Practical, Formal, Software, Engineering Lecture Slideshow.

Based on the text:

Practical formal
software engineering,
by Bruce Mills,

Cambridge University Press. [2009]

This course describes
a generic method for
software engineering

and illustrate its use
through detailed examples.

This course is about
the mind of the engineer,

not about the computer.

The course is in three main parts.

1. Fundamentals:

   generic reasoning with software.

2. Languages:

   form and meaning of software.

3. Practice:

   practical software examples.

Lecture 1

Roman Arithmetic

Arithmetic is not obvious,
it took thousands of years
to develop.

From Chapter 1: arithmetic.

It is hard to look at text
without reading.

It is hard to look at numerals
without seeing numbers.

But, to learn more,
we must do exactly that.

We say there are 2 goats in a field.

We believe that with another 2,
there would be 4 goats in the field.

But, this is physics,

it might not be true.

Let I mean one thing.

Let II mean two things.

Let III mean three things.

Let IIII mean four things.

Let IIIII mean five things.

Abbreviate IIIII as V.

So, VV means ten things.

And, VVII means 12 things.

Order does not matter

IV is the same as VI

VIIVVI = VVIIIV = IIIVVV

IVVIVI = sort(IVVIVI) = VVVIII

Warning:

Some of you heard IV = 4.
As on clocks.
Romans did not use this much.
And neither will we.

IV = VI = 6

Two Hindu numerals
with the same meaning
look exactly the same.

Test for equality
by checking the form is the same.

Two different Roman numerals might have the same meaning.

Sometimes sorting will show this.

sort(VIIVVI) = VVVIII
sort(VVIIIV) = VVVIII

Permuting the symbols

does not change the meaning.

IIIVIIVVII = VVVIIIIIII

Changing IIIII into V

does not change the meaning.

VVVIIIIIII = VVVVII

To *normalise* a Roman numeral,
sort and abbreviate until
no more changes occur.

The result is the
unique minimal numeral
for the given number.

In the Roman system
normalise first,
then
check equality of normal forms
character by character.

Most software data types
are like the Roman system
not the the Hindu system
in their algorithms.

Equality must be defined.

Let + mean to combine collections.

III + II = IIIII = V,

Addition algorithm is catentation.

IIIVII + VVII = IIIVIIVVII

Changing VV into X

does not change the meaning.

VVVVII = XXII

Abbreviate XXXXX as L

Abbreviate LL as C

Abbreviate CCCCC as D

Abbreviate DD as M

Even though we are not sure

what XVXXLCII is
or XXLLVV is
in Hindu numerals,

we can compute the sum.
using pure Roman numerals.

XVXXLCII + XXLLVV

catenation

= XVXXLCIIXXLLVV

sorting

= C LLL XXXXX VVV II

abreviation

= C CL L XV II

abbreviation

= CCC X V II

Multiplication by distribution.

VI * VII = (V+I)*(V+I+I)

And remembering special cases

*	I	V	X
I	I	V	X
V	V	XXV	L
X	X	L	C

Just like the Hindu system.

VI * VII =

VI * (V+I+I) =

V*V + V*I + V*I +

I*V + I*I + I*I =

XXV + V + V + V + I + I =

XXVVVVII =

XXXXII

Algorithms for manipulation of the symbols allow an abstract analogy between the behaviour of the symbols on the page, and the physical goats in the field.

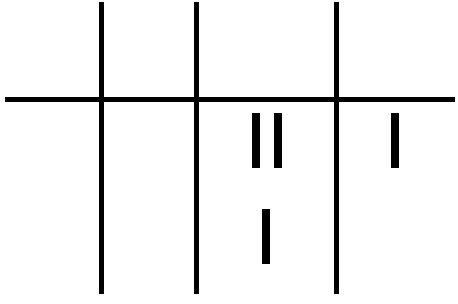
Lecture 2

Tally arithmetic

The decimal system is not the only or the obvious choice of arithmetic.

From Chapter 1: arithmetic.

The tally system uses dashes.



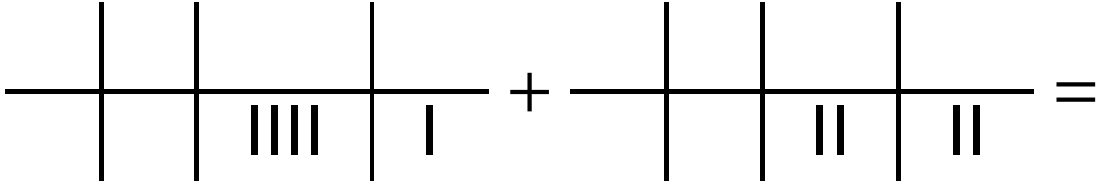
Using base-5,

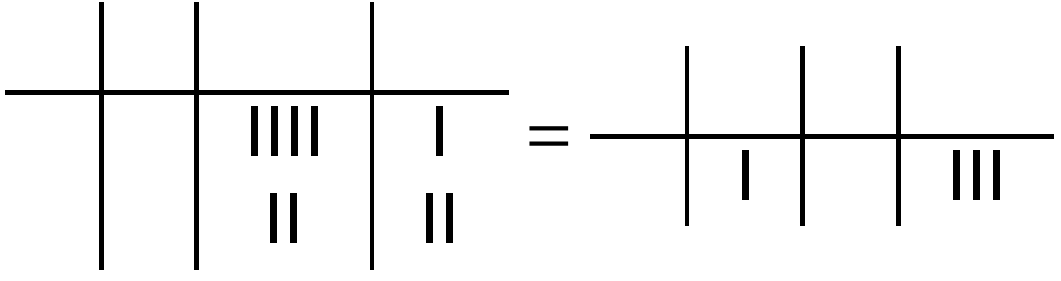
the right-most column means 1,

and each other means 5 times

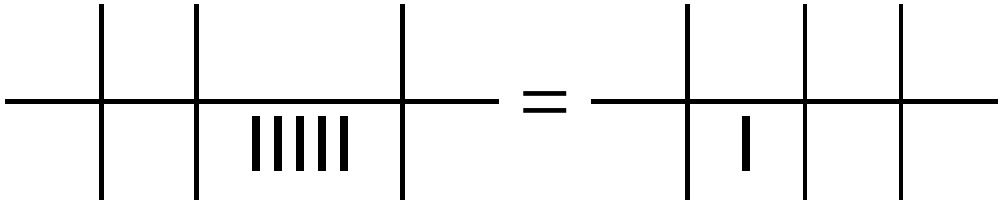
the one on its right.

Addition:





Do not keep track of ‘place value’ just recall the rule ...



Regardless of where this occurs, or in which direction it is used.

Shift left means IIIII becomes I.

Shift right means I becomes IIIII.

Do not give meaning to this.
It is the mechanism of these tables.

Distinct tally forms
can have the same meaning.

Normalisation:
shift to the left when possible.

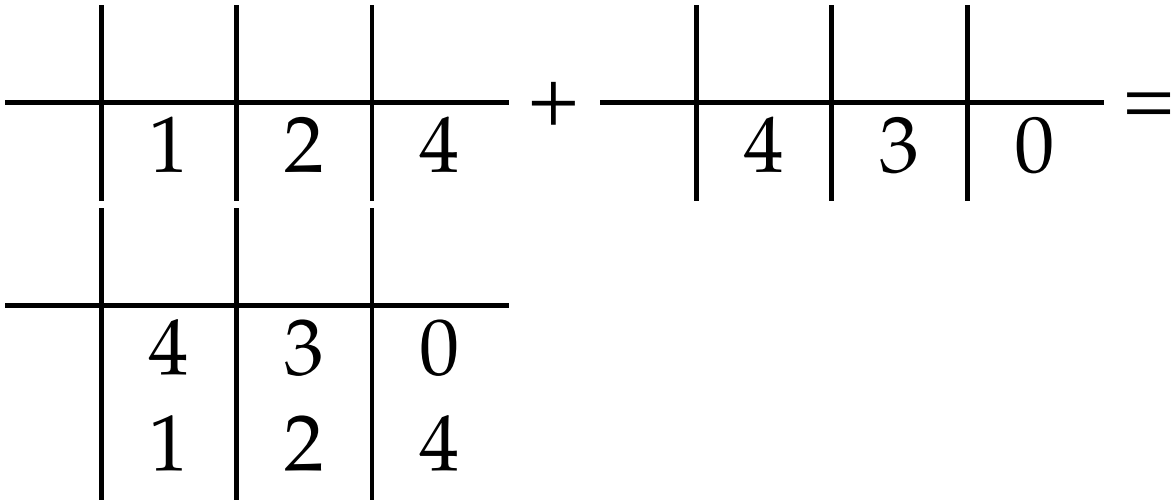
Check equality
by comparing normal forms.

Abbreviate:
I by 1, II by 2, III by 3, IIII by 4.

If a column has nothing in it,
use 0 to emphasise this.

Call 0, 1, 2, 3, 4,
the base-5 digits.

It is acceptable
to have multiple digits in a column.



Normalise from right to left.
Put shifted digits in the top row.

					=			0						
	4	3	0				4	3	4					
	1	2	4				1	2						

	1	0				1	1	0						
	4	0	4		=		1	0	4		=			
	1													

1	1	0		
1	1	0	4	

The regular tableaux, without rubbing out:

1	1	0	0	auxiliary
0	4	3	0	summand
0	1	2	4	summand
1	1	0	4	result

New digits in blue.

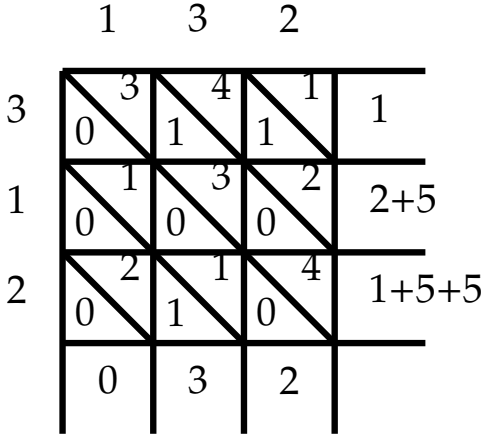
Subtraction:

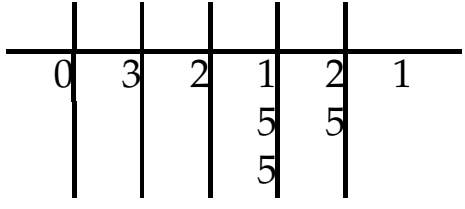
Fill a blank addition tableaux
with one summand and the result.

Work out what
the other summand has to be.

In multiplying,
the column a dash is in
is important.
The dashes multiply,
and the columns add.
The grid system helps
to keep track of this.

The grid system for multiplication.





213 * 132 = 0 ; 3 ; 2 ; 1 5 5 ; 2 5 ; 1

Normalisation:

0	0	2	1	0	0
0	3	2	1	2	1
0	0	0	5	5	0
0	0	0	5	0	0
0	3	4	2	2	1

For arithmetic
there is the formal system
of manipulation of symbols,
and the meaning of these symbols.

A system is formal
if the manipulations
can be completed
without reference to
any meaning for the symbols.

A formal system is
a game with symbols,
like chess, or Rubick's-cube.

It does not have to have meaning.

But, it may have applications.

Lecture 3

Natural Logic.

Logic is the empirical study of correct argument, as such, it is always hypothetical.

From Chapter 2, logic.

It is hard to look at text
without reading.

It is hard to look at argument
without seeing meaning.

But, to learn more,
we must do exactly that.

Logic is not blinkered thinking.

This is from a horror movie:

There must be a *logical* explanation.

There is; this house is haunted.

*When it appears that you have killed the monster, never check to see if it is **really** dead.*

From the guide to
how to survive a horror movie.

Logic is the process.
Statement is the raw material.

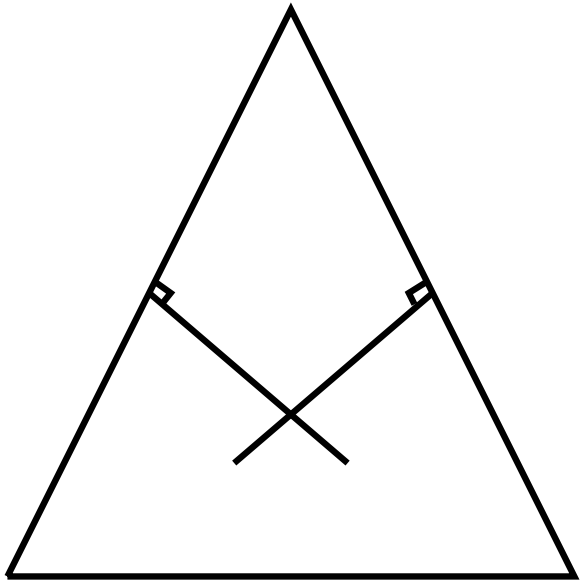
'Statement' cannot be defined unambiguously.

But, very good clues can be given.

Examples of statements:

The cat sat on the mat.
This frog is green.
I like green eggs and Ham.
$1435 \times 1903249 = 6$
The moon is made of green cheese.
'What' is your name.

This is also a statement:



Examples of not statements:

Where is the cat?
Go to your room!
What is your name?
Tra la la.
12391456

Our observations are not pure.

What we expect strongly affects what we observe.

Our state of mind
is a set of statements.

Australia is a country.
A kilometre is 1000 metres.
She is pretty.
I need to go to the toilet.

Argument
changes our state of mind
by addition or removal
of statements.

Our state of mind can also include
questions and commands.

All cats are fish and
all fish are mammals,

becomes

All cats are fish and
all fish are mammals and
all cats are mammals.

Classical logic adds only
So, abbreviate,
write only additions:

All cats are fish and
all fish are mammals,
therefore
all cats are mammals.

Argument does not only add,
it often removes elements.

We change our minds when
we are convinced we are wrong.

Classical logic does not
cover this case.

To classical logic our mind
is often contradictory.
So too is software.

When we find a contradiction,
we might attempt to remove it.

But, often
by meta-logical rules:

rules that avoid
certain lines of thought.

Logic develops rules of reasoning
with the intention that ...

If the premises are all true,
then the conclusions will be also.

If any conclusion is false,
then some premise will be also.

If the premises are false,

then the conclusion *might* be false,

but it could equally be true.

Everyone gives a variant
meaning to their words

If an argument depends
on the exact meaning,
then proof is personal, subjective.

Logic looks for
objective argument.

Argument that can be examined
and accepted by others.

Paradoxically,
it must not depend
on the meaning of the terms.

Classical logic research
found arguments
whose syntax was the same,
even when the semantics was not.

All cats are mammals and
all mammals are hairy,
therefore all cats are hairy.

All men are tall,
all tall people are strong,
therefore all men are strong.

The pattern is

All X are Y and all Y are Z,
so all X are Z.

Not always direct replacement, some allowance must be made for the rules of English grammar.

The truth does not depend
on the details of the meaning
of the inserted words.

If nothing else, this abbreviates
multiple arguments
in a single form.

Logic is like a puzzle:

I have some statements. If I have a statement of this form, and a related statement of this other form, then I may add a statement of yet another form.

Logic is like a puzzle:

A typical goal is that I must find a way to generate a statement with predefined properties.

Premise pattern :

{All X are Y, All Y are Z}

Conclusion pattern :

{All X are Z}

All our precise reasoning
is permutation and substitution.

Permutations and substitutions
are recorded as
pattern matching operations.

I match

All cats are wet to All Xs are Y,

by noting that X=cat and Y=wet.

Allowance must be made for English grammar.

All cats are wet, but
all fish (no ‘s’) are wet
Computer languages could be built this way,
but we choose not to.

Lecture 4

Logical Terms

Logic is a game we play with symbols, in practice this always comes down to terms.

From Chapter 2: Logic

Simple sentences in English are predicate and subject.
The cat is wet:
"The cat" is the subject, and
"is wet" is the predicate.

The cat is wet
can be rewritten as ...
IsWet(TheCat)

wet(cat)

w(c)

The software engineer can accept the conventional and transitory nature of language.

Define and redefine words
to suit the situation.

Words are tools.

They should not control us,
we should control them.

Pure string replacement.

```
replace   ]i   with   i]
replace   0i   with   1
replace   1i   with   i0
replace   [i   with   [1
```

Increment binary numbers

```
[1011]i -> [1011i] ->
[101i0] -> [10i00] -> [1100]

[111]i -> [111i] ->
[11i0] -> [1i00] ->
[i000] -> [1000]
```

Wildcards improve the power.

```
add(X0,Y0,0) -> add(X,Y,0)0
add(X0,Y0,1) -> add(X,Y,0)1
add(X0,Y1,0) -> add(X,Y,0)1
add(X0,Y1,1) -> add(X,Y,1)0
add(X1,Y0,0) -> add(X,Y,0)1
add(X1,Y0,1) -> add(X,Y,1)0
add(X1,Y1,0) -> add(X,Y,1)0
add(X1,Y1,1) -> add(X,Y,1)1
```

This is pure string replacement,
the brackets are not special,
but there is a problem.

the wild card might match across
parts of the expression.

```
add(add(10,1,0),add(1,0,0),0)

X = add(10,1
Y = 0),add(1,0,0)
```

Add respect for brackets to fix this.

In practice all precise symbolic reasoning is based on pure string substitution, but the sugar of wild-cards and punctuation makes a big difference to how efficient and natural the reductions are.

Correctly bracketed terms can
be used to construct,
without using meaning.
If R is a raw symbol,
and X and Y are correct.
Then R, (X), R(X), and (X, Y)
are all correct.

Correctly bracketed terms can

Define rabbits.
0 is a rabbit.
If r is a rabbit, s(r) is a rabbit.

Clearly, 0, s(0), s(s(0))
are all rabbits.

```
wait(x,0) -> x
wait(x,s(y)) ->
s(wait(x,y))
```

So,

```
wait(s(0),s(s(0)))
-> s(wait(s(0),s(0)))
-> s(s(wait(s(0),0))
-> s(s(s(0)))
```

This is addition.

Expression of logical proof explicitly in terms of term reduction on a collection of terms.

The state is a single term,
not a set.

No pre-existing set theory needed

Use implicit lists, and add rules
for permutation and duplication.

Permutation of sub terms is

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

```
(A and (B and C)) ->
((A and B) and C)
```

Duplication of sub terms

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

Implication `A => B` is the rule

```
A and X -> A and B and X
```

Lecture 5

Meta logic

Algebra is the manipulation of symbols, usually intended as a meta logic of some other symbol system.

From Chapter 3, Symbolic Algebra.

A data structure
can be defined concretely.
`0` is natural, and
if `x` is natural
then `s(x)` is natural.

And then operations
defined directly on them.

```
add(x,0) = x
add(x,s(y)) = s(add(x,y))
```

From this we can prove that

`add(0,0) = 0`

and

`add(0,s(0)) =`

`s(add(0,0)) =`

`s(0)`

A longer example is ...

```
add(0,s(s(0))) =
s(add(0,s(0))) =
s(s(add(0,0))) =
s(s(0))
```

What about
`add(0,s(s(s(s(s(0)))))`
What does the proof look like?
Each step moves one s from inside to outside the add.

We need 5 steps to get it to
`s(s(s(s(s(add(0,0))`
and one last step to get
`s(s(s(s(s(0)))))`

If there are n instances of s,
then n steps will get them all
outside the add,
and one more step will
reduce the add(0,0) to 0.

This is meta logic.

Talking ***about*** the proof.

Showing that a proof exists, rather than writing out the proof itself.

This is *mathematical induction*.

Informally: if the process works the first few times, and it seems that it would keep working the same way as long as we kept going, then we conclude that it works indefinitely.

In more formal terms

If `P(0)` is true, and
`P(i)` implies `P(i+1)`, then
`P(n)` is true for all natural `n`.

So we now see that

```
add(0,0)=0
```

and

```
add(0,y)=0 implies
add(0,s(y))=s(y)
```

So we claim that
add(0,y)=y for all y.

This is not a rule of
the original definition.

We claim we can find
a sequence of reductions
for each specific case

This is a *metaphysical* assertion
about the ***infinite*** behaviour
of the sequence of integers.

We cannot be sure
that it is correct.

This claim is a claim about the larger scale behaviour of the original rules.

It is a meta-logical claim.

The logic of logic.

The algebra of
a concrete data type

is the logic that is taken to
describe the behaviour of the
reductions that define that type.

There might be multiple algebras.

An algebra is just some system
to describe some behaviour
of some other system.

An algebra is complete if

all assertions about the reduction system can be produced from using the reductions defined in the algebra.

An abstract datatype is defined by giving a complete algebra for it, a typical task is to generate the concrete version given the abstract version.

We have integers,
but we want rationals.

We want numbers such as 1/2,
but what does it mean?

It is defined by the rule
`(1/2) * 2 = 1`.

For every integer a and
non zero integer b,
the number `(a/b)`
is defined by `(a/b)*b=a`.

Every pair `(a,b)`
defines a unique rational,
We can construct rationals
from pairs.

The laws we need are the rules of fractions:

```
a/b * c/d = ac/bd
```

just rewrite as

```
(a,b) * (c,d) = (a*c, b*d)
```

and so on.

Equality must be defined.

`a/b = c/d`

is defined to mean

`ad = bc`.

An important technicality.

It is the meaning of
the phrase ‘a=b’
that is defined,

not the meaning
of the symbol ‘=’

Let (a,b) mean

a + b $\sqrt{2}$

then

```
(a,b) + (c,d) = (a+c,b+d)

(a,b) * (c,d) =
(a*c+2*b*d,a*c+b*d)
```

In like manner,
any selected number can be give
any positive integer root.
Including the famous case of -1.

Start with 0 and 1.

Build the naturals by induction.
Their use is justified by
the behaviour of
real world computers.

Build the algebraics
by concrete extension.

There are no concrete reals.

Lecture 6

Set theory

Set theory is not a good thing on which to found mathematics,
Skolem.

From Chapter 3: algebra.

The idea of an abstract set
comes from
the idea of a set of spanners.

Each spanner is different.

A set of spanners is defined by
which spanners are in it,
and which are not.

Build a set
one spanner at a time,

no matter the order
they are picked

the resulting set is the same.

From 5 spanners,
32 possible sets can be made.

$$2 \times 2 \times 2 \times 2 \times 2 = 32$$

The set containing no spanners
is still a set.

Describe a set
by writing down the elements.

This gives the names order.
This allows multiple membership.
This is a list.

The list is prior to the set.

The list is the natural foundation of software.

To have a set we ***ignore***
the order and multiplicity.

We define a *different* equality
on a foundation of *lists*.

Let `cons(S,x)` be
the set the elements of `S`,
together with `x`.

The *definitive* operation on sets.

Start with set S,
put in x then y.

The result is the same if we
put in y then x.

```
cons(cons(S,x),y) =
cons(cons(S,y),x)
```

Start with set `S`
Put in `x`, and put it in again.

The result is the same
if we put it in just once.

```
cons(cons(S,x),x) =
cons(S,x)
```

The rules

```
cons(cons(S,x),y) =
cons(cons(S,y),x)

cons(cons(S,x),x) =
cons(S,x)
```

define finite sets.

The procedural version is

```
cons(S,x) ; cons(S,y) ==
cons(S,y) ; cons(S,x)

cons(S,x) ; cons(S,x) ==
cons(S,y)
```

A typical implementation
uses a list,

Sort the list and
remove duplicates
after each operation.

The basic property
of a set is membership.

```
x in S.

x in cons(S,y)
iff x==y or x in S
```

If we know for each thing
whether it is in the set
then we know the set.

This is a policy, not observation.

By definition sets with
the same elements
are the same.

This is called
the axiom of extension,

but it is just the essence of
what is meant by the word ‘set’.

Union:

given two sets
build a set containing
all the items in each of the sets.

```
union(S,cons(X,x)) =
cons(union(S,X),x)
```

```
(x in union(A,B)) ==
(x in A) or (x in B)
```

For any finite boolean expression there is an equivalent set constructor.

Every finite sum has a value.
The value does not depend on the order in which the elements are added.

$1+2+3 = 6$

$3+2+1 = 6$

$2+1+3 = 6$

What is the value of
an infinite sum?

1+1+1+1+ ...

does not have any sum.
Unless we invent new numbers,
such as infinity.

1-1+1-1+1 ...

Can be grouped in 2 ways.

(1-1)+(1-1)+ ... = 0
or
1 - (1-1) - (1-1) - ... = 1

Which is the correct value?

The resolution requires
the development of
a theory of infinite sums.
The main constraint is that this theory must agree with the theory of finite sums, when speaking about finite sums.

Such a theory of infinite sums
is a *conservative extension*
of the theory of finite sums.
There could be many such theories
they could conflict.

There is no concrete definition
of infinite sums,
only an abstract logic.
Infinite sums, in general, cannot
be represented in software.

This is a conservative extension.

The logic of the extension always agrees with the original logic, when the original logic says something.

General infinite sums
do not exist as a concrete form.

There is no finitary system
with all the right properties.

They exist only in the sense that
we can reason with the meta logic.

But, the logic itself exists,
is concrete, and is useful.

Thus also

infinite lists

and infinite sets.

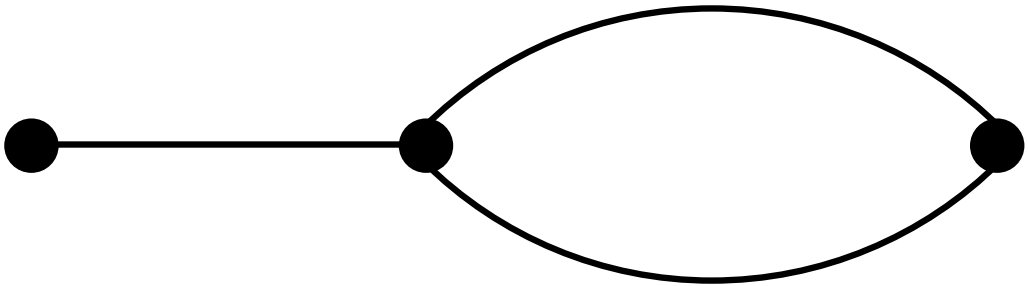
Lecture 7

Network Diagrams

Diagrams can be reasoned with and reasoned about.

From Chapter 4: Diagrams.

A lot of diagrams are networks.



Not a mathematical graph, which allows at most one line between any two points.

Network approach to a puzzle.

A farmer wants to cross a river
using in a small boat,
taking a goat, a wolf, and a lettuce.

Without the farmer,

the wolf would eat the goat,
the goat would eat the lettuce.

The farmer can only take one
thing with him on the boat.

With three items,
there are 8 possible
distributions to the banks,

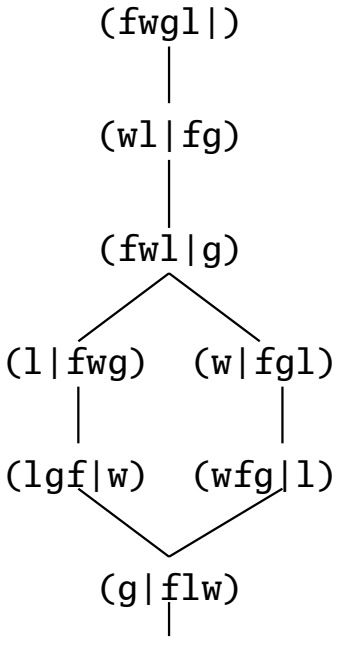
With the farmer, 16.

On each move, the farmer and
at most one other item
is transferred.

The initial state is
`(fwgl|)`

Possible next states are

`(wg|fl)` .. the wolf eats the goat.
`(wl|fg)` .. safe
`(gl|fw)` .. lettuce is eaten.



Build a network.

The data is the same, but easier for a person to see the way.

And
it can be automated.

The action of the network for the farmer puzzle uses the principle of a device moving on a network. The nodes are the state of the puzzle.

All software
is based on this principle.

An addition is done on a grid.

1	1	0	0
0	4	3	0
0	1	2	4
1	1	0	4

With 16 cells and 10 digits,
there are `pow(10,16)`
ways to fill in the grid.

The ways of filling in the cells
are the states of the computation.

The ways of changing the contents
are the transitions.

The states could be written
within circles
on a very large piece of paper.

The transitions could be expressed on lines drawn between the two states.

There are rules that give

exactly

the changes to the grid
based on the current state.
These are the transitions.

This combination of states and transitions is a state machine.

It can be represented on paper with lines and nodes,

as a network.

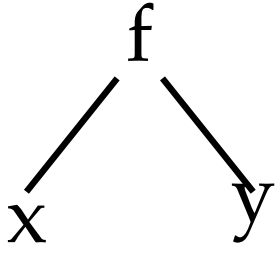
A neumann machine operates in the same way.

The digital memory is the same as the paper grid.

In each case the memory can be increased without clear limits.

The memory
can be a network.

If the state is a term `f(x,y)`
then this network encodes it

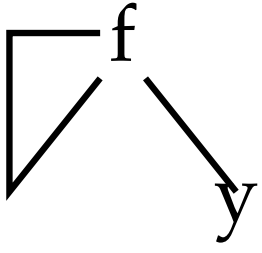


Replacement and permutation of subnetworks performs the same duty as similar work on terms.

An infinite term

```
f(f(f(...,y),y)
```

can also be encoded finitely.

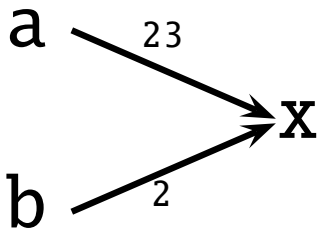


Only some infinite terms can be encoded in this way.

But, reductions on networks give such infinite terms their expected properties.

Similar networks can be used to encode simultaneous equations.

$$x = 23a + 2b$$



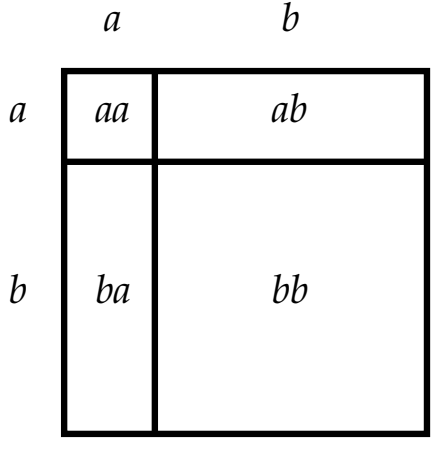
Lecture 8

Solid Algebra.

Solids have
discrete and algebraic aspects

From Chapter 4: Diagrams.

Diagrams can be used to reason about algebra.



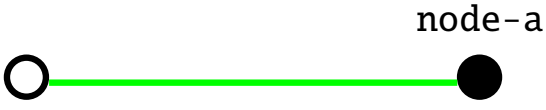
Therefore $(a + b)^2 = a^2 + 2ab + b^2$.

Such diagrams are an aid
to reasoning in the algebra.

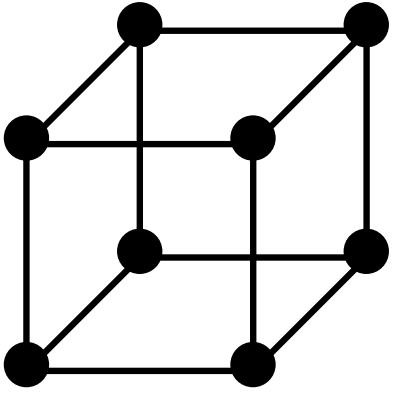
They are not a proof of
properties of physical space.

About physical space,
these conclusions might be false.

The basic diagram is a two dimensional drawing with regions, lines, and nodes, decorated with colour, shape, texture, text, etc.



Sometimes it is seen as solid,
by the human visual system.



Nodes,
edges,
regions,
solids.

With training,
some people can think about
four dimensional diagrams.

But, always a geometric scheme.

The parts are related by closeness.

Elements in a diagram are
recognised by their
approximate shape and position.
Like letters.
A square is a square,
even if it is badly drawn.

Snap-to-grid.

All precise diagrams,

including textual elements,
are made this way.

Draw a planar network.

Draw lines and dots on paper.
Each end of each line is a dot.
Perhaps the same dot.
No line crosses another.

The regions outlined are the faces.

The simplest network
has a single dot,
so it has one face.

faces - lines + nodes =
1 - 0 + 1 = 2

Add a line in a loop,
another face.

$2 - 1 + 1 = 2$

faces - lines + nodes

is the Euler constant for the plane. Every network drawn on a plane gets the same value.

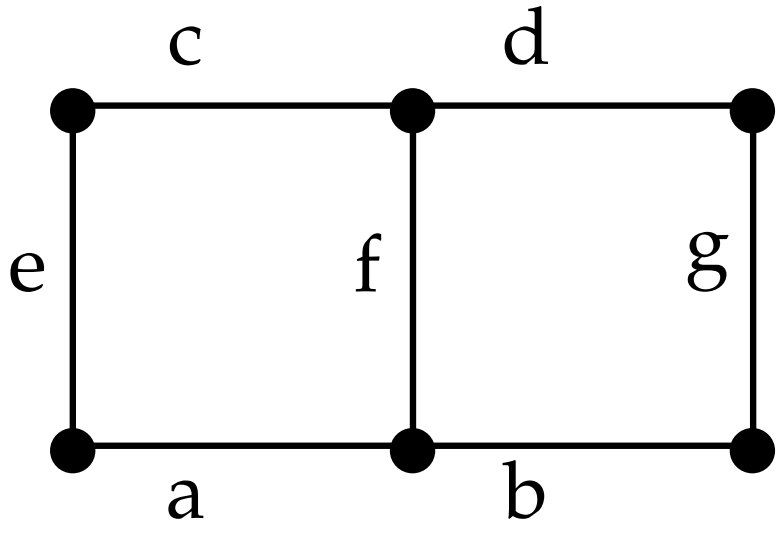
On more complex surfaces
like a torus
a rule is all faces must be
one sheet of rubber with no holes.

The Euler constant for different surfaces is different.

Try it with a torus.

The faces are very important.

One way to describe a network is to describe the faces and how they are stitched together.



Described as
{`aecf,fdgb,gdceab`}.

The edges of the faces are
taken in a standard order:
in this case, clockwise.

This means, the face is
to your right as you walk
around the boundary of the face.

The outside face seems to be going counter clockwise.
But, from inside the outer face, it is clockwise.

Application:

The equations that are set up
to solve for the state of an
electric circuit are selected
one per face, from the diagram.

When the diagram is non planar, select enough loops to cover the network, and call them faces.

The details of this
leads to vector spaces
and to homology

Lecture 9

Object Diagrams

Aspects of object databases can be expressed as diagrams.

From Chapter 5: UML.

Object systems are databases.

A network of objects.

An object is
a record with connections.

A record is a named tupple.

`(x=5,y=6)`, not `(5,6)`

Each entry is a field,
or attribute.

A relational database is
a collection of records
grouped into tables.

Records in the same table
share a type signature.

The type of a field can be another record.

```
(x=(1,2),y=6)
```

We say the type of x is the name of a table with the same type signature.

When the type sig of table A
includes a variable X
whose type is table B.

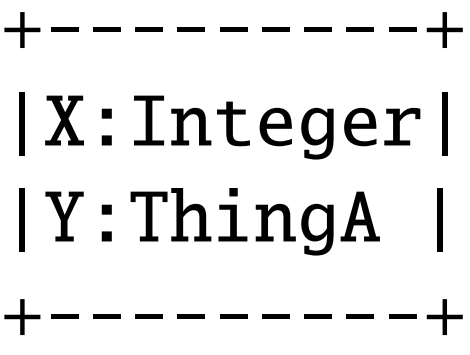
Draw a line from A to B, label it X.

```
A ----[X]----> B
```

In practice, many variables have type Integer.

So the diagram would be clogged.

Instead, make each node a box,
and write the type signature
in the box.



Put this together in a network.
This is the basis of a class diagram.

What makes an object database different from a relational one

is indirect reference.

If the data is not in the record then look for it somewhere else using an inbuilt search strategy.

In object systems this is called inheritance.

It is a special case of the Codasyl structure.

Put links into
the class diagram
that stand for the direction
a search should proceed.

When field value is missing
from a record,
perform a search
of the indirect reference
network structure.

A record can store a function.

```
(mag="x^2+y^2",x=6,y=5)
```

Often this is called
a method or an operation.

An operation can
modify the object.

```
(setx="x=23",x=6,y=5)
```

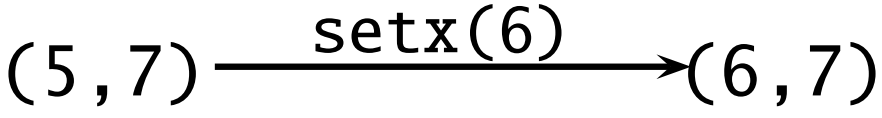
State diagram

An object has state
that can be changed
by operations on it.

For example,
the action `setx(6)`
might map

state `(x=5,y=7)` to `(x=6,y=7)`.

Each call to the object
is an input to a state machine.



Place the states of an object
on a piece of paper, and
draw links labeled by
a method call,
from the state on
which it was called,
to the state that it produces.

This is a statechart diagram.

It is a state machine.

Typically, the states of the object are factored, into a finite number of options.

An operation can change the database structure.

```
new record(x=6,y=45)
```

Activity diagram is
kin to a petri net.
Multiple points of activation.
Explicit decision elements.
Fork and join.

Lecture 10

UML Diagrams

The UML is an OMG standard defining some families of object diagrams.

From Chapter 5: UML.

UML is an open standard
by the OMG.
It defines several types
of diagrams.

The UML diagrams are mostly relation diagrams and petrinets.

The diagrams might be factored.

Relational UML diagrams:

Object diagram
Class diagram
Package diagram
Usecase diagram
Deployment diagram

Petrinet UML diagrams:

State diagram
Activity diagram
Sequence diagram
Timing diagram

Object diagram:
Nodes are objects
Links show relation by value

Class diagram:
Nodes are classes
Links show relation by type.

Links also show inheritance

Package diagram

Nodes are units of deployment.
Links are dependencies.

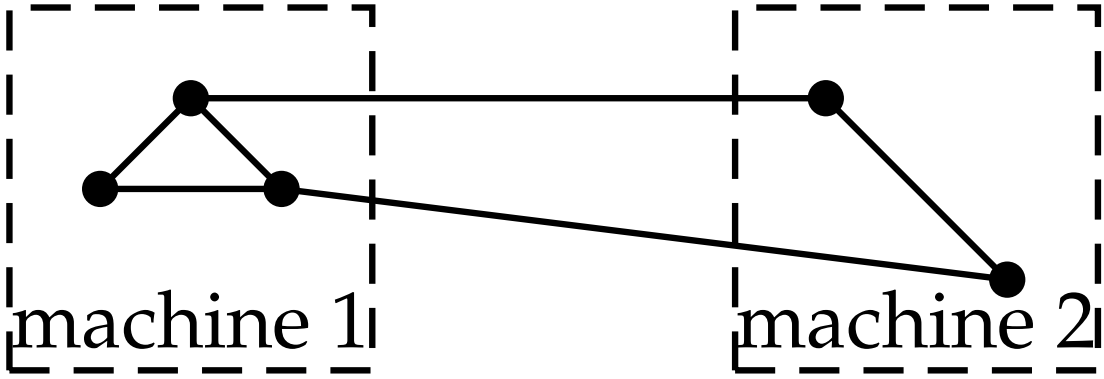
Factor an object diagram to get a class diagram.

Factor a class diagram to get a package diagram.

Deployment Diagram:

Nodes are software tasks.
Links are protocols.
Links are middleware.

Nodes are factored
into machines.


machine 1
machine 2

Usecase diagram

Nodes are actors and usecases
Links show involvement
Links also show special cases

State Diagram

Nodes are objects

Links are decorated with methods.

Tokens are points of execution.

Activity Diagram

Is a factored state diagram,
with extra syntax,
and variable token count.

Activity Diagram

Nodes are places in a program.

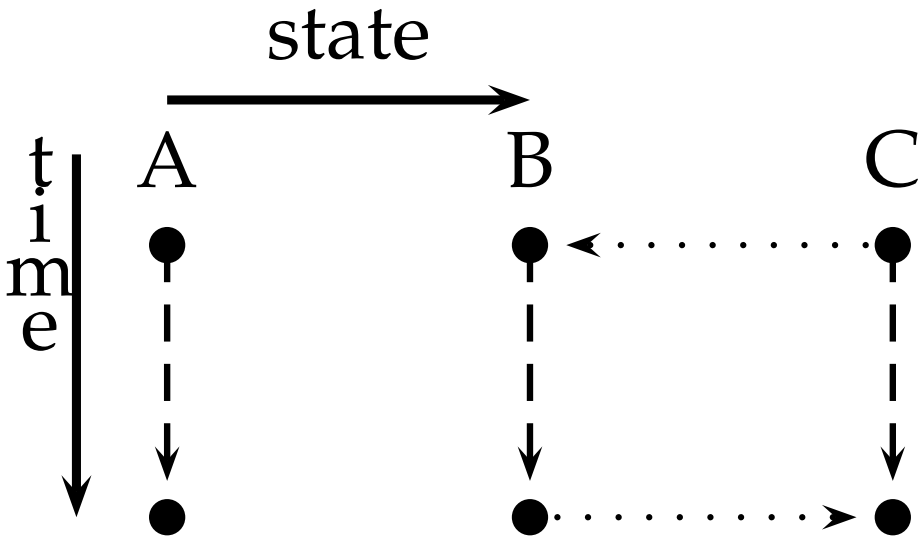
Tokens are points of execution.

Sequence Diagram

Gives the history of transitions.
A trace of a protocol.

Nodes are stretched into lines.

Down the page is a time-axis.

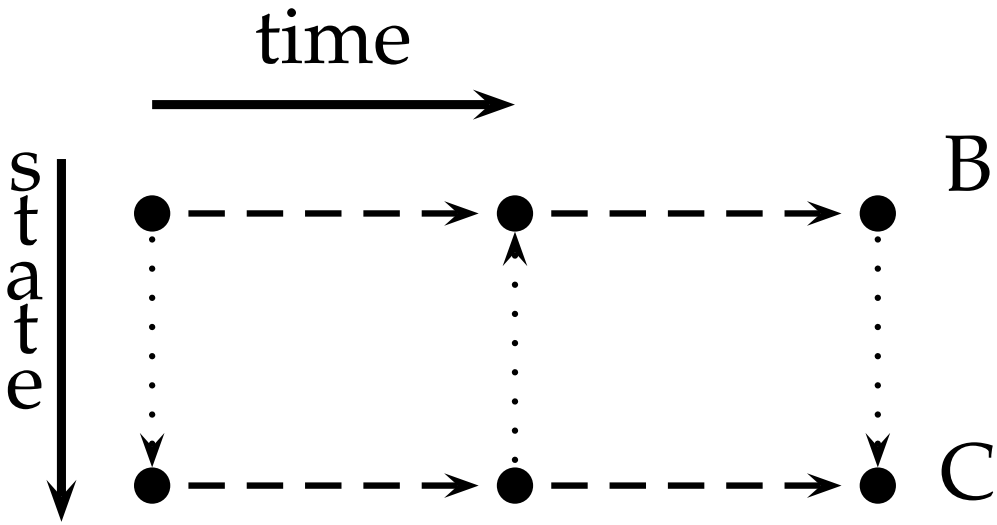


A token moves from state C to B and later moves back to B.

Timing Diagram

For the most part it is
a sequence diagram on its side

Comforms to
electronic engineering practice.



A token moves from B to C
then back to B,
and then to C again.

A timing diagram might also have link-to-link edges

showing cause and effect

between transitions.

Lecture 11

OCL Combinators

OCL is a language with inbuilt finite set theory operations.

From Chapter 6: OCL.

OCL is a language for writing

pure mathematical expressions.

Such as: `y = 2*x + z`.

OCL includes Integers and Reals,
But its core is

pure finite-set theory,

as well as bags and sequences.

Except for some partial
special case exceptions,

such as Integers,

infinite sets do not exist in OCL.

Basic arithmetic uses traditional notation.

`x+y`, `x-y`, `x*y`, and `x/y`.

Expressions such as

```
3*x*x + 4*y + 3*x*y
```

are very clumsy in object notation.

Extended operations use object notation

```
23.mod(5) = 3
```

OCL also has strings,

with the common
standard operations.

```
"this".concat("that") =
"thisthat"
```

For collections:

OCL has

finite sets,
finite bags,
finite sequences,
finite ordered sets.

Take the union of two sets
using object notation.

`A->union(B)`
is an expression for
the union of `A` and `B`.

Note the use of ->
instead of a dot
when the object is a collection.

This is a syntactic convention
the dot and arrow have the same
core meaning.

OCL has no commands
and no control flow.

There are no loops.

There are no assignments.

All variables are parameters.

Given a list X of integers,

get the sum as `X->sum()`

This is a instance of an iterator.

Add a list like this ...

```
sum (1,2,3,4)
=  1 + sum (2,3,4)
=  3 + sum (3,4)
=  6 + sum (4)
= 10 + sum ()
= 10 + 0
= 10
```

There is an implicit 0.

```
   0 + sum (1,2,3,4)
=  1 + sum (2,3,4)
=  3 + sum (3,4)
=  6 + sum (4)
= 10 + sum ()
= 10 + 0
= 10
```

Absorb this into the sum function

```
sum(0,(1,2,3,4))
= sum(0+1,(2,3,4))
= sum(1+2,(3,4))
= sum(3+3,(4))
= sum(6+4,())
= 10
```

The general idea is

```
sum(s, cons(x,X))
= sum(s+x,sum(X))
```

In function notation, this is

```
sum(s,cons(x,X))
= sum(add(s,x),sum(X))
```

The iterator concept:

```
iterate(f,e,empty) = e

iterate(f,e,cons(x,X)) =
iterate(f,f(e,x),X)
```

The OCL syntax for an iterator.

```
X->(x,t=e|f(x,t))
```

The value of an OCL expression is the result that would be returned by performing the implied actions.

So, an iteration over
an Infinite set
is an error.

For example,
there is a ‘forall’ iterator
that checks a predicate on
all elements of a set.

```
forall(Integer, P)
```

Is not valid OCL.

The OCL standard gives
an ‘indeterminate’ value to all

non-terminating calculation.

Lecture 12

OCL Scripts

OCL describes a program written in a target language.

From Chapter 6: OCL.

OCL describes constraints,

on object systems,

using the theory of finite sets.

OCL is declarative.

It does not say “do this”,

it says “this is so”.

OCL syntax for saying
a class exists:

```
context C
```

OCL syntax for saying
a class has an attribute
of a given type:

```
context c::x :  integer
```

OCL syntax for saying
a class has an operation
with given parameter types
and return type:

```
context
C::f(integer x) :  integer
```

An OCL script
describes constraints
on a program which is
written in a target language.

To be concrete, we will use
Java as the target language.

Sometimes
an OCL constraint on attributes
implies that the Java
must have a method.

```
context
a:integer,
b:integer
inv:  a = sqr(b)
```

This says that at all times the value of ‘a’ is the square of the value of ‘b’.

An OCL script describes constraints on an object database. These constraints are of two kinds. A restriction on a value, and a restriction on a transistion.

Conceptually, a value could be an Integer, or a function.

To restrict an integer value

we can say `x>6`, for example.

This eliminates many options,
but leaves an indefinite number.

A constraint such as `x=6` eliminates all but one option.

But, it is not otherwise distinct from any other constraint.

To restrict an integer function

we might say `f(x)>0`, or

`f(x+y)=f(x)*f(y)`,

for example.

A common concept is an invariant.

`p(f(x)) = p(x)`

Where `p` is some property of numbers.

For example, `p(x)` might mean

the parity of `x`,

so `p(f(x)) = p(x)`

means that `f` conserves parity.

In OCL,
parity conservation could be

```
context
C::f(Integer x) :  Integer
post:
parity(result) = parity(x)
```

But, OCL does not handle well constraints using multiple calls.

```
f(x+y) = f(x)*f(y)
```

The OCL post constraint is used more for expressing the state that the database has achieved after the call to the operation.

A pre constraint is a constraint on the values of the arguments.
Effectively it is a type constraint on the argument list.
But, it might be easier to express it as a pre constraint on the function.

If a function makes sense only for positive integers ...

```
context:
C::f(Integer x) :  Integer
pre:  x>0
```

Polymorphism:
A pairing of

a pre and post constraint.

If the pre condition
is true of the call,

then the post condition
is true of the return.

Lecture 13

Zed Core

Zed contains a rich expression language.

From Chapter 7: Zed.

Type declaration is constraint.

```
unsigned int x;
```

means the same as

```
int x; x>=0;
```

except that the latter
is not C-semantics.

In semi-mathematical notation

```
x in Integer and x >= 0
```

In mathematical notation

$x \in \mathbb{Z} \land x \geq 0$

For functions this becomes

```
int f(int x);
```

In mathematical notation

$f : \mathbb{Z} \to \mathbb{Z}$

In Zed notation

$f : \mathbb{Z} \pfun \mathbb{Z}$

The arrow says
f maps int to int.

The bar says that
maybe f does not return
a value for every int.

Zed notes 3 aspects of functions so has eight related operators.

partial function	$\pfun$	total function	$\fun$
partial surjection	$\psurj$	toal surjection	$\surj$
partial injection	$\pinj$	total injection	$\inj$
partial bijection	$\pbij$	total bijection	$\bij$

total/partial
is the domain covered ?

surjective/not
is the range covered ?

injective/not
is the reverse a function ?

Zed is a notation for sets.

Functions and relations are explicity special types of sets.

Much of the notation is from classical mathematics.

Zed has many set operators

Proper Subset	$A \subset B \equiv A \subseteq \wedge A \neq B$	strictly smaller set
Superset	$A \supseteq B \equiv (x \in B \Rightarrow x \in A)$	bigger set
Proper Superset	$A \supset B \wedge A \neq B$	strictly bigger set
Power	$\mathbb{P}(A) = \{B \mid B \subseteq A\}$	all subsets
Power	$\mathbb{P}_1(A) = \{B \mid B \subseteq A \wedge B \neq \phi\}$	non empty subsets
General Join	$\bigcup A = \{x \mid \exists a \in A \bullet x \in a\}$	join of a set of sets
General Meet	$\bigcap A = \{x \mid \forall a \in A \bullet x \in a\}$	meet of a set of sets
Integer Interval	$a..b = \{n \in \mathbb{Z} \mid a \leq n \wedge n \leq b\}$	integers from a to b
Selection	$\{D \mid S \bullet E\}$	selected elements
Cartesian Product	$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$	set of all pairs

A function is a set of ordered pairs.
Think of `f(x){return x*x}`
as {(0,0),(1,1),(2,4) ... }
the list of all pairs of input and output.

The roots of 1 are +1 and -1,

so the list for $\sqrt{}$

includes (1,-1) and (1,+1)

$\sqrt{}$ is a
multi-valued function

A multi-valued function is
usually called a relation.

Relations are any
set of ordered pairs.

The meet of relations
is another relation.

Function usually means
single valued.

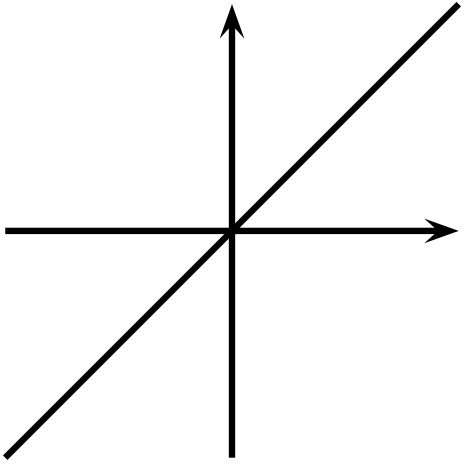
A set of ordered pairs
But for each first element
there is at most one second

The meet of two functions
is another function.

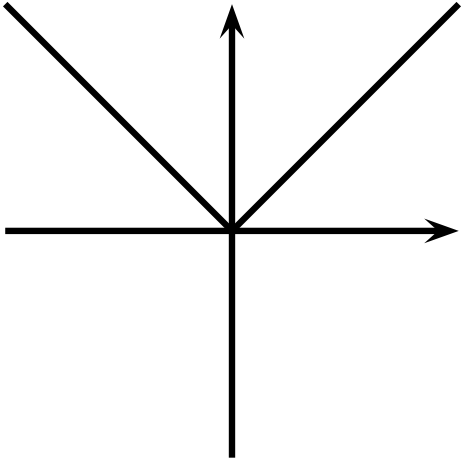
Functions are relations, so
the union of two functions
is a relation.

But the union of two functions
might not be a function.

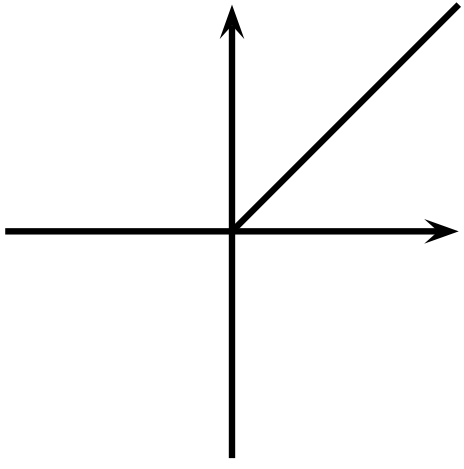
define id(x) = x.



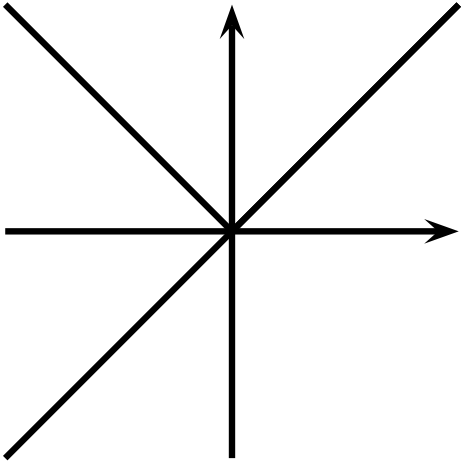
define abs(x) as the absolute value



id meet abs is unsigned id.



id join abs is not a function



The natural domain
of a relation is the set
of all first elements of
its ordered pairs.
The natural range is the
set of all second elements.

A relation might be declared with a domain or range larger than its natural domain or range.

The declared domain of
`int f(int x){return 3/x;}`
covers all ints,

but no value is defined for 0,
0 is not in the range of f.

The natural domain of f
is all non-zero ints.

The *declared* range of
`int f(int x){return 3*x;}`

is int, but it can only return multiples of 3.

Its *natural* range is
[... -9, -6, -3, 0, +3, +6, ...]

Lecture 14

Zed Scripts

Zed is intended for use by humans.

From Chapter 7: Zed.

The differences between Zed expressions and classical mathematical set theory expressions are superficial.

Zed is a set-theory language.

That is the small of Zed.

An entire program can be described by a Zed expression.

The large of Zed are Z-scripts that exist mostly to provide encapsulation.

A Z-script defines a set of tupples of a consistent type signature

A table.

This is the principle
of a database language,
but Z has stronger
virtual tables

A function `f(x)=x*x`
is an infinite table:

f	x
0	0
1	1
2	4
:	:

Store two functions in a table by storing all the combinations.

f	x	g	y
0	0	0	1
0	0	1	2
:	:	:	:
1	1	0	1
1	1	1	2
:	:	:	:

This is the concept of a free-join

f	x
0	0
1	1
2	4
:	:

$\times$

g	y
0	1
1	2
2	3
:	:

Z defines a table as a
free join of other tables.

Define the square function as

```
sqr == [x:int; y:int | y=x*x]
```

This is almost like

```
int f(int x){return x*x;}
```

But the square-root
can be defined as

```
sqrt==[y:int; x:int | x=y*y]
```

Which you cannot do in C.

These expressions are called *schemas* in Zed.

Zed also admits a semi-graphical syntax

Given two schemas

```
sqrt==[y:int;x:int | x=y*y]
sqr==[x:int;y:int | y=x*x]
```

The x and y in sqrt are different from x and y in sqr.

They are local variables
`sqrt.x` and `sqr.x`.

But, they are on an export list.

Use one schema in another ...

```
prog =
[
sqrt ; a:ℤ; b:ℤ
|
a=x ; b=a+2*y
]
```

The export from sqrt become local in prog.

But, sqrt is a copy,
there is no dynamic link
between prog and sqrt.

A schema has no state.

Including schema A
in schemas B and C
does not create
a means of communicating
between B and C.

Schemas allow the larger scale structure of the program to be described.

To define a function
without exporting the variables
define them inside something
that is not a schema.

```
math ==
[
f :  ℤ ⇸ ℤ
|
∀ x∈ℤ • f(x) = x*x
]
```

The quantifiers ∀ and ∃

define truely local variables.