140 Logic

Ian Hodkinson and Alessandra Russo

Department of Computing, Imperial College London

Thanks to Krysia Broda, Robin Hirsch and Dirk Pattinson for additional material

Logic home page:

`http://www.doc.ic.ac.uk/~imh/teaching/140_logic/logic.html`

Slides and exercises: `https://cate.doc.ic.ac.uk/`

Pandora: `http://www.doc.ic.ac.uk/pandora`

`http://www.doc.ic.ac.uk/pandora/newpandora/`

LOST06, LOST07: `programs/lab/Lost` and `programs/lab/LostCD`

Course layout

In about 18 lectures, we will cover:

- Propositional logic
- Predicate logic (also known as 'first-order' logic, or sometimes just 'classical' logic)

There will be weekly tutorials, and you are advised to try the Pandora and (if possible) LOST programs in the labs later on.

There will be marked (but unassessed) pmt exercises some weeks, and unmarked exercises in the rest. You will get solutions later.

There's a logic question in the Xmas test near the end of term, and two logic questions (40 minutes each) in the exams in May.

The material is needed for 141 'Reasoning about programs' next term, and several later courses.
You may need it to answer exam questions in these courses.

Books

- *Reasoned Programming,* K. Broda, S. Eisenbach, H. Khoshnevisan, S. Vickers, Prentice-Hall, 1994. Best buy. Out of print I think. See `http://www.doc.ic.ac.uk/pandora`
- *Logic,* Wilfrid Hodges, Penguin, 1977. Excellent introduction, highly recommended for beginners, but doesn't cover the more technical parts of the course.
- *Software Engineering Mathematics,* J. Woodcock, M. Loomes, Pitman, 1988. Early chapters quite close to course. Natural deduction done Gentzen-style, not with boxes, but you should be able to work it out.
- *Beginning Logic,* E. Lemmon, Van Nostrand Reinhold, 1982. Classic. More for mathematicians, natural deduction in different style from the course. But you might make a go of it with effort.

- *How to prove it,* D. Velleman, Cambridge University Press, 1995. Examples mostly from maths. Early chapters OK but no natural deduction.
- Just go to the library and find one you like.

More advanced:

- *Logic in computer science: modelling and reasoning about systems,* M. Huth and M. Ryan, Cambridge University Press, 2000.
- *Formal Logic: its scopes and limits,*
  R. Jeffrey, McGraw Hill, 1981.
- *Logic: techniques of formal reasoning,* Kalish, Montague, Mar. Harcourt Brace Janovich, 1980 (2nd edn).
- *Methods of Logic,* W. Quine, Harvard UP, 1982.

1. Introduction: what is logic?

Shorter Oxford Dictionary:

> ‘Logic, from Greek *logos* (reasoning, discourse)
> The branch of philosophy that deals with forms of reasoning and thinking, especially inference and scientific method.
> Also, the systematic use of symbolic techniques and mathematical methods to determine the forms of valid deductive argument.’

We are not concerned here with low-level ‘logic gates’ etc in computing — that is for the Hardware course.

We are concerned with using logic to describe, specify, verify, and reason about software.

At higher levels, logic can shed light on ‘forms of thinking’ — relevant to AI.

Why should you learn logic?

- Logic is the ‘calculus of computing’: a good mathematical foundation for dealing with information, and reasoning about program behaviour.
- It also provides a good training in correct reasoning and accurate, unambiguous description.
- It is needed in later courses/areas: Reasoning about programs, Prolog, Databases(?), SE – Design, SE – Systems Verification, Software Reliability, many AI courses, . . .
- Some of the more sophisticated real-world application areas need and use logic. Example: Prolog is a logic programming language that you can buy commercially. The SQL database language, and model-checking, are based on logical ideas.
- Propositional and first-order logic are the most fundamental of all logical systems. If you ever do any logic, you really have to do these first.

So what makes up logic?

A logical system usually consists of three things:

1. Syntax — a *formal* language (like a programming language) that will be used to express concepts.

2. Semantics — provides *meaning* for the formal language.

3. Proof theory — a purely syntactic way of obtaining or identifying the valid statements of the language. It provides a way of *arguing* in the formal language.

Far, far more can be done with logic than this. Logic has now attained the depth and sophistication of modern mathematics: eg Fermat's last theorem.

But this is a first course in logic, and we stick near the ground!

2. Propositional logic

This is the study of 'if-tests':

```
if count>0 and not found then
  decrement count; look for next entry;
end if
```

Features:

- basic (or 'atomic') statements — here: `count>0`, `found` — are true or false depending on circumstances
- can use boolean operations — `and`, `or`, `not`, etc. — to build more and more complex test statements from the atomic propositions
- the final complex statement evaluates to true or false.

Propositional logic is not very expressive. Predicate logic (later) can say much more. Eg. 'every student has a tutor'.

Why do propositional logic?

Anyone who's written imperative programs knows about if-tests. Why study them in university?

- All logics are based on propositional logic in some form. We have to start with it.

- We want to handle arbitrarily complicated if-tests and study their general features.

- We don't just want to *evaluate* if-tests.

  We want to know how to find out when two if-tests mean the same, when one implies another, whether an if-test (or loop test) can ever be made true or not.

- Propositional logic encapsulates an important class of computational problems, and so comes in to algorithms and complexity theory.

2.1 Syntax — the formal language

First, we need to fix a precise definition of the kinds of if-test we will consider. This will constitute the formal language of propositional logic. It has three ingredients.

1st ingredient: Propositional atoms

We are not concerned about which atomic facts (`count>0, found`) are involved. So long as they can get a *truth value* (true or false), that's enough.

So we fix a collection of algebraic symbols to stand for these atomic statements.

The symbols are called *propositional atoms.* Many better-educated people call them *propositional variables* or *propositional letters* instead.
For short, I'll usually call them *atoms.*

They are like variables x, y, z in maths.

But because they are Propositional, we usually use the letters $p, p', p_0, p_1, p_2, \ldots$, and also $q, r, s, \ldots$

2nd ingredient: Boolean connectives

We are interested in the following boolean operations, or *connectives*:

- and: written as $\wedge$ (or sometimes &, and in old books, '$\cdot$')
- not: written as $\neg$ (or sometimes $\sim$)
- or: written as $\vee$ (in old books, $+$)
- 'if-then', or 'implies'. This is written as $\to$ (or sometimes $\supset$, but not as $\Rightarrow$, which is used differently).
- if-and-only-if: written $\leftrightarrow$ (but not as $\iff$ or $\equiv$, which are used differently)
- truth and falsity: written as $\top, \bot$

Warning: 'implies' is just the way to *say* $\to$. We'll discuss the *meaning* of $\to$ later. The same applies to the other connectives.

Examples

- Our test `count>0 and not found` would come out as $(\texttt{count} > 0) \wedge \neg\,\texttt{found}$, or maybe $p \wedge \neg q$.
- Symbols for and, or, implies, if-and-only-if take 2 arguments and are written in infix form: $p \wedge q,\ \ p \vee q,\ \ p \to q,\ \ p \leftrightarrow q$.
- Negation $\neg$ takes 1 argument, which is written to the right of it: eg, $\neg p,\ \ \neg q$.
- $\top, \bot$ take no arguments.
  They are logical *constants* (like π, e).

You'll have to learn these new symbols, so you can read logic books. They are also quicker to write than `and`, `or`, `not`, etc.

3rd ingredient: Punctuation

We need brackets to disambiguate our if-tests.

This is like terms in arithmetic. $1 - 2 + 3$ is ambiguous. We can read it as $(1 - 2) + 3$ or $1 - (2 + 3)$, and the difference matters.

For example, $p \wedge q \vee r$ might be read as

- $(p \wedge q) \vee r$,
- $p \wedge (q \vee r)$,

and the difference matters.

So we start by putting all brackets in, and then deleting those that aren't needed.

Formulas

What I've called if-tests are called *formulas* by logicians. (Some call them *well-formed formulas,* or *wffs,* but I think this is daft.)

A propositional formula is a string of symbols made from propositional atoms using the boolean connectives above, and brackets, in the appropriate way. More accurately:

Definition 2.1 (propositional formula)

1. *Any propositional atom* (p, q, r*, etc) is a (propositional) formula.*
2. $\top$ *and* $\bot$ *are formulas.*
3. *If* A *is a formula then so is* $(\neg A)$*.* Note the brackets!
4. *If* A, B *are formulas then so are* $(A \wedge B)$*,* $(A \vee B)$*,* $(A \to B)$*, and* $(A \leftrightarrow B)$*.*
5. *That's it: nothing is a formula unless built by these rules.*

Examples of formulas

The following are formulas:

- p
- $(\neg p)$
- $((\neg p) \wedge \top)$
- $(\neg((\neg p) \wedge \top))$
- $((\neg p) \to (\neg((\neg p) \wedge \top)))$

You can see the brackets are a pain already. We need some conventions to get rid of (some of) them.
What we'll get are (strictly speaking) *abbreviations* of genuine formulas. But most people seem to think of them as real formulas.

We can always omit the final, outermost brackets:
$\neg p$, and $(\neg p) \to (\neg((\neg p) \wedge \top))$ are unambiguous.

Binding conventions

To get rid of more brackets, we *order* the boolean connectives according to decreasing binding strength:

(strongest) $\neg,\ \wedge,\ \vee,\ \to,\ \leftrightarrow$ (weakest)

This is like in arithmetic, where $\times$ is stronger than $+$. This means that $2 + 3 \times 4$ is usually read as $2 + (3 \times 4)$, not as $(2 + 3) \times 4$. So:

- $p \vee q \wedge r$ is read as $p \vee (q \wedge r)$, not as $(p \vee q) \wedge r$.
- $\neg p \wedge q$ is read as $(\neg p) \wedge q$, not as $\neg(p \wedge q)$.
- $p \wedge \neg q \to r$ is read as $(p \wedge (\neg q)) \to r$, rather than $p \wedge (\neg(q \to r))$ or $p \wedge ((\neg q) \to r)$.

But don't take the conventions too far. Eg, $p \to \neg q \vee \neg\neg r \wedge s \leftrightarrow t$ is a mess! USE BRACKETS if it helps readability, even if they are not strictly needed.

Repeated connectives

What about $p \to q \to r$? The binding conventions are no help now. Should we read it as $p \to (q \to r)$ or as $(p \to q) \to r$?

Haskell would suggest $p \to (q \to r)$, common sense might suggest $(p \to q) \to r$. (See what you think later on.)

There probably is a convention here. But after many years doing logic, I still find it too hard to remember to be worth the space saved. I (and most logicians) would always put brackets in such a formula.

However, $p \wedge q \wedge r$ and $p \vee q \vee r$ are usually OK as they are.

As we'll see, their logical meaning is the same however we bracket them. $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$ are different formulas. But we will see that they always have the same truth value in any situation: they are 'logically equivalent'.
So we don't really care how we disambiguate $p \wedge q \wedge r$. (Usually, it is $(p \wedge q) \wedge r$ — from left.)

Special cases

Sometimes the binding conventions don't work as we would like.

For example, if I saw

$$p \to r \wedge q \to r,$$

I'd probably read it as $(p \to r) \wedge (q \to r)$.

How can I justify this? $\wedge$ is stronger than $\to$. So shouldn't it be $p \to (r \wedge q) \to r$?
Read according to a plausible convention about repeated $\to$, this is $(p \to (r \wedge q)) \to r$.

But I say few sane people would want to write $(p \to (r \wedge q)) \to r$.
So $p \to r \wedge q \to r$ seems more likely to mean $(p \to r) \wedge (q \to r)$.

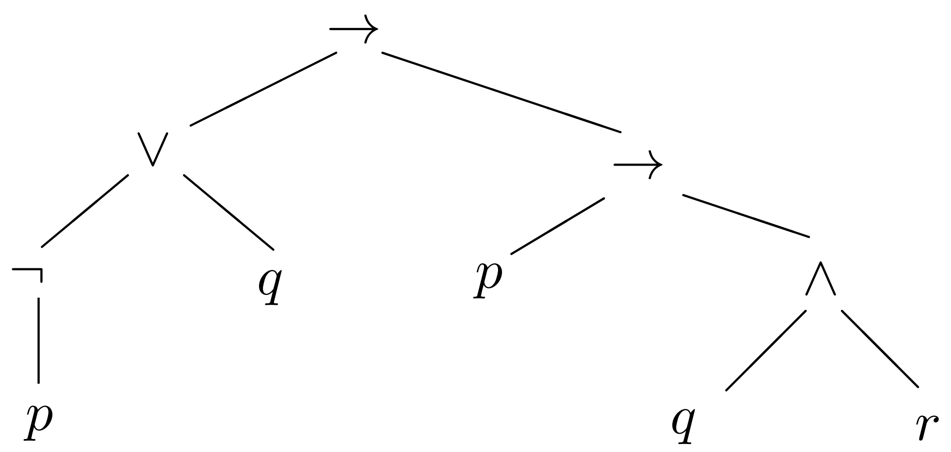
The lesson: formulas like $p \to r \wedge q \to r$ can be misinterpreted without brackets, even if they are not strictly needed. Don't write such formulas without brackets.
USE BRACKETS if it helps readability, even if they are not strictly needed.

Parsing: formation tree, logical form

We have shown how to read a formula unambiguously — to *parse* it.

The information we gain from this can be represented as a tree: the *formation tree* of the formula.

For example, $\neg p \vee q \to (p \to q \wedge r)$ has the formation tree



This is a nicer (but too expensive) way to write formulas.

Principal connective

Note that the connective at the root (top!) of the tree is $\to$. This is the *top connective* or *principal connective* of $\neg p \vee q \to (p \to q \wedge r)$. This formula has the overall *logical form* $A \to B$.

Every non-atomic formula has a principal connective, which determines its overall logical form. You have to learn to recognise it.

- $p \wedge q \to r$ has principal connective $\to$.
  Its overall logical form is $A \to B$.
- $\neg(p \to \neg q)$ has principal connective $\neg$. Its logical form is $\neg A$.
- $p \wedge q \wedge r$ has principal connective $\wedge$ (probably the 2nd one!). Its logical form is $A \wedge B$.
- $p \vee q \wedge r$ has principal connective $\vee$. Its logical form is $A \vee B$.

What are the formation trees of these formulas?

Subformulas

The tree view makes it easy to explain what a subformula is.

The *subformulas* of a formula A are the formulas built in the stages on the way to building A as in definition 2.1.

They correspond to the nodes, or to the subtrees, of the formation tree of A.

The subformulas of $\neg p \vee q \to (p \to q \wedge r)$ are:

$$\begin{array}{cccccc}
 & & \neg p \vee q \to (p \to q \wedge r) & & & \\
 & \neg p \vee q & & & p \to q \wedge r & \\
\neg p & q & p & & q \wedge r & \\
p & & & q & & r
\end{array}$$

Notes: there are *two different* subformulas p.
And $p \vee q$ and $p \to q$ are substrings but NOT subformulas.

Abbreviations

You may see some funny-looking formulas in books:

- $\bigwedge_{1\leq i\leq n} A_i$, $\bigwedge_{i=1}^{n} A_i$, $\displaystyle\bigwedge_{i=1}^{n} A_i$, $\displaystyle\bigwedge_{1\leq i\leq n} A_i$, $\displaystyle\mathop{\bigwedge\!\!\!\bigwedge}_{1\leq i\leq n} A_i$.

  These all abbreviate $A_1 \wedge A_2 \wedge \ldots \wedge A_n$.

  Example: $\displaystyle\bigwedge_{1\leq i\leq n} p_i \to q$ abbreviates $p_1 \wedge \ldots \wedge p_n \to q$.

- $\bigvee_{1\leq i\leq n} A_i$, $\bigvee_{i=1}^{n} A_i$, $\displaystyle\bigvee_{i=1}^{n} A_i$, $\displaystyle\bigvee_{1\leq i\leq n} A_i$, $\displaystyle\mathop{\bigvee\!\!\!\bigvee}_{1\leq i\leq n} A_i$.

  These abbreviate $A_1 \vee A_2 \vee \ldots \vee A_n$.

This is like in algebra:

$$\sum_{i=1}^{n} a_i \text{ abbreviates } a_1 + a_2 + \cdots + a_n.$$

Technical terms for logical forms

To understand logic books, you'll need some jargon. Learning it is tedious but necessary.

Definition 2.2

- *A formula of the form $\top$, $\bot$, or p for an atom p, is (as we know) called* atomic.
- *A formula whose logical form is $\neg A$ is called a* negated formula. *A formula of the form $\neg p$, $\neg\top$, or $\neg\bot$ is sometimes called* negated-atomic.
- *A formula of the form $A \wedge B$ is called a* conjunction, *and A, B are its* conjuncts.
- *A formula of the form $A \vee B$ is called a* disjunction, *and A, B are its* disjuncts.
- *A formula of the form $A \to B$ is called an* implication. *A is called the* antecedent, *B is called the* consequent.

Technical terms ctd. — literals and clauses

Definition 2.3

- *A formula that is either atomic or negated-atomic is called a* literal. *(In maths it's sometimes called a* basic formula.*)*
- *A* clause *is a disjunction* $(\vee)$ *of one or more literals.*

Eg, the formulas p, $\neg r$, $\neg\bot$, $\top$ are all literals.

Four examples of clauses

$$p$$

$$\neg p$$

$$p \vee \neg q \vee r$$

$$p \vee p \vee \neg p \vee \neg\bot \vee \top \vee \neg q$$

Some people allow the empty clause (the disjunction of zero literals!), which by convention is treated the same as $\bot$.

2.2 Semantics — meaning

We know how to read and write formulas. But what do they mean? Posh way: ‘what is their semantics?’

The Boolean connectives (and $\wedge$, not $\neg$, or $\vee$, if-then $\rightarrow$, if and only if $\leftrightarrow$) have *roughly* their English meanings.

But English is a natural language, full of ambiguities, subtleties, and special cases. Translating between English and logic is fraught with difficulties. See (eg) Hodges’ book for many nasty examples.

We are engineers. We need a precise idea of the meaning of formulas — especially as there are infinitely many of them. We may want to implement it. In Haskell.

So we prefer to give a formal mathematical-style definition of meaning.

Giving meaning to formulas

Definition 2.4 (situation) *A situation is simply something that determines whether each propositional atom is true or false.*

For if-tests, a situation is some point in a program execution. The current values of the program variables will determine whether or not each atomic statement of an if-test (such as $x > 0$, $x = y$, etc) is true.

For *'if (it rains) then (it is cloudy)',* a situation is the weather.

For atoms $p, q, \ldots$, a situation just specifies which of them are true and which are false.

Never forget: there is more than one situation. In a different situation, the truth values may be different. Obviously.

Knowing the situation, we can work out the *truth value* of any given propositional formula — whether it is true or false *in this situation.* We work from simple formulas to more complex ones, as follows.

Working out truth values

Definition 2.5 (evaluation) *The truth value of a propositional formula in a given situation is defined as follows.*

- *The situation tells us directly the truth values of atoms.*
- $\top$ *is true, and* $\bot$ *is false.*

Assume A, B *are formulas, and we've already worked out whether they are true or false in the given situation. Then in this situation:*

- $\neg A$ *is true if* A *is false, and false if* A *is true.*
- $A \wedge B$ *is true if* A *and* B *are both true; otherwise it is false.*
- $A \vee B$ *is true if one or both of* A *and* B *are true, false otherwise. (Inclusive or)*
- $A \to B$ *is true if* A *is false or* B *is true (or both). Otherwise — if* A *is true and* B *is false — it is false.* 'If A is true, so is B.'
- $A \leftrightarrow B$ *is true if* A *and* B *have the same truth value (both true, or both false). If they don't, it's false.*

Truth tables for connectives

We can express these rules of meaning using *truth tables* for the connectives.

We write 1 for true and 0 for false. (Some write *tt*, *ff* instead.)
Do not confuse them with $\top$, $\bot$, which are formulas, not truth values.

A	$\neg A$	$\top$	$\bot$
1	0	1	0
0	1	1	0

A	B	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

Aside: general connectives

Actually, *any* truth table gives us a (possibly new) connective.

E.g., the table

A	B	$A \uparrow B$
1	1	0
1	0	1
0	1	1
0	0	1

defines what is called the NAND connective, also known as the Sheffer stroke, and often written $\uparrow$.

Any connective can be expressed with $\wedge, \neg$ (and even with just $\uparrow$). E.g., $A \uparrow B$ can be expressed as $\neg(A \wedge B)$.

See exercises and exam questions for more examples and information.

Examples

Suppose that p is true and q is false in a certain situation. Then in this situation:

- $\neg p$ is false
- $\neg q$ is true
- $p \wedge q$ is false
- $p \vee q$ is true
- $p \rightarrow q$ is false
- $q \rightarrow p$ is true
- $p \leftrightarrow q$ is false, and $\neg p \leftrightarrow q$ is true.

A different situation

Suppose now that p is false, and q is true. In this new situation:

- $\neg p$ is now true
- $\neg q$ is now false
- $p \wedge q$ is still false
- $p \vee q$ is still true
- $p \to q$ is now true
- $q \to p$ is false now
- $p \leftrightarrow q$ is still false, and $\neg p \leftrightarrow q$ is still true.

So do not ask whether your formula is true.
(It's like asking 'is $3x = 7$ true?')
Ask if it is true in such-and-such a situation.

Now it's your turn

Suppose again that p is true and q is false in a certain situation. Are the following true or false?

1. $p \rightarrow \neg q$
2. $q \rightarrow q$
3. $\neg p \vee \neg q$
4. $p \wedge q \rightarrow q$
5. $\neg(p \leftrightarrow \top)$
6. $\neg(q \rightarrow \neg(p \vee q))$

2.3 English correspondence

As I said, translating from English to logic is difficult. But we have to do it for applications.

We can only translate (some) *statements;* not questions, commands/invitations, exclamations.

Basic idea: extract suitable atomic propositions from the English sentence, and join them up appropriately with boolean connectives. But you will need massive amounts of common sense.

Eg: ‘I bought beans and peas’ could be translated as
‘I bought beans $\wedge$ I bought peas’.

‘I am not good at logic’ could be translated as ‘$\neg$(I am good at logic)’.

‘If it’s raining then it’s not sunny’ could be translated as
‘raining $\to$ $\neg$sunny’.

‘The answer is 3 or 6’ could be translated as
‘The answer is 3 $\vee$ the answer is 6’.

English variants

‘But’ means ‘and’.
‘I will go out, but it is raining’ translates to ‘I will go out $\wedge$ it is raining’.

‘Unless’ generally means ‘or’.
‘I will go out unless it rains’ translates to ‘I will go out $\vee$ it will rain’.
(Note the extra ‘will’.)
You could also use ‘$\neg$(it will rain) $\to$ I will go out’.

But you may think that ‘I will go out unless it rains’ implies that if it does rain then I won’t go out. This is a stronger reading of ‘unless’. If you take that view, you’d translate it as ‘I will go out $\leftrightarrow$ $\neg$(it will rain)’.
(This is ‘exclusive or’.)

Deciding whether to take the strong or weak reading is done by individual context, can be very subtle, and is sometimes impossible — English is not always precise.
But in computing, people always prefer the weak form.

Or

In everyday English, ‘or’ sometimes means ‘exclusive or’ (the ‘strong reading’).
You’d probably say ‘I have keys or coins in my pocket’ is false if I have both. So this ‘or’ is strong.

But sometimes it means ‘inclusive or’ (the ‘weak reading’).
‘If I have keys or coins in my pocket, I’d better be careful at night’ implies I’d still better be careful if I have both. So this ‘or’ is weak!

(The way English handles nested connectives is very subtle!)

But in computing we always take the weak reading: inclusive or.

We use English as a technical language. As an initiate/neophyte, you have to learn to accept this and use the weak forms.

But when reading English written by non-computing people, use your common sense to determine what they meant (or ask them!)

Modalities (nasty)

A modality shifts the context of utterance.

Eg time: 'I will be rich and famous' can't be accurately translated as

'I will be rich $\wedge$ I will be famous'.

The formula is true if I get famous only when I lose all my money.

The English probably isn't: it suggests I'll be rich and famous at the same time.

Eg permission: 'You can have chicken or fish' usually means

'You can have chicken $\wedge$ you can have fish'.

Eg obligation: 'I must do topics or a foreign language' (was true) can't be translated as 'I must do topics $\vee$ I must do a language' (was false — you could choose which to take).

Modalities are beyond the scope of this course.
See 4th year course 499H modal logic.

From logic to English

You might think this is straightforward:

- ‘$p \wedge q$’ translates to ‘p and q’
- ‘$\neg p$’ to ‘not p’
- $p \to q$ to ‘if p then q’ or perhaps ‘q if p’,

etc. Eg, ‘It is raining $\to \neg$ I go out’ translates to ‘If it is raining then I do not go out’.

But there are problems.

Complex formulas are hard to render naturally in English.

‘$\to$’ is a nightmare to translate. The semantics of $\to$ works superbly for logic. But in English we use if-then in many different ways, and not always carefully. Don’t read $\to$ as ‘causes’ — too strong.

Eg ‘I am the pope $\to$ I am an atheist’ is true here and now (why?)

But would you say that the English sentence ‘If I am the pope, then I am an atheist’ is true?

A really vicious example

This one deserves careful thought. Consider the formulas

(1) $p \wedge q \to r$

(2) $(p \to r) \vee (q \to r)$.

Let p be ‘I get into the lift’, q be ‘you get into the lift’, and r be ‘the lift breaks, crashes to the ground, and burns’.

Then (1) says that *if we both get in the lift, then the lift breaks.*

(2) says that *if I get in then the lift breaks, or if you get in then the lift breaks.* Right?

Would you say these mean the same? I guess not.

But they do! The formulas (1) and (2) are true in exactly the same situations. They are ‘logically equivalent’! We’ll see this later.

I think the trouble is that in the English version of (2), you read the ‘or’ as ‘and’. You shouldn’t!

3. Arguments, validity

You now know how to read, write, and evaluate propositional formulas. You know a little about translating between English and logic.

But logic is not just about saying things. It allows arguing too. Some say it is the study of valid arguments.

Here is an old argument:

- Socrates is a man.
- Men are mortal.
- Therefore, Socrates is mortal.

Is this a valid argument? What does it mean to say that it is or is not a valid argument?

3.1 Valid arguments

Our answer is, it is a valid argument if for each situation in which Socrates is a man and also men are mortal, Socrates is mortal in this situation.

In our simple propositional logic, a situation simply defines the truth value of each propositional atom (see definition 2.4).

So we propose:

Definition 3.1 (valid argument) *Given formulas $A_1, A_2, \ldots, A_n, B$, an argument*

$$\text{‘}A_1, \ldots, A_n, \textit{ therefore } B\text{’}$$

is valid if: B is true in every situation in which $A_1, \ldots, A_n$ are all true.

If this is so, we write ‘$A_1, \ldots, A_n \models B$’.

The ‘$\models$’ is called ‘double turnstile’. You can read it as ‘logically entails/implies’, or ‘semantically entails’.

Examples of arguments

Let A, B be arbitrary propositional formulas.

- 'A, therefore A' is valid, since in any situation in which A is true, A is true. $A \models A$.
- '$A \wedge B$, therefore A' is valid. $A \wedge B \models A$.
- 'A, therefore $A \wedge B$' is not in general valid: depending on A and B, there might be situations in which A is true but $A \wedge B$ is not. Then, $A \not\models A \wedge B$.
- '$A,\ A \to B$, therefore B' is valid. $A,\ A \to B \models B$. This argument style is so common and old, it has a name: *modus ponens.*
- $A \to B,\ \neg B$, therefore $\neg A$ is also valid: $A \to B, \neg B \models \neg A$. This argument is called *modus tollens.*
- '$A \to B$, B, therefore A' is not valid in general, in spite of what lawyers and politicians may say. $A \to B, B \not\models A$.

3.2 Valid, satisfiable, equivalent formulas

Three important ideas are related to valid arguments.

Definition 3.2 (valid formula) *A propositional formula is* (logically) valid *if it is true in every situation.*

We write '$\models A$' if A is valid.

Valid *propositional* formulas are often called *tautologies.* A tautology is just another name for a valid propositional formula.

Definition 3.3 (satisfiable formula) *A propositional formula is* satisfiable *if it is true in at least one situation.*

Definition 3.4 (equivalent formulas) *Two propositional formulas A, B are* logically equivalent *if they are true in exactly the same situations. Roughly speaking: they mean the same.*

Some people write $A \equiv B$ for this. But $\equiv$ is also used in other ways, so watch out.

Examples

Formula	valid	satisfiable
$\top$	yes	yes
$\bot$	no	no
p	no	yes
$p \land \neg p$	no	no
$p \to p$	yes	yes

Equivalent?	p	$\top$	$p \to q$
$p \land p$	yes	no	no
$p \lor \neg p$	no	yes	no
$\neg p \lor q$	no	no	yes

Some useful validities and equivalences are given later as exercises for you to check.

3.3 Links between the four concepts

Valid arguments and valid, satisfiable, and equivalent formulas are all definable from each other:

argument	validity	satisfiability	equivalence
$A \models B$	$A \to B$ valid	$A \wedge \neg B$ unsatisfiable	$(A \to B) \equiv \top$
$\top \models A$	A valid	$\neg A$ unsatisfiable	$A \equiv \top$
$A \not\models \bot$	$\neg A$ not valid	A satisfiable	$?$
$A \models B$ and $B \models A$	$?$	$A \leftrightarrow \neg B$ unsatisfiable	$A \equiv B$

All four statements in each line amount to the same thing.

4. Checking validity

So we need only deal with (e.g.) valid formulas — we get the other three notions for free.

How do we tell whether a formula is valid?

In principle, we know how: check that the formula is true in every situation.
Finding good ways of doing it in practice will occupy us off and on till Xmas.

For propositional formulas, it can be done, though computationally it's a hard problem.

For predicate logic, coming later, it's much harder, because the situations are more complex and numerous (infinitely many). But we can often show that particular predicate formulas are valid by tricks, inspiration, or good luck.

Ways to check propositional validity

There are several ways to find out if a given propositional formula is valid or not:

- Truth tables — the obvious way. You just go through all possible relevant situations and see if the formula is true in each.
- Direct 'mathematical' argument. This is the fastest of all, once you get used to it. I myself often use this method.
- Equivalences. You make a stock of useful pairs of equivalent formulas. You use them to reduce your formula step by step to $\top$, which is valid. This takes some art but is quite useful. I use it quite often (especially combined with direct argument).
- Various proof systems, including tableaux, Hilbert systems, natural deduction. We will look at natural deduction: it is similar to direct argument. It deserves a section (§5) to itself.

4.1 Truth tables

Let's show that $(p \to q) \leftrightarrow (\neg p \vee q)$ is valid.
We write 1 for true and 0 for false.

We have two atoms, each taking one of two truth values. The truth values of other atoms are irrelevant.
So there are four possible relevant situations. We evaluate all subformulas of the formula in each one:

p	q	$p \to q$	$\neg p$	$\neg p \vee q$	$(p \to q) \leftrightarrow (\neg p \vee q)$
1	1	1	0	1	1
1	0	0	0	0	1
0	1	1	1	1	1
0	0	1	1	1	1

We see that $(p \to q) \leftrightarrow (\neg p \vee q)$ is true (it has value 1) in all four situations. So it's valid.

Equivalence and satisfiability

The same table shows that $p \to q$ and $\neg p \vee q$ are equivalent: in each of the four situations, they have the *same truth value.*

What do we look for to show satisfiability?

Pros and cons of truth tables

They are boring and tedious, dumb, and error-prone (‘write-only’).

But they always work — at least for propositional logic, where only finitely many situations are relevant to a given formula. (This is not true for predicate logic: we have to find other ways there.)

They can also be used for showing satisfiability and equivalence.

They are easy to implement (not recommended).

They illustrate how hard deciding validity, satisfiability, etc can be — even for ‘trivial’ propositional logic. A formula with n atoms needs 2^n lines in its truth table. Adding an atom doubles the length of the table.

No satisfactory way to determine propositional satisfiability is known. You may see in later years that this problem is NP-complete. Breaking RSA public-key encryption is no harder, and is believed to be easier!

4.2 Direct argument

Let’s show that $p \to p \vee q$ is valid.

Take an arbitrary situation. We show $p \to p \vee q$ is true in this situation.

So we have to show that IF p is true in this situation THEN so is $p \vee q$.

Well, if p is true, then one of p, q is true, so $p \vee q$ is true. And there we are.

Exercise 4.1 Show that $A \wedge (A \to B) \to B$ is valid.

Direct argument: another example

Let's show $(A \wedge B) \wedge C$ is logically equivalent to $A \wedge (B \wedge C)$. (Equally (page 44), we could check that $(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$ is valid.)

Take any situation.

$(A \wedge B) \wedge C$ is true in this situation if and only if $A \wedge B$ and C are both true (by definition of semantics of $\wedge$).

This is so if and only if A and B are true, and also C is true — that is, they're all true.

This is so if and only if A is true, and also B and C are true.

This is so if and only if A and $B \wedge C$ are true.

This is so if and only if $A \wedge (B \wedge C)$ is true.

So $(A \wedge B) \wedge C$ and $A \wedge (B \wedge C)$ have the same truth value in this situation. The situation was arbitrary, so they are logically equivalent.

Direct argument: another example

Here's a more complex example. Let's try to show that $((p \to q) \to p) \to p$ (known as 'Peirce's law') is valid.

Take an arbitrary situation. If p is true in this situation, then $((p \to q) \to p) \to p$ is true, since any formula $A \to B$ is true when B is true. We are done.

If not, then p must be false in this situation.

So $p \to q$ is true, because $A \to B$ is true when A is false.

So $(p \to q) \to p$ is false, because the LHS is true and the RHS false: $A \to B$ is false when A is true and B false.

So $((p \to q) \to p) \to p$ is true, because $A \to B$ is true when A is false. We are done again, and finished.

This was an *argument by cases:* p true, or p false. They are exhaustive: this is known as the *'law of excluded middle'*.

4.3 Equivalences

These form a toolbox for simplifying a formula or converting a formula into another, always preserving logical equivalence.

So (eg) if we can simplify a formula to $\top$, we know it's valid.

(We do this kind of thing in arithmetic:

$$\begin{aligned} 3(x+y^2)-2y(y+4) &= 3x+3y^2-2y^2-8y \\ &= 3x+y^2-8y. \end{aligned}$$

The value is preserved in each step.)

We are going to list some examples of equivalences that have been found useful for this.

There are quite a lot, but they all embody basic logical principles that you should know. You should check that the formulas really are logically equivalent, using truth tables or direct argument.

Equivalences involving $\wedge$

In the equivalences, A, B, C will denote arbitrary formulas. For short, I will often say ‘equivalent’ rather than ‘logically equivalent’.

1. $A \wedge B$ is logically equivalent to $B \wedge A$
   (commutativity of $\wedge$)

2. $A \wedge A$ is logically equivalent to A
   (idempotence of $\wedge$)

3. $A \wedge \top$ and $\top \wedge A$ are logically equivalent to A

4. $\bot \wedge A$, $A \wedge \bot$, $A \wedge \neg A$, and $\neg A \wedge A$ are all equivalent to $\bot$

5. $(A \wedge B) \wedge C$ is equivalent to $A \wedge (B \wedge C)$ (associativity of $\wedge$)

Equivalences involving $\vee$

6. $A \vee B$ is equivalent to $B \vee A$
   (commutativity of $\vee$)

7. $A \vee A$ is equivalent to A
   (idempotence of $\vee$)

8. $\top \vee A$, $A \vee \top$, $A \vee \neg A$, and $\neg A \vee A$ are equivalent to $\top$

9. $A \vee \bot$ and $\bot \vee A$ are equivalent to A

10. $(A \vee B) \vee C$ is equivalent to $A \vee (B \vee C)$ (associativity of $\vee$)

Equivalences involving $\neg$

11. $\neg\top$ is equivalent to $\bot$

12. $\neg\bot$ is equivalent to $\top$

13. $\neg\neg A$ is equivalent to A

Equivalences involving $\to$

14. $A \to A$ is equivalent to $\top$

15. $\top \to A$ is equivalent to A

16. $A \to \top$ is equivalent to $\top$

17. $\bot \to A$ is equivalent to $\top$

18. $A \to \bot$ is equivalent to $\neg A$

19. $A \to B$ is equivalent to $\neg A \vee B$, and also to $\neg(A \wedge \neg B)$

20. $\neg(A \to B)$ is equivalent to $A \wedge \neg B$.

Equivalences involving $\leftrightarrow$

21. $A \leftrightarrow B$ is equivalent to
    - $(A \rightarrow B) \wedge (B \rightarrow A)$,
    - $(A \wedge B) \vee (\neg A \wedge \neg B)$,
    - $\neg A \leftrightarrow \neg B$.

22. $\neg(A \leftrightarrow B)$ is equivalent to
    - $A \leftrightarrow \neg B$,
    - $\neg A \leftrightarrow B$,
    - $(A \wedge \neg B) \vee (\neg A \wedge B)$.

    (This is one way to express the *exclusive or* of A, B.)

De Morgan laws

Augustus de Morgan (19th-century logician) did not discover these (they are much older), and he could not even express them in his own notation!

23. $\neg(A \wedge B)$ is equivalent to $\neg A \vee \neg B$

24. $\neg(A \vee B)$ is equivalent to $\neg A \wedge \neg B$

Distributivity of $\wedge, \vee$

25. $A \wedge (B \vee C)$ is equivalent to $(A \wedge B) \vee (A \wedge C)$.
    $(B \vee C) \wedge A$ is equivalent to $(B \wedge A) \vee (C \wedge A)$.

26. $A \vee (B \wedge C)$ is equivalent to $(A \vee B) \wedge (A \vee C)$
    $(B \wedge C) \vee A$ is equivalent to $(B \vee A) \wedge (C \vee A)$

Absorption

27. $A \wedge (A \vee B)$ and $A \vee (A \wedge B)$ are equivalent to A.
    So are $A \wedge (B \vee A)$, $(A \wedge B) \vee A$, etc.

Proving validity by equivalences

Take any subformula of the given formula, and replace it by any equivalent formula.
Repeat. Any formula we get to is logically equivalent to the original.

If we end up with $\top$, the original formula is valid.

Eg: let's show $p \to p \vee q$ is valid.

- It is equivalent to $\neg p \vee (p \vee q)$
  — by the equivalence '$A \to B \equiv \neg A \vee B$'.
- This is equivalent to $(\neg p \vee p) \vee q$ — by $A \vee (B \vee C) \equiv (A \vee B) \vee C$.
- This is equivalent to $\top \vee q$ — by $\neg A \vee A \equiv \top$.
- This is equivalent to $\top$ (by $\top \vee A \equiv \top$), which is valid.

We got to $\top$, a valid formula. So $p \to p \vee q$ is valid.

Another example

Let's show $A \to B$ is equivalent to $\neg B \to \neg A$.

$\neg B \to \neg A$ is equivalent to $\neg(\neg B) \vee \neg A$ $\quad (X \to Y \equiv \neg X \vee Y)$

This is equivalent to $B \vee \neg A$ $\quad (\neg\neg X \equiv X)$.

This is equivalent to $\neg A \vee B$ $\quad$ (commut. of $\vee$: $X \vee Y \equiv Y \vee X$).

And this is equivalent to $A \to B$ $\quad$ ($\neg X \vee Y \equiv X \to Y$ again).

And another

Let’s show that $(p \vee \neg q) \wedge (p \vee q)$ is logically equivalent to p.

First instinct: ‘multiply out’. You’ll get
$(p \wedge p) \vee (p \wedge q) \vee (\neg q \wedge p) \vee (\neg q \wedge q)$,
and you can probably (eventually) simplify this to p.

But it’s faster to go:

$(p \vee \neg q) \wedge (p \vee q)$ is equivalent to $p \vee (\neg q \wedge q)$
(by $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ used ‘backwards’).

This is equivalent to $p \vee \bot$ $\quad$ ($\neg A \wedge A \equiv \bot$).

And this is equivalent to p $\quad$ ($A \vee \bot \equiv A$).

So look out for ‘backwards-uses’ of equivalences.

4.4 Normal forms

Equivalences allow us to rewrite a formula in equivalent *normal form.* There are two common normal forms:

Definition 4.2 (normal forms DNF, CNF)

- *A formula is in* disjunctive normal form *if it is a disjunction of conjunctions of literals, and is not further simplifiable by equivalences without leaving this form.*

  Examples: $p \vee q \vee \neg r$.
  $(p \wedge \neg q) \vee r \vee (\neg p \wedge q \wedge \neg r)$.
  Non-example: $(p \wedge p) \vee (q \wedge \top \wedge \neg q)$.

- *A formula is in* conjunctive normal form *if it is a conjunction of disjunctions of literals (that is, a conjunction of clauses), and is not further simplifiable by equivalences without leaving this form.*

  Example: $(p \vee \neg q) \wedge (q \vee r) \wedge (\neg p \vee q)$.

See definition 2.3 for literals, clauses.

Rewriting a formula in normal form

1. Get rid of $\to, \leftrightarrow$:

   Replace all subformulas $A \to B$ by $\neg A \vee B$.
   Replace all subformulas $A \leftrightarrow B$ by
   $(A \wedge B) \vee (\neg A \wedge \neg B)$.

   It's faster to replace $\neg(A \to B)$ by $A \wedge \neg B$, and $\neg(A \leftrightarrow B)$ by
   $(A \wedge \neg B) \vee (\neg A \wedge B)$.

2. Then use the De Morgan laws to push negations down next to atoms. Delete all double negations (replace $\neg\neg A$ by A).

3. Now rearrange using distributivity to get the desired normal form.

4. Simplify: replace subformulas $p \wedge \neg p$ by $\bot$, and $p \vee \neg p$ by $\top$.
   Replace $\top \vee p$ by $\top$, $\top \wedge p$ by p, $\bot \vee p$ by p, and $\bot \wedge p$ by $\bot$.
   Equivalence 27 is often useful too. Repeat till no further progress.

Eg: write $p \wedge q \to \neg(p \leftrightarrow \neg r)$ in DNF

- $p \wedge q \to \neg(p \leftrightarrow \neg r)$ [the original formula]
- $\neg(p \wedge q) \vee ((p \wedge \neg\neg r) \vee (\neg p \wedge \neg r))$ [got by eliminating $\to, \neg\leftrightarrow$]
- $\neg p \vee \neg q \vee (p \wedge r) \vee (\neg p \wedge \neg r)$ [by De Morgan law & $\neg\neg A \equiv A$.]
- $\neg p \vee (\neg p \wedge \neg r) \vee (p \wedge r) \vee \neg q$ [by commutativity of $\vee$]
- $\neg p \vee (p \wedge r) \vee \neg q$ [$A \vee (A \wedge B) \equiv A$].
  This is in DNF.

We can simplify further if we are willing to leave DNF temporarily:

- $[(\neg p \vee p) \wedge (\neg p \vee r)] \vee \neg q$ [by distributivity]
- $[\top \wedge (\neg p \vee r)] \vee \neg q$ [$\neg A \vee A \equiv \top$]
- $\neg p \vee r \vee \neg q$ [$\top \wedge A \equiv A$]

The drastic simplification from the original shows the benefit of rewriting in normal form.

5. Proof systems: natural deduction

A *proof system* is a way of showing formulas to be valid by using purely syntactic rules — not using meaning at all. A computer should be able to apply the rules.

Modern automated reasoning is a growing and successful area.

There are many proof systems. We will do *natural deduction,* a system invented by F. B. Fitch (and related to work of Gentzen); it is a major topic and will take some time.

Computing students will see a closely-related system in Discrete Structures.

Pandora: `https://www.doc.ic.ac.uk/pandora/newpandora/`

Pandora is a lab program that allows you to do natural deduction proofs as a kind of computer game.

It was originally done by Dan Ehrlich as a 4th-year individual project, c. 1998. It has evolved since then, always developed by students, supervised by *Krysia Broda* and (in the past) Gabrielle Sinnadurai. It's becoming known worldwide.

I asked the 2005, 2006 and 2009 students to rate how useful they found Pandora in learning natural deduction. Responses:

	not tried	$1/5$	$2/5$	$3/5$	$4/5$	$5/5$	total
2005	6	4	7	15	31	12	75
2006	3	3	4	12	23	15	60
2009	2	1	5	14	37	29	86

So please try it, at least until you get the hang of it.

What is natural deduction?

- A formalisation of 'direct argument'.
- Starting perhaps from some given facts — some formulas A_1, $\ldots, A_n$ — we use the rules of the system to reason towards a formula B. (In practice, we reason forwards, backwards, and especially inwards.) If we succeed, we can write $A_1, \ldots, A_n \vdash B$.
- The reasoning works by writing down intermediate formulas using the rules. These formulas form the *proof* of B from the givens $A_1, \ldots, A_n$. Once established, they may be usable later. Each step of the proof is a valid argument.
- Mostly, there are two rules for each connective: one for *introducing* it (for bringing in a formula of which it is the principal connective), and one for *using* it (using a formula of which it is the principal connective). The rules are based on the semantics for the connectives given earlier.

5.1 Natural deduction rules for $\wedge, \to, \vee$

There are two rules for each of these connectives. The rules reflect the meanings of the connectives.
The easiest is $\wedge$ (‘and’).

Rules for $\wedge$

- ($\wedge$-introduction, or $\wedge I$) To introduce a formula of the form $A \wedge B$, you have to have already introduced A and B.

1	A	we proved this. . .
	$\vdots$	(other junk)
2	B	and this. . .
3	$A \wedge B$	$\wedge I(1, 2)$

The line numbers are essential for clarity.

Rules for $\wedge$ (continued)

- ($\wedge$-elimination, or $\wedge E$) If you have managed to write down $A \wedge B$, you can go on to write down A and/or B.

1	$A \wedge B$	we proved this somehow
2	A	$\wedge E(1)$
3	B	$\wedge E(1)$

Assumptions

In ND proofs, we often need to temporarily ‘*assume* for the sake of argument that. . . ’.

An *assumption* here is just a formula, but it’s used in a special way. We use our imagination to conjure up situation(s) in which the formula holds. That is, we *assume* that (we are in a situation in which) the formula is true. Then we may make progress, because we know more about the situation.

Example (‘Silver Blaze’, A. Conan Doyle, 1893): Sherlock Holmes *assumes* for a moment that whoever took the valuable racehorse in the night was a stranger. Then, he is sure, the guard dog would have barked, which would have woken the stable-boys.

But the boys slept all night. Holmes deduces that the dog didn’t bark in the night, so that whoever took the horse was no stranger.

VITAL NOTE: his assumption was (therefore) *wrong* — but he could still assume or imagine it mentally, and it was useful to do so.

Rules for $\to$

- ($\to$-introduction, $\to I$: 'arrow-introduction') To introduce a formula of the form $A \to B$, you *assume* A and then prove B.

  During the proof, you can use A as well as anything already established. But *you can't use A or anything from the proof of B from A later on* (because it's based on an extra assumption). So we isolate the proof of B from A, in a *box:*

1	A	ass
	⟨the proof⟩	hard struggle
2	B	we made it!
3	$A \to B$	$\to I(1,2)$

  Nothing inside the box can be used later.

In natural deduction, boxes are used when we make additional assumptions. The first line inside a box should always be labelled 'ass' (assumption) — with one exception, coming later ($\forall I$).

Rules for $\rightarrow$ (continued)

- ($\rightarrow$-elimination, or $\rightarrow E$) If you have managed to write down A and $A \rightarrow B$, in either order, you can go on to write down B. (This is modus ponens.)

1	$A \rightarrow B$	we got this somehow...
	$\vdots$	other junk
2	A	and this too...
3	B	$\rightarrow E(1, 2)$

Rules for $\vee$

- ($\vee$-introduction, or $\vee I$)
  To prove $A \vee B$, prove A, or (if you prefer) prove B.

1	A	proved this somehow
2	$A \vee B$	$\vee I(1)$

B can be any formula at all!

1	B	proved this somehow
2	$A \vee B$	$\vee I(1)$

A can be any formula at all.

Rules for $\vee$ (continued)

- ($\vee$-elimination, or $\vee E$) To prove something from $A \vee B$, you have to prove it by assuming A, AND prove it by assuming B.

  This is arguing by cases.

1	$A \vee B$				we got this somehow
2	A	ass	5	B	ass
3	$\vdots$	the 1st proof	6	$\vdots$	the 2nd proof
4	C	we got it	7	C	we got it again
8	C				$\vee E(1, 2, 4, 5, 7)$

  The assumptions A, B are not usable later, so are put in (side-by-side) boxes. *Nothing inside the boxes can be used later.*

  C can be any formula at all; but both boxes must end with the same C.

5.2 Examples

With rules for $\wedge, \to, \vee$ we can already do quite a lot. (We still need rules for $\neg, \top, \bot$.)

Example 5.1 Let's prove $A \to A \vee B$ by natural deduction (where A, B are arbitrary formulas).

We did this earlier by direct argument (page 50). Compare the two proofs: they're quite similar!

1	A	ass
2	$A \vee B$	$\vee I(1)$
3	$A \to A \vee B$	$\to I(1,2)$

A similar English argument: 'In some situation, assume A is true. Then $A \vee B$ is true. So, $A \to A \vee B$ is true in *any* situation.'

The rules are natural... but they are still syntactic

Note how natural the rules are — they come from the meaning of the connectives.

HOWEVER: yes, the ND rules are *motivated* by the meaning of $\wedge, \vee, \ldots$, but nonetheless they are just *syntactic rules.*

A natural deduction proof is syntactic. A computer could do it, without knowing about ‘meaning’.

More examples

Example 5.2 Given $A \to (B \to C)$, prove $A \wedge B \to C$.

1	$A \to (B \to C)$	given
2	$A \wedge B$	ass
3	A	$\wedge E(2)$
4	$B \to C$	$\to E(1,3)$
5	B	$\wedge E(2)$
6	C	$\to E(4,5)$
7	$A \wedge B \to C$	$\to I(2,6)$

English paraphrase: ‘Suppose $A \to (B \to C)$ is true (in some situation). To show $A \wedge B \to C$ is true, assume $A \wedge B$ is also true (in this situation), and show C. Well, as $A \wedge B$, we have A, so using $A \to (B \to C)$, we see that $B \to C$ (is true). But $A \wedge B$ tells us B too, so we get C. Done.’

The notation ‘$\vdash$’

We proved $A \wedge B \to C$ from the ‘given’ $A \to (B \to C)$.
We can now write

$$A \to (B \to C) \;\vdash\; A \wedge B \to C.$$

Definition 5.3 *Let* $A_1, \ldots, A_n, B$ *be arbitrary formulas.*

$$A_1, \ldots, A_n \vdash B$$

means that there is a (natural deduction) proof of B*, starting with the formulas* $A_1, \ldots, A_n$ *as ‘givens’.*

- $\vdash B$ (the case $n = 0$) means we can prove B with no givens at all. We then say that B is a ‘theorem’ (of natural deduction).
- Can read $A_1, \ldots, A_n \vdash B$ as ‘B is provable from $A_1, \ldots, A_n$’, and $\vdash B$ as ‘B is a “theorem”’. ‘$\vdash$’ is called ‘single turnstile’.
- Do not confuse $\vdash$ with $\models$.
  $\vdash$ is syntactic and involves proofs.
  $\models$ is semantic and involves situations.

Another example

Example 5.4 $A \to C,\ B \to C\ \vdash\ A \lor B \to C$.

1	$A \to C$	given			
2	$B \to C$	given			
3	$A \lor B$	ass			
4	A	ass	6	B	ass
5	C	$\to E(4,1)$	7	C	$\to E(6,2)$
8	C	$\lor E(3,4,5,6,7)$			
9	$A \lor B \to C$	$\to I(3,8)$			

In English: 'Suppose we're given $A \to C$ and $B \to C$. To show $A \lor B \to C$, assume we have $A \lor B$, and show C. Well, $A \lor B$ tells us we have A or B. If we have A, then remember we have $A \to C$ as well — so we have C. If we have B, then because we have $B \to C$, we have C. This covers all cases. So we have C anyway. Done.'

5.3 Rules for $\neg$

This is the trickiest case. Also, $\neg$ has three rules! The first two treat $\neg A$ like $A \to \bot$.

- ($\neg$-introduction, $\neg I$) To prove $\neg A$, you assume A and prove $\bot$.

  As usual, you can't then use A later on, so enclose the proof of $\bot$ from assumption A in a box:

1	A	ass
2	$\vdots$	more hard work, oh no
3	$\bot$	we got it!
4	$\neg A$	$\neg I(1, 3)$

Rules for $\neg$ (continued)

- ($\neg$-elimination, $\neg E$): From A and $\neg A$, deduce $\bot$.

1	$\neg A$	proved this somehow...
2	$\vdots$	junk
3	A	...and this
4	$\bot$	$\neg E(1,3)$

- ($\neg\neg$-elimination, $\neg\neg$): From $\neg\neg A$, deduce A.

1	$\neg\neg A$	proved this somehow...
2	A	$\neg\neg(1)$

See example 5.8 for a real example of $\neg\neg$ in use.

5.4 Examples of ¬-rules in action

Example 5.5 (Holmes and the racehorse)

- stranger = ‘whoever took the valuable racehorse in the night was a stranger’
- barked = ‘the dog barked in the night’
- woke = ‘the stable-boys woke up’

1	stranger $\rightarrow$ barked	given
2	barked $\rightarrow$ woke	given
3	$\neg$woke	given
4	stranger	ass
5	barked	$\rightarrow E(4,1)$
6	woke	$\rightarrow E(5,2)$
7	$\perp$	$\neg E(3,6)$
8	$\neg$stranger	$\neg I(4,7)$

Another example

Example 5.6 Let's prove $A \vdash \neg\neg A$.

1	A	given
2	$\neg A$	ass
3	$\bot$	$\neg E(1,2)$
4	$\neg\neg A$	$\neg I(2,3)$

And another. . .

Example 5.7 Now prove $\neg(A \vee B) \vdash \neg A$.

1	$\neg(A \vee B)$	given
2	A	ass
3	$A \vee B$	$\vee I(2)$
4	$\perp$	$\neg E(1,3)$
5	$\neg A$	$\neg I(2,4)$

In English: ‘Given $\neg(A \vee B)$, if you had A, then you’d have $A \vee B$, which is a contradiction. So you’ve got $\neg A$.’

This is another illustration of the ‘Law of Excluded Middle’: if you haven’t got A, you’ve got $\neg A$.

This law is true of classical logic, but not of some other logics.

And finally…you can prove anything from $\perp$!

Example 5.8 Let's show $\perp \vdash A$ *for any A.*

$$
\begin{array}{lll}
1 & \perp & \text{given} \\
\hline
\multicolumn{1}{|l}{2} & \neg A & \multicolumn{1}{r|}{\text{ass}} \\
\multicolumn{1}{|l}{3} & \perp & \multicolumn{1}{r|}{\checkmark(1)} \\
\hline
4 & \neg\neg A & \neg I(2,3) \\
5 & A & \neg\neg(4)
\end{array}
$$

In English: 'Suppose we are in a situation where $\perp$ is true. If $\neg A$ is true then $\perp$ is true (as we knew already), and this is impossible. So $\neg\neg A$ and hence A are true in this situation'.
(Note the use of $\checkmark$ in line 3 to justify a formula that's already available.)

Anything follows from falsity!
Anything follows from a contradiction!

Tricky, eh?

You are going to find the argument above hard to accept. You will not be alone: it has led some to reject or reformulate classical logic.

The effect of dividing by zero in arithmetic is the same. It leads to paradox — that is, $\perp$.

Discussion

I could just say that we are just following proof rules. If the rules allow us to get from $\perp$ to A, then fine.

Better: the English paraphrase says SUPPOSE we are in a situation in which $\perp$ is true... then A is true. (And $\neg A$ is true too, so A is false,...)
But there is no such situation. So you can't dispute the crazy things that follow from this supposition.
Compare: $A \to B$ is true if A is false.

The *practical point* is that there is no situation in which $\perp$ is true, so you can only prove $\perp$ under (contradictory) assumptions in a box in the middle of a bigger proof. You often get such assumptions — e.g., in an argument by cases $(\vee E)$. You can then deduce anything you want, so the box is no obstacle to the overall goal. So $\perp$ is a useful formula to aim to prove. See example 5.11 below.

Proving $A \vee \neg A$

Example 5.9 Let's prove $\vdash A \vee \neg A$. This is very useful, but the proof involves more $\neg$s and is moderately complicated.

1	$\neg(A \vee \neg A)$	ass
2	A	ass
3	$A \vee \neg A$	$\vee I(2)$
4	$\bot$	$\neg E(1,3)$
5	$\neg A$	$\neg I(2,4)$
6	$A \vee \neg A$	$\vee I(5)$
7	$\bot$	$\neg E(1,6)$
8	$\neg\neg(A \vee \neg A)$	$\neg I(1,7)$
9	$A \vee \neg A$	$\neg\neg(8)$

In English: 'Assume $\neg(A \vee \neg A)$. If A, then $A \vee \neg A$, which is impossible. So $\neg A$ (line 5). But then, $A \vee \neg A$, again impossible.

So the assumption was wrong, and so $\neg\neg(A \vee \neg A)$. Therefore, $A \vee \neg A$. Done.'

5.5 Rules for $\top$

- ($\top$-introduction, $\top I$) You can introduce $\top$ anywhere (for all the good it does you).
- ($\top$-elimination, $\top E$) You can prove nothing new from $\top$, sorry!

5.6 Rules for $\perp$

- ($\perp$-introduction, $\perp I$) To prove $\perp$, you must prove A and $\neg A$ (for any A you like).

1	A	got this somehow
$\vdots$		
2	$\neg A$	and this
3	$\perp$	$\perp I(1,2)$

  This is the same rule as $\neg E$. (There are two names for this rule!)

- ($\perp$-elimination, $\perp E$) You can prove any formula at all from $\perp$! (See example 5.8.)

1	$\perp$	got this somehow...
2	A	$\perp E(1)$

The rules for $\perp$ can actually be obtained from the other rules already explained.

5.7 Lemmas

A *lemma* is something you prove that helps in proving what you really wanted.

In example 5.9 we proved $\vdash A \vee \neg A$. This is useful in all sorts of proof. It divides the argument into two cases (A, and $\neg A$). This makes it easier, since we know more in each case.

So in ND proofs, I'll generally allow you to quote $A \vee \neg A$ as a lemma, without proving it. Justify by *'Lemma'.*

It's the only lemma I'm going to give you. . . so remember it well.

But always *you* have to choose which A to use.

Warning: Pandora calls this particular lemma 'EM' (standing for Excluded Middle). Click on the 'EM' button to use it.

The Pandora 'lemma' button just chops a proof into two halves (try it!). It lets you make your own lemmas. Don't use it if you want $A \vee \neg A$ — use the 'EM' button!

Example 5.10 Let's prove $\vdash ((A \to B) \to A) \to A$ (Peirce's law). (We did this earlier by direct argument.)

1	$(A \to B) \to A$	ass
2	$A \vee \neg A$	lemma
3	A	ass
4	A	$\checkmark(3)$
5	$\neg A$	ass
6	A	ass
7	$\bot$	$\neg E(5,6)$
8	B	$\bot E(7)$
9	$A \to B$	$\to I(6,8)$
10	A	$\to E(9,1)$
11	A	$\vee E(2,3,4,5,10)$
12	$((A \to B) \to A) \to A$	$\to I(1,11)$

5.8 Rules for $\leftrightarrow$

Roughly, we treat $A \leftrightarrow B$ as $(A \to B) \land (B \to A)$.

- ($\leftrightarrow$-introduction, $\leftrightarrow I$)
  To prove $A \leftrightarrow B$, prove both $A \to B$ and $B \to A$.

1	$B \to A$	proved this somehow...
	$\vdots$	
2	$A \to B$	...and this
3	$A \leftrightarrow B$	$\leftrightarrow I(1,2)$

Rules for $\leftrightarrow$ (continued)

- ($\leftrightarrow$-elimination, $\leftrightarrow E$)
  From $A \leftrightarrow B$ and A, you can prove B.
  From $A \leftrightarrow B$ and B, you can prove A.

1	$A \leftrightarrow B$	proved this somehow. . .
2	A	. . . and this
3	B	$\leftrightarrow E(1,2)$

1	$A \leftrightarrow B$	proved this somehow. . .
2	B	. . . and this
3	A	$\leftrightarrow E(1,2)$

5.9 Derived rules

These are NOT necessary, but they do help to speed proofs up.

A good one is PC (Proof by Contradiction):

To prove A, assume $\neg A$ and prove $\perp$.

The effect of PC is to combine applications of $\neg I$ and $\neg\neg$:

$$\begin{array}{|l l r|}
\hline
1 & \neg A & \text{ass} \\
2 & \vdots & \text{hard slog} \\
3 & \bot & \text{got it!} \\
\hline
\multicolumn{1}{l}{4} & \neg\neg A & \multicolumn{1}{r}{\neg I(1,3)} \\
\multicolumn{1}{l}{5} & A & \multicolumn{1}{r}{\neg\neg(4)}
\end{array}$$

Can replace this by:

$$\begin{array}{|l l r|}
\hline
1 & \neg A & \text{ass} \\
2 & \vdots & \text{as before} \\
3 & \bot & \text{got it!} \\
\hline
\multicolumn{1}{l}{4} & A & \multicolumn{1}{r}{PC(1,3)}
\end{array}$$

Using PC cuts out a line. This is surprisingly helpful.

5.10 Advice for doing natural deductions

1. Learn the rules. Practise! Use Pandora.
2. Think of a direct argument to prove what you want. Then translate it into natural deduction.
3. If really stuck in proving A, it can help to:
   - Assume $\neg A$ and prove $\bot$. (‘Reductio ad absurdum’, or ‘proof by contradiction’, PC.)
     We did this when proving $\vdash A \vee \neg A$ in example 5.9.
   - Use the lemma $B \vee \neg B$, for suitable B. It amounts to arguing by cases. See example 5.10 for an application of this lemma.

The ‘Reasoned Programming’ book and `www.doc.ic.ac.uk/~imh/teaching/140_logic/advice.pdf` gives more advice, but the above is the most important.

A typical nasty-ish example

Example 5.11 Let's show

$$A \vee B, \ \ \neg C \to \neg A, \ \ \neg(B \wedge \neg C) \ \vdash \ C.$$

Baffled, eh?

Well, assume for the sake of argument that you had $\neg C$.
Then you'd have $\neg A$ — but you're given A or B, so you get B.
Now you've got both B and $\neg C$, which you're told you don't: contradiction.

SO, you must have C.

This is quite easy to translate into ND. There's a scuffle with the $\vee$, but you get used to ND's idiosyncrasies like this.

Example 5.11: $A \vee B,\ \neg C \to \neg A,\ \neg(B \wedge \neg C)\ \vdash\ C$

1	$A \vee B$	given			
2	$\neg C \to \neg A$	given			
3	$\neg(B \wedge \neg C)$	given			
4	$\neg C$	ass			
5	$\neg A$	$\to E(2,4)$			
6	A	ass	9	B	ass
7	$\bot$	$\neg E(5,6)$			
8	B	$\bot E(7)$	10	B	$\checkmark(9)$
11	B	$\vee E(1,6,8,9,10)$			
12	$B \wedge \neg C$	$\wedge I(11,4)$			
13	$\bot$	$\neg E(3,12)$			
14	C	$PC(4,13)$			

Boxes: things to remember

A box is ‘its own little world’ with its own assumptions. Care is needed.

1. A box *always* starts with an assumption (the *only* exception is in $\forall I$ in first-order logic later).

2. An assumption can occur *only* on the first line of a box.

3. Inside a box, you can use any earlier formulas (except formulas in completed earlier boxes — see (5)).

4. The *only* ways of exporting information from a box are by the rules $\rightarrow I$, $\vee E$, $\neg I$, and PC (and also $\exists E$ and $\forall I$ in first-order logic). The first line after a box *must* be justified by one of these.

5. No formula inside a box can be used outside, except via (4).

You can check these conditions for yourself. If your box doesn’t meet them, your proof is wrong.

Example of how not to use boxes: $\neg A \vdash \neg A$ (!)

$$\begin{array}{llll}
1 & \neg A & \text{given} & \\
2 & \neg A & \checkmark(1) & \text{the best proof!}
\end{array}$$

$$\begin{array}{llll}
1 & \neg A & \text{given} & \\
\multicolumn{3}{l}{\boxed{\begin{array}{lll} 2 & A & \text{ass} \\ 3 & \bot & \neg E(2,1) \end{array}}} & \\
4 & \neg A & \neg I(2,3) & \text{correct but silly}
\end{array}$$

$$\begin{array}{lll}
1 & \neg A & \text{given} \\
\multicolumn{3}{l}{\boxed{\begin{array}{lll} 2 & A & \text{ass} \\ 3 & \bot & \neg E(2,1) \end{array}}} \\
4 & \neg A & \bot E(3)
\end{array}
\quad \leftarrow \text{WRONG} \rightarrow \quad
\begin{array}{lll}
1 & \neg A & \text{given} \\
\multicolumn{3}{l}{\boxed{\begin{array}{lll} 2 & A & \text{ass} \\ 3 & \bot & \neg E(2,1) \\ 4 & \neg A & \bot E(3) \end{array}}} \\
5 & \neg A & \checkmark(4)
\end{array}$$

Variants (see exam questions!!)

The ND system we've seen can be varied by

- changing the rules (carefully!)
- introducing new connectives and giving rules for them.

E.g. (exam 2007): The IF connective can be defined as

$$IF(p,q,r) = (p \to q) \wedge (\neg p \to r).$$

Here's an introduction rule for IF, based on the rules $\to I$ and $\wedge I$:

1	A	ass	3	$\neg A$	ass
	$\vdots$			$\vdots$	
2	B	got this somehow…	4	C	…or other
5	$IF(A,B,C)$				$IFI(1,2,3,4)$

Exercise: what would a good elimination rule (or rules) be?

5.11 $\vdash$ versus $\models$

Our main concern is $\models$:
recall $A_1, \ldots, A_n \models B$ if B is true in all situations in which $A_1, \ldots, A_n$ are true.

$\vdash$ is useless unless it helps to establish $\models$.

Definition 5.12 *A* theorem *is a formula that can be established ('proved') by a given proof system.*

We are concerned with natural deduction. So a theorem is any formula A such that $\vdash A$.

Definition 5.13 *A proof system is* sound *if every theorem is valid, and* complete *if every valid formula is a theorem.*

Soundness and completeness

In fact, it can be shown that natural deduction is sound and complete:

Theorem 5.14 (soundness) *Let $A_1, \ldots, A_n, B$ be any propositional formulas. If $A_1, \ldots, A_n \vdash B$, then $A_1, \ldots, A_n \models B$.*

Slogan (for the case $n = 0$):
'Any provable propositional formula is valid.'
'Natural deduction never makes mistakes.'

Theorem 5.15 (completeness)
Let $A_1, \ldots, A_n, B$ be any propositional formulas. If $A_1, \ldots, A_n \models B$, then $A_1, \ldots, A_n \vdash B$.

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

So we can use natural deduction to check validity.

Consistency

This is an important (fundamental) logical idea.

Definition 5.16 (consistency) *A formula A is said to be consistent if $\nvdash \neg A$.*

A collection $\mathcal{C}$ of formulas is said to be consistent if for every integer $n \geq 1$ and $A_1, \ldots, A_n$ in $\mathcal{C}$, the formula $\bigwedge_{1\leq i\leq n} A_i$ is consistent.

By soundness and completeness (theorems 5.14 and 5.15), we get:

Theorem 5.17 *A formula is consistent if and only if it is satisfiable.*

See definition 3.3 for ‘satisfiable’.

Interlude: very sketchy history of logic

Logic has been studied since ancient Greek times (especially Aristotle). It began as analysis of human reasoning, to put notion of ‘valid and invalid argument’ (the ‘laws of thought’) on a solid foundation. Some say it should return to these roots. . .

Aristotle laid down logical principles called *syllogisms,* (eg ‘All women are human, no human is immortal, *so* no woman is immortal’) which were taught and used to analyse/verify arguments through the middle ages. (See Lemmon’s book.) Famous medieval logicians include Peter Abelard, William of Ockham, and Ibn Sīnā.

In the 19th century, algebraic versions of logic were developed by, e.g., George Boole. People realised the limitations of ‘the syllogism’, and algebraic-style logics to handle binary relations (eg ‘x is a daughter of y’) were developed. The algebraic tradition survives today but is no longer so popular.

Around the end of the 19th century, Gottlob Frege developed *quantifier logic* and much of modern logic depends on his work, even though Bertrand Russell wrote him a letter showing his system to be inconsistent. Charles S. Peirce (pronounced ‘Purse’) independently followed a similar path about the same time.

The ‘Russell paradox’ led to a crisis in the foundations of mathematics, which was resolved slowly over the early 20th century. First-order logic (and set theory) now serves as the chief foundation for modern mathematics.

The limitations of first-order logic (as a result of its being too powerful) were revealed by work of Kurt Gödel, Alan Turing, and Alonzo Church in the 1930s.

Gödel also proved soundness and completeness of a deductive system for first-order logic (1929). Alfred Tarski gave semantics (formal meaning) for it (1933–1950s). Gentzen and others extended its proof theory. At this point, first-order logic takes its modern form.

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 PC* and *Frank bought grapes* as atomic, without internal structure.

Now we look inside them.

We regard being a PC 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)* `PC`.
  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 `PC(Heron)` and `bought(Frank,grapes)` are two new atomic formulas.

6.2 Quantifiers

So what? You may think that writing

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

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 particular 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 PCs’ 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 $\mathtt{PC}(\mathtt{Heron})$, we'll write ones like $\mathtt{PC}(x)$.

- Now, to say 'Everything is a PC', we'll write $\forall x\, \mathtt{PC}(x)$.
  This is read as: 'For all x, x is a PC'.
- 'Something is a PC', can be written $\exists x\, \mathtt{PC}(x)$.
  'There exists x such that x is a PC.'
- 'Frank bought a PC', can be written
  $$\exists x(\mathtt{PC}(x) \land \mathtt{bought}(\mathtt{Frank}, x)).$$
  'There is an x such that x is a PC and Frank bought x.'
  Or: 'For some x, x is a PC 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 `Frank`, `Susan`, `Tony`, `Heron`, `Texel`, `Clyde`, `Room-308`, and c
- unary relation symbols `PC`, `human`, `lecturer` (arity 1)
- a binary relation symbol `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 computing 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. $\texttt{bought}(\texttt{Frank}, x)$
   We read this as: ‘Frank bought x.’

2. $\exists x\, \texttt{bought}(\texttt{Frank}, x)$
   ‘Frank bought something.’

3. $\forall x(\texttt{lecturer}(x) \to \texttt{human}(x))$
   ‘Every lecturer is human.’ [Important eg!]

4. $\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{PC}(x))$
   ‘Everything Tony bought is a PC,’ or ‘Tony bought only PCs’.

Formation trees and subformulas (for examples see slides 134 & 138 later), 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 do sets and relations in Discrete Structures, course 142.

Example of a structure

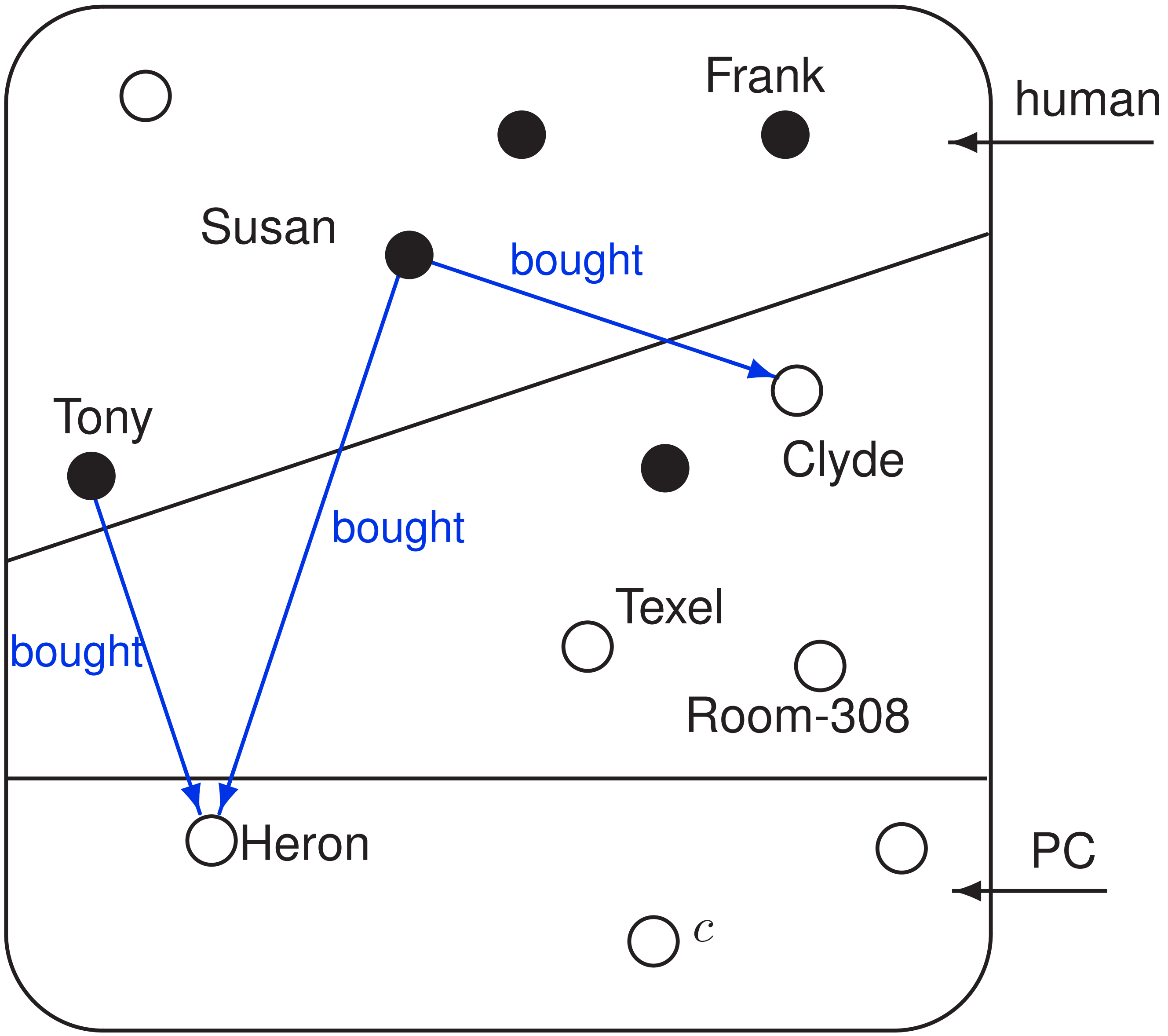
For our handy L, an L-structure should say:

- which objects are in its domain
- which of its objects are `Tony`, `Susan`, ...
- which objects are `human`, `PC`, `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 `PC`, `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 `Tony`?

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

They are quite different things. `Tony` *is syntactic.* ● *is semantic.*
In the context of M, `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 `Tony`) for this ●.
In a different structure, `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?

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

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

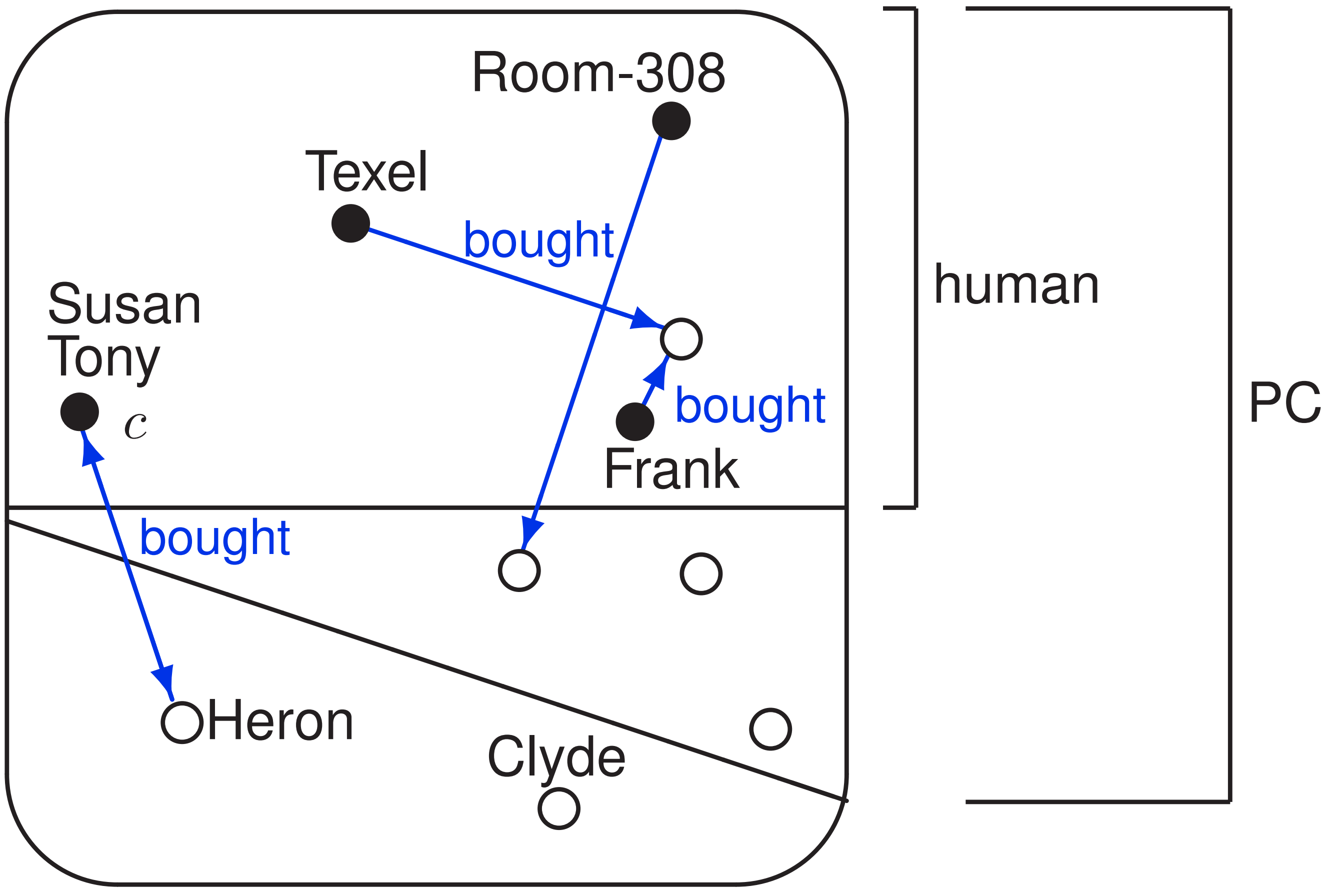
- bought(Susan,Susan) is false in M, because M does not say that the constant Susan names an object ● that bought itself.
  In symbols, $M \not\models \mathtt{bought}(\mathtt{Susan}, \mathtt{Susan})$.

From our knowledge of propositional logic,

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

Another structure

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



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

Some statements about M'

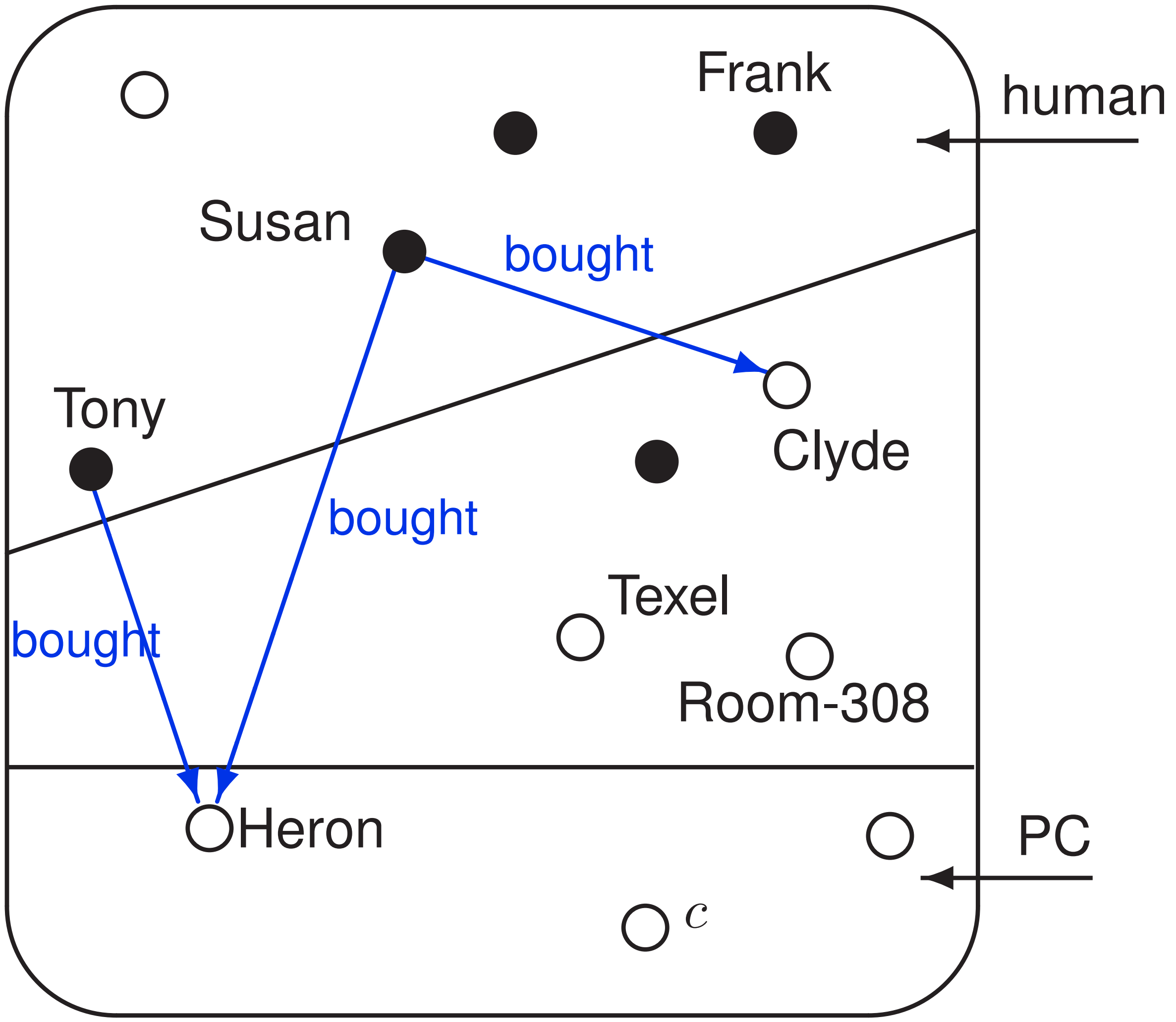
- $M' \not\models \texttt{bought}(\texttt{Susan}, \texttt{Clyde})$ this time.
- $M' \models \texttt{Susan} = \texttt{Tony}$.
- $M' \models \texttt{human}(\texttt{Texel}) \wedge \texttt{PC}(\texttt{Texel})$.
- $M' \models \texttt{bought}(\texttt{Tony}, \texttt{Heron}) \wedge \texttt{bought}(\texttt{Heron},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

How about

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

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.

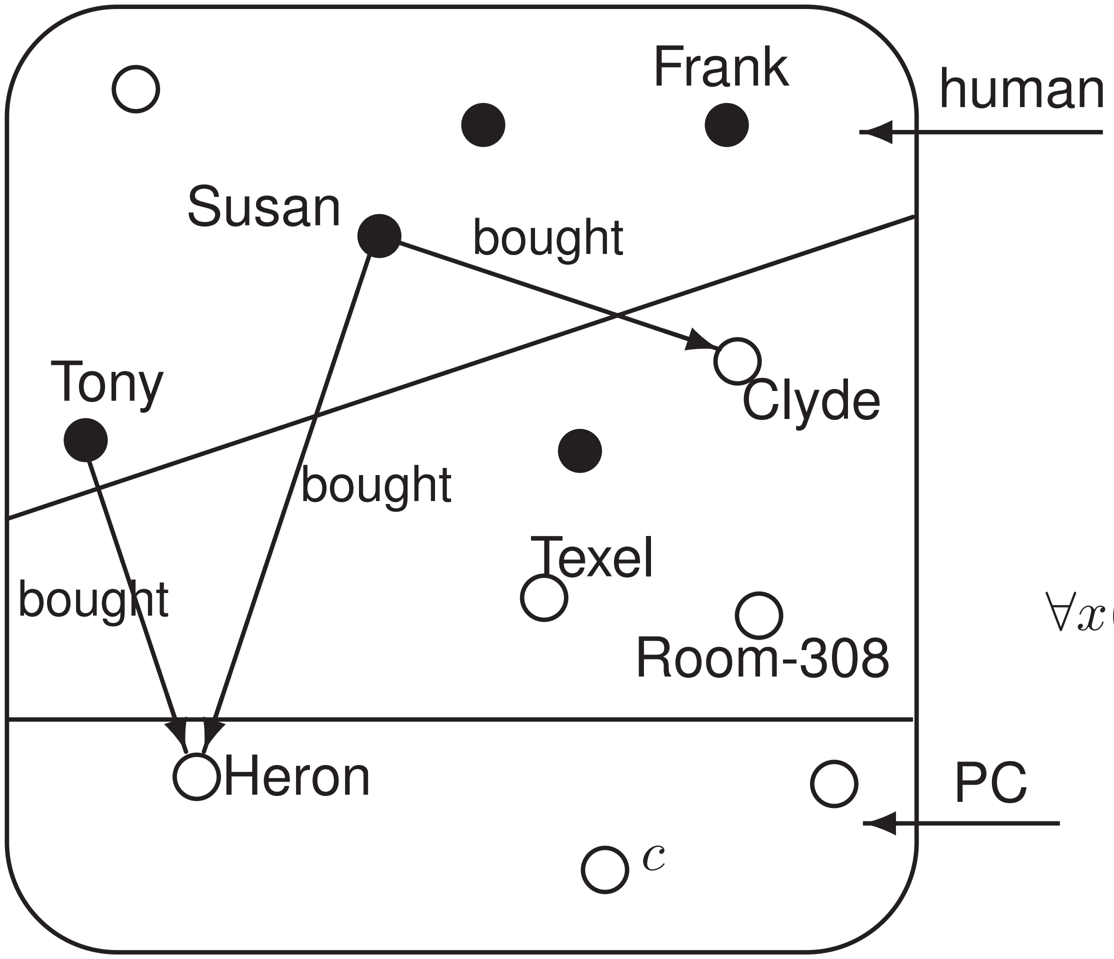
For this $\bigcirc$, $\texttt{bought}(\texttt{Susan}, \bigcirc)$ is true in M.
So $\texttt{bought}(\texttt{Tony}, \bigcirc) \to \texttt{bought}(\texttt{Susan}, \bigcirc)$ is true in M.

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



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

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

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

Rough guide: advice

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

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).

$$\begin{tikzcd}[arrows=dash]
 & \exists x(\texttt{PC}(x) \wedge \exists y\ \texttt{bought}(y,x)) \arrow[d] & \\
 & \texttt{PC}(x) \wedge \exists y\ \texttt{bought}(y,x) \arrow[dl] \arrow[dr] & \\
\texttt{PC}(x) & & \exists y\ \texttt{bought}(y,x) \arrow[d] \\
 & & \texttt{bought}(y,x)
\end{tikzcd}$$

$$\begin{tikzcd}[arrows=dash]
 & \text{something bought a PC} \arrow[d] & \\
 & x \text{ is a PC \& something bought } x \arrow[dl] \arrow[dr] & \\
x \text{ is a PC} & & \text{something bought } x \arrow[d] \\
 & & y \text{ bought } x
\end{tikzcd}$$

This is often a good guide to evaluating the formula.
E.g., the formula here says that there is an x that's a PC 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

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

Its formation tree is:

$$\begin{tikzcd}
& \forall x \arrow[d, no head] & \\
& \to \arrow[dl, no head] \arrow[dr, no head] & \\
\texttt{bought}(\texttt{Tony}, x) & & \texttt{PC}(x)
\end{tikzcd}$$

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

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

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

What's going on?

Free and bound variables

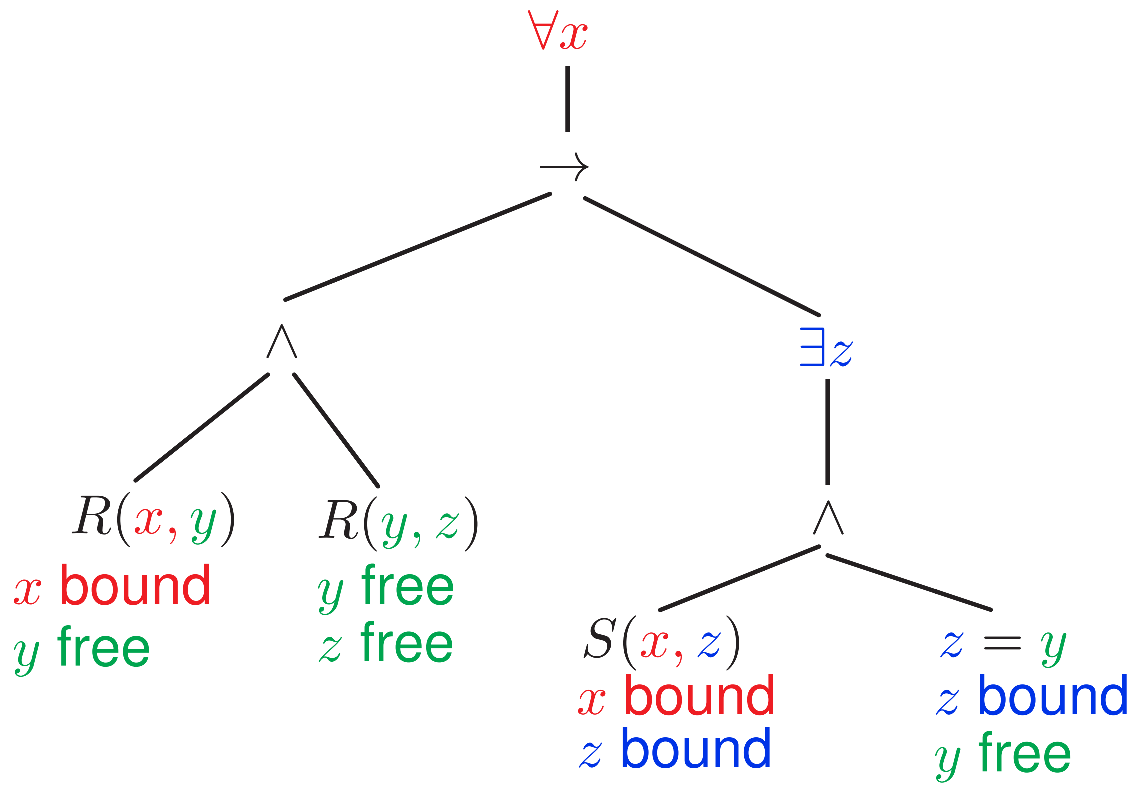
We'd better investigate how variables can arise in formulas.

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

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



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

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

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

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

  $\texttt{PC}(x)$

  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.

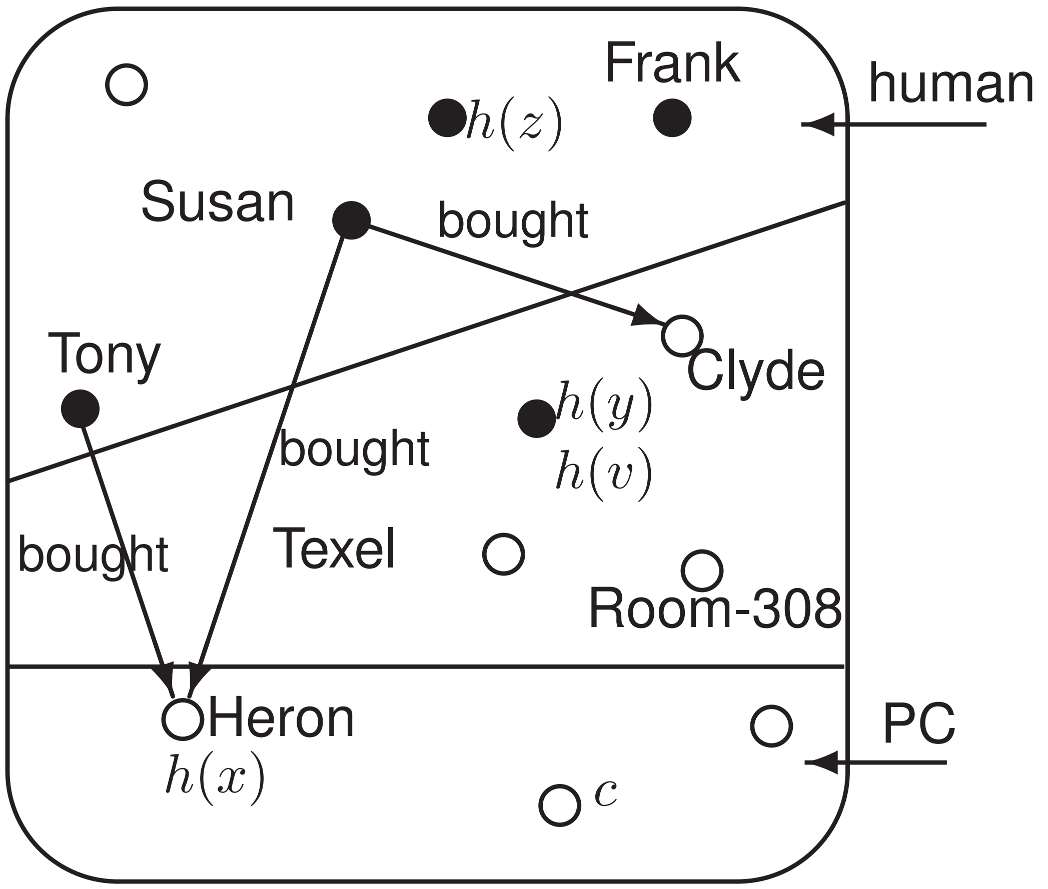
Definition 8.6 (value of term) Let L be a signature, M an L-structure, and h an assignment into M.

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.

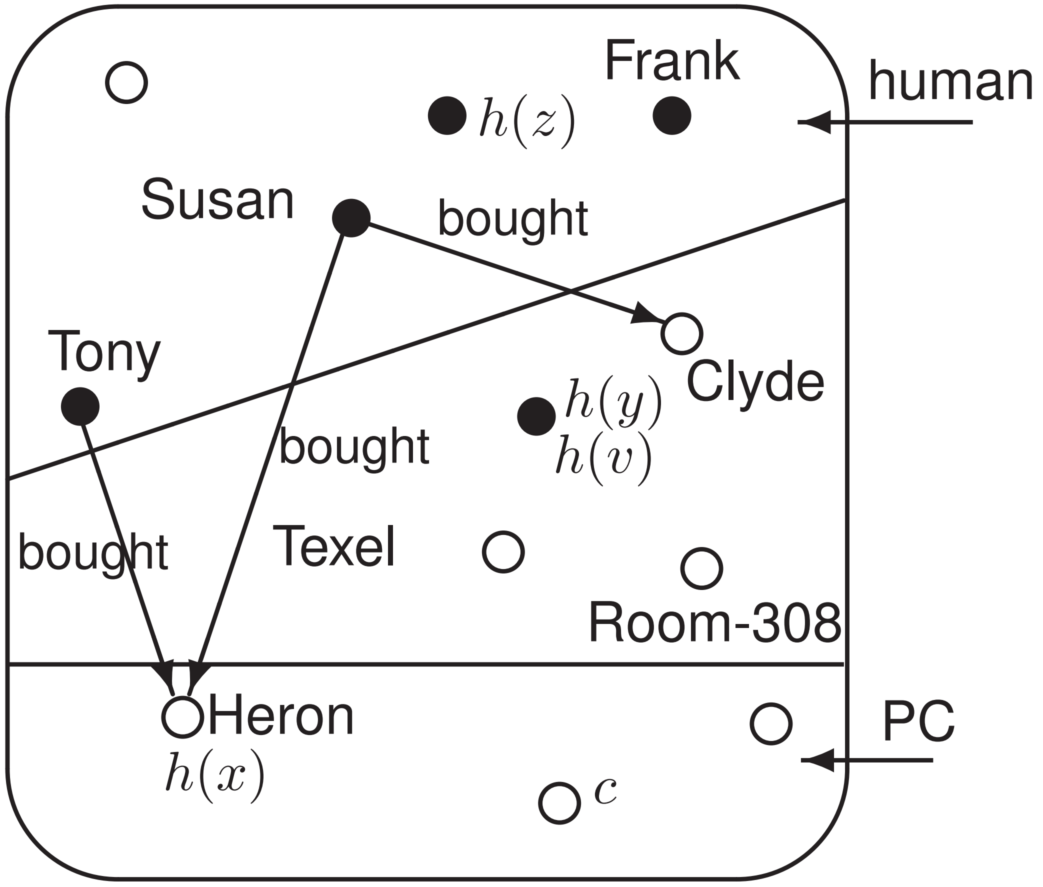
Semantics of quantifier-free formulas

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!)

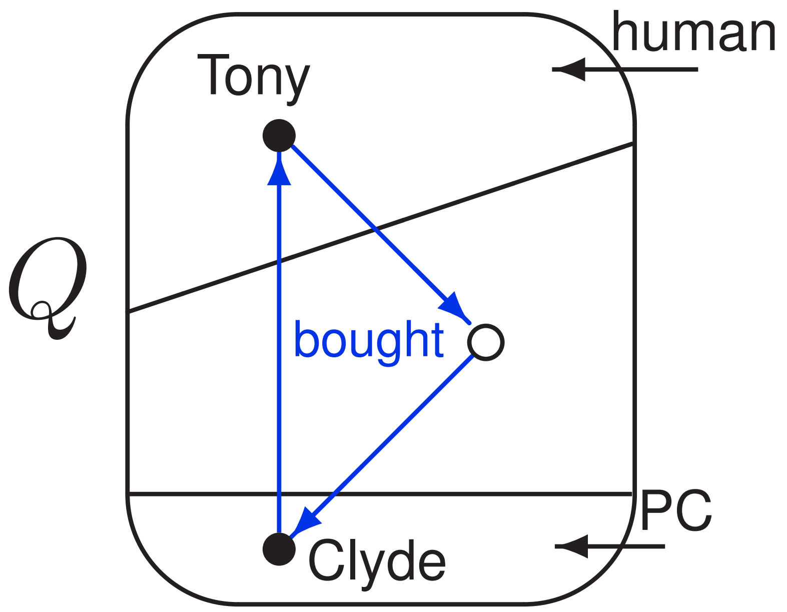
$\boxed{g =_x h \text{ means `} g \text{ agrees with } h \text{ on all variables except possibly } x\text{'.}}$

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	○	Clyde	
Tony	h_1	h_2	h_3	$=_x$
○	h_4	h_5	h_6	$=_x$
Clyde	h_7	h_8	h_9	$=_x$
	$\|_y$	$\|_y$	$\|_y$	

Eg: $h_2(x) = ○$, 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: $Q, h_4 \models \forall x \exists y\, \mathtt{bought}(x, y)$

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

These are: h_4, h_5, h_6.

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

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

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 $M, h \models A$ depends only 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 $B = (\exists x A)(a_1, \ldots, a_n)$ {
  repeat for all $b$ in $\mathrm{dom}(M)$:
    {if istrue($M, A(b, a_1, \ldots, a_n)$) 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 $\forall x(\mathtt{lecturer}(x) \to \mathtt{PC}(x))$, run through all x and check that every x that's a lecturer is a PC.

- Rewrite the formula in a more understandable form using equivalences (see later).

- Use a combination of the three.

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 $\texttt{prime}$, $\texttt{even}$, $>$, $+$, 2.

No-one knows whether

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

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'*: $\exists y\, \texttt{bought}(y, x)$.

- Any bought thing isn't human:
  $\forall x(\exists y\, \texttt{bought}(y, x) \to \neg\, \texttt{human}(x))$.
  Important: $\forall x \exists y(\texttt{bought}(y, x) \to \neg\, \texttt{human}(x))$ would not do.
- Every PC is bought: $\forall x(\texttt{PC}(x) \to \exists y\, \texttt{bought}(y, x))$.
- Some PC has a buyer: $\exists x(\texttt{PC}(x) \wedge \exists y\, \texttt{bought}(y, x))$.

- No lecturer bought a PC:
  $\neg \exists x(\texttt{lecturer}(x) \wedge \underbrace{\exists y(\texttt{bought}(x, y) \wedge \texttt{PC}(y)))}_{x \text{ bought a PC}}$.

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 $\exists x(\texttt{lecturer}(x) \to \texttt{human}(x))$.
- Frank bought a PC: $\exists x(\texttt{PC}(x) \wedge \texttt{bought}(\texttt{Frank}, x))$

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 PC: $\exists x\ \mathtt{PC}(x)$.
- There are at least two PCs: $\exists x \exists y(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \wedge x \neq y)$,
  or (more deviously) $\forall x \exists y(\mathtt{PC}(y) \wedge y \neq x)$.
- There are at least three PCs:
  $\exists x \exists y \exists z(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \wedge \mathtt{PC}(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z)$,
  or $\forall x \forall y \exists z(\mathtt{PC}(z) \wedge z \neq x \wedge z \neq y)$.
- There are no PCs: $\neg \exists x\ \mathtt{PC}(x)$
- There is at most one PC: 3 ways:
  1. $\neg \exists x \exists y(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \wedge x \neq y)$
     This says 'not(there are at least two PCs)' — see above.
  2. $\forall x \forall y(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \to x = y)$
  3. $\exists x \forall y(\mathtt{PC}(y) \to y = x)$
- There's exactly one PC: 3 ways:
  1. 'There's at least one PC' $\wedge$ 'there's at most one PC'
  2. $\exists x(\mathtt{PC}(x) \wedge \forall y(\mathtt{PC}(y) \to y = x))$
  3. $\exists x \forall y(\mathtt{PC}(y) \leftrightarrow y = x)$

10. Function symbols and sorts

— the icing on the cake (‘syntactic sugar’)

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

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 PCs, 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 $\mathrm{s} \rightarrow \mathrm{s}'$ 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 $\exists x A$

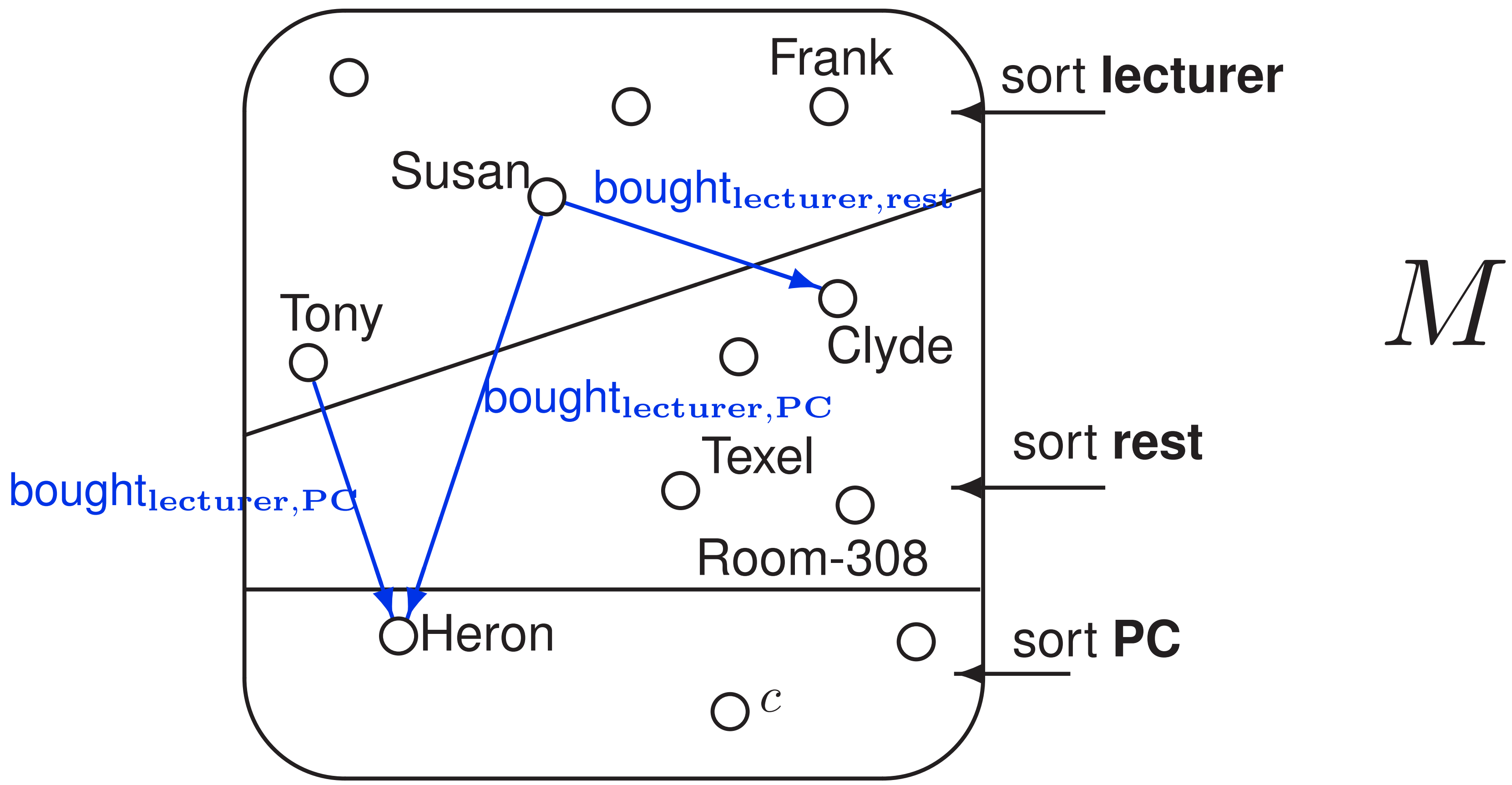
  if x has sort s. Alternatively, declare the variables of each sort.

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

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 164), but additionally it allocates *each* object in its domain to *a single sort.* No sort should be empty.

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



Frank
sort **lecturer**
Susan
$\text{bought}_{\mathbf{lecturer,rest}}$
M
Tony
Clyde
$\text{bought}_{\mathbf{lecturer,PC}}$
Texel
sort **rest**
$\text{bought}_{\mathbf{lecturer,PC}}$
Room-308
Heron
sort **PC**
c

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: $\mathtt{human}_{\mathbf{lecturer}}$, $\mathtt{human}_{\mathbf{PC}}$, $\mathtt{human}_{\mathbf{rest}}$.

   But if (e.g.) you don't want to talk about human objects of sort $\mathbf{PC}$, you can omit $\mathtt{human}_{\mathbf{PC}}$.

3. We need a binary relation symbol $\mathtt{bought}_{\mathbf{s},\mathbf{s}'}$ *for each pair* $(\mathbf{s}, \mathbf{s}')$ *of sorts* (unless $\mathbf{s}$-objects are not expected to buy $\mathbf{s}'$-objects).

4. Messy alternative: use sorts for human lecturer, PC-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, $\forall x : \mathbf{lecturer}\ \exists y : \mathbf{PC}(\mathtt{bought}_{\mathtt{lecturer,PC}}(x, y))$ is true in a structure if every object of sort $\mathbf{lecturer}$ bought an object of sort $\mathbf{PC}$.

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

It does not say that every $\mathbf{PC}$-object bought a $\mathbf{PC}$-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.)

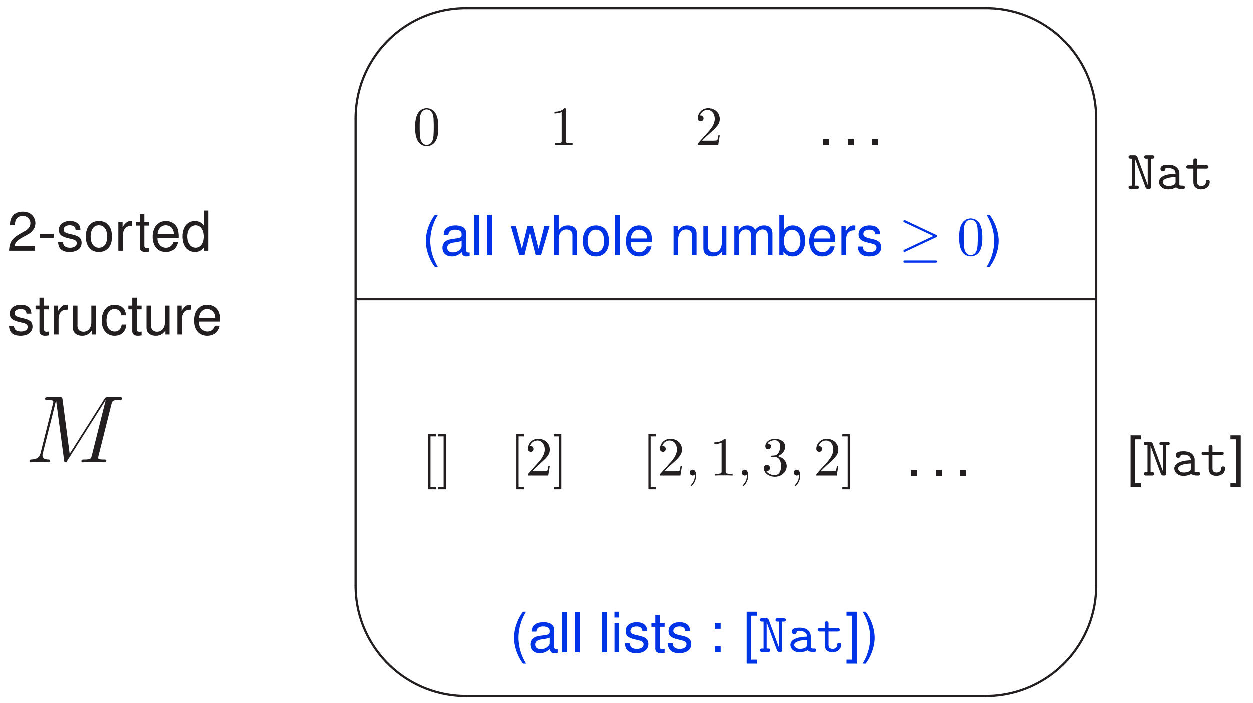
Next term: gory details.
This term: a gentle taster (but still very powerful).

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:


2-sorted
structure
M
$0 \quad 1 \quad 2 \quad \ldots$
(all whole numbers ≥ 0)
Nat
$[] \quad [2] \quad [2,1,3,2] \quad \ldots$
(all lists : [Nat])
[Nat]

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, length (which we write as $\sharp$), !!.

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 $\texttt{tail} : [\texttt{Nat}] \to [\texttt{Nat}]$.
   Choose an arbitrary value (of the right sort) for $\texttt{tail}([])$.
2. Use a relation symbol `Rtail`([`Nat`],[`Nat`]) instead.
   Make $\texttt{Rtail}(xs, ys)$ true just when ys is the tail of xs.
   If xs has no tail, $\texttt{Rtail}(xs, ys)$ 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`].

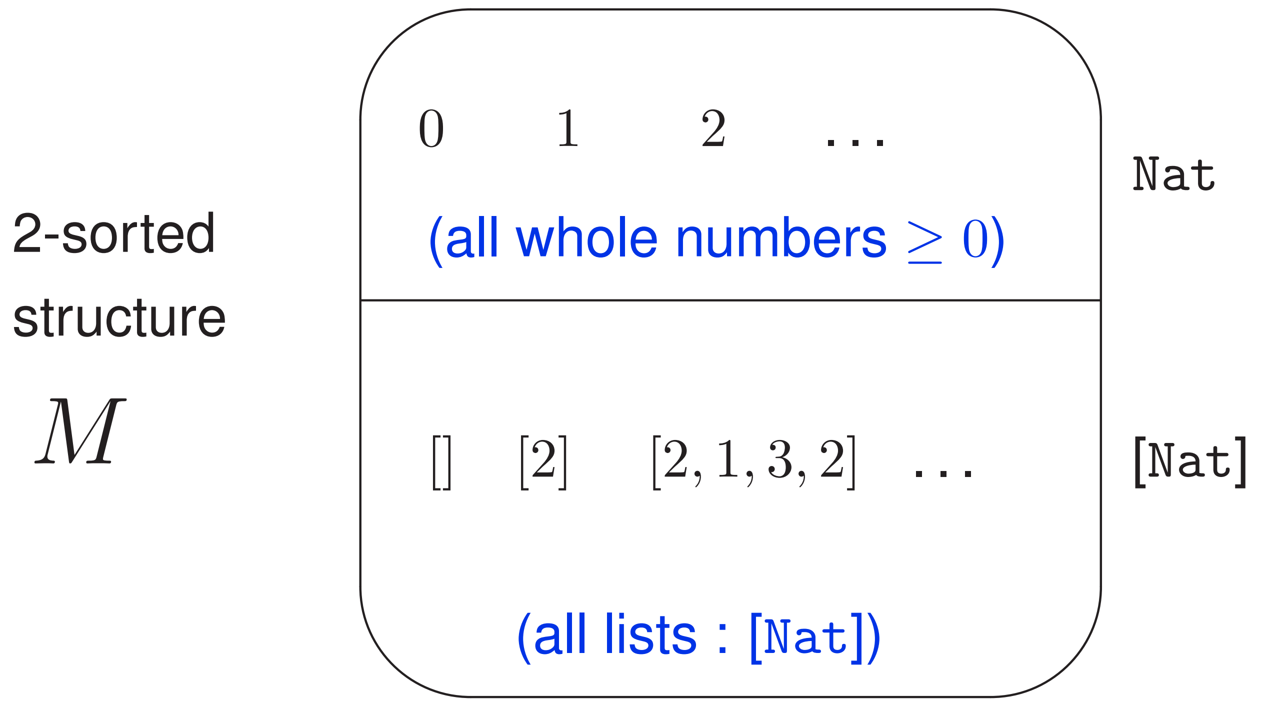
- L has constants $\underline{0}, \underline{1}, \ldots : \mathtt{Nat}$, relation symbols $<, \leq, >, \geq$ of sort (`Nat`,`Nat`), and function symbols
  - $+, -, \times : (\mathtt{Nat}, \mathtt{Nat}) \to \mathtt{Nat}$
  - $[] : [\mathtt{Nat}]$ (a constant to name the empty list)
  - $\mathtt{cons}(:)\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

  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


2-sorted
structure
M
$0 \quad 1 \quad 2 \quad \ldots$
(all whole numbers ≥ 0)
Nat
$[] \quad [2] \quad [2,1,3,2] \quad \ldots$
(all lists : [Nat])
[Nat]

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.

$\forall x \forall xs((\sharp(x:xs) = \sharp(xs) + \underline{1}) \wedge ((x:xs)!!\underline{0} = x))$
$\forall x \forall xs \forall n(n < \sharp(xs) \to (x:xs)!!(n + \underline{1}) = xs!!n)$
The '$n < \sharp(xs)$' is necessary. $xs!!n$ could be anything if $n \geq \sharp(xs)$.

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

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

$$\forall xs \forall ys \forall zs\big(xs = ys +\!\!+\, zs \leftrightarrow \sharp(xs) = \sharp(ys) + \sharp(zs) \wedge \forall n(n < \sharp(ys) \to xs!!n = ys!!n) \wedge \forall n(n < \sharp(zs) \to xs!!(n + \sharp(ys)) = zs!!n)\big).$$

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 $M \models \forall x \forall y (pre(x, y) \rightarrow \exists z \; post(x, y, z))$, 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 $\texttt{isin}(x, xs) \leftrightarrow \exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
  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
  $M \models \exists k : \texttt{Nat}(k < \sharp(bs) \wedge bs!!k = a)$ just when a occurs in bs.
  So we have the intended post-condition for `isin`.
- $\forall x \forall xs(\texttt{isin}(x, xs) \leftrightarrow \exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x))$ 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 $\exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
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’

Use *lists of pointers* (below, they are ts, us):

$$\begin{aligned}\texttt{merge}(xs, ys, zs) \;\leftrightarrow\quad & \sharp xs + \sharp ys = \sharp zs \\ & \wedge \exists ts \exists us \big(\sharp ts = \sharp xs \wedge \sharp us = \sharp ys \\ & \qquad \wedge \forall n (n < \sharp zs \to in(n, ts{+}{+}us)) \\ & \qquad \wedge sorted(ts) \wedge sorted(us) \\ & \qquad \wedge \forall n (n < \sharp xs \to zs!!(ts!!n) = xs!!n) \\ & \qquad \wedge \forall n (n < \sharp ys \to zs!!(us!!n) = ys!!n) \big)\end{aligned}$$

ts gets the *positions* in zs that come from xs, and us gets the ones from ys.

For example, merge([7,7], [6,8,5], [6,8,7,5,7]),
and indeed we can take $ts =$ [2, 4]
and $us =$ [0,1, 3].

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 (slide 57) 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,

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

A horse is an animal

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

Deciding if an 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 a particular 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

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

Take any M such that

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

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

To do it, we take arbitrary a in $\text{dom}(M)$ and show $M \models \texttt{lecturer}(a)$.

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

Harder example

Let's show

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

Take any M. Assume that (1) $M \models \forall x(\texttt{horse}(x) \to \texttt{animal}(x))$.
Show
$M \models \forall x[\exists y(\texttt{headof}(x,y) \wedge \texttt{horse}(y)) \to \exists y(\texttt{headof}(x,y) \wedge \texttt{animal}(y))]$.

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

By (2), there is some h in $\mathrm{dom}(M)$ with $M \models \texttt{headof}(b,h) \wedge \texttt{horse}(h)$.
Then $M \models \texttt{headof}(b,h)$ and $M \models \texttt{horse}(h)$.
By (1), $M \models \texttt{horse}(h) \to \texttt{animal}(h)$.
So $M \models \texttt{animal}(h)$.
So $M \models \texttt{headof}(b,h) \wedge \texttt{animal}(h)$, and this h is living proof that $M \models \exists y(\texttt{headof}(b,y) \wedge \texttt{animal}(y))$, 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 variables not occurring free

Suppose that x *doesn't occur free in* A (for example, when x doesn't occur in A at all — see slide 137 for free variables). Then 34–36 below hold. *The restriction is necessary:* see slide 205.

34. $\forall x A$ and $\exists x A$ are logically equivalent to A.
    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. $\exists x(A \wedge B)$ is equivalent to $A \wedge \exists x B$, and
    $\forall x(A \vee B)$ is equivalent to $A \vee \forall x B$.
36. $\forall x(A \to B)$ is equivalent to $A \to \forall x B$, and
    $\exists x(A \to B)$ is equivalent to $A \to \exists x B$.

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

Why is equivalence 35 (1st half, $\exists x(A \wedge B) \equiv A \wedge \exists x B$) true?

Suppose x doesn't occur free in A. Let M, h be arbitrary.

1. Assume $M, h \models \exists x(A \wedge B)$.
   Then obviously $M, h \models \exists x A$ and $M, h \models \exists x B$.
   But by equivalence 34, $\exists x A \equiv A$.
   So $M, h \models A$ and $M, h \models \exists x B$. Therefore, $M, h \models A \wedge \exists x B$.
2. Now assume $M, h \models A \wedge \exists x B$.
   Then $M, h \models A$, and there is some $g =_x h$ with $M, g \models B$.
   Take such a g.
   By equivalence 34, $A \equiv \forall x A$.
   So since $M, h \models A$, we get $M, h \models \forall x A$, and so $M, g \models A$.
   We now have $M, g \models A$ and $M, g \models B$. So $M, g \models A \wedge B$.
   But $g =_x h$. Therefore, $M, h \models \exists x(A \wedge B)$.

So indeed, $\exists x(A \wedge B) \equiv A \wedge \exists x B$.

The other half of equivalence 35, and equivalences 36 and 37, now follow by earlier equivalences.

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 $\forall z \exists v\, \texttt{bought}(z, v)$.

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

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, 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 $\exists x \neg C \equiv \neg \forall x C$
- $A \to \forall x B$ — by propositional equiv. $\neg D \to \neg C \equiv C \to D$
- $\forall x(A \to B)$ — 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
	$\langle$the proof$\rangle$	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 ∃-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, without the box (bad): Suppose $\exists x(P(x) \wedge Q(x))$.
Let c be some object such that $P(c) \wedge Q(c)$.
So $P(c)$ and $Q(c)$. 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'.

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 $\forall$-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: Suppose $P \to \forall x Q(x)$. 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	$\forall y \exists x G(x, y)$	$\forall I(2, 6)$

In English, without the boxes (bad): Suppose $\exists x \forall y G(x, y)$.

Let c be an object 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.

Another wrong ‘proof’ (often written by students in exams)

You may prefer a simpler $\exists E$ rule with no box. But…

$$\begin{array}{|lll|l}
\multicolumn{1}{l}{1} & \forall x \exists y(x < y) & \multicolumn{1}{r}{\text{given}} & \\
\hline
2 & c & \forall I \text{ const} & \\
3 & \exists y(c < y) & \forall E(1) & \\
4 & c < d & \exists E(3) & \leftarrow \text{WRONG} \\
\hline
\multicolumn{1}{l}{5} & \forall x(x < d) & \multicolumn{1}{r}{\forall I(2,4)} & \\
\multicolumn{1}{l}{6} & \exists y \forall x(x < y) & \multicolumn{1}{r}{\exists I(5)} &
\end{array}$$

Again, this ‘proves’ an invalid argument: $\forall x \exists y(x < y) \not\models \exists y \forall x(x < y)$. So $\exists E$ does need its box.

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	$A(t/x) \to B(t/x)$	$\forall E(1)$
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 $\forall\!\rightarrow\!E$ in action

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

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

We used $\forall\!\rightarrow\!E$ 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 $\forall\!\rightarrow\!E$!!

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.

  $$\begin{array}{lll} & \vdots & \text{bla bla bla} \\ 1 & t = t & \text{refl} \end{array}$$

  (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. Compare with the Leibniz principle, slide 203.)

Symmetry of $=$

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

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

Letting A be $x = c$, then line 2 is $A(c/x)$ and line 3 is $A(d/x)$.

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 5.14.
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. . .

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.