = <[]>                inf + X = inf

[<[]>] = <[]>                ln inf = inf

(<[]>) = <[]>                e^inf = inf       (<[]>) = 1/0

[[]] = J <[]>                ln ln 0 = ln -inf = J inf
```

```
1^inf = ((([1 ]][inf ]))      hybrid
      = ((([()]][<[]>]))
      = (([    ][<[]>]))
      = (              ) = 1
```

Here we can see that `1` raised to any power will result in `1`. However:

```
0^inf = ((([]][<[]>]))
      = ((([]] <[]> ))
      = ((     <[]> ))
      = <[]>                                  = inf

0^0 = ((([]][])) = (([])) = ()                = 1

0/0 = ([]<[]>) = ([]) = void                  = 0

1/0 = ([()] <[]>) = (<[]>) = <[]>             = inf

0^(1/0) = ((([]] [([()] <[]>)]))
        = ((([]]        <[]>  )) = <[]>       = inf
```

Differentiation

The rules of differentiation in the James calculus follow. Let `'A'` be `dA/dx`.

```
'c' = void                                    dc = 0

'x' = ()                                      dx = 1

'(A)' = (['A']  A  )                          de^A = e^A dA

'[A]' = (['A']<[A]>)                          dln A = 1/A dA

'<A>' = <'A'>                                 d-A = -dA

'A B' = 'A' 'B'                               d(A+B) = dA + dB
```

Proof of the Chain Rule of Differential Calculus

```
d(a*b) = b da + a db

'([a][b])'

 ([a][b]['[a][b]'])

 ([a][b] [(<[a]>['a']) (<[b]>['b'])])

 ([a][b] [(<[a]>['a'])]) ([a][b] [(<[b]>['b'])])

 ([a][b]   <[a]>['a'])   ([a][b]  <[b]> ['b']  )

 (   [b]        ['a'])   ([a]           ['b']  )
```

Using James differentiation:

```
y = e^(ax)              dy = ae^(ax)

y = (([a][x]))

dy = '(([a][x]))'
   =  ([ '([a]  [x])'        ] ([a][x]))
   =  ([(['[a]  [x]'] [a][x])] ([a][x]))
   =  (  ['[a]  [x]'] [a][x]   ([a][x]))
   =  (  ['[a]''[x]'] [a][x]   ([a][x]))
   =  (  [(['a']<[a]>)(['x']<[x]>)] [a][x] ([a][x])  )
   =  (  [([   ]<[a]>)(([() ]<[x]>)] [a][x] ([a][x])  )
   =  (  [            (     <[x]>)] [a][x] ([a][x])  )
   =  (                     <[x]>   [a][x] ([a][x])  )
   =  (                             [a]    ([a][x])  )
```

Interpreting:

```
dy =  ([a]  ([a][x])  )
dy =  ([a][(([a][x]))]) = a*e^(a*x)
```

```
y = x^n                 dy = nx^(n-1)

y = (([[x]][n]))        dy = ([n][(([[x]][n <()>]))])
                        dy = ([n]  ([[x]][n <()>])  )

dy ='(([[x]][n]))'
   = (['([[x]][n])'            ] ([[x]][n]))
   = ((([['[[x]]  [n]'] [[x]][n])] ([[x]][n]))
   = (  ['[[x]]  [n]'] [[x]][n]   ([[x]][n]))
   = (  ['[[x]]''[n]'] [[x]][n]   ([[x]][n]))

   = (  [(['[x]'       ]<[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [((['x']<[x]>)]<[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(  ['x']<[x]>  <[[x]]>) (['n']<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(  [() ]<[x]>  <[[x]]>) ([   ]<[n]>)] [[x]][n] ([[x]][n]))
   = (  [(       <[x]>  <[[x]]>)             ] [[x]][n] ([[x]][n]))
   = (           <[x]>  <[[x]]>                [[x]][n] ([[x]][n]))
   = (           <[x]>                              [n] ([[x]][n]))

   = ([n] ([[x]][<()>]) ([[x]][n     ]))
   = ([n]               ([[x]][n <()>]))
```

Interpreting:

```
dy = ([n]   ([[x]][n <()>])  )
dy = ([n] [(([[x]][n <()>]))]) = n*x^(n-1)
```

The next derivation illustrates the use of `J`:

```
y = J = [<()>]

dy = '[<()>]'
   = (['<()>'] <[<()>]>)
   = ([<'()'>] <[<()>]>)
   = ([<    >] <[<()>]>)
   = ([      ] <[<()>]>) = void


'<A>' = '(J [A])'
      = (['J''[A]'           ] J [A])
      = ([   (['A'](J [[A]]))] J [A])
      = (     ['A']           J    )
      =       <'A'>
```

Graphs

Graphs and graph algorithms are pervasive in Computer Science. The graph is such a general data structure that almost all computational problems can be formulated using one of the primary graph processing algorithms. Lists and trees are subsets of graphs.

The major problems in hardware synthesis, operating system scheduling and sharing, compiler optimization, software design and minimization, network communication and synchronization, and requirements and specification modeling are graph problems. Variations of the fundamental graph algorithms of traversal, covering, and coloring model most of the problems in these application areas.

Definition

A ***graph*** is a collection of vertices `V` (also called nodes) and edges `E` (also called arcs or links) which connect the vertices. A single edge can be described by the two vertices which it connects:

```
e = (u,v)
```

Thus, a graph consists of two sets drawn from the carrier set of nodes, `S`. The first set is simply the set of nodes; the second set is a set of unordered pairs of nodes, representing edges.

```
G  =  <S,(S,S)>
```

Note that the single set of unordered pairs is sufficient to characterize the graph, assuming that edges meet at a "virtual node". In the simplest from, graphs are unordered pairs.

A ***generator*** (ie inductive definition) for fully connected graphs:

let `Sn` be the set of nodes of cardinality `n`,
`Ni` be the `i`th node
`Ei` be the edges that have `Ni` as an endpoint.

Base: `N1`
Generator: `Si+1 =` insert `Ni+1` into `Si`
`Ei+1 =` insert one pair `(Ni+1 Nk)` into `Ei`
for each `k` ranging over all `n` nodes.

In a complete graph of n nodes, the number of edges is

```
sum[n] = n(n+1)/2
```

Types of Graphs

undirected graph:
The edges can be traversed in either direction. Undirected graphs are either trees or cyclic.

cyclic graph:
The graph contains loops than permit a node to be visited recurrently.

directed graph:
The edges are directional and support only one direction of traversal.

directed acyclic graph (DAG):
No paths permit visiting a node more than once.

weighted graph:
The edges have a numerical value which may represent the cost of traversing that edge. Weights can be both positive and negative.

connected graph:
All of the vertices can be visited when starting from any vertex. The list of vertices visited between a starting vertex and an ending vertex is called a *path*.

sparce graph:
There are many more vertices than edges, that is `E << V^2`.

dense graph:
The number of edges is about the same as the number of vertices squared; `E =~= V^2`

complete graph:
Every pair of vertices has an edge connecting them. The graph is thus maximally connected.

clique:
A subgraph (that is a subset of nodes and edges in a graph) which is complete.

bipartite graph:
A graph with two sets of vertices, such that each edge connects vertices from different sets. No edges connect vertices in the same set.

hypergraph:
A graph with edges that connect more than two vertices. All hypergraphs can be converted into regular bipartite graphs.

graph complement:
The complement of a graph is a graph with the same vertices, but all edges exchanged (ie if an edge is present, then delete it; if an edge is missing, then add it).

Representation of Graphs

The choice of graph data structures depends on the density of the graph connectivity. The complexity of different algorithms depends both on the number of edges and on the number of vertices. Unlike sets, graphs often have a direct representation in memory. When the data field of a memory cell contains an address of another cell, the cell can be interpreted as a node and the shared address can be interpreted as a link.

adjacency lists:

A collection of `V` lists (usually stored as an array and often hashed), each listing the adjacent vertices of a particular vertex.

adjacency matrix:

A square matrix of `VxV` boolean entries, with a `1` recording adjacent vertices, and a `0` recording non-adjacent vertices.

indexed arrays:

For graphs which change during processing, two arrays are used. The first array contains the `V` vertices, and provides an index for each vertex. This array is changed only when the status of the vertex changes, the vertex index is never changed. The second array contains the `E` edges, and provides an index for each edge. The data field of the second array contains the indices of the two vertices connected by that edge. The connectivity of the graph is changed by changing the edge array entries.

Parenthesis Structure

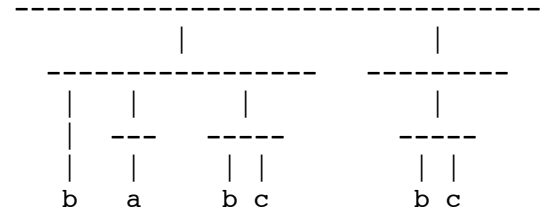
Graphs can be expressed in a linear parenthesis notation which is convenient for computation and analysis.

Example

```
V = {1,2,3,4,5,a,b,c}          E = {12,13,24,25,2b,35,4a,5b,5c}

      G = ( ( b ( a ) ( b c ) ) ( ( b c ) ) )
          1 2   4     5          3 5

          (                                  )
            (                  ) (          )
              b (    ) (      )     (      )
                   a     b c          b c
```



Structure sharing in graphs is represented by multiple labels:

```
G = ( ( b ( a )    5  ) (      5    ) )
5 = ( b c )
```

Topological Sort

Sorting lists and trees makes them easier to process, since the vertices have a consistent ordering. Directed acyclic graphs (DAGs) can also be sorted by the partial ordering implicit in the graph connectivity. Cyclic and undirected graphs do not have a consistent sort.
A ***topological sort*** of a DAG is a consistent ordering of the vertices that serves as an ordering for processing vertices. One convenient sorting is by length of shortest paths from the root vertex to the leaf vertices.

Primary Graph Algorithms

searching

The search algorithm ***traverses a graph***, usually identifying the shortest (or longest) path between two vertices. ***Spanning trees*** list paths that connect all vertices. A ***Hamiltonian path*** is a path which visits each vertex while not traversing the same edge more than once. The travelling saleman problem (ie, what is the shortest route available to visit a collection of sites) is an example of computing a minimal spanning tree.

covering

A ***vertex cover*** is a subset of vertices of an undirected graph such that every edge has at least one end in the subset. Minimization of boolean expressions (by Karnaugh maps or by algebraic techniques) is an example of graph covering. Each product term in a sum-of-products (SOP) representation is a subset of vertices of a boolean hypercube. The covering problem is to include each vertex in the SOP using a minimum number of sub-cubes of the hypercube.

coloring

A ***vertex coloring*** is a labeling of the vertices of an undirected graph such that no edge has two endpoints with the same label. An example of graph coloring is scheduling and resource allocation using a *resource conflict graph*. Vertices in a resource conflict graph represent operations to perform; edges represent pairs of operations which are in conflict because they cannot use the same resource.

The Thirsty Archeologist

An archeologist was digging in a Paleozoic mudflat. She came across an imprint of a raindrop that fell 400 million years ago. The sun was hot, and she took a drink from her canteen.

How many molecules of the original Paleozoic raindrop did she drink?

You have 15 minutes to answer this question. Write down your assumptions, choices, and decisions. (No justifications are needed.)