Classical first-order predicate logic

This is a powerful extension of propositional logic. It is the most important logic of all.

In the remaining lectures, we will:

- explain predicate logic syntax and semantics carefully
- do English–predicate logic translation, and see examples from computing (pre- and post-conditions)
- generalise *arguments* and *validity* from propositional logic to predicate logic
- consider ways of establishing validity in predicate logic:
  - truth tables — they don't work
  - direct argument — very useful
  - equivalences — also useful
  - natural deduction (sorry).

Why?

Propositional logic is quite nice, but not very expressive.
Statements like

- the list is ordered
- every worker has a boss
- there is someone worse off than you

need something more than propositional logic to express.

Propositional logic can't express arguments like this one of De Morgan:

- A horse is an animal.
- Therefore, the head of a horse is the head of an animal.

6. Predicate logic in a nutshell

6.1 Splitting the atom — new atomic formulas

Up to now, we have regarded phrases such as *the computer is a Sun* and *Frank bought grapes* as atomic, without internal structure.

Now we look inside them.

We regard being a Sun as a *property* or *attribute* that a computer (and other things) may or may not have. So we introduce:

- A *relation symbol* (or *predicate symbol)* `Sun`.
  It takes 1 argument — we say it is *unary* or its 'arity' is 1.
- We can also introduce a relation symbol `bought`.
  It takes 2 arguments — we say it is *binary,* or its arity is 2.
- *Constants,* to name objects.
  Eg, `Heron`, `Frank`, `Room-308`, `grapes`.

Then `Sun(Heron)` and `bought(Frank,grapes)` are two new atomic formulas.

6.2 Quantifiers

So what? You may think that writing

$$\texttt{bought}(\texttt{Frank}, \texttt{grapes})$$

is not much more exciting than what we did in propositional logic — writing

$$\texttt{Frank bought grapes.}$$

But predicate logic has machinery to vary the arguments to `bought`.

This allows us to express properties of the relation ‘`bought`’.

The machinery is called *quantifiers.* (The word was introduced by De Morgan.)

What are quantifiers?

A quantifier specifies a quantity (of things that have some property).

Examples

- *All* students work hard.
- *Some* students are asleep.
- *Most* lecturers are crazy.
- *Eight out of ten* cats prefer it.
- *No one* is worse off than me.
- *At least six* students are awake.
- *There are infinitely many* prime numbers.
- *There are more* PCs than there are Macs.

Quantifiers in predicate logic

There are just two:

- $\forall$ (or `(A)`): 'for all'
- $\exists$ (or `(E)`): 'there exists' (or 'some')

Some other quantifiers can be expressed with these. (They can also express each other.)
But quantifiers like *infinitely many* and *more than* cannot be expressed in first-order logic in general. (They can in, e.g., second-order logic. And even first-order logic can sometimes express them in special cases.)

How do they work?

We've seen expressions like `Heron`, `Frank`, etc. These are *constants,* like π, or e.

To express 'All computers are Suns' we need *variables* that can range over all computers, not just Heron, Texel, etc.

6.3 Variables

We will use *variables* to do quantification. We fix an infinite collection (or 'set') V of variables: eg, $x, y, z, u, v, w, x_0, x_1, x_2, \ldots$
Sometimes I write x or y to mean 'any variable'.

As well as formulas like $\mathtt{Sun}(\mathtt{Heron})$, we'll write ones like $\mathtt{Sun}(x)$.

- Now, to say 'Everything is a Sun', we'll write $\forall x\, \mathtt{Sun}(x)$.
  This is read as: 'For all x, x is a Sun'.
- 'Something is a Sun', can be written $\exists x\, \mathtt{Sun}(x)$.
  'There exists x such that x is a Sun.'
- 'Frank bought a Sun', can be written

$$\exists x(\mathtt{Sun}(x) \wedge \mathtt{bought}(\mathtt{Frank}, x)).$$

  'There is an x such that x is a Sun and Frank bought x.'
  Or: 'For some x, x is a Sun and Frank bought x.'

See how the new internal structure of atoms is used.

We will now make all of this precise.

7. Syntax of predicate logic

As in propositional logic, we do the syntax first, then the semantics.

7.1 Signatures

Definition 7.1 (signature) *A signature is a collection (set) of constants, and relation symbols with specified arities.*

Some call it a *similarity type*, or *vocabulary,* or (loosely) *language.*

It replaces the collection of propositional atoms we had in propositional logic.

We usually write L to denote a signature. We often write $c, d, \ldots$ for constants, and $P, Q, R, S, \ldots$ for relation symbols.

Later (§10), we'll throw in function symbols.

A simple signature

Which symbols we put in L depends on what we want to say.

For illustration, we'll use a handy signature L consisting of:

- constants $\mathtt{Frank}$, $\mathtt{Susan}$, $\mathtt{Tony}$, $\mathtt{Heron}$, $\mathtt{Texel}$, $\mathtt{Clyde}$, $\mathtt{Room\text{-}308}$, and c
- unary relation symbols $\mathtt{Sun}$, $\mathtt{human}$, $\mathtt{lecturer}$ (arity 1)
- a binary relation symbol $\mathtt{bought}$ (arity 2).

Warning: things in L are just symbols — syntax. They don't come with any meaning. To give them meaning, we'll need to work out (later) what a *situation* in predicate logic should be.

7.2 Terms

To write formulas, we'll need *terms,* to name objects.
Terms are not formulas. They will not be true or false.

Definition 7.2 (term) *Fix a signature L.*

1. *Any constant in L is an L-term.*
2. *Any variable is an L-term.*
3. *Nothing else is an L-term.*

A closed term *or (as computer people say)* ground term *is one that doesn't involve a variable.*

Examples of terms

`Frank`, `Heron` (ground terms)
x, y, x_{56} (not ground terms)

Later (§10), we'll throw in function symbols.

7.3 Formulas of first-order logic

Definition 7.3 (formula) *Fix L as before.*

1. *If R is an n-ary relation symbol in L, and $t_1, \ldots, t_n$ are L-terms, then $R(t_1, \ldots, t_n)$ is an atomic L-formula.*
2. *If t, t' are L-terms then $t = t'$ is an atomic L-formula. (Equality — very useful!)*
3. *$\top, \bot$ are atomic L-formulas.*
4. *If A, B are L-formulas then so are $(\neg A)$, $(A \wedge B)$ $(A \vee B)$, $(A \to B)$, and $(A \leftrightarrow B)$.*
5. *If A is an L-formula and x a variable, then $(\forall x\, A)$ and $(\exists x\, A)$ are L-formulas.*
6. *Nothing else is an L-formula.*

Binding conventions: as for propositional logic, plus: $\forall x, \exists x$ have same strength as $\neg$.

Examples of formulas

Below, we write them as the cognoscenti do.

Use binding conventions to disambiguate.

1. $\mathtt{bought}(\mathtt{Frank}, x)$
   We read this as: ‘Frank bought x.’
2. $\exists x\, \mathtt{bought}(\mathtt{Frank}, x)$
   ‘Frank bought something.’
3. $\forall x(\mathtt{lecturer}(x) \to \mathtt{human}(x))$
   ‘Every lecturer is human.’ [Important eg!]
4. $\forall x(\mathtt{bought}(\mathtt{Tony}, x) \to \mathtt{Sun}(x))$
   ‘Everything Tony bought is a Sun.’

Formation trees and subformulas, literals and clauses, etc., can be done much as before.

More examples

5. $\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x))$
   'Susan bought everything that Tony bought.'

6. $\forall x\, \texttt{bought}(\texttt{Tony}, x) \to \forall x\, \texttt{bought}(\texttt{Susan}, x)$
   'If Tony bought everything, so did Susan.' Note the difference!

7. $\forall x \exists y\, \texttt{bought}(x, y)$
   'Everything bought something.'

8. $\exists y \forall x\, \texttt{bought}(x, y)$
   'There is something that everything bought.' Note the difference!

9. $\exists x \forall y\, \texttt{bought}(x, y)$
   'Something bought everything.'

You can see that predicate logic is rather powerful — and terse.

8. Semantics of predicate logic

As in propositional logic, we have to specify

1. what a *situation* is for predicate logic,

2. how to evaluate predicate logic formulas in a given situation.

We have to handle: new atomic formulas; quantifiers and variables.

8.1 Structures (situations in predicate logic)

Let's deal with the new-style atomic formulas first.

Definition 8.1 (structure) *Let L be a signature. An L-structure (or sometimes (loosely) a model) M is a thing that*

- *identifies a non-empty collection (set) of objects that M 'knows about'. It's called the domain or universe of M, written $\mathrm{dom}(M)$.*
- *specifies what the symbols of L mean in terms of these objects.*

The interpretation in M of a constant is an *object in* $\mathrm{dom}(M)$.

The interpretation in M of a relation symbol is a *relation on* $\mathrm{dom}(M)$.

CS1 will soon see sets and relations in Discrete Maths, course 142.

Example of a structure

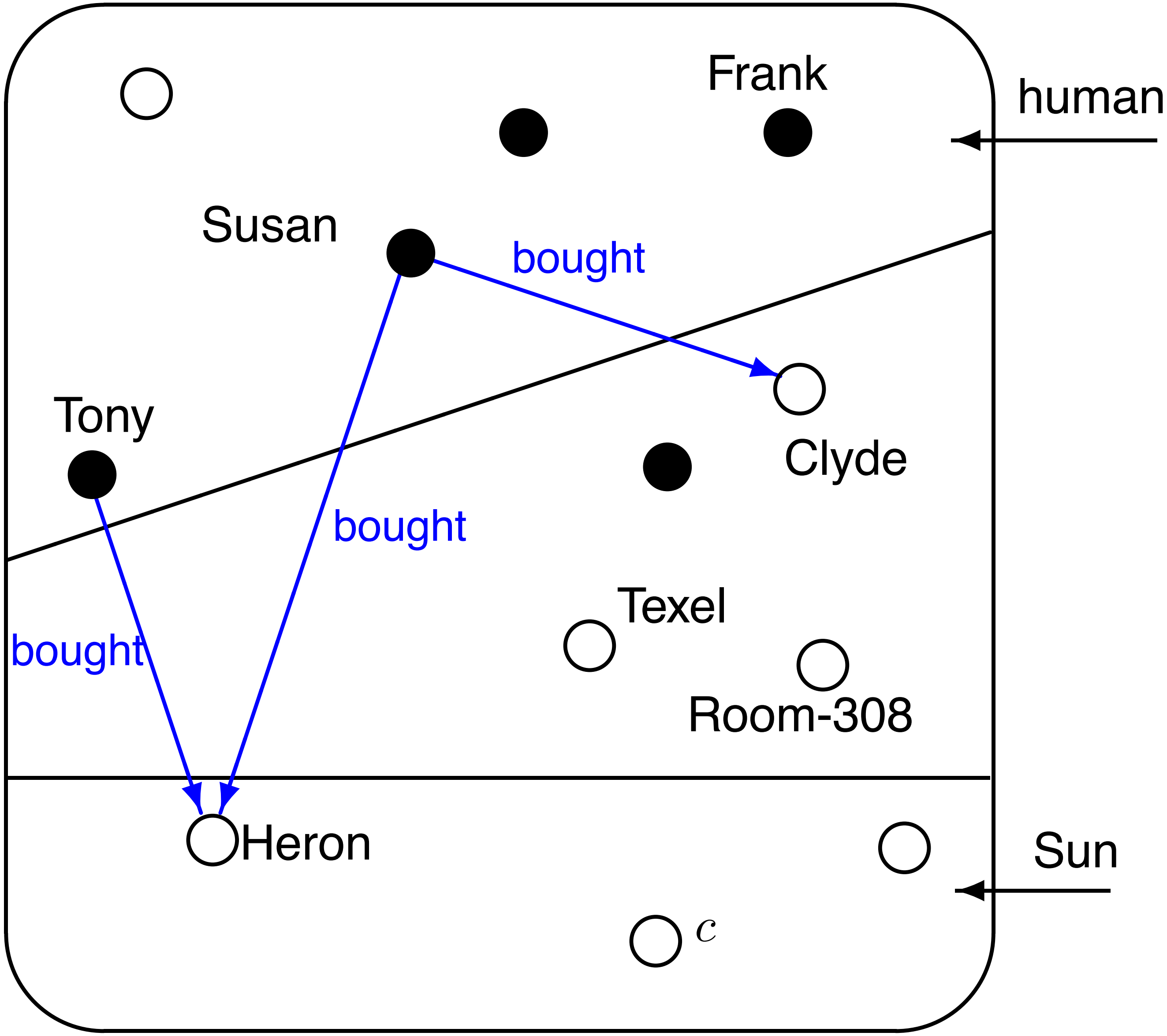
For our handy L, an L-structure should say:

- which objects are in its domain
- which of its objects are `Tony`, `Susan`, ...
- which objects are `human`, `Sun`, `lecturer`
- which objects `bought` which.

Below is a diagram of a particular L-structure, called M (say).

There are 12 objects (the 12 dots) in the domain of M.
Some are labelled (eg ‘Frank’) to show the meanings of the constants of L (eg `Frank`).
The interpretations (meanings) of `Sun`, `human` are drawn as regions.
The interpretation of `lecturer` is indicated by the black dots.
The interpretation of `bought` is shown by the arrows between objects.

The structure M



Tony or $\texttt{Tony}$?

Do not confuse the object ● marked 'Tony' in $\mathrm{dom}(M)$ with the constant $\texttt{Tony}$ in L. (I use different fonts, to try to help.)

They are quite different things. $\texttt{Tony}$ *is syntactic.* ● *is semantic.*
In the context of M, $\texttt{Tony}$ is a *name* for the object ● marked 'Tony'.

The following notation helps to clarify:

Notation 8.2 *Let M be an L-structure and c a constant in L. We write c^M for the interpretation of c in M. It is the object in $\mathrm{dom}(M)$ that c names in M.*

So $\texttt{Tony}^M =$ the object ● marked 'Tony'. I will usually write just 'Tony' or $\texttt{Tony}^M$ (but NOT $\texttt{Tony}$) for this ●.
In a different structure, $\texttt{Tony}$ may name (mean) something else.

The meaning of a constant c *IS* the object c^M assigned to it by a structure M. A constant (and any symbol of L) has as many meanings as there are L-structures.

Drawing other symbols

Our signature L has only constants and unary and binary relation symbols.

For this L, we drew an L-structure M by

- drawing a collection of objects (the domain of M)
- marking which objects are named by which constants in M
- marking which objects M says satisfy the unary relation symbols (`human`, etc)
- drawing arrows between the objects that M says satisfy the binary relation symbols. The arrow direction matters.

If there were several binary relation symbols in L, we'd *really need* to label the arrows.
In general, there's no easy way to draw interpretations of 3-ary or higher-arity relation symbols.
0-ary (nullary) relation symbols are the same as propositional atoms.

8.2 Truth in a structure (a rough guide)

When is a formula *without quantifiers* true in a structure?

- $\mathtt{Sun(Heron)}$ is true in M, because $\mathtt{Heron}^M$ is an object ○ that M says is a $\mathtt{Sun}$.
  We write this as $M \models \mathtt{Sun}(\mathtt{Heron})$.
  Can read as 'M says $\mathtt{Sun(Heron)}$'.

  Warning: This is a quite different use of $\models$ from definition 3.1.
  '$\models$' is *overloaded* — it's used for two different things.

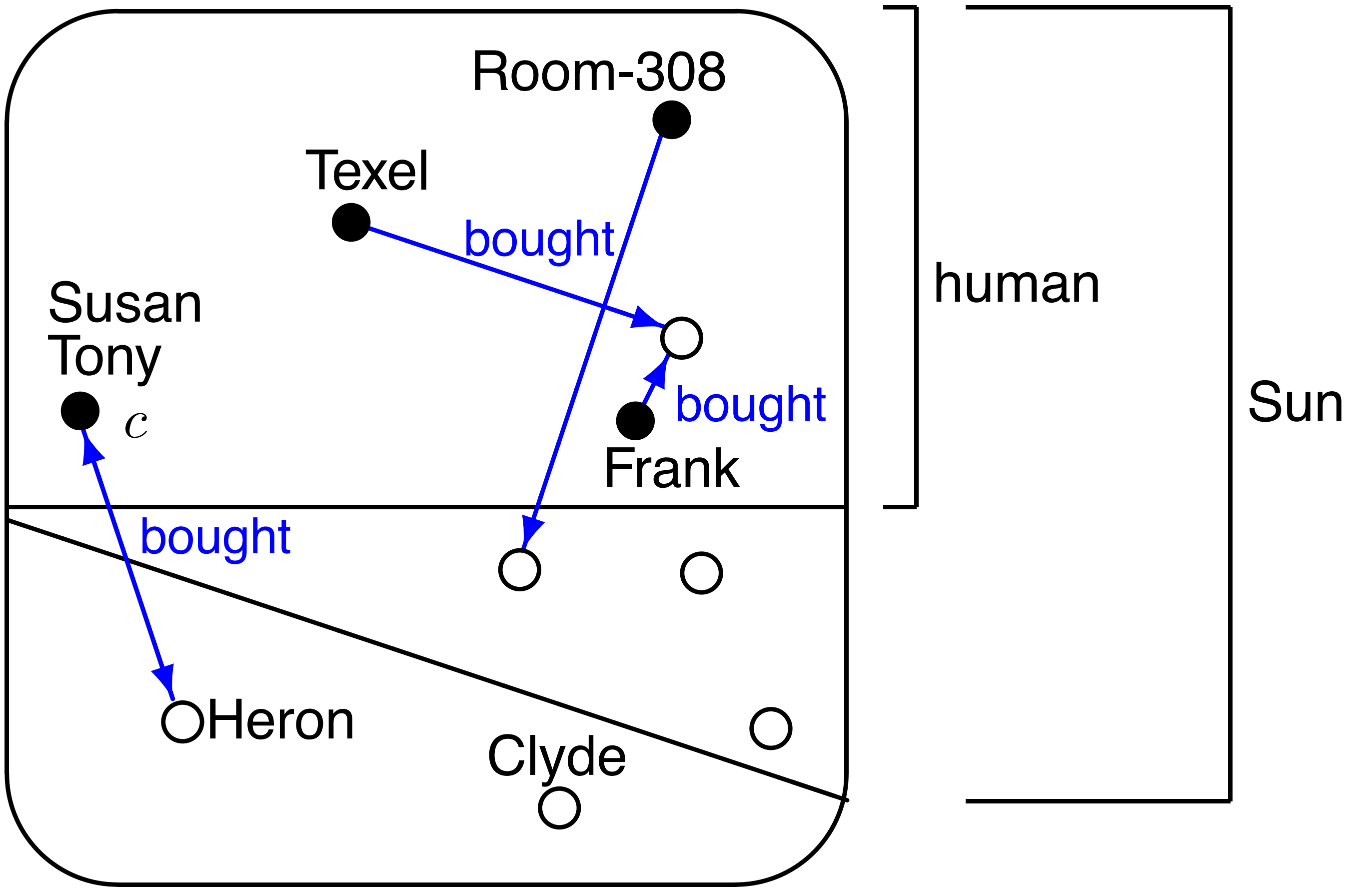
- $\mathtt{bought(Susan,Susan)}$ is false in M, because M does not say that the constant $\mathtt{Susan}$ names an object ● that bought itself.
  In symbols, $M \not\models \mathtt{bought}(\mathtt{Susan}, \mathtt{Susan})$.

From our knowledge of propositional logic,

- $M \models \neg\, \mathtt{human}(\mathtt{Room\text{-}308})$,
- $M \not\models \mathtt{Sun}(\mathtt{Tony}) \vee \mathtt{bought}(\mathtt{Frank}, \mathtt{Clyde})$.

Another structure

Here's another L-structure, called M'.



Now, there are only 10 objects in $\mathrm{dom}(M')$.

Some statements about M'

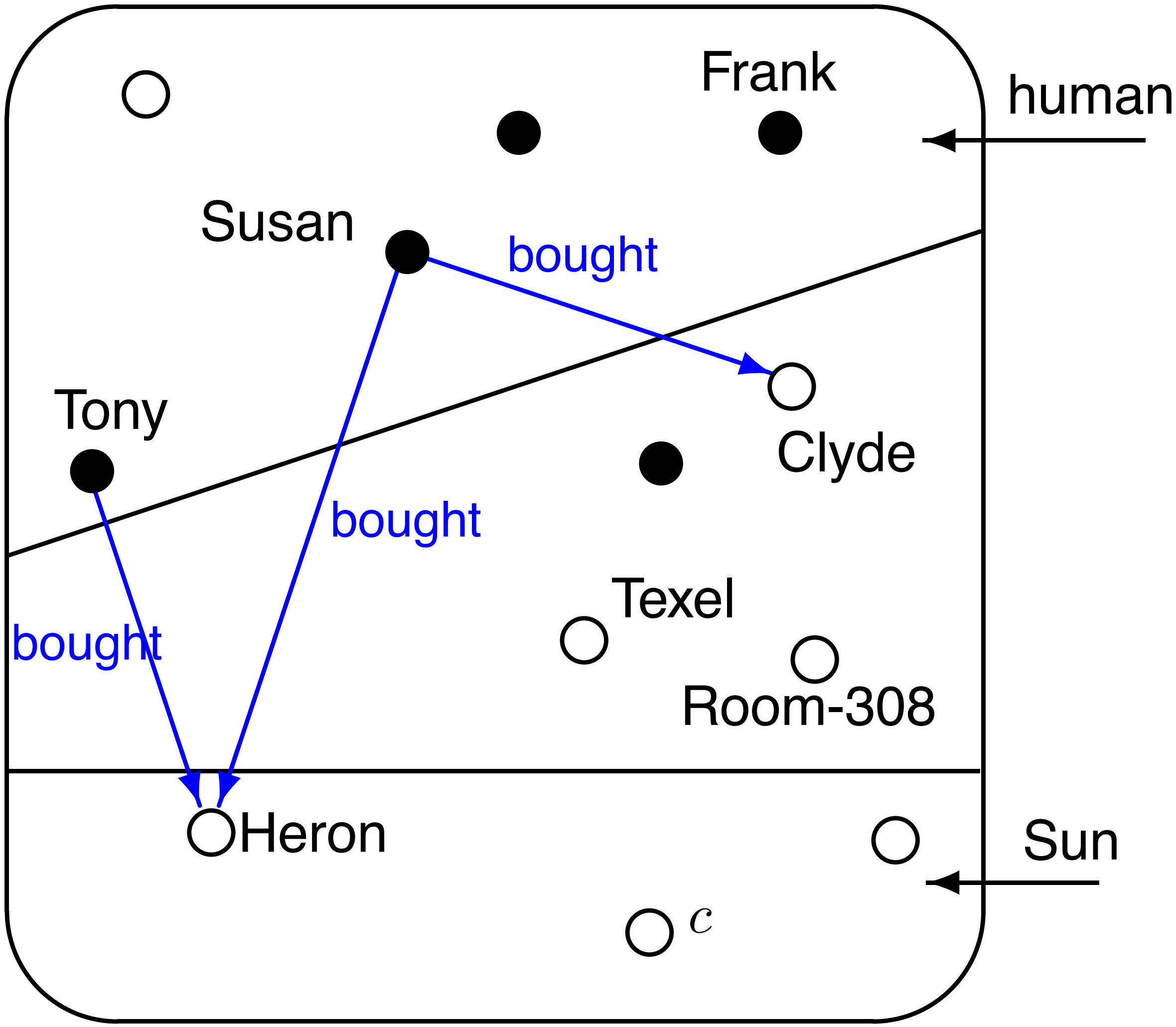
- $M' \not\models \texttt{bought}(\texttt{Susan}, \texttt{Clyde})$ this time.
- $M' \models \texttt{Susan} = \texttt{Tony}$.
- $M' \models \texttt{human}(\texttt{Texel}) \wedge \texttt{Sun}(\texttt{Texel})$.
- $M' \models \texttt{bought}(\texttt{Tony}, \texttt{Heron}) \wedge \texttt{bought}(\texttt{Heron}, c)$.

How about

- $\texttt{bought}(\texttt{Susan}, \texttt{Clyde}) \to \texttt{human}(\texttt{Clyde})$?
- $\texttt{bought}(c, \texttt{Heron}) \to \texttt{Sun}(\texttt{Clyde}) \vee \neg\texttt{human}(\texttt{Texel})$?

Evaluating formulas with quantifiers — rough guide

When is a formula *with quantifiers* true in a structure?



M again

Evaluating quantifiers

How can we tell if $\exists x\ \texttt{bought}(x, \texttt{Heron})$ is true in M?
In symbols, do we have $M \models \exists x\ \texttt{bought}(x, \texttt{Heron})$?
In English, 'does M say that something bought Heron?'.

Well, for this to be so, there must be an object x in $\mathrm{dom}(M)$ such that $M \models \texttt{bought}(x, \texttt{Heron})$ — that is, M says that x $\texttt{bought}$ $\texttt{Heron}^M$.

There is: we have a look, and we see that we can take (eg.) x to be (the ● marked) Tony.

So yes indeed, $M \models \exists x\ \texttt{bought}(x, \texttt{Heron})$.

Another example: $M \models \forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x))$?

That is, 'is it true that for every object x in $\mathrm{dom}(M)$, $\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M'?

In M, there are 12 possible x. We need to check whether $\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M for each of them.

BUT: $\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ will be true in M for any object x such that $\texttt{bought}(\texttt{Tony}, x)$ is false in M. ('False $\to$ anything is true.') So we only need check those x — here, just the object $\bigcirc = \texttt{Heron}^M$ — for which $\texttt{bought}(\texttt{Tony}, x)$ is true.

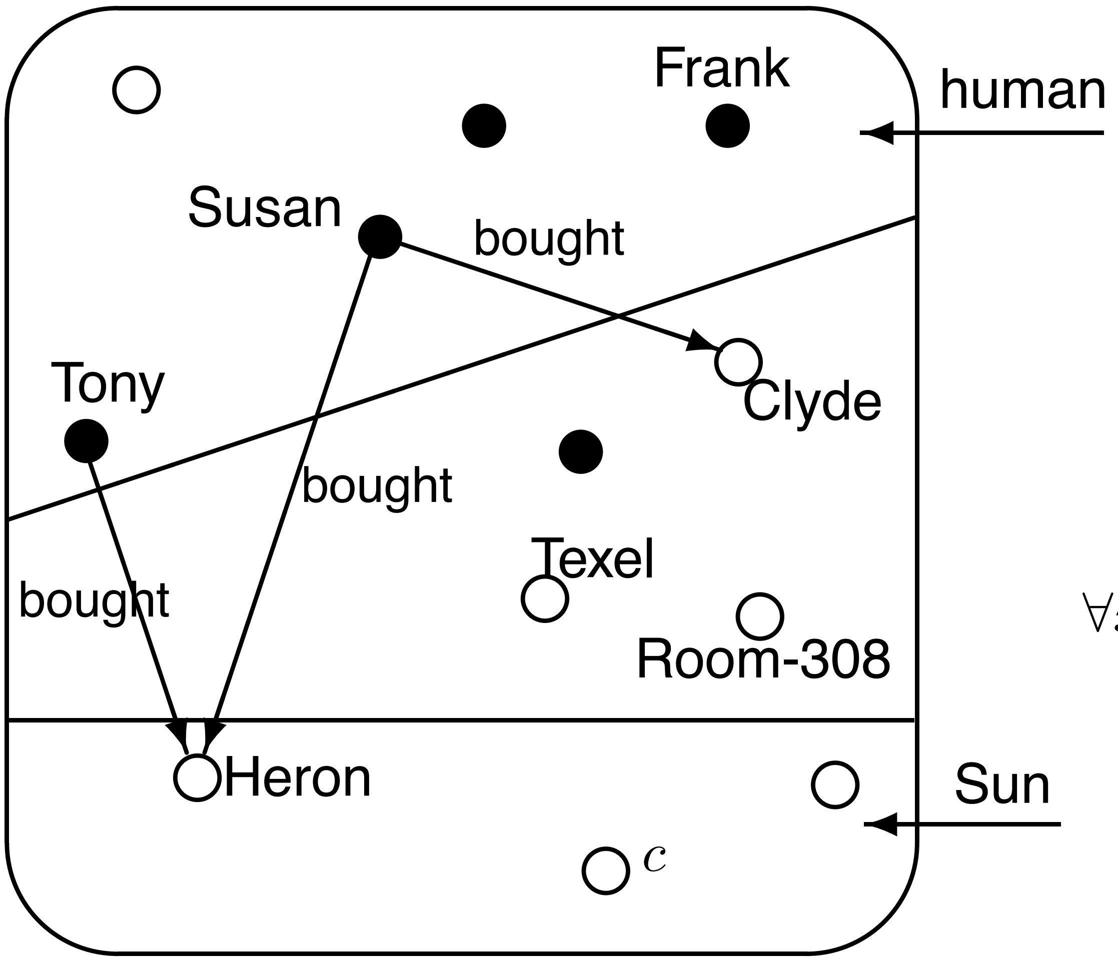
For this $\bigcirc$, $\texttt{bought}(\texttt{Susan},\bigcirc)$ is true in M.
So $\texttt{bought}(\texttt{Tony},\bigcirc)\to\texttt{bought}(\texttt{Susan},\bigcirc)$ is true in M.

So $\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M for *every* object x in M. Hence, $M \models \forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x))$.

The effect of '$\forall x(\texttt{bought}(\texttt{Tony}, x) \to \cdots$' is to *restrict the* $\forall x$ to those x that Tony bought. *This trick is extremely useful. Remember it!*

Exercise: which are true in M?

Reminder: the ●s are the lecturers



$$\exists x(\texttt{Sun}(x) \wedge \texttt{bought}(\texttt{Frank}, x))$$

$$\forall x(\texttt{lecturer}(x) \rightarrow \texttt{human}(x))$$

$$\forall x(\exists y\ \texttt{bought}(y, x) \rightarrow \neg\texttt{human}(x))$$

Rough guide: advice

For a fairly complex formula like $\exists x(\texttt{Sun}(x) \wedge \exists y\ \texttt{bought}(y, x))$:

Work out what each subformula says in English, working from atomic subformulas (leaves of formation tree) up to the whole formula (root of formation tree).

$\exists x(\texttt{Sun}(x) \wedge \exists y\ \texttt{bought}(y, x))$	something bought a Sun
$\texttt{Sun}(x) \wedge \exists y\ \texttt{bought}(y, x)$	x is a Sun & something bought x
$\texttt{Sun}(x)$ $\exists y\ \texttt{bought}(y, x)$	x is a Sun something bought x
$\texttt{bought}(y, x)$	y bought x

This is often a good guide to evaluating the formula.
E.g., the formula here says that there is an x that's a Sun and that something bought (it's pointed to by an arrow). So look for one.

8.3 Truth in a structure — formally!

We saw how to evaluate some formulas in a structure ‘by inspection’.

But as in propositional logic, English can only be a rough guide. For engineering, this is not good enough.

We need a more formal way to evaluate all predicate logic formulas in structures.

In propositional logic, we calculated the truth value of a formula in a situation by working up through its formation tree — from the atomic subformulas (leaves) up to the root.

For predicate logic, thing are not so simple…

A problem

$\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x))$ is true in the structure M on slide 133.

Its formation tree is:

$$\begin{tikzcd}[column sep=small] & \forall x \arrow[d, dash] & \\ & \to \arrow[dl, dash] \arrow[dr, dash] & \\ \texttt{bought}(\texttt{Tony}, x) & & \texttt{Sun}(x) \end{tikzcd}$$

Can we evaluate the main formula by working up the tree?

Is $\texttt{bought}(\texttt{Tony}, x)$ true in M?!
Is $\texttt{Sun}(x)$ true in M?!

Not all formulas of predicate logic are true or false in a structure!

What's going on?

Free and bound variables

We'd better investigate how variables can arise in formulas.

Definition 8.3 *Let A be a formula.*

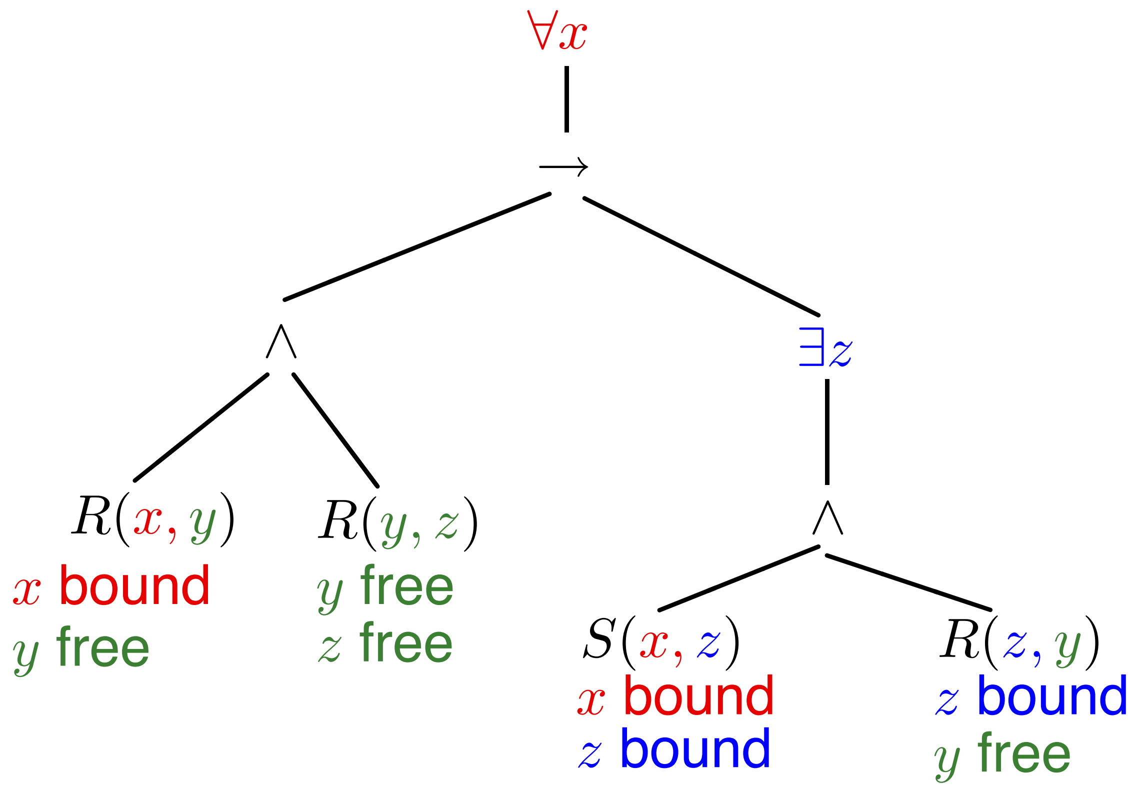
1. *An occurrence of a variable x in an atomic subformula of A is said to be* bound *if it lies under a quantifier $\forall x$ or $\exists x$ in the formation tree of A.*

2. *If not, the occurrence is said to be* free.

3. *The* free variables of A *are those variables with free occurrences in A.*

Example

$$\forall x(R(x,y) \wedge R(y,z) \rightarrow \exists z(S(x,z) \wedge R(z,y)))$$



The free variables of the formula are y, z.

Note: z has both free and bound occurrences.

Sentences

Definition 8.4 *A sentence is a formula with no free variables.*

Examples

- $\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x))$ is a sentence.
- Its subformulas

  $\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x)$

  $\texttt{bought}(\texttt{Tony}, x)$

  $\texttt{Sun}(x)$

  are not sentences.

Which are sentences?

- $\texttt{bought}(\texttt{Frank}, \texttt{Texel})$
- $\texttt{bought}(\texttt{Susan}, x)$
- $x = x$
- $\forall x(\exists y(y = x) \to x = y)$
- $\forall x \forall y(x = y \to \forall z(R(x, z) \to R(y, z)))$

Problem 1: free variables

Sentences are true or false in a structure.
But non-sentences are not!
A formula with free variables is neither true nor false in a structure M, because the free variables have no meaning in M.
It's like asking 'is $x = 7$ true?'

So *the structure is not a 'complete' situation* — it doesn't fix the meanings of free variables. (They are *variables,* after all!)

Handling values of free variables

So we must specify values for free variables, before evaluating a formula to true or false.
This is so even if it turns out that the values do not affect the answer (like $x = x$).

Assignments to variables

We supply the missing values of free variables using something called an *assignment.*

What a structure does for constants, an assignment does for variables.

Definition 8.5 (assignment) *Let M be a structure. An* assignment (or 'valuation') into M *is something that allocates an object in* $\mathrm{dom}(M)$ *to each variable.*
For an assignment h and a variable x, we write $h(x)$ for the object assigned to x by h.

[Formally, $h : V \to \mathrm{dom}(M)$ is a function.]

Given an L-structure M *plus* an assignment h into M, we have a 'complete situation'. We can then evaluate:

- any L-term, to *an object in* $\mathrm{dom}(M)$,
- any L-formula with no quantifiers, to *true or false*.

Evaluating terms (easy!)

We do the evaluation in two stages: first terms, then formulas.

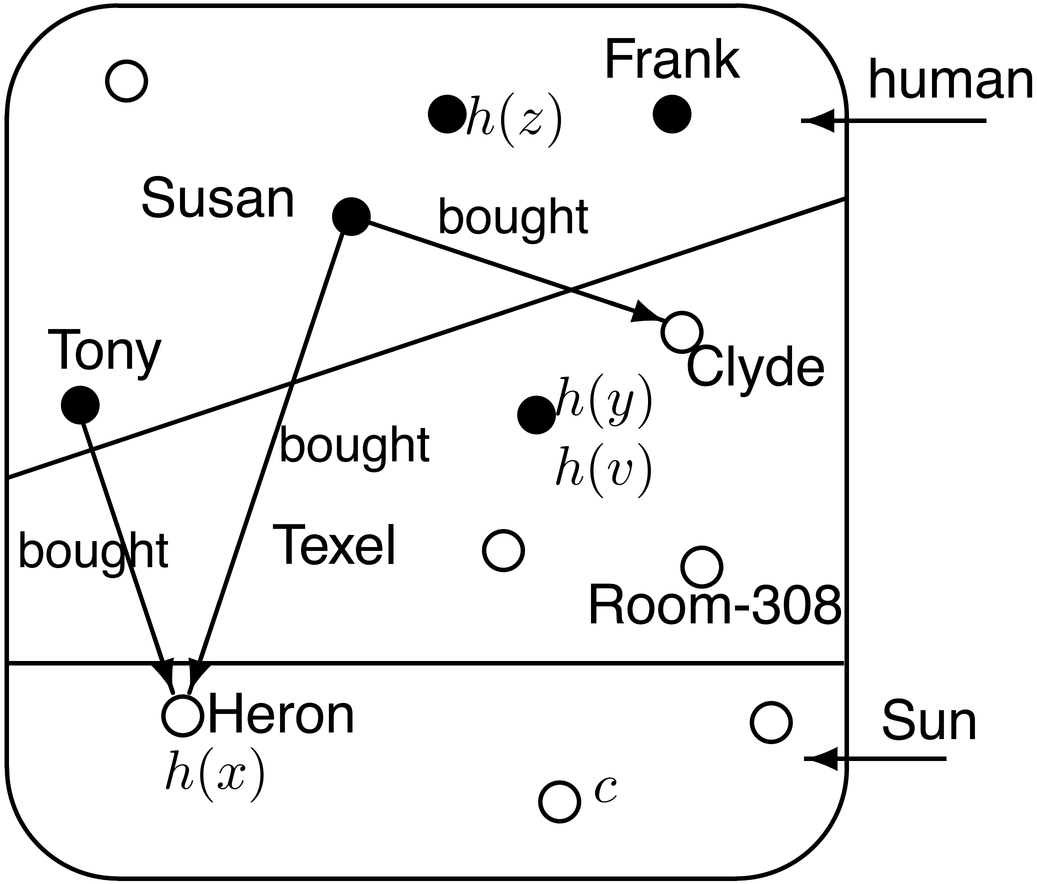
Definition 8.6 (value of term) Let L be a signature, M an L-structure, and h an assignment into M.

Then for any L-term t, the *value of t in M under h* is the object in M allocated to t by:

- M, if t is a constant — that is, t^M,
- h, if t is a variable — that is, $h(t)$.

Dead easy!

Evaluating terms: example



M, h

The value in M under h of the term `Tony` is (the ● marked) 'Tony'.

The value in M under h of the term x is Heron.

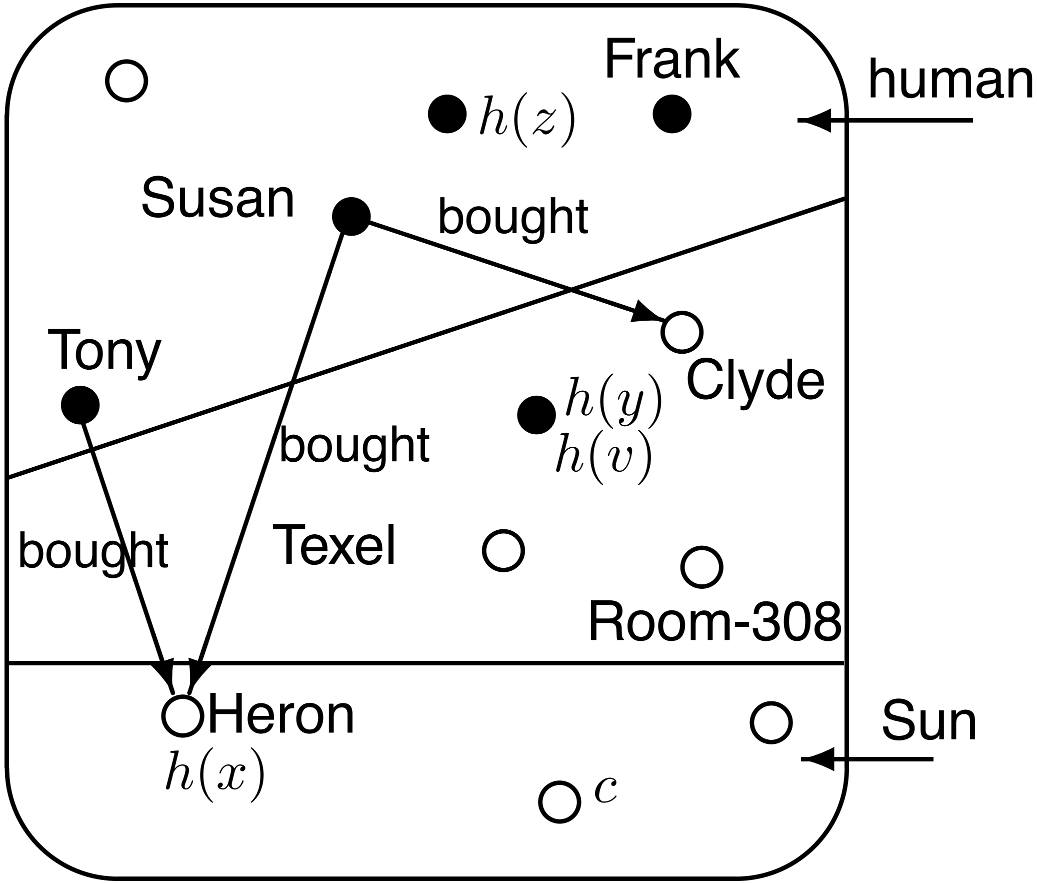
Semantics of quantifier-free formulas

We can now evaluate any formula without quantifiers.
Fix an L-structure M and an assignment h.
We write $M, h \models A$ if A is true in M under h, and $M, h \not\models A$ if not.

Definition 8.7

1. *Let R be an n-ary relation symbol in L, and $t_1, \ldots, t_n$ be L-terms. Suppose that the value of t_i in M under h is a_i, for each $i = 1, \ldots, n$ (see definition 8.6).*
   Then $M, h \models R(t_1, \ldots, t_n)$ if M says that the sequence $(a_1, \ldots, a_n)$ is in the relation R. If not, then $M, h \not\models R(t_1, \ldots, t_n)$.
2. *If t, t' are terms, then $M, h \models t = t'$ if t and t' have the same value in M under h. If they don't, then $M, h \not\models t = t'$.*
3. *$M, h \models \top$, and $M, h \not\models \bot$.*
4. *$M, h \models A \wedge B$ if $M, h \models A$ and $M, h \models B$.*
   Otherwise, $M, h \not\models A \wedge B$.
5. *$\neg A$, $A \vee B$, $A \to B$, $A \leftrightarrow B$ — similar: as in propositional logic.*

Evaluating quantifier-free formulas: example



M, h

- $M, h \models \texttt{human}(z)$
- $M, h \models x = \texttt{Heron}$
- $M, h \not\models \texttt{bought}(\texttt{Susan}, v) \lor z = \texttt{Frank}$

Problem 2: bound variables

We now know how to specify values for *free variables:* with an assignment. This allowed us to evaluate all quantifier-free formulas.

But most formulas involve quantifiers and *bound variables.* Values of bound variables are not — and should not be — given by the situation, as they are controlled by quantifiers.

How do we handle this?

Answer:

We let the assignment vary. Rough idea:

- for $\exists$, want *some* assignment to make the formula true;
- for $\forall$, demand that *all* assignments make it true.

Semantics of non-atomic formulas (definition 8.7 ctd.)

Notation (not very standard): Suppose that M is a structure, g, h are assignments into M, and x is a variable. We write $g =_x h$ if $g(y) = h(y)$ for all variables y *other than* x. (Maybe $g(x) = h(x)$ too!)

$g =_x h$ means 'g agrees with h on all variables except possibly x'.

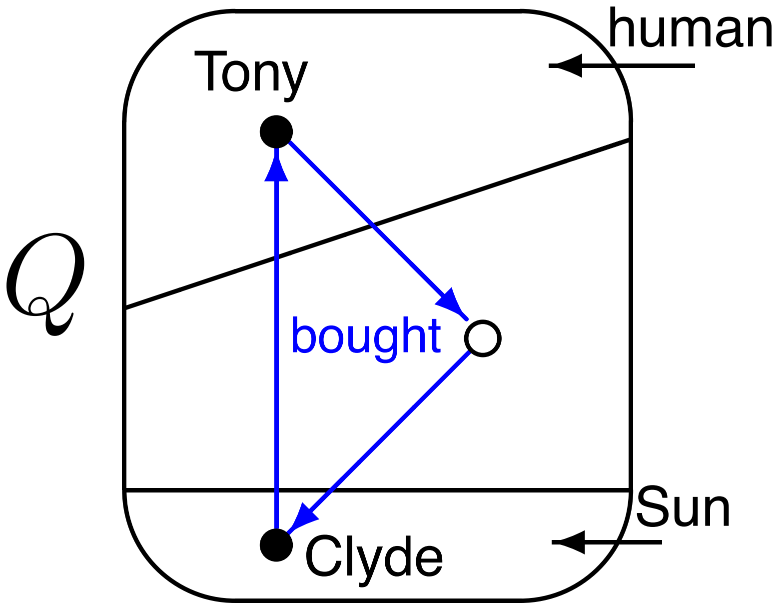
Warning: don't be misled by the '$=$' sign in $=_x$.
$g =_x h$ does not imply $g = h$, because we may have $g(x) \neq h(x)$.

Definition 8.7 (continued) Suppose we already know how to evaluate the formula A in M under any assignment. Let x be any variable, and h be any assignment into M. Then:

6. $M, h \models \exists x A$ *if* $M, g \models A$ *for* some *assignment* g *into* M *with* $g =_x h$. *If not, then* $M, h \not\models \exists x A$.

7. $M, h \models \forall x A$ *if* $M, g \models A$ *for* every *assignment* g *into* M *with* $g =_x h$. *If not, then* $M, h \not\models \forall x A$.

Evaluating formulas with quantifiers: simple example



$y \backslash x$	Tony	$\bigcirc$	Clyde	
Tony	h_1	h_2	h_3	$=_x$
$\bigcirc$	h_4	h_5	h_6	$=_x$
Clyde	h_7	h_8	h_9	$=_x$
	$\Vert_y$	$\Vert_y$	$\Vert_y$	

Eg: $h_2(x) = \bigcirc$, and $h_2(y) =$ Tony.

- $Q, h_2 \not\models \texttt{human}(x)$
- $Q, h_2 \models \exists x\ \texttt{human}(x)$, because there is an assignment g with $g =_x h_2$ and $Q, g \models \texttt{human}(x)$ — namely, $g = h_1$
- $Q, h_7 \not\models \forall x\ \texttt{human}(x)$, because it is not true that $Q, g \models \texttt{human}(x)$ for all g with $g =_x h_7$: e.g., $h_8 =_x h_7$ and $Q, h_8 \not\models \texttt{human}(x)$.

A more complex one: $Q, h_4 \models \forall x \exists y\, \texttt{bought}(x, y)$

For this to be true, we require $Q, g \models \exists y\, \texttt{bought}(x, y)$ for every assignment g into Q with $g =_x h_4$.

These are: h_4, h_5, h_6.

- $Q, h_4 \models \exists y\, \texttt{bought}(x, y)$, because
  - $h_4 =_y h_4$ and $Q, h_4 \models \texttt{bought}(x, y)$
- $Q, h_5 \models \exists y\, \texttt{bought}(x, y)$, because
  - $h_8 =_y h_5$ and $Q, h_8 \models \texttt{bought}(x, y)$
- $Q, h_6 \models \exists y\, \texttt{bought}(x, y)$, because
  - $h_3 =_y h_6$ and $Q, h_3 \models \texttt{bought}(x, y)$

So indeed, $Q, h_4 \models \forall x \exists y\, \texttt{bought}(x, y)$.

Useful notation for free variables

The following notation is useful for writing and evaluating formulas.

The books often write things like

'Let $A(x_1, \ldots, x_n)$ be a formula.'

This indicates that the free variables of A are among $x_1, \ldots, x_n$.

Note: $x_1, \ldots, x_n$ should all be different. And not all of them need actually occur free in A.

Example: if C is the formula

$$\forall x(R(x, y) \to \exists y S(y, z)),$$

we could write it as

- $C(y, z)$
- $C(x, z, v, y)$
- C (if we're not using the useful notation)

but not as $C(x)$.

Notation for assignments

Fact 8.8 *For any formula A, whether or not $M, h \models A$ only depends on $h(x)$ for those variables x that occur free in A.*

So for a formula $A(x_1, \ldots, x_n)$, if $h(x_1) = a_1$, ..., $h(x_n) = a_n$, it's OK to write $M \models A(a_1, \ldots, a_n)$ instead of $M, h \models A$.

- Suppose we are explicitly given a formula $C(y, z)$, such as

  $$\forall x(R(x, y) \to \exists y S(y, z)).$$

  If $h(y) = a$, $h(z) = b$, say, we can write

  $$M \models C(a, b), \text{ or } M \models \forall x(R(x, a) \to \exists y S(y, b)),$$

  instead of $M, h \models C$. Note: only the *free* occurrences of y in C are replaced by a. The bound y is unchanged.

- For a sentence S, whether $M, h \models S$ does not depend on h at all. So we can just write $M \models S$.

Working out $\models$ in this notation

Suppose we have an L-structure M, an L-formula $A(x, y_1, \ldots, y_n)$, and objects $a_1, \ldots, a_n$ in $\mathrm{dom}(M)$.

- To establish that $M \models (\forall x A)(a_1, \ldots, a_n)$ you check that $M \models A(b, a_1, \ldots, a_n)$ for each object b in $\mathrm{dom}(M)$.

  You have to check even those b with no constants naming them in M. 'Not just Frank, Texel, . . . , but all the other ○ and ● too.'

  We can summarise this as a *recursive procedure:*

  function istrue(M, B) : bool

  ...

  if $B = (\forall x A)(a_1, \ldots, a_n)$ {

  {repeat for all b in $\mathrm{dom}(M)$:

  {if not istrue($M, A(b, a_1, \ldots, a_n)$) then return false}

  return true}

  }

The case $\exists x A$, for $A(x, y_1, \ldots, y_n)$

- To establish $M \models (\exists x A)(a_1, \ldots, a_n)$, you try to find some object b in the domain of M such that $M \models A(b, a_1, \ldots, a_n)$.

```
function istrue(M, B) : bool
...
 if B = (∃xA)(a_1,...,a_n) {
  {repeat for all b in dom(M):
    {if istrue(M, A(b, a_1,...,a_n)) then return true}
  return false}
 }
```

A is simpler than $\forall x A$ or $\exists x A$. So you can recursively work out if $M \models A(b, a_1, \ldots, a_n)$, in the same way. The process terminates.

Exercise: write the whole function `istrue`. Then implement in Haskell!

So how to evaluate in practice?

We've just seen the formal definition of truth in a structure (due to Alfred Tarski, 1933–1950s).

But how best to work out whether $M \models A$ in practice?

- Often you can do it by working out the English meaning of A and checking it against M. We did this in section 2.2. See slide 134 for advice.
- Use definition 8.7 and check all assignments. Tedious, but can often do mentally with practice. E.g., in $\forall x(\mathtt{lecturer}(x) \to \mathtt{Sun}(x))$, run through all x and check that every x that's a lecturer is a Sun.
- Rewrite the formula in a more understandable form using equivalences (see later).
- Use a combination of the three.
- Try LOST. . .

LOST — LOgic Semantics Tutor

LOST lets you create and load signatures, structures, and sentences. You can evaluate sentences in structures

- automatically,
- interactively, using ‘Hintikka games’ (not covered in lectures).

It gives you instant feedback, and help.

It complements Pandora.

Leon Mouzourakis’s 2007 version should be available in the labs: try typing ‘lost &’ at a Linux terminal.

Please try LOST. It will help you learn semantics.

How hard is first-order evaluation?

In most practical cases, with a sentence written by a (sane) human, it's easy to do the evaluation mentally, once used to it.

If the sentence makes no sense, you may have to evaluate it by checking all assignments. Tedious but straightforward.

But in general, evaluation is hard.

It is generally believed that $\forall x \exists y \forall z \exists t \forall u \exists v A$ is just too difficult to understand.

Suppose that N is the structure whose domain is the natural numbers and with the usual meanings of $\mathtt{prime}$, $\mathtt{even}$, $>$, $+$, 2.

No-one knows whether

$$N \models \forall x(\mathtt{even}(x) \wedge x > 2 \rightarrow \exists y \exists z(\mathtt{prime}(y) \wedge \mathtt{prime}(z) \wedge x = y + z)).$$

9. Translation into and out of logic

Translating predicate logic sentences *from logic to English* is not much harder than in propositional logic.

But you need to use standard English constructions when translating certain logical patterns.
Example: $\forall x(A \to B)$. Rough translation: 'every A is a B'.

Also, you can end up with a mess that needs careful simplifying.
You'll need common sense!

Variables must be eliminated: English doesn't use them.

Examples

$\forall x(\texttt{lecturer}(x) \wedge \neg(x = \texttt{Frank}) \to \texttt{bought}(x, \texttt{Texel}))$

'For all x, if x is a lecturer and x is not Frank then x bought Texel.'

'Every lecturer apart from Frank bought Texel.' (Maybe Frank did too.)

$\exists x \exists y \exists z(\texttt{bought}(x, y) \wedge \texttt{bought}(x, z) \wedge \neg(y = z))$

'There are x, y, z such that x bought y, x bought z, and y is not z.'

'Something bought at least two different things.'

$\forall x(\exists y \exists z(\texttt{bought}(x, y) \wedge \texttt{bought}(x, z) \wedge \neg(y = z)) \to x = \texttt{Tony})$

'For all x, if x bought two different things then x is equal to Tony.'

'Anything that bought two different things is Tony.'

Care: it doesn't say Tony did buy 2 things, just that noone else did.

Over to you...

1. $\forall x(\texttt{lecturer}(x) \rightarrow \texttt{bought}(x, \texttt{Clyde}))$

2. $\forall x(\texttt{lecturer}(x) \wedge \texttt{bought}(x, \texttt{Clyde}))$

3. $\exists x(\texttt{lecturer}(x) \wedge \texttt{bought}(x, \texttt{Clyde}))$

4. $\exists x(\texttt{lecturer}(x) \rightarrow \texttt{bought}(x, \texttt{Clyde}))$

English to logic translation: advice I

Express the sub-concepts in logic. Then build these pieces into a whole logical sentence.

- Sub-concept *'x is bought'/'x has a buyer'*: $\exists y\ \mathtt{bought}(y, x)$.
- Any bought thing isn't human:
  $\forall x(\exists y\ \mathtt{bought}(y, x) \to \neg\,\mathtt{human}(x))$.
  Important: $\forall x \exists y(\mathtt{bought}(y, x) \to \neg\,\mathtt{human}(x))$ would not do.
- Every Sun was bought: $\forall x(\mathtt{Sun}(x) \to \exists y\ \mathtt{bought}(y, x))$.
- Some Sun has a buyer: $\exists x(\mathtt{Sun}(x) \wedge \exists y\ \mathtt{bought}(y, x))$.
- No lecturer bought a Sun:
  $\neg\exists x(\mathtt{lecturer}(x) \wedge \underbrace{\exists y(\mathtt{bought}(x, y) \wedge \mathtt{Sun}(y))}_{x \text{ bought a Sun}})$.

English-to-logic translation: advice II (common patterns)

You often need to say things like:

- ‘All lecturers are human’: $\forall x(\texttt{lecturer}(x) \to \texttt{human}(x))$.
  NOT $\forall x(\texttt{lecturer}(x) \wedge \texttt{human}(x))$.
  NOT $\forall x\, \texttt{lecturer}(x) \to \forall x\, \texttt{human}(x)$.
- ‘Some lecturer is human’: $\exists x(\texttt{lecturer}(x) \wedge \texttt{human}(x))$.
  NOT $\exists x(\texttt{lecturer}(x) \to \texttt{human}(x))$.

The patterns $\forall x(A \to B)$ and $\exists x(A \wedge B)$, are therefore very common.

$\forall x(A \wedge B)$, $\forall x(A \vee B)$, $\exists x(A \vee B)$ also crop up: they say everything/something is A and/or B.

But $\exists x(A \to B)$, especially if x occurs free in A, is *extremely rare.* If you write it, check to see if you've made a mistake.

English-to-logic translation: advice III (counting)

- There is at least one Sun: $\exists x\ \texttt{Sun}(x)$.
- There are at least two Suns: $\exists x \exists y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge x \neq y)$, or (more deviously) $\forall x \exists y(\texttt{Sun}(y) \wedge y \neq x)$.
- There are at least three Suns: $\exists x \exists y \exists z(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge \texttt{Sun}(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z)$, or $\forall x \forall y \exists z(\texttt{Sun}(z) \wedge z \neq x \wedge z \neq y)$.
- There are no Suns: $\neg \exists x\ \texttt{Sun}(x)$
- There is at most one Sun: 3 ways:
  1. $\neg \exists x \exists y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge x \neq y)$

     This says 'not(there are at least two Suns)' — see above.
  2. $\forall x \forall y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \to x = y)$
  3. $\exists x \forall y(\texttt{Sun}(y) \to y = x)$
- There's exactly one Sun: 2 ways:
  1. 'There's at least one Sun' $\wedge$ 'there's at most one Sun'.
  2. $\exists x \forall y(\texttt{Sun}(y) \leftrightarrow y = x)$.

10. Function symbols and sorts

— the icing on the cake.

10.1 Function symbols

In arithmetic (and Haskell) we are used to *functions,* such as $+, -, \times, \sqrt{x}, ++$, etc.

Predicate logic can do this too.

A *function symbol* is like a relation symbol or constant, but it is interpreted in a structure as a *function* (to be defined in discr math).

Any function symbol comes with a fixed arity (number of arguments).

We often write f, g for function symbols.

From now on, we adopt the following extension of definition 7.1:

Definition 10.1 (signature) *A* signature *is a collection of constants, and relation symbols and function symbols with specified arities.*

Terms with function symbols

We can now extend definition 7.2:

Definition 10.2 (term) *Fix a signature L.*

1. *Any constant in L is an L-term.*
2. *Any variable is an L-term.*
3. *If f is an n-ary function symbol in L, and $t_1, \ldots, t_n$ are L-terms, then $f(t_1, \ldots, t_n)$ is an L-term.*
4. *Nothing else is an L-term.*

Example

Let L have a constant c, a unary function symbol f, and a binary function symbol g. Then the following are L-terms:

- c
- $f(c)$
- $g(x, x)$ (x is a variable, as usual)
- $g(f(c), g(x, x))$

The first two are closed, or ground, terms. The last two are not.

Semantics of function symbols

We need to extend definition 8.1 too: if L has function symbols, an L-structure must additionally define their meaning.

For any n-ary function symbol f in L, an L-structure M *must* say which object (in $\mathrm{dom}(M)$) f associates with each sequence $(a_1, \ldots, a_n)$ of objects in $\mathrm{dom}(M)$.
We write this object as $f^M(a_1, \ldots, a_n)$. There must be such a value.

[Formally, f^M is a function $f^M : \mathrm{dom}(M)^n \to \mathrm{dom}(M)$.]
A 0-ary function symbol is like a constant.

Examples

In arithmetic, M *might* say $+, \times$ are addition and multiplication of numbers: it associates 5 with $2+3$, 8 with 4×2, etc.

If the objects of M are vectors, M might say $+$ is addition of vectors and $\times$ is cross-product. M doesn't have to say this — it could say $\times$ is addition — but nobody would want such an M.

Evaluating terms with function symbols

We can now extend definition 8.6:

Definition 10.3 (value of term) The value of an L-term t in an L-structure M under an assignment h into M is defined as follows:

- If t is a constant, then its value is the object t^M in M allocated to it by M,
- If t is a variable, then its value is the object $h(t)$ in M allocated to it by h,
- If t is $f(t_1, \ldots, t_n)$, and the values of the terms $t_1, \ldots, t_n$ in M under h are already known to be $a_1, \ldots, a_n$, respectively, then the value of t in M under h is $f^M(a_1, \ldots, a_n)$.

So the value of a term in M under h is always *an object in* $\mathrm{dom}(M)$*, rather than true or false!*

Definition 8.7 needs no amendment, apart from using it with the extended definition 10.3.

We now have the standard system of first-order logic (as in books).

Example: arithmetic terms

A useful signature for arithmetic and for programs using numbers is the L consisting of:

- constants $\underline{\mathtt{0}}$, $\underline{\mathtt{1}}$, $\underline{\mathtt{2}}$, ... (I use underlined typewriter font to avoid confusion with actual numbers $0, 1, \ldots$)
- binary function symbols $+, -, \times$
- binary relation symbols $<, \leq, >, \geq$.

We interpret these in a structure with domain $\{0, 1, 2, \ldots\}$ in the obvious way. But (eg) $34 - 61$ is unpredictable — can be any number.

We'll abuse notation by writing L-terms and formulas in infix notation:
- $x + y$, rather than $+(x, y)$,
- $x > y$, rather than $>(x, y)$.

Everybody does this, but it's breaking definitions 10.2 and 7.3.

Some terms: $x + \underline{\mathtt{1}}$, $\underline{\mathtt{2}} + (x + \underline{\mathtt{5}})$, $(\underline{\mathtt{3}} \times \underline{\mathtt{7}}) + x$. Not $x + y + z$.
Formulas: $\underline{\mathtt{3}} \times x > \underline{\mathtt{0}}$, $\forall x(x > \underline{\mathtt{0}} \to x \times x > x)$.

10.2 Many-sorted logic

As in typed programming languages, it sometimes helps to have structures with objects of different types. In logic, types are called *sorts.*

Eg some objects in a structure M may be lecturers, others may be Suns, numbers, etc.

We can handle this with unary relation symbols, or with *'many-sorted first-order logic'.* We'll use many-sorted logic mainly to specify programs.

Fix a collection $\mathrm{s}, \mathrm{s}', \mathrm{s}'', \ldots$ of sorts. How many, and what they're called, are determined by the application.

These sorts do *not* generate extra sorts, like $\mathrm{s} \to \mathrm{s}'$ or $(\mathrm{s}, \mathrm{s}')$.
If you want extra sorts like these, add them explicitly to the original list of sorts. (Their meaning would not be automatic, unlike in Haskell.)

Many-sorted terms

We adjust the definition of 'term' (definition 10.2), to give each term a sort:

- each variable and constant comes with a sort s. To indicate which sort it is, we write $x : \mathbf{s}$ and $c : \mathbf{s}$. There are infinitely many variables of each sort.

- each n-ary function symbol f comes with a template

$$f : (\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s},$$

  where $\mathrm{s}_1, \ldots, \mathrm{s}_n$, and s are sorts.
  Note: $(\mathrm{s}_1, \ldots, \mathrm{s}_n) \to \mathrm{s}$ is not itself a sort.

- For such an f and terms $t_1, \ldots, t_n$, if t_i has sort s_i (for each i) then $f(t_1, \ldots, t_n)$ is a term of sort s.

  Otherwise (if the t_i don't all have the right sorts), $f(t_1, \ldots, t_n)$ is not a term — it's just rubbish, like $)\forall)\to$.

Formulas in many-sorted logic

- Each n-ary relation symbol R comes with a template $R(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, where $\mathrm{s}_1, \ldots, \mathrm{s}_n$ are sorts.
  For terms $t_1, \ldots, t_n$, if t_i has sort s_i (for each i) then $R(t_1, \ldots, t_n)$ is a formula. Otherwise, it's rubbish.
- $t = t'$ is a formula if the terms t, t' have the same sort. Otherwise, it's rubbish.
- Other operations ($\wedge, \neg, \forall, \exists$, etc) are unchanged. But it's polite to indicate the sort of a variable in $\forall, \exists$ by writing

  $\forall x : \mathbf{s}\ A$ and $\exists x : \mathbf{s}\ A$

  instead of just

  $\forall x A$ and $\exists x A$

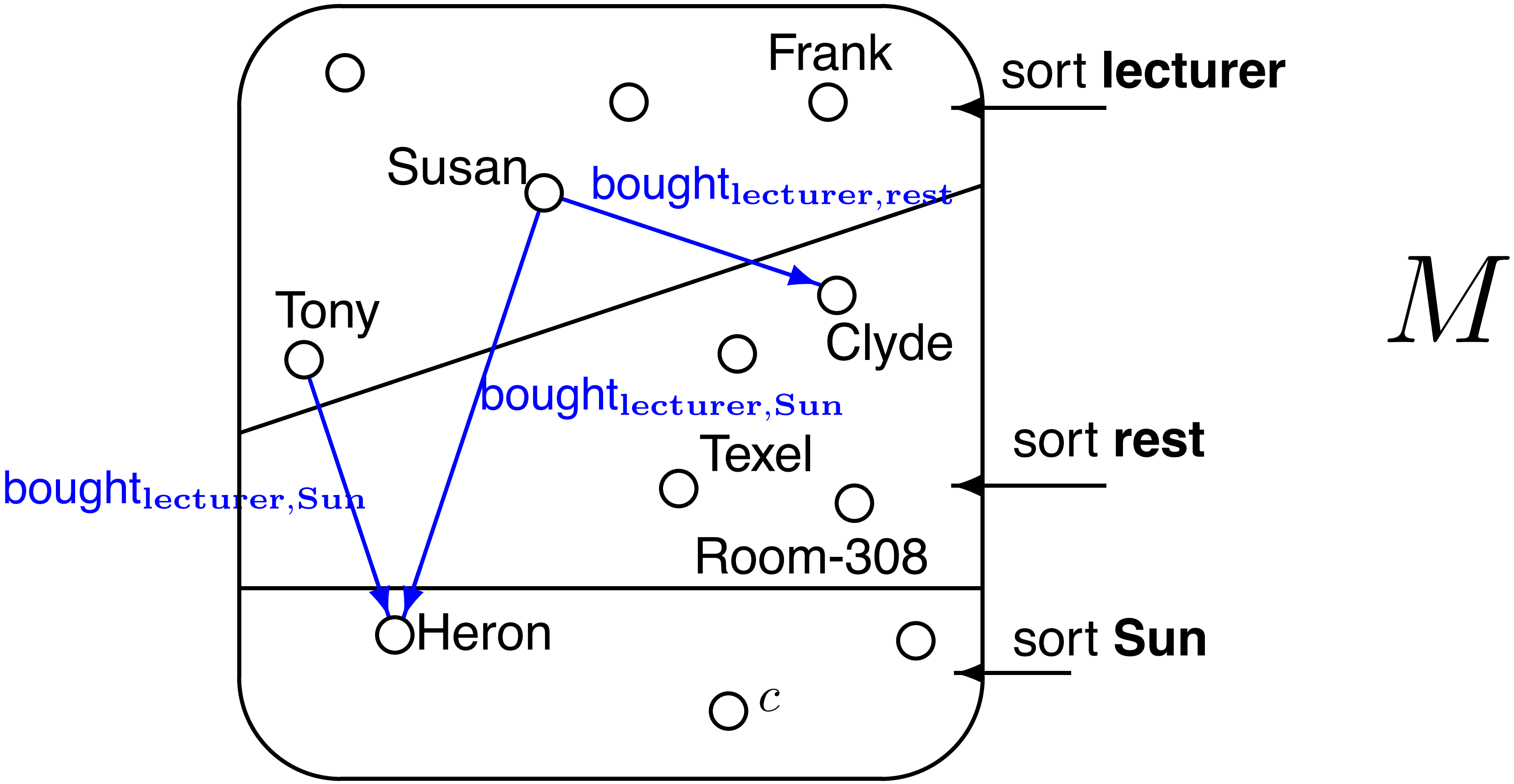
  if x has sort s.

This all sounds complicated, but it's very simple in practice.
Eg, roughly, you can write $\forall x : \mathbf{lecturer}\ \exists y : \mathbf{Sun}(\mathtt{bought}(x, y))$ instead of $\forall x(\mathtt{lecturer}(x) \to \exists y(\mathtt{Sun}(y) \wedge \mathtt{bought}(x, y)))$.

L-structures for many-sorted L — example

Let L be a many-sorted signature. An L-structure is defined as before (definition 8.1 + slide 165), but additionally it allocates *each* object in its domain to *a single sort.* No sort should be empty.

Eg if L has sorts $\mathbf{lecturer}$, $\mathbf{Sun}$, $\mathbf{rest}$, an L -structure looks like:



Interpretation of L-symbols in L-structures

Let M be a many-sorted L-structure.

- For each constant $c : \mathbf{s}$ in L, M must say which object of sort s in $\mathrm{dom}(M)$ is 'named' by c.
- For each function symbol $f : (\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s}$ in L and all objects $a_1, \ldots, a_n$ in $\mathrm{dom}(M)$ of sorts $\mathrm{s}_1, \ldots, \mathrm{s}_n$, respectively, M must say which object $f^M(a_1, \ldots, a_n)$ of sort s is associated with $(a_1, \ldots, a_n)$ by f.

  M doesn't say anything about $f(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.
- For each relation symbol $R(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ in L, and all objects $a_1, \ldots, a_n$ in $\mathrm{dom}(M)$ of sorts $\mathrm{s}_1, \ldots, \mathrm{s}_n$, respectively, M must say whether $R(a_1, \ldots, a_n)$ is true or not.

  M doesn't say anything about $R(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.

Notes

1. Sorts can replace some or all unary relation symbols.
2. As in Haskell, each object has only 1 sort, not 2.

   So for M above, $\mathtt{human}$ would have to be implemented as three unary relation symbols: $\mathtt{human}_{\mathbf{lecturer}}$, $\mathtt{human}_{\mathbf{Sun}}$, $\mathtt{human}_{\mathbf{rest}}$.

   But if (e.g.) you don't want to talk about human objects of sort $\mathbf{Sun}$, you can omit $\mathtt{human}_{\mathbf{Sun}}$.
3. We need a binary relation symbol $\mathtt{bought}_{\mathbf{s},\mathbf{s}'}$ *for each pair* $(\mathrm{s}, \mathrm{s}')$ *of sorts* (unless s-objects are not expected to buy s'-objects).
4. Messy alternative: use sorts for human lecturer, Sun-lecturer, etc — all possible types of object.

Quantifiers in many-sorted logic

Semantics of formulas is defined as before (definition 8.7), but assignments must respect sorts of variables.

In a nutshell: if variable x has sort s, then $\forall x$ and $\exists x$ range over objects of sort s only.

For example, $\forall x : \mathbf{lecturer}\ \exists y : \mathbf{Sun}(\mathtt{bought}_{\mathtt{lecturer},\mathtt{Sun}}(x, y))$ is true in a structure if every object of sort $\mathbf{lecturer}$ bought an object of sort $\mathbf{Sun}$.

It is not the same as $\forall x\, \exists y\ \mathtt{bought}(x, y)$.

It does not say that every $\mathbf{Sun}$-object bought a $\mathbf{Sun}$-object as well (etc etc).

Do not get worried about many-sorted logic. It looks complicated, but it's easy once you practise. It is there to help you (like types in programming), and it can make life easier.

11. Application of logic: specifications

A *specification* is a description of what a program should do.

It should state the inputs and outputs (and their types).

It should include conditions on the input under which the program is guaranteed to operate. This is the *pre-condition.*

It should state what is required of the outcome in all cases (output for each input). This is the *post-condition.*

- The type (in the function header) is part of the specification.
- The pre-condition refers to the inputs (only).
- The post-condition refers to the outputs and inputs.

Precision is vital

A specification should be unambiguous. It is a *CONTRACT!*

Programmer wants pre-condition and post-condition to be the same — less work to do! The weaker the pre-condition and/or stronger the post-condition, the more work for the programmer — fewer assumptions (so more checks) and more results to produce.

Customer wants weak pre-condition and strong post-condition, for added value — less work before execution of program, more gained after execution of it.

Customer guarantees pre-condition so program will operate. Programmer guarantees post-condition, provided that the input meets the pre-condition.

If customer (user) provides the pre-condition (on the inputs), then provider (programmer) will guarantee the post-condition (between inputs and outputs).

11.1 Logic for specifying Haskell programs

A very precise way to specify properties of Haskell programs is to use first-order logic.
(Logic can also be used for Java, etc.)

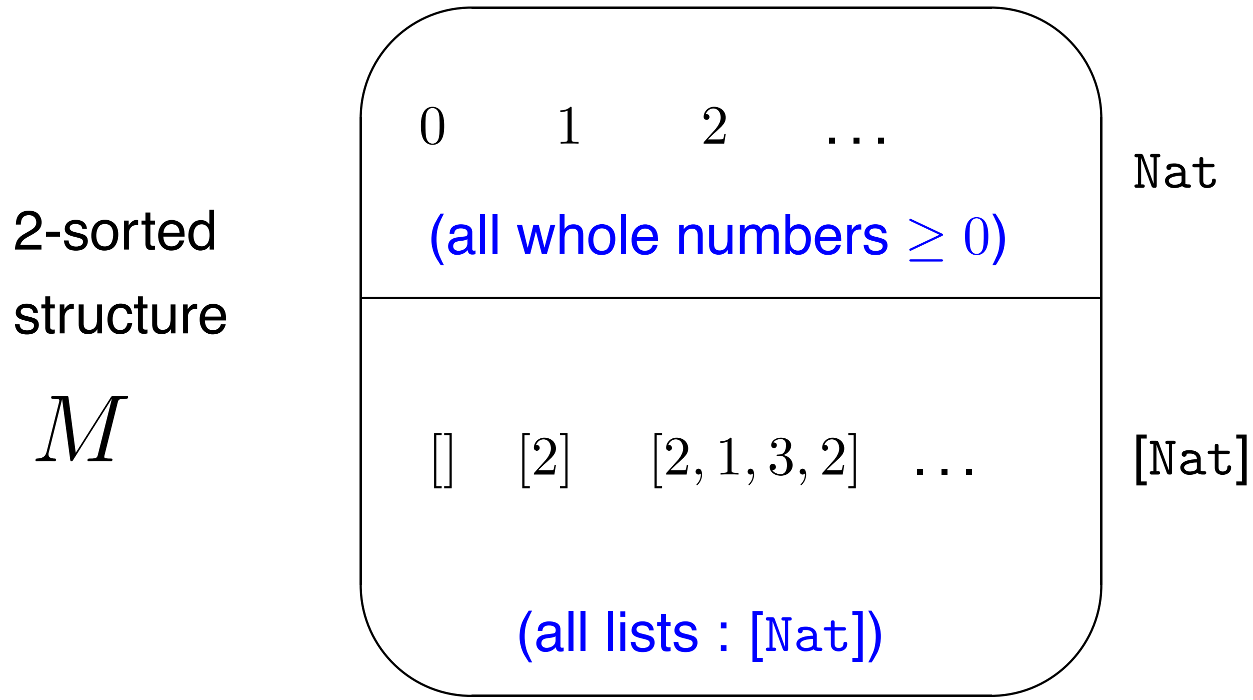
Next term: gory details.
This term: a gentle taster (but still very powerful).

We use many-sorted logic, so we can have a sort for each Haskell type we want.

Example: lists of type [`Nat`]

Let's have a sort `Nat`, for $0, 1, 2, \ldots$, and a sort [`Nat`] for lists of natural numbers.
(Using the actual Haskell `Int` is more longwinded: must keep saying $n \geq 0$ etc.)

The idea is that the structure's domain should look like:



11.2 Signature for lists

The signature should be chosen to provide access to the objects in such a structure.

We want [], : (cons), ++, head, tail, $\sharp$, !!.

And $+$, $-$, etc., for arithmetic.

How do we represent these using constants, function symbols, or relation symbols?

Problem: `tail` etc are partial operations

In first-order logic, a structure *must* provide a meaning for function symbols *on all possible arguments* (of the right sorts).
But what is the head or tail of the empty list? What is $xs\,!!\,\sharp(xs)$?
What is $34 - 61$?

Two solutions (for `tail`; the others are similar):

1. Use a function symbol $\mathtt{tail} : [\mathtt{Nat}] \to [\mathtt{Nat}]$.
   Choose an arbitrary value (of the right sort) for $\mathtt{tail}([])$.

2. Use a relation symbol Rtail([Nat],[Nat]) instead.
   Make $\mathtt{Rtail}(xs, ys)$ true just when ys is the tail of xs.
   If xs has no tail, $\mathtt{Rtail}(xs, ys)$ will be false for all ys.

We'll take the function symbol option (1), as it leads to shorter formulas. But always beware:

Warning: values of functions on 'invalid' arguments are 'unpredictable'.

Lists in first-order logic: summary

Now we can define a signature L suitable for lists of type [`Nat`].

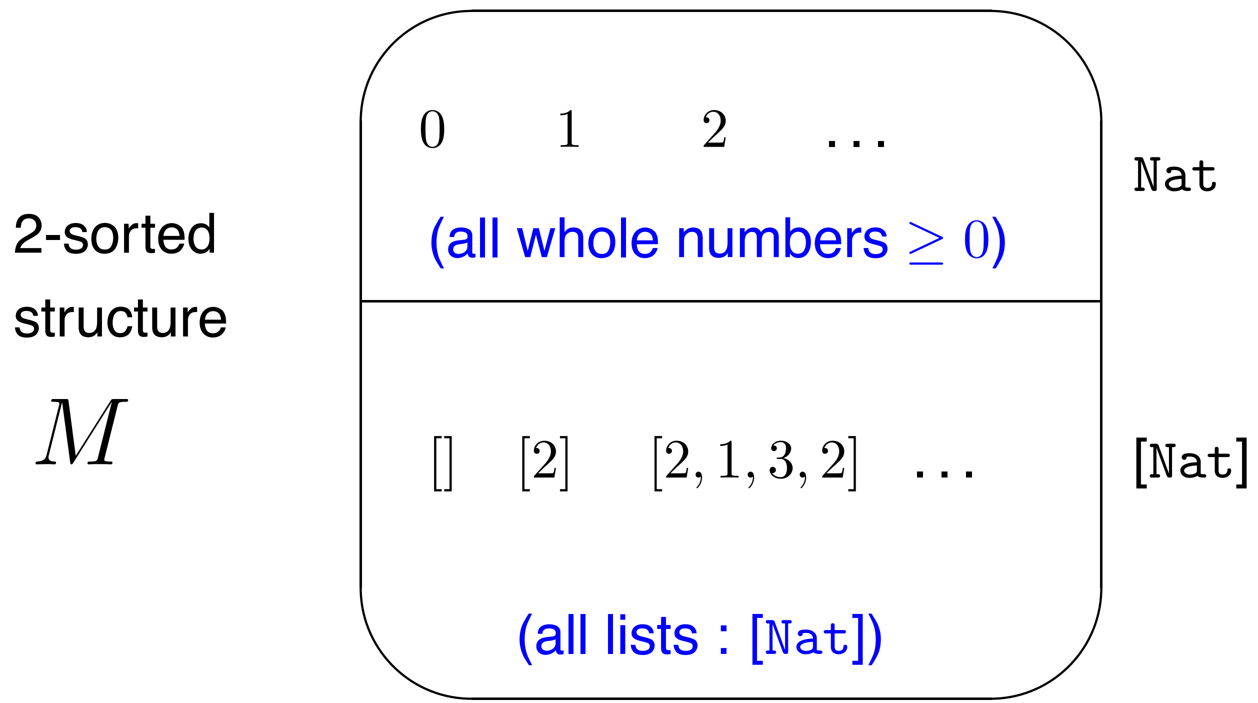
- L has constants $\underline{0}, \underline{1}, \ldots : \mathtt{Nat}$, relation symbols $<, \leq, >, \geq$ of sort (`Nat`,`Nat`), and function symbols
  - $+, -, \times : (\mathtt{Nat}, \mathtt{Nat}) \to \mathtt{Nat}$
  - $[] : [\mathtt{Nat}]$ (a constant to name the empty list)
  - $\mathtt{cons}(:) \;: (\mathtt{Nat}, [\mathtt{Nat}]) \to [\mathtt{Nat}]$
  - $++ : ([\mathtt{Nat}], [\mathtt{Nat}]) \to [\mathtt{Nat}]$
  - $\mathtt{head} : [\mathtt{Nat}] \to \mathtt{Nat}$
  - $\mathtt{tail} : [\mathtt{Nat}] \to [\mathtt{Nat}]$
  - $\sharp : [\mathtt{Nat}] \to \mathtt{Nat}$
  - $!! : ([\mathtt{Nat}], \mathtt{Nat}) \to \mathtt{Nat}$

  We write the constants as $\underline{0}, \underline{1}, \ldots$ to avoid confusion with actual numbers $0, 1, \ldots$

- Let $x, y, z, k, n, m \ldots$ be variables of sort `Nat`.
  Let $xs, ys, zs, \ldots$ be variables of sort $[\mathtt{Nat}]$.

Semantics

Let M be the L-structure



The L-symbols are interpreted in the natural way: ++ as concatenation of lists, etc.

We define $34 - 61$, $\texttt{tail}([])$, etc. arbitrarily. So don't assume they have the values you might expect.

11.3 Saying things about lists

Now we can say *a lot* about lists.

E.g., the following L-sentences, expressing the definitions of the function symbols, are true in M, because (as we said) the L-symbols are interpreted in M in the natural way:

$\sharp([\,]) = \underline{0}$ Indeed, $\forall xs(\sharp(xs) = \underline{0} \leftrightarrow xs = [\,])$ is also true.

$\forall x \forall xs((\sharp(x : xs) = \sharp(xs) + \underline{1}) \wedge ((x : xs)!!\underline{0} = x))$

$\forall x \forall xs \forall n(n < \sharp(xs) \rightarrow (x : xs)!!(n + \underline{1}) = xs!!n)$

The '$n < \sharp(xs)$' is necessary. $xs!!n$ could be anything if $n \geq \sharp(xs)$.

$\forall xs(xs \neq [\,] \rightarrow \mathtt{head}(xs) = xs!!\underline{0})$, and $\forall x \forall xs(\mathtt{head}(x : xs) = x)$

$\forall x \forall xs(\mathtt{tail}(x : xs) = xs)$

$$\begin{aligned}\forall xs \forall ys \forall zs\big(xs = ys +\!\!+\, zs \leftrightarrow &\\ &\sharp(xs) = \sharp(ys) + \sharp(zs) \wedge \forall n(n < \sharp(ys) \rightarrow xs!!n = ys!!n)\\ &\wedge \forall n(n < \sharp(zs) \rightarrow xs!!(n + \sharp(ys)) = zs!!n)\big).\end{aligned}$$

11.4 Specifying Haskell functions

Now we know how to use logic to say things about lists, we can use logic to specify Haskell functions. There are three bits to it.

1. Type information

This is stuff like ‘the first argument is a `Nat` and the second a [`Nat`]’.

It is determined by the program header.

It is *not* part of the pre-condition (coming next).

2. Pre-conditions in logic

The pre-condition expresses restrictions on the arguments or parameters that can be legally passed to a function.

To do a pre-condition in logic, *you write a formula* $A(x_1, \ldots, x_n)$ so that any arguments $a_1, \ldots, a_n$ satisfy the intended pre-condition ('are legal') if and only if $A(a_1, \ldots, a_n)$ is true.

E.g., for the function $\log(x)$, you'd want a pre-condition of $x > \underline{0}$.
For $\max xs$ you'd want $xs \neq [\,]$.

Pre-conditions are usually very easy to write:

- xs is not empty: use $xs \neq [\,]$.
- n is positive: use $n > \underline{0}$.

If there are no restrictions on the arguments beyond their type information, you can write 'none', or $\top$, as pre-condition. This is perfectly normal and is no cause for alarm.

3. Post-conditions in logic

The post-condition expresses the required connection between the input and output of a function.

It expresses *what* the program does, but not *how* the program works. It can look completely different from the program code.

To do a post-condition in logic, *you write a formula* expressing the intended value of a function in terms of its arguments.

The formula should have free variables for the arguments, and should involve the function call so as to describe the required value.

The formula should be true if and only if the output is as intended.

Existence, non-uniqueness of result

Suppose you have a post-condition $A(x, y, z)$, where the variables x, y represent the input, and z represents the output.

Idea: for inputs a, b in M satisfying the pre-condition (if any), the function should return some c such that $M \models A(a, b, c)$.

There is no requirement that c be unique! We could well have $M \models A(a, b, c) \wedge A(a, b, d) \wedge c \neq d$. Then the function could legally return c or d. It can return *any* value satisfying the post-condition.

But should arrange that $M \models \exists z A(a, b, z)$ whenever a, b meet the pre-condition: otherwise, the function cannot meet its post-condition.

So need $M \models \forall x \forall y (pre(x, y) \to \exists z \; post(x, y, z))$, for functions of 2 arguments with pre-, post-conditions given by formulas $pre, post$.

Example: specifying the function ‘isin’

```
isin :: Nat -> [Nat] -> Bool
-- pre:none
-- post: isin x xs <--> (E)k:Nat(k<#xs & xs!!k=x)
```

- This is $\mathtt{isin}(x, xs) \leftrightarrow \exists k : \mathtt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
  I used `(E)` and `&`, as I can't type $\exists, \wedge$ in Haskell.
  Similarly, use \/ or | for $\vee$, `(A)` for $\forall$, and `~` or `!` for $\neg$.
- For any number a and list bs in M, we have
  $M \models \exists k : \mathtt{Nat}(k < \sharp(bs) \wedge bs!!k = a)$ just when a occurs in bs.
  So we have the intended post-condition for `isin`.
- $\forall x \forall xs(\mathtt{isin}(x, xs) \leftrightarrow \exists k : \mathtt{Nat}(k < \sharp(xs) \wedge xs!!k = x))$ would be better, but it's traditional to use free variables for the function arguments. Implicitly, though, they are universally quantified.
- We treat functions with boolean values (like `isin`) as relation symbols. Functions that return number or list values (values in $\mathrm{dom}(M)$) are treated as function symbols.

Least entry

Write $in(x, xs)$ for the formula $\exists k : \mathtt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
Then $in(m, xs) \wedge \forall n(in(n, xs) \to n \geq m)$
expresses that (is true in M iff) m is the least entry in list xs.

So could specify a function `least`:

```
least :: [Nat] -> Nat
-- pre: input is non-empty
-- post: in(m,xs) & (A)n(in(n,xs) -> n>=m), where m = least xs
```

Ordered (or sorted) lists

$\forall n \forall m(n < m \wedge m < \sharp(xs) \to xs!!n \leq xs!!m)$ says that xs is ordered.
So does $\forall ys \forall zs \forall m \forall n(xs = ys{+}{+}(m:(n:zs)) \to m \leq n)$.

Exercise: specify a function

```
sorted ::  [Nat] -> Bool
```

that returns true if and only if its argument is an ordered list.

Merge

Informal specification:

```
merge :: [Nat] -> [Nat] -> [Nat] -> Bool
-- pre:none
-- post:merge(xs,ys,zs) holds when xs, ys are
-- merged to give zs, the elements of xs and ys
-- remaining in the same relative order.
```

merge([1,2], [3,4,5], [1,3,4,2,5]) and
merge([1,2], [3,4,5], [3,4,1,2,5]) are true.

merge([1,2], [3,4,5], [1]) and
merge([1,2], [3,4,5], [5,4,3,2,1]) are false.

Specifying ‘merge’

Quite hard to specify explicitly (challenge for you!).

But can write an implicit specification:

$$\forall ys \forall zs(\texttt{merge}([], ys, zs) \leftrightarrow ys = zs)$$

$$\begin{aligned}\forall x \ldots zs[\texttt{merge}(x : xs, y : ys, z : zs) \leftrightarrow (&x = z \wedge \texttt{merge}(xs, y : ys, zs) \\ &\vee y = z \wedge \texttt{merge}(x : xs, ys, zs))]\end{aligned}$$

This pins down `merge` exactly: there exists a unique way to interpret a 3-ary relation symbol `merge` in M so that these sentences are true. So I suppose they could form a post-condition.

Count

Can use `merge` to specify other things:

```
count : Nat -> [Nat] -> Nat
-- pre:none
-- post (informal): count x xs  = number of x's in xs
-- post: (E)ys,zs(merge ys zs xs
--         & (A)n:Nat(in(n,ys) -> n=x)
--         & (A)n:Nat(in(n,zs) -> n<>x)
--         & count x xs = #ys)
```

Idea: ys takes all the x from xs, and zs takes the rest. So the number of x is $\sharp(ys)$.

Conclusion

First-order logic is a valuable and powerful way to specify programs precisely, by writing first-order formulas expressing their pre- and post-conditions.

More on this in 141 'Reasoning about Programs' next term.

12. Arguments, validity

Predicate logic is a big jump up from propositional logic.
But still, our experience with propositional logic tells us how to define 'valid argument' etc.

Definition 12.1 (valid argument)
Let L be a signature and $A_1, \ldots, A_n, B$ be L-formulas.
An argument '$A_1, \ldots, A_n$, therefore B' is valid *if for any L-structure M and assignment h into M,*
if $M, h \models A_1$, $M, h \models A_2$, ..., and $M, h \models A_n$, then $M, h \models B$.
We write $A_1, \ldots, A_n \models B$ in this case.

This says: in any situation (structure + assignment) in which $A_1, \ldots, A_n$ are all true, B must be true too.

Special case: $n = 0$. Then we write just $\models B$. It means that B is true in every L-structure under every assignment into it.

Validity, satisfiability, equivalence

These are defined as in propositional logic. Let L be a signature.

Definition 12.2 (valid formula)
An L-formula A is (logically) valid *if for every L-structure M and assignment h into M, we have $M, h \models A$.*
We write '$\models A$' (as above) if A is valid.

Definition 12.3 (satisfiable formula)
An L-formula A is satisfiable *if for some L-structure M and assignment h into M, we have $M, h \models A$.*

Definition 12.4 (equivalent formulas)
L-formulas A, B are logically equivalent *if for every L-structure M and assignment h into M, we have $M, h \models A$ if and only if $M, h \models B$.*

The links between these (page 44) also hold for predicate logic.
So (eg) the notions of valid/satisfiable formula, and equivalence, can all be expressed in terms of valid arguments.

Which arguments are valid?

Some examples of valid arguments:

- valid propositional ones: eg, $A \wedge B \models A$.
- many new ones: for example,

$$\forall x(\texttt{horse}(x) \to \texttt{animal}(x)) \models \forall y[\exists x(\texttt{head-of}(y,x) \wedge \texttt{horse}(x)) \\ \to \exists x(\texttt{head-of}(y,x) \wedge \texttt{animal}(x))].$$

A horse is an animal

$\models$ the head of a horse is the head of an animal.

Deciding if a supposed argument $A_1, \ldots, A_n \models B$ is valid is extremely hard in general.
We can't just check that all L-structures + assignments that make $A_1, \ldots, A_n$ true also make B true (like truth tables),
because there are infinitely many L-structures (some are infinite!)

Theorem 12.5 (Church, 1936) *No computer program can be written to identify precisely the valid arguments of predicate logic.*

Useful ways of validating arguments

In spite of theorem 12.5, we can often verify in practice that an argument in predicate logic is valid. Ways to do it include:

- direct reasoning (the easiest, once you get used to it)
- equivalences (also useful)
- proof systems like natural deduction

The same methods work for showing a formula is valid. (A is valid if and only if $\models A$.)

Truth tables no longer work. You can't tabulate all structures — there are infinitely many.

12.1 Direct reasoning

Let's show

$$\left.\begin{array}{r}\forall x(\texttt{human}(x) \to \texttt{lecturer}(x))\\ \forall x(\texttt{Sun}(x) \to \texttt{lecturer}(x))\\ \forall x(\texttt{human}(x) \vee \texttt{Sun}(x))\end{array}\right\} \models \forall x\, \texttt{lecturer}(x).$$

Take any M such that

1) $M \models \forall x(\texttt{human}(x) \to \texttt{lecturer}(x))$,
2) $M \models \forall x(\texttt{Sun}(x) \to \texttt{lecturer}(x))$,
3) $M \models \forall x(\texttt{human}(x) \vee \texttt{Sun}(x))$.

Show $M \models \forall x\, \texttt{lecturer}(x)$.

Take arbitrary a in $\mathrm{dom}(M)$. We require $M \models \texttt{lecturer}(a)$.

Well, by (3), $M \models \texttt{human}(a) \vee \texttt{Sun}(a)$.
If $M \models \texttt{human}(a)$, then by (1), $M \models \texttt{lecturer}(a)$.
Otherwise, $M \models \texttt{Sun}(a)$. Then by (2), $M \models \texttt{lecturer}(a)$.
So either way, $M \models \texttt{lecturer}(a)$, as required.

Harder example

Let's show

$$\forall x(\mathtt{horse}(x) \to \mathtt{animal}(x)) \models \forall y[\exists x(\mathtt{head}(y,x) \wedge \mathtt{horse}(x)) \\ \to \exists x(\mathtt{head}(y,x) \wedge \mathtt{animal}(x))].$$

Take any L-structure M (where L is as before). Assume that (1) $M \models \forall x(\mathtt{horse}(x) \to \mathtt{animal}(x))$. Show $M \models \forall y[\exists x(\mathtt{head}(y,x) \wedge \mathtt{horse}(x)) \to \exists x(\mathtt{head}(y,x) \wedge \mathtt{animal}(x))]$.

So take any object b in $\mathrm{dom}(M)$. We show the blue formula for $y = b$. So we assume that (2) $M \models \exists x(\mathtt{head}(b,x) \wedge \mathtt{horse}(x))$, and try our best to show that $M \models \exists x(\mathtt{head}(b,x) \wedge \mathtt{animal}(x))$.

By (2), there is some h in $\mathrm{dom}(M)$ with $M \models \mathtt{head}(b,h) \wedge \mathtt{horse}(h)$.
Then $M \models \mathtt{head}(b,h)$ and $M \models \mathtt{horse}(h)$.
By (1), $M \models \mathtt{horse}(h) \to \mathtt{animal}(h)$.
So $M \models \mathtt{animal}(h)$.
So $M \models \mathtt{head}(b,h) \wedge \mathtt{animal}(h)$, and this h is living proof that $M \models \exists x(\mathtt{head}(b,x) \wedge \mathtt{animal}(x))$, as required.

Direct reasoning with equality

Let's show $\forall x \forall y(x = y \wedge \exists z R(x, z) \to \exists v R(y, v))$ is valid.

Take any structure M, and objects a, b in $\mathrm{dom}(M)$. We need to show

$$M \models a = b \wedge \exists z R(a, z) \to \exists v R(b, v).$$

So we need to show that
IF $M \models a = b \wedge \exists z R(a, z)$ THEN $M \models \exists v R(b, v)$.

But IF $M \models a = b \wedge \exists z R(a, z)$, then a, b are the same object.
So $M \models \exists z R(b, z)$.

So there is an object c in $\mathrm{dom}(M)$ such that $M \models R(b, c)$.

Therefore, $M \models \exists v R(b, v)$. We're done.

12.2 Equivalences

As well as the propositional equivalences seen before, we have extra ones for predicate logic. A, B denote arbitrary predicate formulas.

28. $\forall x \forall y A$ is logically equivalent to $\forall y \forall x A$.

29. $\exists x \exists y A$ is (logically) equivalent to $\exists y \exists x A$.

30. $\neg \forall x A$ is equivalent to $\exists x \neg A$.

31. $\neg \exists x A$ is equivalent to $\forall x \neg A$.

32. $\forall x (A \wedge B)$ is equivalent to $\forall x A \wedge \forall x B$.

33. $\exists x (A \vee B)$ is equivalent to $\exists x A \vee \exists x B$.

Equivalences involving bound variables

34. If x does not occur free in A, then $\forall x A$ and $\exists x A$ are logically equivalent to A. (See slide 137 for free variables.)
E.g., $\forall x \underbrace{\exists x P(x)}_{A}$ and $\exists x \underbrace{\exists x P(x)}_{A}$ are equivalent to $\underbrace{\exists x P(x)}_{A}$.

35. If x doesn't occur free in A, then
$\exists x(A \wedge B)$ is equivalent to $A \wedge \exists x B$, and
$\forall x(A \vee B)$ is equivalent to $A \vee \forall x B$.

36. If x does not occur free in A then
$\forall x(A \rightarrow B)$ is equivalent to $A \rightarrow \forall x B$, and
$\exists x(A \rightarrow B)$ is equivalent to $A \rightarrow \exists x B$.

37. *Note:* if x does not occur free in B then
$\forall x(A \rightarrow B)$ is equivalent to $\exists x A \rightarrow B$, and
$\exists x(A \rightarrow B)$ is equivalent to $\forall x A \rightarrow B$.
The quantifier changes! Watch out!

Renaming bound variables

38. Suppose that x is any variable, y is a variable that does not occur in A, and B is got from A by

    - replacing all *bound* occurrences of x in A by y,
    - replacing all $\forall x$ in A by $\forall y$, and
    - replacing all $\exists x$ in A by $\exists y$.

    Then A is equivalent to B.

Eg $\forall x \exists y\, \texttt{bought}(x, y)$ is equivalent to $\forall z \exists v\, \texttt{bought}(z, v)$.

$\texttt{human}(x) \land \exists x\, \texttt{lecturer}(x)$ is equivalent to $\texttt{human}(x) \land \exists y\, \texttt{lecturer}(y)$.

Equivalences/validities involving equality

39. $t = t$ is valid (equivalent to $\top$), for any term t.

40. For any terms t, u,
    $t = u$ is equivalent to $u = t$

41. (Leibniz principle) If A is a formula in which x occurs free, y doesn't occur in A at all, and B is got from A by replacing one or more free occurrences of x by y, then

$$x = y \to (A \leftrightarrow B)$$

    is valid.

    Example:
    $x = y \to (\forall z R(x, z) \leftrightarrow \forall z R(y, z))$ is valid.

Examples using equivalences

These equivalences form a toolkit for transforming formulas.

Eg: let's show that if x is not free in A then $\forall x(\exists x \neg B \to \neg A)$ is equivalent to $\forall x(A \to B)$.

Well, the following formulas are equivalent:

- $\forall x(\exists x \neg B \to \neg A)$
- $\exists x \neg B \to \neg A$ — by $\forall x D \equiv D$ when x is not free in D
- $\neg \forall x B \to \neg A$ — by $\exists x \neg C \equiv \neg \forall x C$
- $A \to \forall x B$ (example on p. 60)
- $\forall x(A \to B)$ — this *is* equivalence 36 (x is not free in A)

Warning: non-equivalences

Depending on A, B, the following need *NOT* be logically equivalent (though always, the first $\models$ the second):

- $\forall x(A \to B)$ and $\forall x A \to \forall x B$
- $\exists x(A \wedge B)$ and $\exists x A \wedge \exists x B$.
- $\forall x A \vee \forall x B$ and $\forall x(A \vee B)$.

Can you find a 'countermodel' for each one? (Find suitable A, B and a structure M such that $M \models$ 2nd but $M \not\models$ 1st.)

12.3 Natural deduction for predicate logic

This is quite easy to set up. We keep the old propositional rules — e.g., $A \vee \neg A$ for any first-order sentence A ('lemma') — and add new ones for $\forall, \exists, =$.

You construct natural deduction proofs as for propositional logic: first think of a direct argument, then convert to ND.

This is *even more important than for propositional logic.* There's quite an art to it.

Validating arguments by predicate ND can sometimes be harder than for propositional ones, because the new rules give you wide choices, and at first you may make the wrong ones!
If you find this depressing, remember, it's a hard problem, there's no computer program to do it (theorem 12.5)!

$\exists$-introduction, or $\exists I$

Notation 12.6 *For a formula A, a variable x, and a term t, we write $A(t/x)$ for the formula got from A by replacing all* free *occurrences of x in A by t.*

To prove a sentence $\exists x A$, you can prove $A(t/x)$, for some closed term t of your choice.

	$\vdots$		
1	$A(t/x)$	we got this somehow...	
2	$\exists x A$		$\exists I(1)$

Recall a *closed term* (or ground term) is one with no variables.

This rule is reasonable. If in some structure, $A(t/x)$ is true, then so is $\exists x A$, because there exists an object in M (namely, the value in M of t) making A true.

But choosing the 'right' t can be hard — that's why it's such a good idea to think up a 'direct argument' first!

$\exists$-elimination, $\exists E$ (tricky!)

Let A be a formula. If you have managed to write down $\exists x A$, you can prove a sentence B from it by

- assuming $A(c/x)$, where c is a *new* constant not used in B or in the proof so far,
- proving B from this assumption.

During the proof, you can use anything already established.
But once you've proved B, you cannot use any part of the proof, *including* c, later on.
So we isolate the proof of B from $A(c/x)$, in a box:

1	$\exists x A$	got this somehow
2	$A(c/x)$	ass
	⟨the proof⟩	hard struggle
3	B	we made it!
4	B	$\exists E(1, 2, 3)$

c is often called a Skolem constant. Pandora uses sk1, sk2, ...

Justification of $\exists E$

Basically, 'we can give any object a name'.

Given any formula $A(x)$, if $\exists x A$ is true in some structure M, then there is an object a in $\mathrm{dom}(M)$ such that $M \models A(a)$.

Now a may not be named by a constant in M. But we can add a new constant to name it — say, c — and add the information to M that c names a.

c must be new — the other constants already in use may not name a in M.

And of course, if $M \models A(c/x)$ then $M \models \exists x A$.

So $A(c/x)$ *for new c* is really *no better or worse than $\exists x A$.*

Therefore, if we can prove B from the assumption $A(c/x)$, it counts as a proof of B from the already-proved $\exists x A$.

Example of $\exists$-rules

Show $\exists x(P(x) \wedge Q(x)) \vdash \exists x P(x) \wedge \exists x Q(x)$.

1	$\exists x(P(x) \wedge Q(x))$	given
2	$P(c) \wedge Q(c)$	ass
3	$P(c)$	$\wedge E(2)$
4	$\exists x P(x)$	$\exists I(3)$
5	$Q(c)$	$\wedge E(2)$
6	$\exists x Q(x)$	$\exists I(5)$
7	$\exists x P(x) \wedge \exists x Q(x)$	$\wedge I(4,6)$
8	$\exists x P(x) \wedge \exists x Q(x)$	$\exists E(1,2,7)$

In English: Assume $\exists x(P(x) \wedge Q(x))$.
Then there is a with $P(a) \wedge Q(a)$.
So $P(a)$ and $Q(a)$. So $\exists x P(x)$ and $\exists x Q(x)$.
So $\exists x P(x) \wedge \exists x Q(x)$, as required.

Note: only sentences occur in ND proofs. They should never involve formulas with free variables!

$\forall$-introduction, $\forall I$

To introduce the sentence $\forall x A$, for some $A(x)$, you introduce a *new* constant, say c, not used in the proof so far, and prove $A(c/x)$.
During the proof, you can use anything already established.
But once you've proved $A(c/x)$, you can no longer use the constant c later on.
So isolate the proof of $A(c/x)$, in a box:

1	c	$\forall I$ const
	$\langle$the proof$\rangle$	hard struggle
2	$A(c/x)$	we made it!
3	$\forall x A$	$\forall I(1,2)$

This is the *only* time in ND that you write a line (1) containing a *term,* not a formula. And it's the *only* time a box doesn't start with a line labelled 'ass'. (Pandora gives no label.)

Justification

To show $M \models \forall x A$, we must show $M \models A(a)$ for every object a in $\mathrm{dom}(M)$.

So choose an arbitrary a, add a new constant c naming a, and prove $A(c/x)$. As a is arbitrary, this shows $\forall x A$.

c must be new, because the constants already in use may not name this particular a.

$\forall$-elimination, or $\forall E$

Let $A(x)$ be a formula. If you have managed to write down $\forall x A$, you can go on to write down $A(t/x)$ for any closed term t. (It's your choice which t!)

	$\vdots$	
1	$\forall x A$	we got this somehow...
2	$A(t/x)$	$\forall E(1)$

This is easily justified: if $\forall x A$ is true in a structure, then certainly $A(t/x)$ is true, for any closed term t.

However, choosing the 'right' t can be hard — that's why it's such a good idea to think up a 'direct argument' first!

Example of ∀-rules

Let's show $P \to \forall x Q(x) \vdash \forall x(P \to Q(x))$.
Here, P is a 0-ary relation symbol — that is, a propositional atom.

1	$P \to \forall x Q(x)$	given
2	c	$\forall I$ const
3	P	ass
4	$\forall x Q(x)$	$\to E(3,1)$
5	$Q(c)$	$\forall E(4)$
6	$P \to Q(c)$	$\to I(3,5)$
7	$\forall x(P \to Q(x))$	$\forall I(2,6)$

In English: Assume $P \to \forall x Q(x)$. Then for any object a, if P then $\forall x Q(x)$, so $Q(a)$.

So for any object a, if P, then $Q(a)$.

That is, for any object a, we have $P \to Q(a)$. So $\forall x(P \to Q(x))$.

Example with all the quantifier rules

Show $\exists x \forall y G(x, y) \vdash \forall y \exists x G(x, y)$.

1	$\exists x \forall y G(x, y)$	given
2	d	$\forall I$ const
3	$\forall y G(c, y)$	ass
4	$G(c, d)$	$\forall E(3)$
5	$\exists x G(x, d)$	$\exists I(4)$
6	$\exists x G(x, d)$	$\exists E(1, 3, 5)$
7	$\forall y \exists x G(x, y)$	$\forall I(2, 6)$

English: Assume $\exists x \forall y G(x, y)$. Then there is some object c such that $\forall y G(c, y)$.

So for any object d, we have $G(c, d)$, so certainly $\exists x G(x, d)$.

Since d was arbitrary, we have $\forall y \exists x G(x, y)$.

Breaking the quantifier rules

I hope you know by now that $\forall x \exists y(x < y) \not\models \exists y \forall x(x < y)$.
E.g., in the natural numbers, $\forall x \exists y(x < y)$ is true; $\exists y \forall x(x < y)$ isn't.

So the following must be WRONG:

1	$\forall x \exists y(x < y)$	given
2	c	$\forall I$ const
3	$\exists y(c < y)$	$\forall E(1)$
4	$c < d$	ass
5	$c < d$	$\checkmark(4)$
6	$c < d$	$\exists E(3,4,5)$ $\leftarrow$ WRONG
7	$\forall x(x < d)$	$\forall I(2,6)$
8	$\exists y \forall x(x < y)$	$\exists I(7)$

The 'Skolem constant' d, introduced on line 4, must not occur in the conclusion (lines 5, 6): see slide 208. So the $\exists E$ on line 6 is illegal.

The restrictions in the rules are necessary for sound proofs!

Derived rule $\forall{\to}E$

This is like PC: it collapses two steps into one. Useful, but not essential.

Idea: often we have proved $\forall x(A(x) \to B(x))$ and $A(t/x)$, for some formulas $A(x), B(x)$ and some closed term t.

We know we can derive $B(t/x)$ from this:

1	$\forall x(A(x) \to B(x))$	(got this somehow)
2	$A(t/x)$	(this too)
3	$A(t) \to B(t)$	$\forall E(1)$
4	$B(t/x)$	${\to}E(2,3)$

So let's just do it in 1 step:

1	$\forall x(A(x) \to B(x))$	(got this somehow)
2	$A(t/x)$	(this too)
3	$B(t/x)$	$\forall{\to}E(2,1)$

Example of $\forall\!\to\! E$ in action

Show $\forall x \forall y (P(x,y) \to Q(x,y)), \quad \exists x P(x,a) \quad \vdash \quad \exists y Q(y,a)$.

1	$\forall x \forall y (P(x,y) \to Q(x,y))$	given
2	$\exists x P(x,a)$	given
3	$P(c,a)$	ass
4	$Q(c,a)$	$\forall\!\to\! E(3,1)$
5	$\exists y Q(y,a)$	$\exists I(4)$
6	$\exists y Q(y,a)$	$\exists E(2,3,5)$

We used $\forall\!\to\! E$ on 2 $\forall$s at once. This is even more useful. There is no limit to how many $\forall$s can be covered at once with $\forall\!\to\! E$!!

Rules for equality

There are two: refl and $=$sub. We also add a derived rule, $=$sym.

- Reflexivity of equality (refl).
  Whenever you feel like it, you can introduce the sentence $t = t$, for any closed L-term t and for any L you like.

	$\vdots$	bla bla bla
1	$t = t$	refl

  (Idea: any L-structure makes $t = t$ true, so this is sound.)

More rules for equality

- Substitution of equal terms ($=$sub).

  If $A(x)$ is a formula, t, u are closed terms, you've proved $A(t/x)$, and you've also proved either $t = u$ or $u = t$, you can go on to write down $A(u/x)$.

1	$A(t/x)$	got this somehow...
2	$\vdots$	yada yada yada
3	$t = u$	...and this
4	$A(u/x)$	$=$sub$(1, 3)$

  (Idea: if t, u are equal, there's no harm in replacing t by u as the value of x in A.)

Symmetry of $=$

Show $c = d \vdash d = c$. (c, d are constants.)

1	$c = d$	given
2	$d = d$	refl
3	$d = c$	$=$sub$(2, 1)$

This is often useful, so make it a derived rule ‘symmetry of $=$’:

1	$c = d$	given
2	$d = c$	$=$sym(1)

A hard-ish example

Show $\exists x \forall y (P(y) \to y = x), \quad \forall x P(f(x)) \vdash \exists x (x = f(x))$.

1	$\exists x \forall y (P(y) \to y = x)$	given
2	$\forall x P(f(x))$	given
3	$\forall y (P(y) \to y = c)$	ass
4	$P(f(c))$	$\forall E(2)$
5	$f(c) = c$	$\forall{\to}E(4,3)$
6	$c = f(c)$	$=$sym(5)
7	$\exists x (x = f(x))$	$\exists I(6)$
8	$\exists x (x = f(x))$	$\exists E(1,3,7)$

English: assume there is an object c such that all objects a satisfying P (if any) are equal to c, and for *any* object b, $f(b)$ satisfies P.

Taking 'b' to be c, $f(c)$ satisfies P, so $f(c)$ is equal to c.

So c is equal to $f(c)$.

As $c = f(c)$, we obviously get $\exists x (x = f(x))$.

Soundness and completeness

We did this for propositional logic — definition 5.13.
Natural deduction is also sound and complete for predicate logic:

Theorem 12.7 (soundness) *Let* $A_1, \ldots, A_n, B$ *be any first-order sentences. If* $A_1, \ldots, A_n \vdash B$*, then* $A_1, \ldots, A_n \models B$*.*

Slogan (for the case $n = 0$):
'Any provable first-order sentence is valid.'
'Natural deduction never makes mistakes.'

Theorem 12.8 (completeness)
Let $A_1, \ldots, A_n, B$ *be any first-order sentences. If* $A_1, \ldots, A_n \models B$*, then* $A_1, \ldots, A_n \vdash B$*.*

Slogan (for $n = 0$): 'Any first-order validity can be proved.'
'Natural deduction is powerful enough to prove all valid first-order sentences.'

So we can use natural deduction to check validity.

Final remarks

Now you've done sets, relations, and functions in other courses(?), here's what an L-structure M really is.

It consists of the following items:

- a non-empty set, $\mathrm{dom}(M)$
- for each constant $c \in L$, an element $c^M \in \mathrm{dom}(M)$
- for each n-ary function symbol $f \in L$, a function $f^M : \mathrm{dom}(M)^n \to \mathrm{dom}(M)$
- for each n-ary relation symbol $R \in L$, an n-ary relation R^M on $\mathrm{dom}(M)$ — that is, $R^M \subseteq \mathrm{dom}(M)^n$.

Recall for a set S, S^n is $\overbrace{S \times S \times \cdots \times S}^{n \text{ times}}$.

Another name for a relation (symbol) is a *predicate (symbol).*

What we did (all can be in Xmas test!)

Propositional logic

- Syntax

  Literals, clauses (see Prolog later in 1st year!)
- Semantics
- English–logic translations
- Arguments, validity
  - †truth tables
  - direct reasoning
  - equivalences, †normal forms
  - natural deduction

Classical first-order predicate logic

same again (except †), plus

- Many-sorted logic
- Specifications, pre- and post-conditions (continued in Reasoning about Programs)

Some of what we didn't do. . .

- normal forms for first-order logic
- proof of soundness or completeness for natural deduction
- theories, compactness, non-standard models, interpolation
- Gödel's theorems
- non-classical logics, eg. intuitionistic logic, linear logic, modal & temporal logic, model checking
- finite structures and computational complexity
- automated theorem proving

Do later years for some of these. Happy holidays.

Modern logic at research level

- Advanced computing uses classical, modal, temporal, and dynamic logics. Applications in AI, databases, concurrent and distributed systems, multi-agent systems, knowledge representation, automated theorem proving, specification and verification (eg with model checking), ... Theoretical computing (complexity, finite model theory) needs logic.
- In mathematics, logic is studied in *set theory, model theory,* and *recursion theory.* Each of these is an entire field, with dozens or hundreds of research workers. Other logical areas include *non-standard analysis, universal algebra, proof theory*, ...
- In philosophy, logic is studied for its contribution to formalising truth, validity, argument, in many settings: e.g., involving time or other possible worlds.
- Logic provides the foundation for several modern theories in linguistics. This is nowadays relevant to computing.