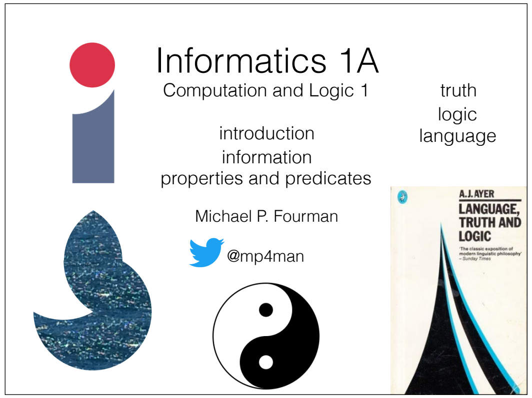
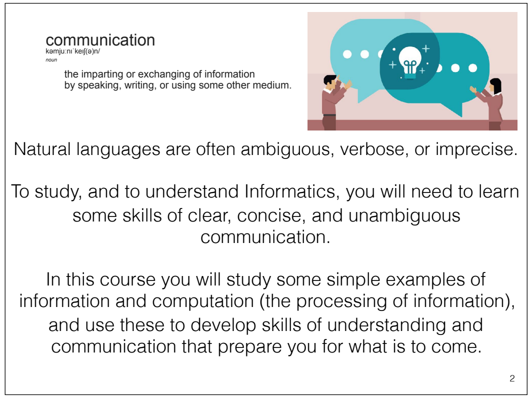


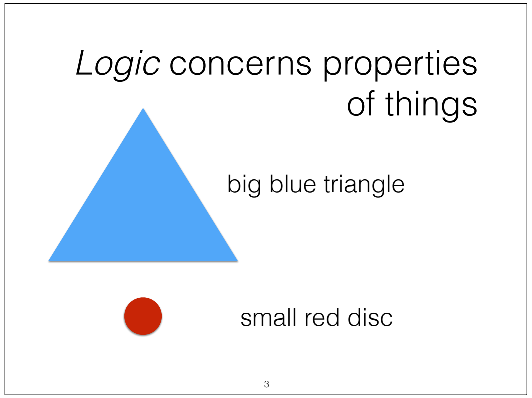
This course provides a first glimpse of the deep connections between computation and logic.



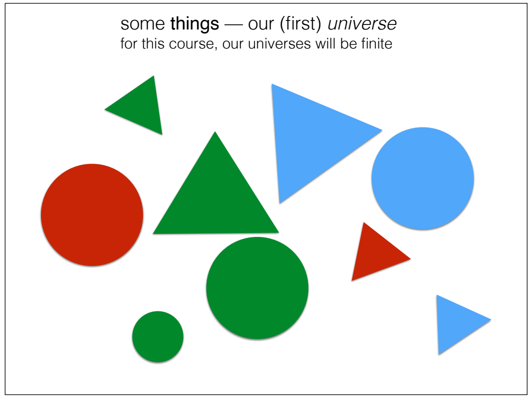
To work together we need to learn to communicate.

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.

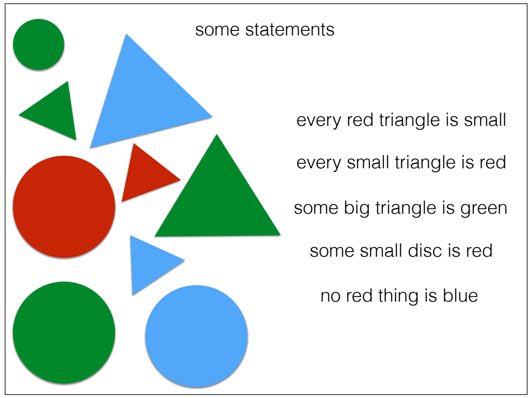
It also turns out that using such languages we can get computers to do much of our reasoning.



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 …

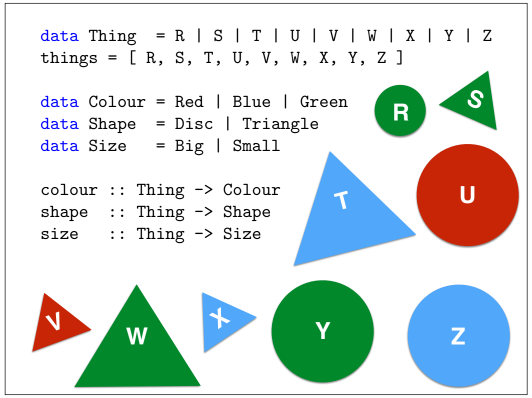


If everything is
either red or blue or green
and
either small or big (not small)
and
either disc or triangle (not disc)
then we have 12 = 3 x 2 x 2 possible combinations of three features.
Only some of these combinations appear in our universe.



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 our how we can turn such statements into Haskell code that will produce the correct answers.



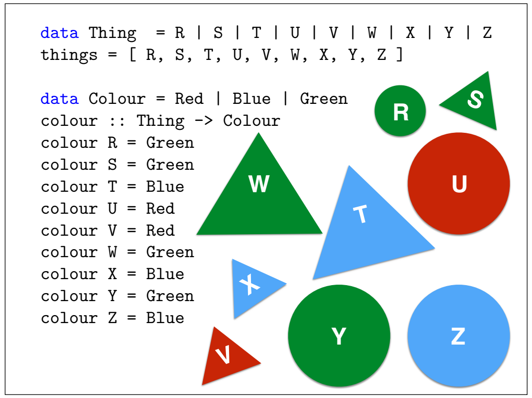
To code this problem in Haskell, we first introduce a type to represent our things.

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 then *constructors.*

We can only introduce a value of type Thing by useing 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.



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

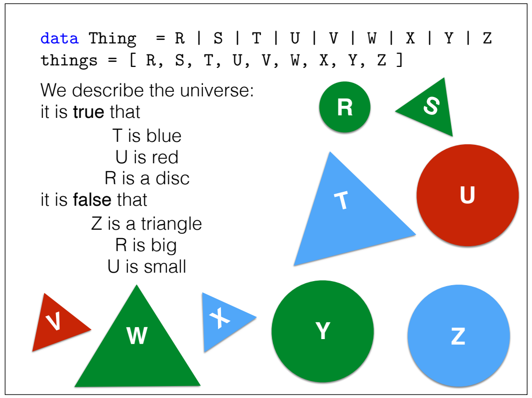
then r would act as a variable.

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

Variables begin with lowercase ; constructors begin with uppercase.

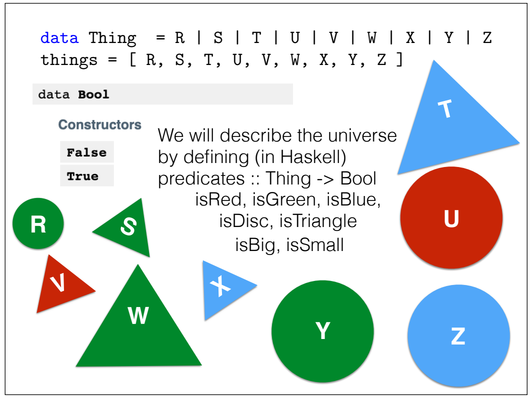
Logic is the science of *pure reasoning*

reasoning that doesn't depend on the subject matter.



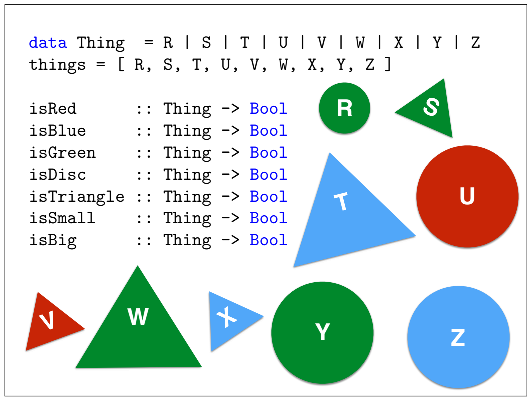
Having three features of different types — makes it difficult to separate the logic from the notions of colour, shape and size.

To study logic we use a different representation — we want a logic that is independent of the subject matter.



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.

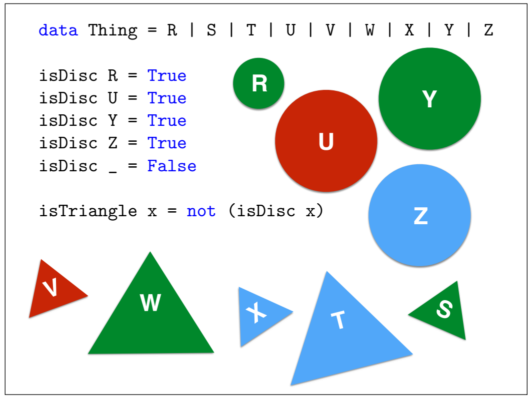
Any tools we produce for reasoning about this situation will work for any other (finite) universe of Things with predicates :: Thing -> Bool.



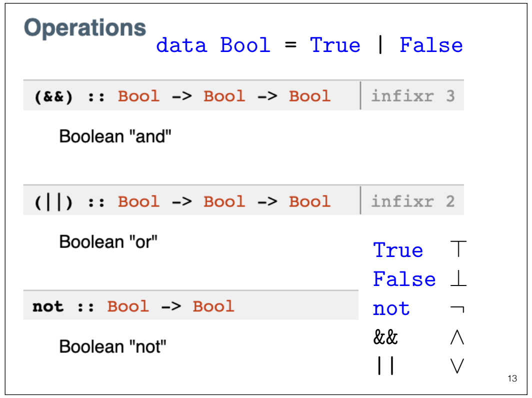
To code this problem in Haskell, we first introduce a type to represent our things.

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

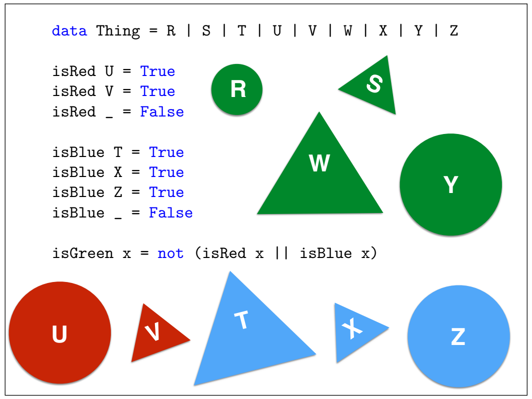
Then we will define a number of Boolean-valued functions to complete our representation.



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



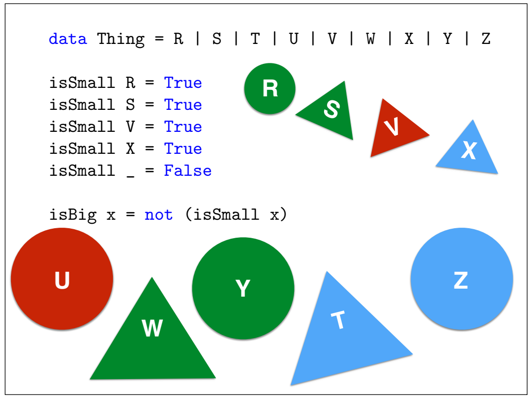
These operations will suffice for a (short) while.



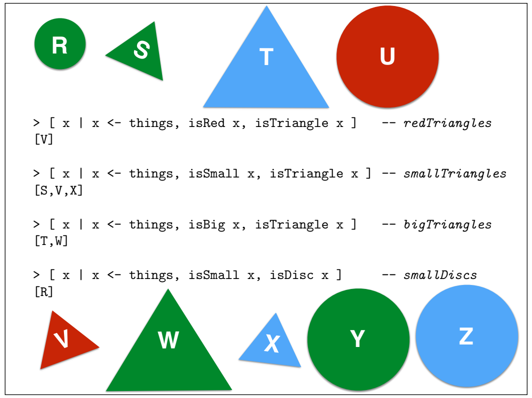
We list the red things, and say each one isRed — everything else is not.

Similarly for isBlue.

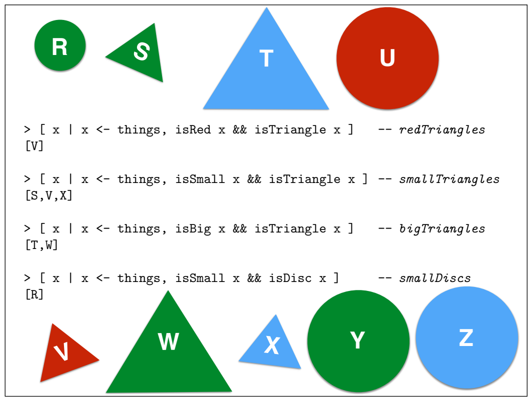
We can use a little boolean logic to define isGreen. || is Boolean 'or'



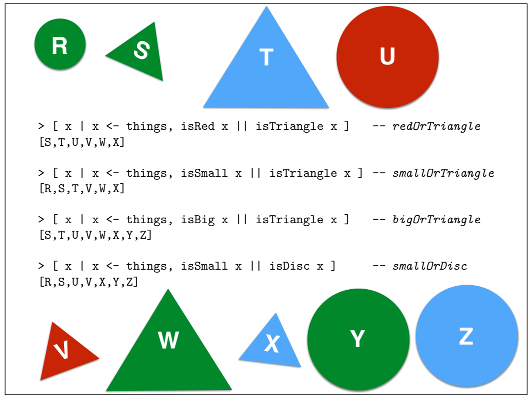
This example is similar.



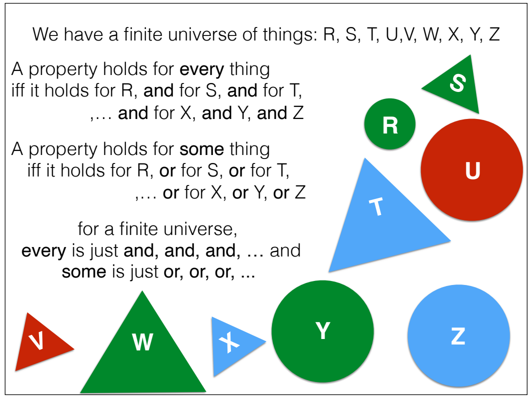
List comprehensions allow us to pick out subsets of things with particular combinations of properties.



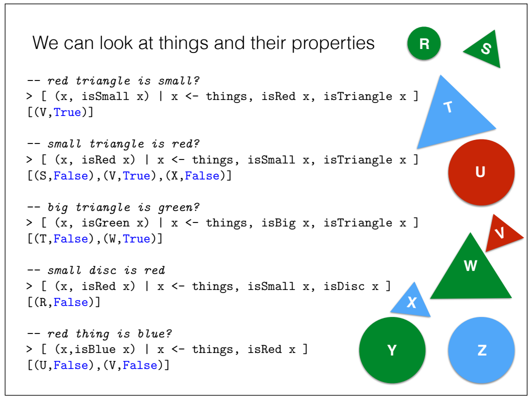
Using && in place of some commas gives the same results. Note that is now easy to see that every red triangle is small, but no small triangle is red.



using || instead of && gives different results

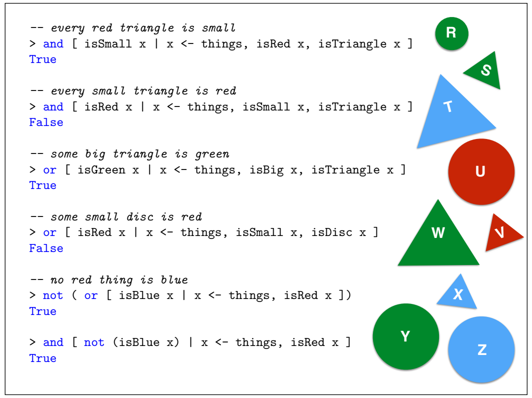


We can check whether a property holds for every thing, or for some thing, in a finite universe by checking whether it holds for each thing in turn.



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.



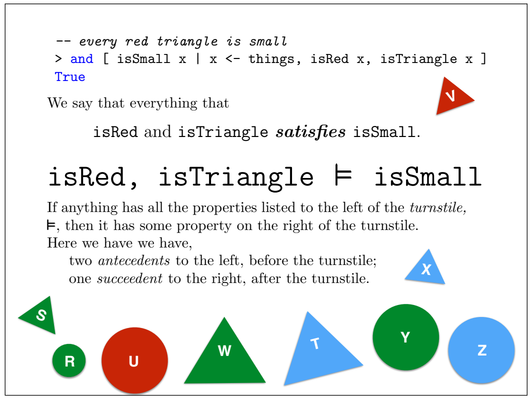
To get Haskell to produce the result directly, we produce the list of booleans, and use the functions

and :: [Bool] -> Bool

or :: [Bool] -> Bool

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

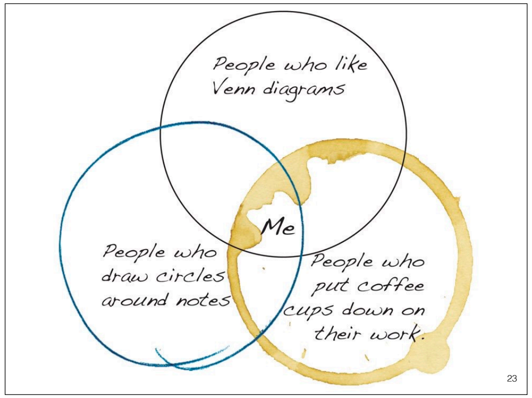
or xs returns true iff some value in the list, xs, is True

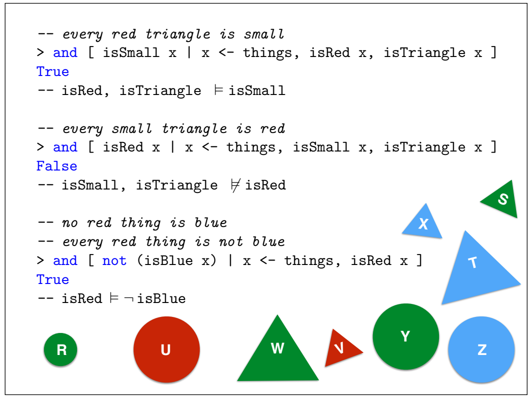


if something isRed and IsTriangle then it isSmall

isRed, isTriangle $\models$ isSmall — every satisfies $\models$

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

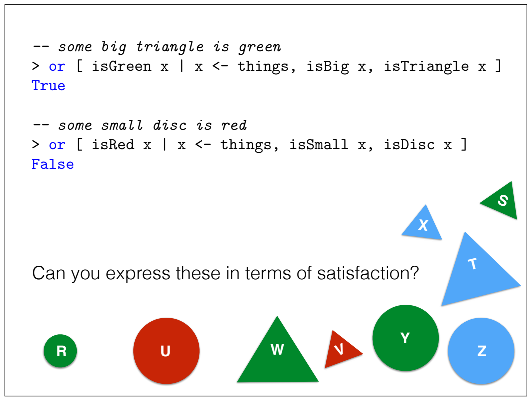




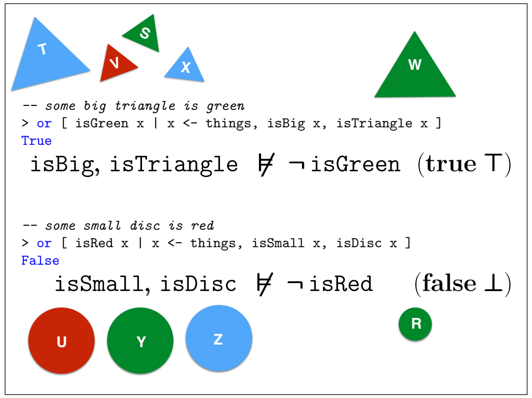
if something isRed and IsTriangle then it isSmall

isRed, isTriangle $\models$ isSmall — every satisfies $\models$

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



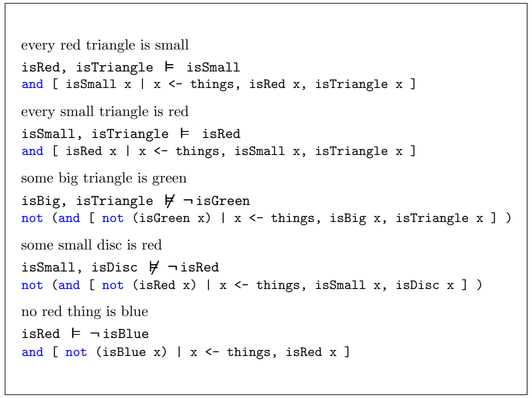
Can you write these in terms of satisfaction ?



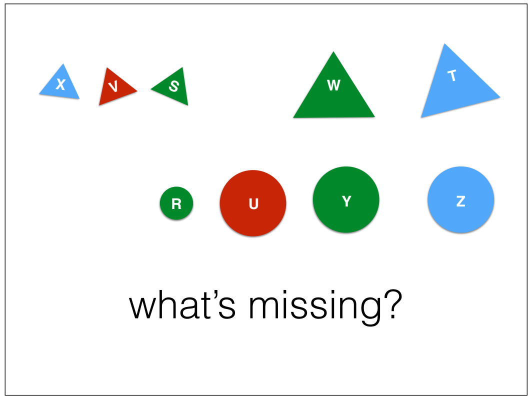
if something isRed and isTriangle then it isSmall

isRed, isTriangle $\models$ isSmall — satisfies $\models$

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

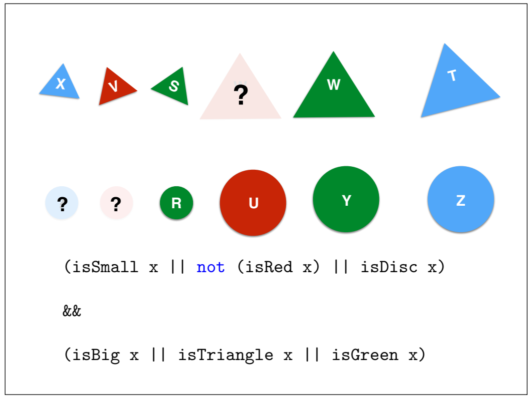


Here are some statements we might make about this universe.
Some true, some false. For this small universe it is easy to work out which are true, just by looking at the picture.
Our first task is to work our how we can turn such statements into Haskell code that will produce the correct answers.



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



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
```

p	¬ p
⊥	⊤
⊤	⊥

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

∧	⊥	⊤
⊥	⊥	⊥
⊤	⊥	⊤

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

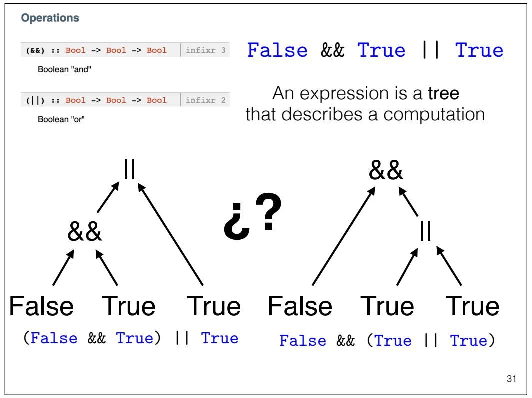
∨	⊥	⊤
⊥	⊥	⊤
⊤	⊤	⊤

```
or  :: [Bool] -> Bool -- OR  ⋁    or  [] = False
and :: [Bool] -> Bool -- AND ⋀    and [] = True
```

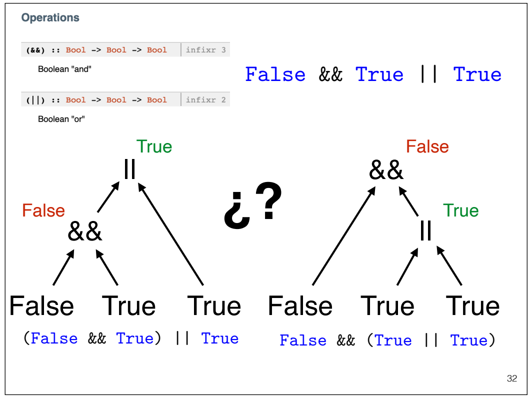
30

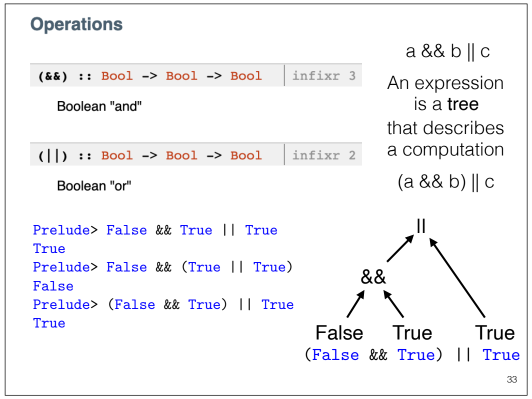
Here, for reference, are the boolean operations we will use, together with their mathematical notations, and possible implementations of ¬∧∨. Implementations for **or** and **and**, and alternative implementations of ¬∧∨ are given later.

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





alternative implementations

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

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

(||) :: 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.