

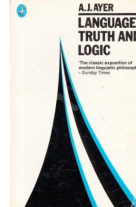


Informatics 1A
Computation and Logic 1

introduction
information
properties and predicates

truth
logic
language


Michael P. Fourman
@mp4man

This course provides a first glimpse of the deep connections between computation and logic.

communication
harigō ni mojiagari
noun

the imparting or exchanging of information by speaking, writing, or using some other medium.



Natural languages are often ambiguous, verbose, or imprecise.

To study, and to understand Informatics, you will need to learn some skills of clear, concise, and unambiguous communication.

In this course you will study some simple examples of information and computation (the processing of information), and use these to develop skills of understanding and communication that prepare you for what is to come.

2

To work together we need to learn to communicate.

We will study some formal systems that allow us to communicate clearly, concisely, and unambiguously. They will allow us to describe and reason about our systems. They may help you to communicate less ambiguously in natural language.

It also turns out that using such languages we can get computers to do much of our reasoning.

Logic concerns properties
of things



big blue triangle



small red disc

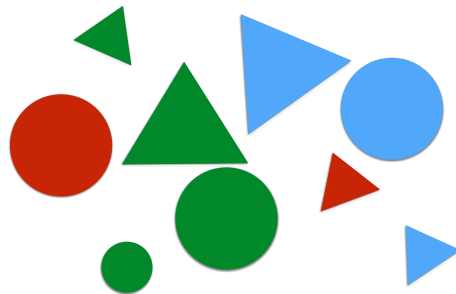
3

We will use logic to describe sets of states in terms of their properties.

We consider a very simple 'universe', where everything is either red, blue, or green, either big or small, and either a triangle or a disc.

Moreover, there is at most one thing of each kind: at most one big blue triangle, and so on ...

some **things** — our (first) *universe*
for this course, our universes will be finite



If everything is

either red or blue or green

and

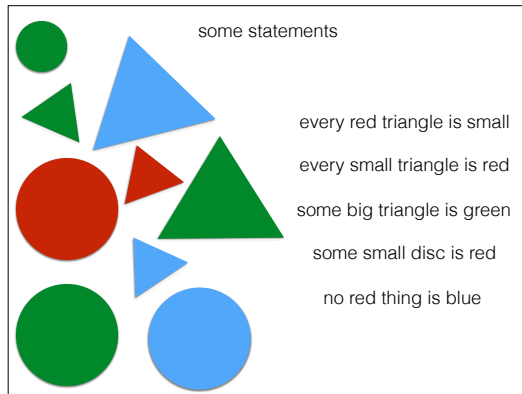
either small or big (not small)

and

either disc or triangle (not disc)

then we have $12 = 3 \times 2 \times 2$ possible combinations of three features.

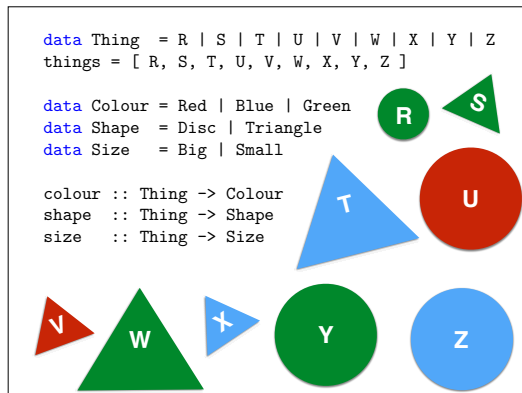
Only some of these combinations appear in our universe.



Here are some statements we might make about this universe.

Some true, some false. For this small universe it is easy to work out which are true, just by looking at the picture.

Our first task is to work out how we can turn such statements into Haskell code that will produce the correct answers.



To code this problem in Haskell, we first introduce a type to represent our things.

We use the data keyword to introduce this type and a name for each thing in our collection.

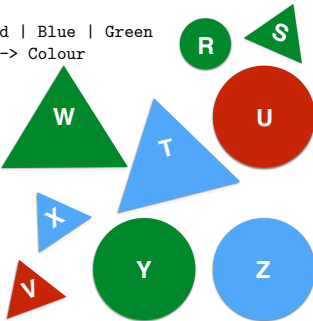
The names start with uppercase letters — we call them *constructors*.

We can only introduce a value of type Thing by using one of its constructors.

We can describe this universe in terms of three different features. For each feature we introduce a type and a function that gives the feature value for each Thing.

```
data Thing = R | S | T | U | V | W | X | Y | Z
things = [ R, S, T, U, V, W, X, Y, Z ]
```

```
data Colour = Red | Blue | Green
colour :: Thing -> Colour
colour R = Green
colour S = Green
colour T = Blue
colour U = Red
colour V = Red
colour W = Green
colour X = Blue
colour Y = Green
colour Z = Blue
```



Constructors can be used in function declarations. It's a bit tiresome, but we can define the function by listing its value for each thing.

If we wrote

```
colour r = Green
```

then `r` would act as a variable.

This single line would say that everything is Green — not what we want.

Variables begin with lowercase ; constructors begin with uppercase.

logic -ology It's all greek to me!

-λογία • (-logiá) *f* (genitive -λογιάς);

1. Base for nouns denoting the study of something, or the branch of knowledge of a discipline.

The suffix *-ology* is commonly used in the English language to denote a field of study. Wikipedia gives [hundreds of examples](#).

— here is a small selection of those starting with *a*

[acarology](#)

The study of mites and ticks.

[acidology](#)

The study of grasshoppers and locusts

[aerolithology](#)

The study of meteorites.

[agathology](#)

The science or theory of the good or goodness.

[agrology](#)

The science and art of agriculture.

[anesthesiology](#)

The study of anesthesia and anesthetics.

[arachnology](#)

The scientific study of spiders and related animals.

[aristology](#)

The art or study of cooking and dining.

If we abstract away from the discipline to find universal laws of reasoning, logic is what remains.

Logic is the science of *pure reasoning*

reasoning that doesn't depend on the subject matter.

```
data Thing = R | S | T | U | V | W | X | Y | Z
things = [ R, S, T, U, V, W, X, Y, Z ]
```

We describe the universe:
it is **true** that
 T is blue
 U is red
 R is a disc
it is **false** that
 Z is a triangle
 R is big
 U is small

Having three features of different types — makes it difficult to separate the logic from the notions of colour, shape and size.
To study logic we use a different representation — we want a logic that is independent of the subject matter.

```
data Thing = R | S | T | U | V | W | X | Y | Z
things = [ R, S, T, U, V, W, X, Y, Z ]
```

```
data Bool
Constructors
  False
  True
```

We will describe the universe by defining (in Haskell)
predicates :: Thing -> Bool
 isRed, isGreen, isBlue,
 isDisc, isTriangle
 isBig, isSmall

We have more predicates than properties, so it looks more complex, but they all have the same type so in this way this representation is simpler.

Any tools we produce for reasoning about this situation will work for any other (finite) universe of Things with predicates :: Thing -> Bool.

```
data Thing = R | S | T | U | V | W | X | Y | Z
things = [ R, S, T, U, V, W, X, Y, Z ]
```

```
isRed    :: Thing -> Bool
isBlue   :: Thing -> Bool
isGreen  :: Thing -> Bool
isDisc   :: Thing -> Bool
isTriangle :: Thing -> Bool
isSmall  :: Thing -> Bool
isBig    :: Thing -> Bool
```

To code this problem in Haskell, we first introduce a type to represent our things.

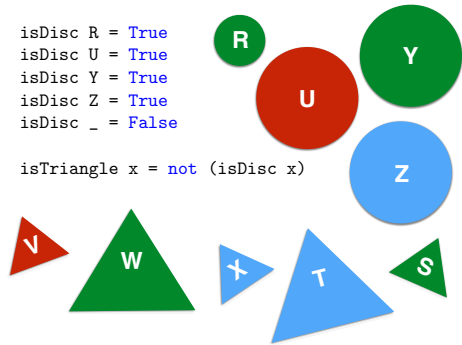
We use the data keyword to introduce this type and a name for each thing in our collection.

Then we will define a number of Boolean-valued functions to complete our representation.

```
data Thing = R | S | T | U | V | W | X | Y | Z

isDisc R = True
isDisc U = True
isDisc Y = True
isDisc Z = True
isDisc _ = False

isTriangle x = not (isDisc x)
```



Here we define isDisc by enumeration. The logic for isTriangle is simple.

Operations

```
data Bool = True | False
```

```
(&&) :: Bool -> Bool -> Bool | infixr 3
```

Boolean "and"

```
(||) :: Bool -> Bool -> Bool | infixr 2
```

Boolean "or"

```
not :: Bool -> Bool
```

Boolean "not"

True	T
False	⊥
not	¬
&&	∧
	∨

13

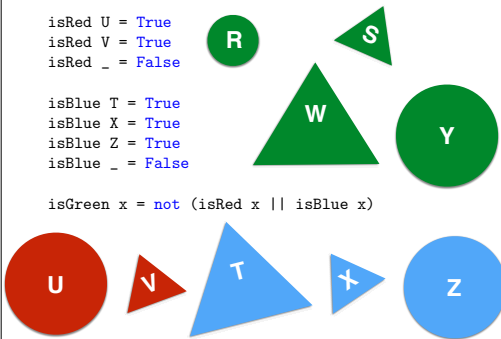
These operations will suffice for a (short) while.

```
data Thing = R | S | T | U | V | W | X | Y | Z
```

```
isRed U = True
isRed V = True
isRed _ = False
```

```
isBlue T = True
isBlue X = True
isBlue Z = True
isBlue _ = False
```

```
isGreen x = not (isRed x || isBlue x)
```



We list the red things, and say each one isRed — everything else is not.

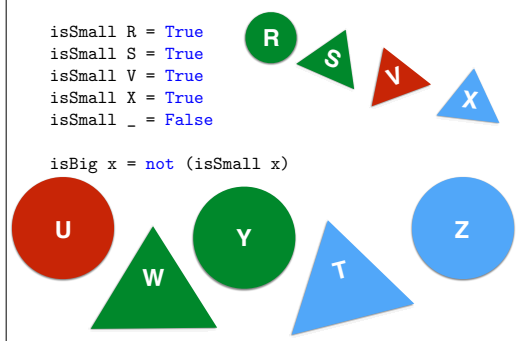
Similarly for isBlue.

We can use a little boolean logic to define isGreen. || is Boolean 'or'


```
data Thing = R | S | T | U | V | W | X | Y | Z

isSmall R = True
isSmall S = True
isSmall V = True
isSmall X = True
isSmall _ = False

isBig x = not (isSmall x)
```



This example is similar.




```
> [ x | x <- things, isRed x, isTriangle x ] -- redTriangles
[V]


> [ x | x <- things, isSmall x, isTriangle x ] -- smallTriangles
[S,V,X]

> [ x | x <- things, isBig x, isTriangle x ] -- bigTriangles
[T,W]

> [ x | x <- things, isSmall x, isDisc x ] -- smallDiscs
[R]
```



List comprehensions allow us to pick out subsets of things with particular combinations of properties.




```
> [ x | x <- things, isRed x && isTriangle x ] -- redTriangles
[V]

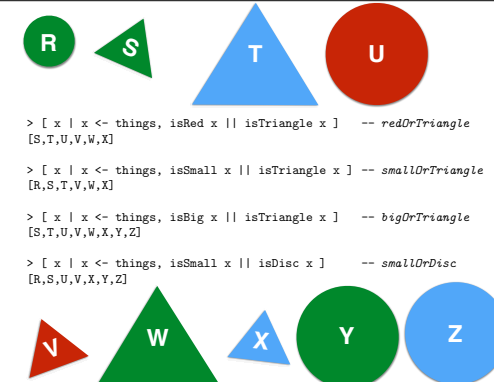
> [ x | x <- things, isSmall x && isTriangle x ] -- smallTriangles
[S,V,X]

> [ x | x <- things, isBig x && isTriangle x ] -- bigTriangles
[T,W]

> [ x | x <- things, isSmall x && isDisc x ] -- smallDiscs
[R]
```



Using && in place of some commas gives the same results.
Note that is now easy to see that every red triangle is small,
but no small triangle is red.



```

> [ x | x <- things, isRed x || isTriangle x ] -- redOrTriangle
[S,T,U,V,W,X]

> [ x | x <- things, isSmall x || isTriangle x ] -- smallOrTriangle
[R,S,T,V,W,X]

> [ x | x <- things, isBig x || isTriangle x ] -- bigOrTriangle
[S,T,U,V,W,X,Y,Z]

> [ x | x <- things, isSmall x || isDisc x ] -- smallOrDisc
[R,S,U,V,X,Y,Z]

```

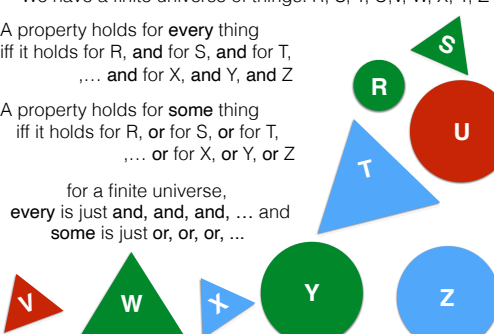
using || instead of && gives different results

We have a finite universe of things: R, S, T, U, V, W, X, Y, Z

A property holds for **every** thing
iff it holds for R, **and** for S, **and** for T,
... **and** for X, **and** Y, **and** Z

A property holds for **some** thing
iff it holds for R, **or** for S, **or** for T,
... **or** for X, **or** Y, **or** Z

for a finite universe,
every is just **and**, **and**, **and**, ... **and**
some is just **or**, **or**, **or**, ...



We can check whether a property holds for every thing, or for some thing, in a finite universe by checking whether it holds for each thing in turn.

We can look at things and their properties

```

-- red triangle is small?
> [ (x, isSmall x) | x <- things, isRed x, isTriangle x ]
[(V,True)]

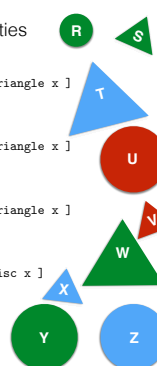
-- small triangle is red?
> [ (x, isRed x) | x <- things, isSmall x, isTriangle x ]
[(S,False),(V,True),(X,False)]

-- big triangle is green?
> [ (x, isGreen x) | x <- things, isBig x, isTriangle x ]
[(T,False),(W,True)]

-- small disc is red
> [ (x, isRed x) | x <- things, isSmall x, isDisc x ]
[(R,False)]

-- red thing is blue?
> [ (x, isBlue x) | x <- things, isRed x ]
[(U,False),(V,False)]

```



going back to &&, we can make it even easier to check properties by getting Haskell to give the property alongside each thing. Here we see immediately that there is only one red triangle, V, and it is indeed small. — so every red triangle is small. On the following line, we see there are three small triangles, only one of which is red;

so it is not the case that every small triangle is red, but it is the case that some small triangle is red.

```
-- every red triangle is small
> and [ isSmall x | x <- things, isRed x, isTriangle x ]
True

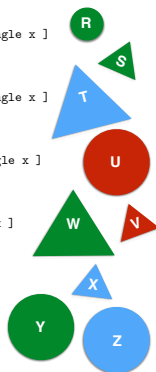
-- every small triangle is red
> and [ isRed x | x <- things, isSmall x, isTriangle x ]
False

-- some big triangle is green
> or [ isGreen x | x <- things, isBig x, isTriangle x ]
True

-- some small disc is red
> or [ isRed x | x <- things, isSmall x, isDisc x ]
False

-- no red thing is blue
> not ( or [ isBlue x | x <- things, isRed x ] )
True

> and [ not (isBlue x) | x <- things, isRed x ]
True
```



To get Haskell to produce the result directly, we produce the list of booleans, and use the functions

and :: [Bool] -> Bool

or :: [Bool] -> Bool

and xs returns True only if every value in the list, xs, is True

or xs returns true iff some value in the list, xs, is True

```
-- every red triangle is small
> and [ isSmall x | x <- things, isRed x, isTriangle x ]
True
```

We say that everything that

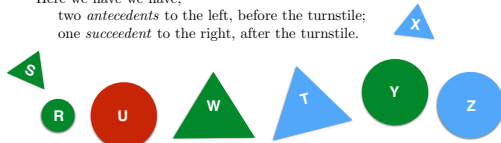
isRed and isTriangle *satisfies* isSmall.

isRed, isTriangle \models isSmall

If anything has all the properties listed to the left of the turnstile, \models , then it has some property on the right of the turnstile.

Here we have we have,

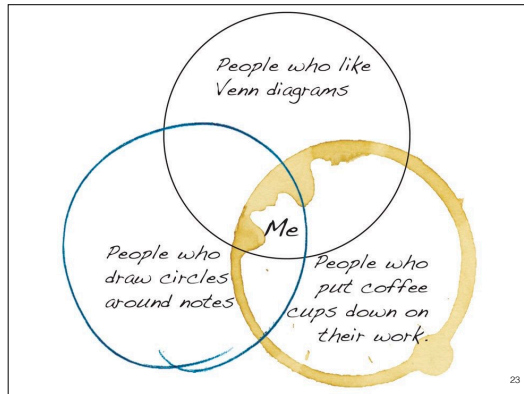
two *antecedents* to the left, before the turnstile;
one *succedent* to the right, after the turnstile.



if something isRed and isTriangle then it isSmall

isRed, isTriangle \models isSmall — every satisfies \models

isSmall, isTriangle $\not\models$ isRed — some does not satisfy $\not\models$



```

-- every red triangle is small
> and [ isSmall x | x <- things, isRed x, isTriangle x ]
True
-- isRed, isTriangle  $\models$  isSmall

-- every small triangle is red
> and [ isRed x | x <- things, isSmall x, isTriangle x ]
False
-- isSmall, isTriangle  $\not\models$  isRed

-- no red thing is blue
-- every red thing is not blue
> and [ not (isBlue x) | x <- things, isRed x ]
True
-- isRed  $\models \neg$  isBlue

```

if something isRed and isTriangle then it isSmall

isRed, isTriangle \models isSmall — every satisfies \models

isSmall, isTriangle $\not\models$ isRed — some does not satisfy $\not\models$

```

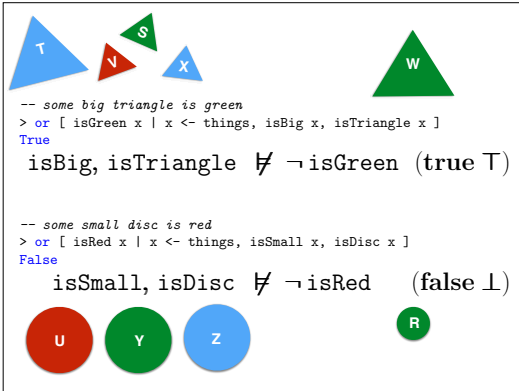
-- some big triangle is green
> or [ isGreen x | x <- things, isBig x, isTriangle x ]
True

-- some small disc is red
> or [ isRed x | x <- things, isSmall x, isDisc x ]
False

```

Can you express these in terms of satisfaction?

Can you write these in terms of satisfaction ?



```

-- some big triangle is green
> or [ isGreen x | x <- things, isBig x, isTriangle x ]
True
isBig, isTriangle  $\not\models$   $\neg$  isGreen (true T)

-- some small disc is red
> or [ isRed x | x <- things, isSmall x, isDisc x ]
False
isSmall, isDisc  $\not\models$   $\neg$  isRed (false  $\perp$ )

```

if something isRed and isTriangle then it isSmall

isRed, isTriangle \models isSmall — satisfies \models

isSmall, isTriangle $\not\models$ isRed — does not satisfy $\not\models$

```

every red triangle is small
isRed, isTriangle  $\models$  isSmall
and [ isSmall x | x <- things, isRed x, isTriangle x ]

every small triangle is red
isSmall, isTriangle  $\models$  isRed
and [ isRed x | x <- things, isSmall x, isTriangle x ]

some big triangle is green
isBig, isTriangle  $\not\models$   $\neg$  isGreen
not (and [ not (isGreen x) | x <- things, isBig x, isTriangle x ])

some small disc is red
isSmall, isDisc  $\not\models$   $\neg$  isRed
not (and [ not (isRed x) | x <- things, isSmall x, isDisc x ])

no red thing is blue
isRed  $\models$   $\neg$  isBlue
and [ not (isBlue x) | x <- things, isRed x ]

```

Here are some statements we might make about this universe.

Some true, some false. For this small universe it is easy to work out which are true, just by looking at the picture.

Our first task is to work out how we can turn such statements into Haskell code that will produce the correct answers.



what's missing?

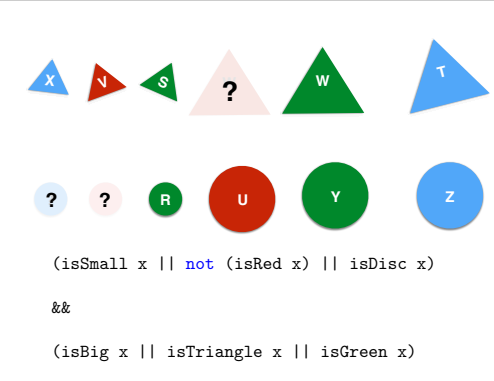
With three colours, two shapes and two sizes, we have 12 possibilities,

but we have only 9 things.

How can we say what's missing?

We can list the missing things :

Large Red Triangle, Small Blue Disc, Small Red Disc



```

(isSmall x || not (isRed x) || isDisc x)
&&
(isBig x || isTriangle x || isGreen x)

```

We can list the missing things :

Big Red Triangle, Small Blue Disc, Small Red Disc

`not (isBig x && isRed x && isTriangle x) &&`

`not (isSmall x && isTriangle x && not(isGreen x))`

Or equivalently

`(isSmall x || not(isRed x) || isDisc x) &&`

`(isBig x || isDisc x || isGreen x)`

some Boolean Operations

```

not :: Bool -> Bool -- not ¬
not False = True
not True = False

(&&) :: Bool -> Bool -> Bool -- and ∧
True && True = True
_ && _ = False

(||) :: Bool -> Bool -> Bool -- or ∨
False || False = False
_ || _ = True

or :: [Bool] -> Bool -- OR ∨ or [] = False
and :: [Bool] -> Bool -- AND ∧ and [] = True

```

p	¬p
⊥	⊤
⊤	⊥

∧	⊥	⊤
⊥	⊥	⊥
⊤	⊥	⊤

∨	⊥	⊤
⊥	⊥	⊤
⊤	⊤	⊤

30

Here, for reference, are the boolean operations we will use, together with their mathematical notations, and possible implementations of $\neg \wedge \vee$. Implementations for **or** and **and**, and alternative implementations of $\neg \wedge \vee$ are given later.

You don't need to use either set of implementations as these functions are all provided in the standard Preamble

An expression describes a computation. La

Operations

```
(&&) :: Bool -> Bool -> Bool   infixr 3
Boolean "and"
```

```
(||) :: Bool -> Bool -> Bool   infixr 2
Boolean "or"
```

False && True || True

An expression is a **tree** that describes a computation

(False && True) || True False && (True || True)

31

Operations

```
(&&) :: Bool -> Bool -> Bool   infixr 3
Boolean "and"
```

```
(||) :: Bool -> Bool -> Bool   infixr 2
Boolean "or"
```

False && True || True

(False && True) || True False && (True || True)

32

Operations

```
(&&) :: Bool -> Bool -> Bool   infixr 3
Boolean "and"
```

```
(||) :: Bool -> Bool -> Bool   infixr 2
Boolean "or"
```

a && b || c

An expression is a **tree** that describes a computation

(a && b) || c

```
Prelude> False && True || True
True
Prelude> False && (True || True)
False
Prelude> (False && True) || True
True
```

(False && True) || True

33

alternative implementations

```
not :: Bool -> Bool
not a = if a then False else True

(&&) :: Bool -> Bool -> Bool
a && b = if a then b else False

(||) :: Bool -> Bool -> Bool
a || b = if a then True else b

or :: [Bool] -> Bool
or (x : xs) = if x then True else or xs
or [] = False

and :: [Bool] -> Bool
and (x : xs) = if x then and xs else False
and [] = True

ite :: Bool -> Bool -> Bool -> Bool
ite a b c = if a then b else c
```



34

Here we define everything in terms of if then else.

The ite function is sufficient to define all Boolean functions.