Logic for Linguists; Eight Lectures in the Michaelmas Term 2013

Thomas Forster

October 15, 2014

Contents

Modern Symbolic Logic, as developed since the days of Bertrand Russell, and particularly since the "*linguistic turn*" of Frege, is a sibling discipline to Linguistics. Linguists have wider concerns than logicians, but many of the problems to which Linguistics addresses itself are shared with modern Logic, and many of the techniques available to logicians can be useful to students of linguistics.

This being so, it is possible to design a logic course for linguists that is designed not to poach linguistics students but to put them in possession of skills they will find useful.

Among the concerns of logicians that are shared with linguists is a desire to understand the levels of complexity of syntax. Linguists know the Chomsky hierarchy, and logicians [.....]

I envisage three lectures on languages, grammars and machines

The Chomsky hierarchy of languages has nothing to say about semantics: languages can be assigned to a level in the hierarchy without reference to the denotations of items in their lexicon. This of course means that it addresses only a small fragment of the concerns of linguists. Fortunately Modern logic concerns itself with semantics as well, and the early chapters of any logic text are concerned with the elaboration of well-behaved model syntaxes (propositional logic and first-order logic) and a detailed account of how meaning can be breathed into this naked syntax.

I envisage four lectures on the syntax and semantics of propositional and first-order logic.

Indeed the variety of ways in which this can be done is a major preoccupation of modern Symbolic Logic. Some of the techniques developed by Logicians have attracted interest from linguists: one thinks of Montague semantics, λ-caculus,

These last two are an advanced topic, not for fainthearts, and i would envisage three or four lectures on them

One of the syntaxes developed by logicians in the early 20th century is modal logic. It was a while before a satisfactory semantics was developed for it—by Kripke in the 1960s—but this semantics of Kripke's ("possible world semantics") has proved to be beautiful, powerful and illuminating, and no theoretical linguist should be without it.

I envisage three lectures on modal logic and possible world semantics

This year the course is eight lectures not sixteen, so not all of the material prefigured above will actually be covered.

I think all we will get through is

Lectures 1-4
RLFA
Propositional logic; evaluation strategies
Predicate logic, formal semantics for predicate logic

Lectures 5-8
Enhanced syntax: modal logic
Possible world semantics;
λ-calculus

0.0.1 Further Resources

Have a look at
www.dpmms.cam.ac.uk/~tf/chchlectures.pdf
People have mentioned
"Mathematical Methods in Linguistics" by Barbara B.H. Partee, A.G. ter Meulen, R. Wall (Studies in Linguistics and Philosophy) Springer
but i don't know it.
`public_html/cam_only/langs-and-automata/main.html`
I've tutored courses using the material on
`http://www.cl.cam.ac.uk/Teaching/2002/RLFA/reglfa.ps.gz`
This is the course material used at the Computer Laboratory in Cambridge for their 12-lecture course. It looks daunting and mathematical, but it's actually very well designed and thought through, so as long as you read it *carefully* you will be OK. However, it does make use of ϵ transitions (which i `hate`! and do not make much use of here)

Gules, a cross ermine between four lions passant guardant Or, charged with a closed book fesswise of the first, clasped and garnished of the second, the clasps to base.
A description of Oxford University's coat of arms. Presumably in a regular language.

JFLAP is a complete package for doing almost anything relating to fsa, cfg, tm, pda, L-system etc etc. Go to `http://www.jflap.org/`

Stuff to fit in

Linguists often have to make the point to lay people that words do not have magical powers. A word might have strong tapu connotations in one language and not in another. Hence 'reclaiming' words. Logicians make the same point.

Natural languages have both literal and metaphorical meaning; formal languages have *only* literal meaning.

Chapter 1

Introduction

One of the engines for the development of Logic in the last century was the idea that bad (= fallacious, erroneous) Philosophy could be abolished if we cleaned up the language in which we did Philosophy. ("I saw nobody on the road"). Some enthusiasts took this idea far too seriously. Leibnitz; Neurath. Orwell's *1984* does not satirise Stalinism only, it satirises this idea too.

(In this spirit we shouldn't expect the various gadgets in formal logic ($\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$ etc—all of which I will explain to you in due course) to correspond exactly to the gadgets in ordinary language that they replace. Indeed on the Neurath View the lack of correspondence could be taken as evidence both that the programme was needed and that it was succeeding.)

Both Logicians and Linguists study languages. Linguists study *natural* languages, and Logicians study—and design—artificial languages. There isn't a great deal of traffic between the two communities, and this is largely because the difference between artificial and natural languages is so vast that the two programmes experience different weather, and get driven in different directions. However the idea that logicians and linguists share an interest in language is an important one, and it is the chief reason why linguists (of some persuasions at least) can profit by studying formal logic.

There is movement in the other direction as well: at various times logicians have become interested in linguistics—Richard Montague is a famous example and my NZ colleague Max Cresswell another (perhaps less famous, but at least still alive).

I am a student of logic, and to me the tasks confronting the student of natural languages seem mind-bogglingly hard: the kind of questions I ask about artificial languages (and for which I expect answers) are—all of them—far too difficult when asked of natural languages. Since we all of us work equally hard one obvious inference is that linguists have to set their sights much lower. They have to: their subject matter is a lot less tractable. There are questions one can ask about languages that are tractable when asked about artificial languages but are completely intractable when asked about natural languages. Examples here.(Semantics?)

From the point of view of linguists, the assumptions logicians make about the nature of their subject matter come across as comically simplistic.

The languages of formal logic include the programming languages that we find in IT. Some of you may have encountered some of them and perhaps even written programs in them.

From a linguist's perspective, the languages of formal logic look a bit like Basic English: they are fantastically impoverished. This is a key observation: their very impoverishment makes their study much easier. Because the material studied by logicians is so much more tractable, progress has been made with it in a way that has not been made with natural languages. I think this is the chief reason why some linguists think that a bit of input from Logic might be useful: logicians are further down their road than linguists are down theirs, and they might have some useful hints. Not all linguists think this, and even those that do don't necessarily think they need to take any interest in Logic for their particular research areas. However the idea that the shared experiences of logicians and linguists might make logic useful to linguists is the idea behind courses like this. And it is a good idea.

Examples here

Let's look at some of the contrasts:

Natural Languages	**Languages of Formal Logic**
Syntax ascertained by fieldwork	Syntax given by stipulation
Syntax changes over time	Syntax fixed once for all
Linguistics is descriptive	Logic is prescriptive
Problem of individuating languages	No problem individuating languages
All lexical items have pre-assigned meanings	Most lexical items do not have pre-assigned meanings
Some grammatical categories (nouns, adjectives . . .) are open; some (prepositions . . .) closed	All grammatical categories closed
Semantics involves complex layers of feedback and error-correction etc	Straightforward recursive ("compositional") semantics
Alienating adjectives ("*fake* Van Gogh")	No alienating adjectives
First, second and third person pronouns	Third person pronouns only
Semantically closed	Semantically open

semantics for natural language done in real time?

Talk over the slide:

One feature of the typical uses of Logic that sets it off from your concerns is that logics tend to be tailored to specific needs. In contrast Linguists tend to think of natural languages as being things with completely open-ended semantics (and open-ended lexicons for that matter). The languages studied by logicians are very specific and designed for highly specific tasks: for arithmetics of various kinds for example. Formal languages are now used for all sorts of things they weren't used for 50 years ago. *cf* Neurath.

In natural languages almost all words have pre-determined meanings which cannot be altered by the language user. The exceptions are these things called *indexicals* or *token-reflexives*, namely the pronouns, possessive adjectives and a few temporal adverbs ("here", "now") 'this' 'over there', and even their meaning—depending as it does on context—depends on it in a systematic way that does not give the user any genuine freedom. No Humpty Dumpty!

Semantics for artificial languages can be done smoothly, and theoretical considerations can lead to the design of programming languages with nice behaviour. Semantics of natural languages is a problematic business with multiple levels of error-correction, feedback, pragmatics etc. "I love it when you play the piano" might not mean that at all. It might mean "don't annoy the neighbours".

Grice's conditions should be dealt with later

Although a lot of semantics for natural languages is recursive (or "compositional" as the linguists say[1] a significant part of it isn't. There are various ways in which semantics can fail to be compositional. For example, people can use a distinctive vocabulary to announce affiliation to a linguistically defined community–at least in cases where use of that vocabulary was optional, because then it represents a choice made by the speaker and it can convey information to the hearer. People engaged in sports discourse will signal this fact by calling a good player of the game under discussion 'useful'. People who write for the financial press often write 'heading south' for 'decreasing', or "going forward" instead of "in the future" to signal their membership of this community. Elsewhere, 'represents' for 'is' and 'propose' for 'suggest' mark out the speaker as engaged in scientific discourse, as in the following examples:

> Massif-type anorthosites are large igneous complexes of Proterozoic age. They are almost monomineralic, representing [*sic*] vast accumulations of plagioclase ... the 930-Myr-old Rogaland anorthosite province in Southwest Norway represents [*sic*] one of the youngest known expressions of such magmatism. (Nature, **405** p.781.)

Writing 'denotes' for 'is' marks out the user as a mathematician.

> To divide a number a by a number b means to find, if possible, a number x such that $bx = a$. If such a number exists it is denoted [*sic*] by a/b. H. Davenport, [**?**]

[1] Apparently this terminology goes back to [**?**] or do i mean [8]?.

The grammar of an artificial language is fixed and doesn't change over time. (It's also nailed down and blindingly clear beyond quibbles, tho' that is a separate point). In contrast the syntax of natural languages is a matter to be ascertained. The grammar of natural languages changes over time, significant changes occurring in the timescale of a single human lifetime. We are losing negative and interrogative inversion in English; 'like' is replacing 'as if'; 'likely' is starting to be used as an adverb ... and the use of *some* versus *any* is changing even as we speak. "If anybody can do it, Jones can". [The *lexicon* changes too, and so does the semantics for words in the lexicon, but those are separate points]. This gives rise to nontrivial questions about whether the language spoken at time t is the same language as the (slightly different) language spoken at time t'.[2] Specific answers to these questions (Is Ancient Greek the same language as New Testament Greek? Is New Testament Greek the same language as Modern Greek?) are not particularly interesting to linguists (they are uninteresting at any rate in the sense that answers to them will not help linguists do their job better): nevertheless we need to make decisions: if we can't individuate languages we don't know what our subject matter is. "No entity without identity" said Quine (and he was a logician, tho' wearing his philosopher-of-Science hat at the time). Certainly one of the expressive resources a sophisticated speaker of a language has is judicious movement between registers, and this can include exploitation of the different associations in the minds of their hearers of archaic *vs* current *vs* fashionable constructions and lexical items. If we think that giving an account of how this can be done is part of the linguists' job then that means that we are thinking of a language that has lots of overlapping syntaxes and overlapping lexicons rather than just one. more here

(Also the question of individuating languages can have huge political significance!)

In natural languages the meaning of individual lexical items is something to be ascertained, not stipulated by fiat. You do fieldwork to find out what a word means.

That is to say: in a natural language the meaning of the words is part of the language. For logicians this is not true: a language for a logician—in most cases—is a naked piece of syntax, waiting to be clothed in meaning. It is true that many formal languages have what the computer scientists call **reserved words**, which are signs ('=' is one) that are only ever allowed to mean one thing, but in natural languages almost all lexical items are reserved in this sense of having pre-assigned meaning, the exceptions being indexicals, whose denotation is determined by the speaker in real-time.

In artificial languages, in contrast, the meaning of particular lexical items can indeed be stipulated by fiat. Indeed the idea that syntax had a life of its own, and that it existed independently of our need and desire to assign meaning to it (or to invent it in order to bear meaning) was one of the most important insights driving the study of Logic from the very outset of its renaissance in the last century. You might think that the symbol '$\leq$' means "less than or equal

[2] Ask Wikipædia about *Theseus' ship*.

to" as applied to numbers, or that the symbol '=' means "equal to", and you might reckon that you know what '+' and '×' mean—and you would be right, because those symbols were brought in precisely to bear those meanings, but they could perfectly well have borne other meanings instead: there is nothing about those symbols in themselves that tell you that they have to bear those interpretations. The great insight of C20th logic was that in order to understand how symbols can bear meaning *at all* it is important first of all to study them *entirely stripped* of their meaning. They have to be—as it were—*born again*!

To summarise:

One reason for linguists to study formal logic is because it is a very very simple instance of their general task, and it's tractable. It looks hard only beco's one can aspire to do it properly!

Another reason is that some of the more complicated logics resemble natural languages sufficently for them to be invoked as part of a theory of natural language.

1.1 Some basics

type-token, use-mention, language-metalanguage

de re and *de dicto*

Talk over this. Details in chchlectures.pdf

I am not going to expose you to any new Mathematics in this chapter, but that doesn't mean you should skip it. Read this first, to prepare you for what is to come. There are a number of things I want to warn you about. I am a great believer in *Naming the Devil*: philosophers have argued for a long time about the relation between thought and language but we do all agree that many things become easier to see and recognise once you have a name for them. In the next few sections I shall be introducing you to some terminology. You won't have to prove anything using it, but it will help you once you come to proving other things.

1.2 Philosophical Introduction: the pedagogical difficulties

It might seem odd to kick off a folder of course materials on discrete structures with a section that has a title like this, but there is a reason. Some proofs just are hard, and people experience difficulty accordingly. But there are bottlenecks where people experience difficulties with the underlying concepts, and particularly with the notation.

The hard part of doing discrete maths is not learning the proofs of the theorems. By and large the proofs are not particularly difficult at this level—though they can appear daunting. The hard part is making a certain kind

of mental jump. Once you have made this jump, everything is easy. Let me explain.

Mathematicians often complain that lay people think that mathematics is about numbers. It isn't, and they are right to complain. Not just because it's a mistake, but because it's a mistake that throws people off the scent. Mathematics is a process of formalisation and abstraction that can be applied to all sorts of things, not just numbers. It just so happens that the only bits of mathematics that the average lay person encounters is mathematics as applied to *numerosity*, which is where numbers come from. In fact we can apply mathematical methods to all sorts of other ideas. (Geometry, for example).

Variables and Things

In applying mathematical methods to a topic we find that we take a number of steps. One of them is summarised in a famous remark of the twentieth century philosopher W.V.Quine: "To be is to be the value of a variable". You are probably quite happy if I say "Let x be a number between 1 and 1000" or (if you are old enough to have done geometry at school) "Let ABC be a triangle". You are almost certainly not happy if I say "Let R be a binary relation on a set X". Why is this? It is because numbers (and perhaps triangles) are mathematical objects in your way of thinking, whereas relations aren't. And what has this got to do with being the value of a variable? Quine's criterion for a species of object to be a *mathematical* object (in the way that numbers (or perhaps triangles are) is that variables can range over mathematical objects. From your point of view, the utility of the observations of Quine's is that it enables you to tell which things you are comfortable thinking of mathematically.

Computer Scientists in their slang make the distinction (from the point of view of a programming language)— between **first class objects** and the rest. First class objects are the kinds of things that the variables of the language can take as values. Typically, for a programming language, numbers will be first-class objects but operations on numbers will not.

This distinction between first-class objects and the rest is echoed in ordinary language (well, in all the ordinary languages known to me, at least) by the difference between nouns and verbs.

Let us take a live example, one that bothers many beginners in discrete mathematics. Relations are not mathematical objects for most people. ("Let R be a binary relation on $\mathbb{N}\ldots$"!) In consequence many people are not happy about being asked to perform operations on relations. The problem is not that they are unacquainted with the fact that—for example—the uncle-of relation is the composition of the brother-of relation with the parent-of relation. This is, after all, something you can explain easily to any foreigner who asks you what the word 'uncle' means! The problem is that they don't know that this fact is a fact about composition of relations. This is because they don't think of relations as being the kind of things you perform operations on, and that in turn is because they don't think of relations as **things** at all!

You are quite happy applying operations to numbers. The conceptual leap

you have to make here is to be willing to apply operations to relations. Although thinking of them as things (rather than as relations between things) and then thinking of them as the substrates of operations are two steps rather than one, it's probably best to think of them as two parts of a single move.

But relations are only one example of entities that you are now going to have to think of mathematically. Others are sets, functions and graphs.

Null objects

Another sign that a species of object (number, set, line ...) has become a mathematical object for you is when you are happy about degenerate or *null* objects of that species. You may remember being told in school that the discovery that zero was a number was a very important one. An analogous discovery you will be making here is that the empty set is a set. (Perhaps this is the same discovery, since numbers like 1,2,3, ... (though not 1.5, 5/3, π ...) are answers to questions about how many elements there are in a set. "0" is the answer to "How many things are there in the empty set?".) If you think that the empty set "isn't there" it's because you don't think that sets have any existence beyond the existence of their members. Contrast this with the relaxed feeling you have about an empty folder or file in a directory on your computer. Files and folders on your computer are things that, for you, are unproblematic in a way that makes it possible for you to think of empty ones. To that extent you are thinking of them as mathematical objects: the ability to be relaxed about the empty set is one of the things you will acquire when you start thinking of sets as mathematical objects. (The grin remains after the cat has gone.)

We build formulæ by taking conjunctions or disjunctions of collections of formulæ. What is the conjunction of the empty set of formulæ? The disjunction of the empty set of formulæ? Never mind about the answer to this *just* yet (though we will soon); for the moment I am trying to impress you with (i) the novel idea that the question is a sensible one and (ii) that accepting the fact that it is a sensible question is part of thinking of sets as mathematical objects, which in turn is part of doing discrete maths.

Integrate this last para into the preceding para

(My Ph.D. thesis has the shortest title on record: "N.F." A title is a string of characters. So the shortest possible title is the empty string of characters. Note that having the empty string as your title is not the same as having no title!) TTBA "untitled".)

Envoi

I've inflicted on you this brief digression on Philosophy and Foundations beco's every student, in mastering the skills and ideas of Computer Science, has to go through a hugely speeded-up version of the journey that Mathematics and Philosophy went through in dreaming up these objects in the first place.

1.3 The Type-Token distinction

The terminology 'type-token' is due to the remarkable nineteenth century American philosopher Charles Sanders Peirce. (You may have heard the tautology $((A \to B) \to A) \to A$ referred to as *Peirce's Law*). The two ideas of token and type are connected by the relation "is an instance of". Tokens are instances of types.

It's the distinction we reach for in situations like the following

- (i) "I wrote a *book* last year"
  (ii) "I bought two **books** today"

  In (ii) the two things I bought were physical objects, but the thing I wrote in (i) was an abstract entity. What I wrote was a *type*. The things I bought today with which I shall curl up tonight are *tokens*. This important distinction is missable because we typically use the same word for both the type and the token.

- A best seller is a book large numbers of whose *tokens* have been sold. There is a certain amount of puzzlement in copyright law about ownership of tokens of a work versus ownership of the type. James Hewitt owns the copyright in Diana's letters to him but not the letters themselves. (Or is it the other way round? Either way the man's a bounder.)

- I remember being very puzzled when I was first told about printing. I was told that each piece of type could only be used once. Once for each book, in the sense of once for each print run Not once for each *copy* of a book. The copies from any one print run are all tokens of a type.

- I read somewhere that " ... next to Mary Woollstonecroft was buried Shelley's heart, *wrapped in one of his poems*." To be a bit more precise, it was wrapped in a *token* of one of his poems.

- You have to write an essay of 5000 words. That is 5000 word tokens. On the other hand, there are 5000 words used in this course material that come from latin. Those are word types.

- Grelling's paradox: a **heterological** word is one that is not true of itself. 'long' is heterological: it is not a *long* word. 'English' is not heterological but *homological*, for it is an English word. Notice that it is word *types* not word *tokens* that are heterological (or homological!) It doesn't make any sense to ask whether or not 'italicised' is heterological. Only word *tokens* can be italicised!

- What is the difference between "unreadable" and "illegible"? A book (type) is unreadable if it so badly written that one cannot force oneself to read it. A book (token) is illegible if it is so defaced or damaged that one cannot decypher the (tokens of) words on the page.

- We must not forget the difference between a program (type) and the tokens of it that run on various machines.
- Genes try to maximise the number of tokens of themselves in circulation. We attribute the intention to the gene *type* because it is not the action of any *one* token that invites this mentalistic metaphor, but the action of them all together. However it is the number of *tokens* that the type appears to be trying to maximise.

1.4 Copies

Buddhas

It is told that the Buddha could perform miracles. But—like Jesus—he felt they were vulgar and ostentatious, and they displeased him.

> A merchant in a city of India carves a piece of sandalwood into a bowl. He places it at the top of some bamboo stalks which are high and very slippery, and declares that he will give the bowl to anyone who can get it down. Some heretical teachers try, but in vain. They attempt to bribe the merchant to say they had succeeded. The merchant refuses, and a minor disciple of the Buddha arrives. (His name is not mentioned except in this connection). The disciple rises through the air, flies six times round the bowl, then picks it up and delivers it to the merchant. When the Buddha hears the story he expels the disciple from the order for his frivolity.

But that didn't stop him from performing them himself when forced into a corner. In *Siete Noches* (from which the above paragraph is taken) J. L. Borges procedes to tell the following story, of a miracle of *courtesy*. The Buddha has to cross a desert at noon. The Gods, from their thirty-three heavans, each send him down a parasol. The Buddha does not want to slight any of the Gods, so he turns himself into thirty-three Buddhas. Each God sees a Buddha protected by a parasol he sent.[3]

Apparently he did this routinely whenever he was visiting a city with several gates, at each of which people would be waiting to greet him. He would make as many copies of himself as necessary to be able to appear at all the gates simultaneously, and thereby not disappoint anyone.

1.4.1 Minis

Q: How many elephants can you fit in a mini?
A: Four: two in the front and two in the back.

Q: How many giraffes can you fit in a mini?
A: None: it's full of elephants.

[3] As is usual with Borges, one does not know whether he has a source for this story in the literature, or whether he made it up. And—again, as usual—it doesn't matter.

Q: How can you tell when there are elephants in the fridge?
A: Footprints in the butter.

Q: How can you tell when there are *two* elephants in the fridge?
A: You can hear them giggling when the light goes out.

Q: How can you tell when there are *three* elephants in the fridge?
A: You have difficulty closing the fridge door.

Q: How can you tell when there are *four* elephants in the fridge?
A: There's a mini parked outside.

Sets

If A is a set with three members and B is a set with four members, how many ordered pairs can you make whose first component is in A and whose second component is in B?

Weeeeell . . . you pick up a member of A and you pair it with a member of B . . . that leaves two things in A so you can do it again . . . . The answer must be three!

Wrong! Once you have picked up a member of A and put it into an ordered pair—it's still there!

One would tend not to use the word *token* in this connection. One would be more likely to use a word like *copy*. One makes lots of copies of the members of A. Just as the Buddha made lots of copies of himself rather than lots of *tokens* of himself. I suppose you could say that the various tokens of a type are copies of each other.

It is possible to do a lot of rigorous analysis of this distinction, and a lot of refinements suggest themselves. However, in the culture into which you are moving the distinction is a piece of background slang useful for keeping your thoughts on an even keel, rather than something central you have to get absolutely straight. In particular we will need it when making sense of ideas like *disjoint union*.

1.5 The Use-Mention Distinction

We must distinguish words from the things they name: the word 'butterfly' is not a butterfly. The distinction between the word and the insect is known as the "use-mention" distinction. The word 'butterfly' has nine letters and no wings; a butterfly has two wings and no letters. The last sentence *uses* the word 'butterfly' and the one before that *mentions* it. Hence the expression 'use-mention distinction'. (It is a bit more difficult to illustrate the difference between using and mentioning butterflies!)

As so often the standard example is from *Alice through the looking-glass*.

> [...] The name of the song is called 'Haddock's eyes'".
>
> "Oh, that's the name of the song is it", said Alice, trying to feel interested.
>
> "No, you don't understand," the Knight said, looking a little vexed. "That's what the name is *called*. The name really is *'The agèd, agèd man'*."
>
> "Then I ought to have said, 'That's what the *song* is called'?" Alice corrected herself.
>
> "No you oughtn't: that's quite another thing! The *song* is called '*Ways and means*', but that's only what it is *called*, you know!"
>
> "Well, what *is* the song, then?" said Alice, who was by this time completely bewildered.
>
> "I was coming to that," the Knight said. "The song really is *'A-sitting on a Gate'* and the tune's my own invention".

The situation is somewhat complicated by the dual use of single quotation marks. They are used both as a variant of ordinary double quotation marks for speech-within-speech (to improve legibility)—as in "Then I ought to have said, 'That's what the *song* is called'?"—and also to make names of words—'butterfly'. Even so, it does seem clear that the White Knight has got it wrong. At the very least if the name of the song is "An agèd agèd man" as he says then clearly Alice was right to say that was what the song was called, It might have more names than just that one—such as 'Ways and means'—but that was no reason for him to tell her she had got it wrong. And again, if his last utterance is to be true he should leave the single quotation marks off the title, or failing that (as Martin Gardner points out in *The Annotated Alice*) burst into song. These mistakes must be mistakes of the White Knight not Lewis Carroll, but it is hard to see what purpose these errors serve, beyond multiplying in Alice's head the sense of nightmare and confusion that she already feels ... Perhaps he had the reader in his sights too.

In the following example Ramsey uses the use-mention distinction to generate something very close to paradox: the child's last utterance is an example of what used to be called a "self-refuting" utterance: whenever this utterance is made, it is not expressing a truth.

PARENT: Say 'breakfast'.

CHILD: Can't.

PARENT: What can't you say?

CHILD: Can't say 'breakfast'.

Rather like the paradoxes, the use-mention distinction can end up being a source of humour. A nice illustration is the joke about the compartment in the commuter train, where the passengers have travelled together so often that they have all told all the jokes they know, and have been reduced to numbering the jokes and reciting the numbers instead. In most versions of this story, an outsider arrives and attempts to join in the fun by announcing "*Fifty-six!*" which is met with a leaden silence and he is told "It's not the joke, it's the way you tell it". In another version he then tries "*Forty-two!*" and the train is convulsed with laughter. Apparently they hadn't heard that one before.

We make a fuss of this distinction because we should always be clear about the difference between a thing and its representation. Thus, for example, we distinguish between numerals and the numbers that they represent. If we write numbers in various bases (Hex, binary, octal ...) the numbers stay the same, but the numerals we associate with each number change. Thus the numerals '`XI`', 'B', '11', '13' '1011' all represent the same number.

Exercise 1. *What is that number, and under which systems do those numerals represent it?*

1.6 Semantic optimisation and the principle of charity

When a politician says "We have found evidence of weapons-of-mass-destruction programme-related activities", you immediately infer that that have *not* found weapons of mass destruction (whatever they are). Why do you draw this inference?

Well, it's *so* much easier to say "We have found weapons of mass destruction" than it is to say "We have found evidence of weapons-of-mass-destruction-related programme-related activities" that the only conceivable reason for the politician to say the second is that he won't be able to get away with asserting the first. After all, why say something longer and less informative when you can say something shorter and more informative? We here, doing a course in discrete mathematics, will tend to see this as a principle about maximising the amount of information you convey while minimising the amount of energy you expend in conveying it. We will be doing a teeny weeny bit of optimisation theory (in chapter **??**) but only a very teeny-weeny bit (just enough for you to develop a taste for it) and certainly not enough to come to grips with all the complexities of human communication. But it's not a bad idea to think of ourselves as generally trying to minimise the effort involved in conveying whatever information it is that we want to convey.

Quine used the phrase "The Principle of Charity" for the assumption one makes that the people one is listening to are trying to minimise effort in this way. It's a useful principle, in that by charitably assuming that they are not being unneccessarily verbose it enables one to squeeze a lot more information out of one's interlocutors' utterances than one otherwise might, but it's dangerous. Let's look at this more closely.

Suppose I hear you say

> We have found evidence of weapons-of-mass-destruction programme-related activities. (1)

Now you *could* have said

> We have found weapons of mass destruction. (2)

Naturally I will of course put two and two together and infer that you were not in a position to say (2), and therefore that you have *not* found weapons of mass destruction. However, you should notice that (1) emphatically does *not* imply that

> We have *not* found weapons of mass destruction. (3)

After all, had you been lucky enough to have found weapons of mass destruction then you have most assuredly found evidence of weapons-of-mass-destruction programme-related activities: the best possible evidence indeed. So what is going on?

What's going on is that (1) does not imply (3), but that (4) does!

> We have chosen to say "We have found evidence of weapons-of-mass-destruction programme-related activities" instead of "We have found weapons of mass destruction ". (4)

Notice that 1 and 4 are not the same!

Now the detailed ways in which this optimisation principle is applied in ordinary speech do not concern us here—beyond one very simple consideration. I want you to understand this optimisation palaver well enough to know when you are tempted to apply it, and to lay off. The formal languages we use in mathematics and computer science are languages of the sort where this kind of subtle reverse-engineering of interlocutors' intentions is a hindrance not a help. Everything is to be taken literally.

1.6.1 Overloading

Not quite the same as ambiguity.

+ on reals and on natural numbers are different operations. They look sort-of similar, because they obey some of the same rules, so there is a temptation to think are the same thing—and certainly to use the same symbol for them. A symbol used in this way is said to be **overloaded**, and it's not quite the

same as the symbol being **ambiguous** because there is a connection of meaning between the two uses which there might not be when a symbol is being used ambiguously.

Overloading is a way of being thrifty in our use of notation. The drawback is that it gets us into the habit of expecting ambiguities even in settings where there is none. This leads us to...

1.7 Fault-tolerant pattern-matching

Explain what it is!

Fault-tolerant pattern matching is very useful in everyday life but absolutely no use at all in the lower reaches of computer science. Too easily fault-tolerant pattern matching can turn into overenthusiastic pattern matching—otherwise known as *syncretism*: the error of making spurious connections between ideas. A rather alarming finding in the early days of experiments on sensory deprivation was that people who are put in sensory deprivation tanks start hallucinating: their receptors expect to be getting stimuli, and when they don't, they wind up their sensitivity until they start getting positives. Since they are in a sensory deprivation chamber, those positives are one and all spurious.

1.7.1 Overinterpretation

My brother-in-law once heard someone on the bus say "My mood swings keep changing." He—like you or I on hearing the story—knew at once that what the speaker was trying to say was that they suffer from mood swings!

Reinterpreting silly utterances like this so that they make sense is something that we are incredibly good at. And by 'incredibly good' I mean that this is one of the things we can do *vastly* better that computers do (in contrast to the things like multiplying 100-digit numbers in our head, which computers can do very much better than we can). In fact we are *so* good at it that nobody has yet quite worked out how we do it, though there is a vast literature on it, falling under the heading of what people in linguistics call "pragmatics". Interesting though that literature is I am mentioning it here only to draw your attention to the fact that learning to do this sort of thing *better* is precisely what we are *not* going to do. I want you to recognise this skill, and know when you are using it, in order not to use it *at all!*

Why on earth might we *not* want to use it?? Well, one of the differences between the use of symbols in mathematics (eg in programming languages) and the use of symbols in everyday language is that in maths we use symbols formally and rigidly and we suffer for it if we don't. If you write a bit of code with a grammatical error in it the O/S will reject it: "Go away and try again." One of the reasons why we design mathematical language (and programming languages) in this po-faced fault-intolerant way is that that is the easiest way to do it. Difficult though it is to switch off the error-correcting pattern-matching software that we have in our heads, it is much more difficult still to discover how it works and thereby emulate it on a machine—which is what we would have to

do if we were to have a mathematical or programming language that is fault-tolerant and yet completely unambiguous. In fact this enterprise is generally regarded as so difficult as to be not worth even attempting. There may even be some deep philosophical reason why it is impossible even in principle: I don't know.

Switching off our fault-tolerant pattern-matching is difficult for a variety of reasons. Since it comes naturally to us, and we expend no effort in doing it, it requires a fair amount of self-awareness even to realise that we *are* doing it. Another reason is that one feels that to refrain from sympathetically reinterpreting what we find being said to us or displayed to us is unwelcoming, insensitive, autistic and somehow not fully human. Be that as it may, you have to switch all this stuff off all the same. Tough!

So we all need some help in realising that we do it. I've collected in the appendix to this chapter a few examples that have come my way. I'm hoping that you might find them instructive.

1.7.2 Scope ambiguities

Years ago when I was about ten a friend of my parents produced a German quotation, and got it wrong. (I was a horrid child, and I blush to recall the episode). I corrected him, and he snapped "All right, everybody isn't the son of a German Professor") (My father was Professor of German at University College London at the time). Quick as a flash I replied "What you mean is 'Not everybody is the son of a professor of German".'.

I was quite right. (Let's overlook the German professor/professor of German bit). He said that Everybody Isn't the son of a professor of German. That's not true. Plenty of people are; I am, for one. What he meant was "Not everybody is ...". It's the difference between "$(\forall x)(\neg \ldots)$" and "$\neg(\forall x)(\ldots)$"—the difference is real, and it matters.

The difference is called a matter of **scope**. 'Scope'? The point is that in "$(\forall x)(\neg \ldots)$" the "scope" of the '$\forall x$' is the whole formula whereas in the '$\neg(\forall x)(\ldots)$ it isn't.

For you, the moral of this story is that you have to identify with the annoying ten-year old rather than with the adult that he annoyed: it's the annoying 10-year-ld that is your rôle model here!

It is a curious fact that humans using ordinary language can be very casual about getting the bits of the sentence they are constructing in the right order so that each bit has the right scope. We often say things that we don't literally mean. ("Everybody isn't the son of ..." when we mean "Not everybody is ...") On the receiving end, when trying to read things like $(\forall x)(\exists y)(x$ loves $y)$ and $(\exists y)(\forall x)(x$ loves $y)$, people often get into tangles because they try to resolve their uncertainty about the scope of the quantifiers by looking at the overall meaning of the sentence rather than by just checking to see which order they are in!

Exercise 2. *Match up the formulæ on the left with their English equivalents*

on the right.

(i) $(\forall x)(\exists y)(x \text{ loves } y)$	*(a) Everyone loves someone*
(ii) $(\forall y)(\exists x)(x \text{ loves } y)$	*(b) There is someone everyone loves*
(iii) $(\exists y)(\forall x)(x \text{ loves } y)$	*(c) There is someone that loves everyone*
(iv) $(\exists x)(\forall y)(x \text{ loves } y)$	*(d) Everyone is loved by someone*

In the real world people make mistakes and say things that aren't exactly what they mean ("Everybody isn't the son of a german professor") so listeners have to get quite good at spotting these errors and correcting them. So good, in fact, that we don't notice we do it. In mathematics (and, in particular, with programming languages) errors of the kind we are so skillful at correcting are never tolerated, so there is no need to have lots of clever software to detect and correct them. The fault-tolerant pattern-matching skill is no longer an asset and its deployment merely distracts us from the task of reading the formula in question. The result is that when we encounter a formula with nasty alternations of quantifiers and tricky scoping, such as the Pumping Lemma from Languages-and-Automata, we think "This looks ghastly; it can't be what he means. Life isn't that bad: let's reach for the rescoping software" whereas what we should be doing is just trying to read it as it is. Sadly life—or at any rate the Pumping Lemma—really is that bad! The Pumping Lemma is less than completely straightforward to read even with the best of intentions (it has more quantifiers in it than we are used to) and attempting to read it without first switching off your rescoping software is a sure recipe for disaster.

1.8 Intension and Extension

The intension-extension distinction is an informal device but it is a standard one which we will need at several places. We speak of **functions-in-intension** and **functions-in-extension** and in general of **relations-in-intension** and **relations-in-extension**. There are also 'intensions' and 'extensions' as nouns in their own right.

Consider two properties of being *human* and being a *featherless biped*–a creature with two legs and no feathers. There is a perfectly good sense in which these concepts are the same (or can be taken to be, for the sake of argument: one can tell that this illustration dates from before the time when the West had encountered Australia with its kangaroos!), but there is another perfectly good sense in which they are different. We name these two senses by saying that 'human' and 'featherless biped' are the same property in extension but are different properties in intension.

A more modern and more topical illustration is as follows. A piece of code that needs to call another function can do it in either of two ways. If the function being called is going to be called often, on a restricted range of arguments, and is hard to compute, then the obvious thing to do is compute the set of values in advance and store them in a look-up table in line in the code. On the other

hand if the function to be called is not going to be called very often, and the set of arguments on which it is to be called cannot be determined in advance, and if there is an easy algorithm available to compute it, then the obvious strategy is to write code for that algorithm and call it when needed. In the first case the embedded subordinate function is represented as a function-in-extension, and in the second case as a function-in-intension.

Functions-in-extension are sometimes called the **graph**s of the corresponding functions-in-intension: the graph of a function f is $\{\langle x, y\rangle : x = f(y)\}$. One cannot begin to answer exercise **??.??** unless one realises that the question must be, "How many binary relations-*in-extension* are there on a set with n elements?" (There is no answer to "how many binary relations-in-intension ...")

It turns up nowadays in the connection with the idea of evaluation. In recent times there has been increasingly the idea that intensions are the sort of things one *evaluates* and that the things to which they evaluate are extensions. Propositions evaluate to truth-values. Truth-values (`true` and `false`) are propositions-in-extension.

We do need both. Some operations are more easily understood on relations-in-intension than relations-in-extension (composition for example) Ditto ancestral

1.9 Appendix: Howlers of overinterpretation

$\lambda x.\lambda y.2$

What is $\lambda x.\lambda y.2$? Overinterpretation will probably make you think this should simplify to 2; it doesn't. It really just is the function whose constant value is the function whose constant value is 2. What happens if you apply this function to 3? It's actually idiotically simple. It's the result of applying to 3 the function whose constant value is the function whose constant value is 2. And please do not make the mistake of thinking that the function with constant value 2 (the one that returns 2 whatever it is given as argument) is the same as the number 2. There is an important difference between a pint of Guinness and the magic Guinness glass (given by the Leprichaun to the Irishman who released him from a bottle wherein he'd been trapped since the Bronze Age) that automatically refills itself with Guinness every time anyone drinks from it. This difference is not *quite* the same as the difference between 2 and $\lambda x.2$ but it might help to remind you of that difference.

The square of a relation

What is the square of the $<$ relation on $\mathbb{N}$? Well, one thing it ain't is $\{\langle x, y\rangle : x^2 < y^2\}$, whihc is the answer one of my students gave once. You get into this mess if you forget what the square of a relation is, and fault-tolerantly match to something you do know, such as squaring of numbers.

The power set of the empty set

The empty set is the set with no elements. We write it as '$\emptyset$' or occasionally as '$\{\}$'. I say *the* empty set because there is in fact only *one* empty set. There is the criterion of identity for sets: two sets are the same set if they have the same members. (Not true for multisegts or lists for example). So what is the power set of the empty set? Let's take this question slowly, in stages, and answer it carefully by reading *entirely literally* all the definitions we need to refer to. Recall that the power set, $\mathcal{P}(X)$ of a set X is the set of all subsets of X. Y is a subset of X (written '$Y \subseteq X$') if everything that is in Y is in X. So in order to devise the power set of the empty set we are going to open up a bag and put into it everything we can find that is a subset of the empty set.

Q: So what are the subsets of $\emptyset$?

Were you about to say "none"? If you were about to say that, then I want to smack your wrist. *Obviously* the empty set is a subset of itself–just look at the definition!! So why did you leave it out? Because it didn't sound like a sensible answer. Why didn't it sound like a sensible answer? Because somewhere in your mind is the unspoken assumption that if I ask you for the something-or-others of X, you should come up with something *new*. "He can't be wanting me to mention X co's he already knows *that*." But recall the definition of 'subset of': $x \subseteq y$ holds precisely when everything that is in x is also in y. So every set is (trivially) a subset of itself. So in particular the empty set is a subset of itself.

Very well, we are agreed there is one subset of $\emptyset$, namely $\emptyset$ itself. Also, by extensionality, it is the *only* subset. (That bit is unproblematic)

Q: So what is the power set of the emptyset?

Were you about to say $\emptyset$? If so I'm going to smack your wrist yet *again*. You were probably thinking something like "...the empty set is $\{\}$ so the set containing the empty set must be $\{\{\}\}$ and the curly brackets can't be doing anything so that must be the same as $\{\}$". This mistake arises from your overinterpretation of data that you feel to be suspect, namely $\{\{\}\}$. But if you are careful, you will see that it isn't suspect at all, and it won't be hard to explain this to yourself. The set that contains all the subsets of the empty set has as one of its members (its sole member as it happens) the empty set. So it isn't empty! So it *can't* be the same as the empty set, by extensionality!

So the mistake of thinking that $\mathcal{P}(\emptyset) = \emptyset$ arises from thinking that the expression '$\{\emptyset\}$' is a bit of suspect data to which you need to apply your fault-tolerant pattern-matching software. The idea that '$\{\emptyset\}$' is a bit of suspect data is a seperate mistake that deserves an analysis of its own. The trap is the trap of thinking that because the empty set *has nothing inside it* then it *actually isn't there at all*. Why do you think this? Because you are not happy with the idea of a set being *empty*. But—I put it to you—there is nothing any odder about the idea of an empty *set* than there is about the idea of an empty *folder*, or an empty *file*. It's no odder than the idea that zero is an integer. "How

many strawberries have you got in that punnet?" "None, sadly!". Once you are thinking about sets properly this difficulty goes away.

The power set of $\{1, 2, 3\}$

I once had a student who, when asked in an exam to write down all the subsets of $\{1, 2, 3\}$, supplied only $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$. My guess is that

- She omitted $\{1, 2, 3\}$ on the grounds that 'subset' probably meant 'proper subset'. We saw this mistake earlier;
- She omitted the singleton subsets because she was probably thinking something like "why would anyone want to write down '{1}'? That's silly. Anyone writing that down probably really means '1', and that isn't a set, so I can leave it out".
- She left out the empty set because she didn't think it was there. We've seen this too.

Affirming the consequent

Years ago I was teaching elementary Logic to a class of first-year law students, and I showed them this syllogism:

> "If George is guilty he'll be reluctant to answer questions; George is reluctant to answer questions. Therefore George is guilty."

Then I asked them: Is this argument valid? A lot of them said 'yes'.

We all know that an obvious reason—the first reason that comes to mind—why someone might be reluctant to answer questions is that they might have something to hide. And that something might be their guilt. So if they are reluctant to answer questions you become suspicious at once. Things are definitely not looking good for George. Is he guilty? Yeah—string him up!

But what has this got to do with the question my first-years were actually being asked? Nothing whatever. They were given a premiss of the form $P \to Q$, and another premiss Q. Can one deduce P from this? Clearly not. Thinking that you can is the fallacy of *affirming the consequent*.

There are various subtle reasons for us to commit this fallacy, and we haven't got space to discuss them here. The question before the students in this case was not: do the premisses (in conjunction with background information) give evidence for the conclusion? The question is whether or not the inference from the premisses to the conclusion is logically valid. And that it clearly isn't. The mistake my students were making was in misreading the question, and specifically in misreading it as a question to which their usual fault-tolerant pattern-matching software would give them a swift answer.

$A \times \emptyset$

What is $A \times \emptyset$? Do the sensible thing: try to form ordered pairs whose first components are in A and whose second components are in $\emptyset$. So you pick up a member of A to put into your *chase* (that's the thing printers hold in their hand and put bits of type in when setting something up in type) and then you reach for a member of $\emptyset$. Ouch!

It is possible to get into a tangle by trying to find the correct description of the set of *defective* ordered pairs: things that ought to be ordered pairs but are defective in that altho' they have a first component in A they somehow lack a second component. $A \times \emptyset$ is the obvious place to put these discards and offcuts. You might well think that such a discard-or-offcut is just a member of A, and conclude that $A \times \emptyset$ is A.

There are no ordered pairs whose first component is in A and whose second component is in $\emptyset$: any attempt to assemble one fails. A long way of saying this is to say that the set of all such pairs—namely $A \times \emptyset$—is $\emptyset$, and that of course is the correct answer.

The discards and offcuts never actually get as far as existing (they are the discards and offcuts—after all—from the things that are being brought into existence) so you are not obliged to find anywhere to put them. So you shouldn't be surprised that obvious place to think to put them—namely $A \times \emptyset$—should turn out to be empty!

Is the Identity Relation a Function?

"Well, functions give you back values when you feed them arguments. The identity relation obviously doesn't do anything so it can't be a function!"

Is that what you were thinking? Shame on you! *Of course* the identity relation is a function. Look at the definition of a function: each argument is related to one and only one value. The identity relation relates each thing to one and only one thing, namely itself!

1.9.1 Is the Identity Relation a Partial Order?

Lots of people say: 'no'! They think that because it doesn't order things then it can't be a partial order. However, if you read the definitions you will see that it is.

Can a Relation be both Symmetrical and Antisymmetrical?

"Well, obviously not, beco's the words sound as if the conditions are mutually contradictory. On the other hand, why would they be asking me if it was that easy? Er ... is this a multiple choice question ...? Will I be penalised for guessing? ... can I ask a friend?"

How about just answering the question? It might be quicker!

Suppose R is both symmetrical and antisymmetrical. Then, whenever x is related to y by R, y is related to x by R—by symmetry. But if x is related to y and y to x, then $x = y$ by antisymmetry. So if x is related to y, $x = y$. So R must be a subset of the identity relation. So perhaps the identity relation itself might be both symmetrical and antisymmetrical, and so indeed it turns out.

And that is about all you can say!

Is the empty relation transitive?

I have had students get in a tangle over the question whether or not the empty relation is transitive. Or over the question of whether or not the relation $\{\langle 1, 2\rangle\}$ is transitive. It's the same tangle. The tangle is this: for a relation to be transitive, it's necessary for it to contain the ordered pair $\langle x, z\rangle$ whenever it contains the ordered pairs $\langle x, y\rangle$ and $\langle y, z\rangle$. It's mechanical to check that the empty relation and the relation $\{\langle 1, 2\rangle\}$ are transitive. "But what if it doesn't contain any ordered pair?" they wail. Or "What if it contains an ordered pair $\langle x, y\rangle$ but no pair $\langle y, z\rangle$?" This is overinterpretation. Nobody said it in order to be transitive it had to contain an ordered pair $\langle x, y\rangle$ or had to contain pairs $\langle x, y\rangle$ and $\langle y, z\rangle$. Merely that **if** it contained pairs $\langle x, y\rangle$ and $\langle y, z\rangle$ **then** it would have to contain the ordered pair $\langle x, z\rangle$. If it doesn't satisfy the antecedent of the conditional then the condition is trivially satisfied. Ian Stewart's example *If you pick a guinea pig up by its tail its eyes fall out*—is true. Conditionals whose antecedents are false are vacuously true: in the nature of things these conditionals are unlikely to be *useful* but that doesn't make the *false*. Overinterpretation.

You only get into a tangle if you try to be too clever, and overinterpret.

Coda

If you get these questions wrong it's almost certainly not beco's you are ignorant or stupid, but beco's you are approaching them the wrong way.

Chapter 2

Languages and Automata

2.0.2 New stuff to fit in

Where do we talk about actual live regular languages? Roman numerals? phonology rules

Ambiguous parses; decidability of language equivalence

No ambiguous parses in regular languages. Why not?

Phonetic rules say things like: if the last two phonemes were x followed by y you cannot then have a z next; you cannot begin a word with an x; you cannot end it with a y; everything else is OK. Not hard to show that if all the phonological rules look like that then the set of permitted strings forms a regular language. Suppose your alphabet is Σ and the number of previous characters you have to remember in order to comply with these rules is n. Then you can form an NFA whose set of states is Σ^n. This strategy will even accommodate rules like those for vowel harmony: you make copies of each string: one for a front-vowel environment, one for a back-vowel environment ...

Introduction

Mathematical Background

Turn this into a DM-for-linguists section

Could have a section on Discrete maths for linguists, explaining equivalence relations and congruence relations—e.g., phonemes as equivalence classes. Congruence relations are mentioned on page 39. Disjoint unions appear too.

We will be using the asterisk symbol: *, in more than one way. We will do the same with the two vertical bars ||. When a symbol is used in two related ways, we say it is **overloaded**.

Automata are going to be interesting to us because they are a way of understang **parsing**.

'Automata' (singular: *automaton*) is a Greek word for 'machines'. The automata in this course are all *discrete* rather than *continuous* machines. Or perhaps one should say *digital*-rather-than-*analogue*.

If you are happy about this difference just check that you and I have the same take on it: a car is an *analogue* machine, and a computer is a *digital* machine.

There are two ways into machines, and one can profitably run both—one does not have to restrict oneself to one only.

- One can draw a couple of FSAs and talk through what they do.
- One can start with a well-motivated story from life and say how this might be a machine. Try for example:

  `http://www.philosophy.uncc.edu/logic/projectTALLC/tallc2/applets/puzzles/wj/wj_jar.html` or try googling 'water jugs'

Finite state machines can be quite useful in describing games, like chess, or Go, snakes-and-ladders or draughts. This is because many games (for example, those I have just mentioned) have machines naturally associated with them: the board positions can be thought of as states of a machine. However, it's not always clear what the input alphabet is!

Snakes-and-ladders. At least if i remember properly, a snakes-and-ladders board has 100 squares, numbered 1–100, and the game is played by each player placing a piece on square 1 and, turn and turn about, rolling a die and advancing their piece the number of places indicated by the number on the die, and then hopping on a snake or a ladder should there be one at the destination square.

The point of departure is this: machines are things with finite descriptions that have states, and they move from one state to another on receiving an input, which is a character from an input alphabet.

The best way to present machines is through pictures.

To be formal about it: A machine $\mathfrak{M}$ is a set S of states, together with a family of transition operations, one for each $w \in \Sigma$, the input alphabet. These transition operations are usually written as one function of two arguments rather than lots of unary operations. Thus $\delta(s, c)$ is the state that machine is in after it received character c when it was in state s:

$$\delta : S \times \Sigma \to S$$

There must of course be a designated start state $s_0 \in S$, and there is a set A of accepting states $A \subseteq S$, whose significance we will explain later. We will also only be interested in machines with only finitely many states. There are technicalities to do with infinity, but we don't need to worry about them just yet. All we mean is that for our machine $\mathfrak{M}$ the number $|S|$ (the size of the set of states) must be a natural number: $|S| = 1$ or $|S| = 2$ or ....

Clearly any game of snakes and ladders can be represented by a FSA over the alphabet $\{1, 2, 3, 4, 5, 6\}$ with 100 states and precisely one accepting state.[1]

[1] Is the converse true? Can any FSA over the alphabet $\{1, 2, 3, 4, 5, 6\}$ with 100 states and precisely one accepting state be thought of as a snakes and ladders board?

2.0.3 Languages Recognised by Machines

Languages, parsing, compilers etc are the chief motivation for this course. You might think that a *language* is something like English, or Spanish, or perhaps PASCAL or JAVA or something like that: a naturally occurring set of strings-of-letters with some natural definition and a sensible reason for being there—such as meaning something! Who could blame you? It's a very reasonable thing to expect. Unfortunately for us the word 'language' has been hijacked by the formal-language people to mean something much more general than that. What they mean is the following.

We have to be careful here not to confuse the punters. 'language' in this context is different even from the word as used by logicians!

We start with an **alphabet** Σ (and for some reason they always *are* called Σ, don't ask me why) which is a finite set of **characters**. We are interested in **string**s of characters from the alphabet in question. A string of characters is not the same as a *set* of characters. Sets do not have order—the set $\{a, b\}$ is the same set as the set $\{b, a\}$ but strings do: the string ab is not the same as the string ba. Sets do not have *multiplicity*: the set $\{a, a, b\}$ is the same set as the set $\{a, b\}$ (which is why nobody writes things like '$\{a, a, b\}$') but strings definitely do have multiplicity: the string aab is not the same string as the string ab. Also, slightly confusingly, although we have this '{' and '}' notation for sets, there is no corresponding delimiter for strings. Notation was not designed rationally, but just evolved haphazardly.

We write 'Σ^*' for the set of all strings formed from characters in Σ; a **language** ("over Σ") is just a subset of Σ^*: any subset at all. A subset of Σ^* doesn't have to have a sensible definition in order to be called a language by the formal-language people. While we are about it, notice also that people use the word 'word' almost interchangeably with 'string'.

(Aside on notation: the use of the asterisk in this way to mean something like "finite repetitions of" is widespread. If you have done a course in discrete maths you might connect this with the notation 'R^*' for the transitive closure of R. We will also see the notation 'δ^*' for the function that takes a state s, and a string w and returns the state that the machine reaches if we start it in s and then perform successively all the transitions mandated by the characters in w. Thus, for example, for characters a, b and c, $\delta^*(s, ab) = \delta(\delta(s, a), b)$ and $\delta^*(s, abc) = \delta(\delta(\delta(s, a), b), c)$ and so on. In fact $\delta^* : S \times \Sigma^* \to S$. It's easier than it looks!)

A word of warning. Many people confuse $\subseteq$ and $\in$, and there is a parallel confusion that occurs here. (This is not the same as the confusion between sets and strings: the confusion I am talking about here is the common confusion between sets and their members!) A *language* is not a string: it is a *set* of strings. This may be because they are thinking that strings are sets of characters and that accordingly a language—on this view—is a set of sets. If in addition you think that everything there is to be said about sets can be drawn in Venn diagrams, this will confuse you. Venn diagrams give you pictures of sets of *points*, but not sets of *sets*. A Venn diagram picture displaying three levels of

sets is impossible.

We will write '$|w|$' for the length of the string w. This may remind you of the notation '$|A|$' for the number of elements of A when A is a set.

Although a language is any old subset of Σ^*, on the whole we are interested only in languages that are infinite. And here I find that students need to be tipped off about the need to be careful. Print out this warning and pin it to the wall:

> **The languages we are interested in are usually infinite, but the strings in them are always finite!**

The set of grammatical English sentences is infinite (people sometimes say "potentially infinite") but each individual grammatical sentence is finite!

Let's have some examples. Let Σ be the alphabet $\{a, b\}$.

The language $\{aa, ab, ba, bb\}$ is the language of two-letter words over Σ.

$\{a^n : n \in \mathbb{N}\}$ is the set of all strings consisting entirely of a's which is the (yes, it's infinite) language $\{\epsilon, a, aa, aaa, \ldots\}$.

$\{a^n b^n : n \in \mathbb{N}\}$ is the set of all strings that consist of a's followed by the same number of b's.

Question: What might the symbol 'ϵ' mean above (in "$\{\epsilon, a, aa, aaa, \ldots\}$").?
`Click here to submit`

Answer: It must mean the empty string. What else can it be!?

Mathematics is full of informal conventions that people respect because they find them helpful. Some of them are applicable here, and we will respect them.

1. We tend to use letters from near the beginning of the alphabet, a, b, ... for characters in alphabets;
2. We tend to use lower-case letters from near the end of the alphabet—like 'u', 'v' and 'w'—for variables to range over strings;
3. 'q' is often a variable used to vary over states;
4. We tend to use upper-case letters from the middle of the alphabet—like 'K', 'L'—for variables to range over languages.

Concatenation

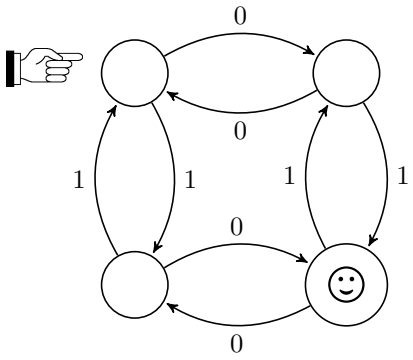
If w and u are strings then wu is w with u stuck on the end, and uw is u with w stuck on the end. Naturally $|uw| = |wu| = |w| + |u|$. ϵ is just the empty string (which we met on page 34) so ϵw—the empty string with w concatenated on the end—is just w. So ww, www and $wwwww$ are respectively the result of stringing two, three and five copies of w together. Instead of writing '$wwwww$' we write 'w^5' and we even use this notation for variable exponents, so that w^n is n copies of w strung together.

2.0.4 Languages from Machines

Some languages have sensible definitions in terms of machines. There is an easy and natural way of associating languages with machines. It goes like this.

For each machine we single out one state as the designated "start" state.

Then we decorate some of the states of our machines with labels saying "accepting". My (highly personal and nonstandard!) notation for this is a smiley smacked on top of the circle corresponding to the state in question. The more usual notation is much less evocative: states are represented as circles, and accepting states are represented by a pair of concentric circles.



Two Asides on Notation

- Some people write pictures of machines from which arrows might be missing, in that there could be a state s and a character c such that there is no arrow from s labelled c. With some people this means that you stay in s, and with some it means that you go from s to a terminally unhappy state—which I notate with a scowlie. My policy is to put in all arrows. This is a minority taste, but I think it makes things clearer. It's important to remember that the alternatives of putting in all arrows versus omitting arrows to terminally unhappy states afford us not two-different-concepts-of-machine, but two different-notations-for-the-same-concept. I must emphasise that my practice of putting in all arrows is not the usual practice in the literature (it should be but it isn't) and readers should get used to seeing pictures with missing arrows.

- For some reason the expression 'final state' is sometimes used for 'accepting state'. I don't like this notation, since it suggests that once you get the machine into that state it won't go any further or has to be reset or something, and this is not true. But you will see this nomenclature in the literature.

The use of the word "accepting" is a give-away for the use we will put this labelling to. We say that a machine **accepts** a string **if**, when the machine is powered up in the designated start state, and is then fed the characters from

s in order, **then** when it has read the last character of s it is in an accepting state.

This gives us a natural way of making machines correspond to languages. When one is shown a machine $\mathfrak{M}$ one's thoughts naturally turn to the set of strings that $\mathfrak{M}$ can accept—which is of course a language. We say that this set is the language **recognised by** $\mathfrak{M}$, and we write it $L(\mathfrak{M})$.

Pin this to the wall too:

> **The set of strings accepted by a machine $\mathfrak{M}$ is the language recognised by $\mathfrak{M}$**

People often confuse a machine-recognising-a-language with a machine-accepting-a-string. It is sort-of OK to *end up* confusing the two words 'recognise' and 'accept' once you really know what's going on: lots of people do. However if you *start off* confusing them you will become hopelessly lost.

Two points to notice here. It's obvious that the language recognised by a machine is uniquely determined. However, if you start from the other direction, with a language and ask what machine determines it then it's not clear that there will be a unique machine that recognises it. There may be lots or there may be none at all. The first possibility isn't one that will detain us long, but the second possibility will, for the difference between languages that have machines that accept them and languages that don't is a very important one. We even have a special word for them.

Definition:

> If there is a finite state machine which recognises L then L is **regular**.

Easy exercise. For any alphabet Σ whatever, the language Σ^* is regular. Exhibit a machine that recognises Σ^*. This is so easy you will probably suspect a trick.

`click here to submit`

Answer: The machine with one state, and that an accepting state.

I'm not sure why Kleene (for it was he) chose the word 'regular' for languages recognised by finite state machines. It doesn't seem well motivated.

2.0.5 Some exercises

1. (a) Draw a machine that recognises the language $\{\epsilon\}$. (This is easy but you might have to think hard about the notation!)

   (b) Draw a machine that recognises $\emptyset$, the empty language.

   `click here to submit`

Answer:

(a) The machine has two states. The start state is accepting, and the other state is not. The transition function takes only one value, namely the second—nonaccepting—state. Nonaccepting states like this—from which there is no escape—I tend to write with a scowlie.

(b) The machine has one state, and that is a nonaccepting state (a scowlie).

2. Explain what languages the following notations represent

   (a) $\{a^{2n} : n \in \mathbb{N}\}$;

   (b) $\{(ab)^n : n \in \mathbb{N}\}$;

   (c) $\{a^n b^m : n < m \in \mathbb{N}\}$

   You might like to design machines to recognise the first two.

   `click here to submit`

Answer:

(a) is the set of strings of even length consisting entirely of as. The corresponding machine has three states: the first means "I have seen an even number of as" (this state is the start state and also the unique accepting state); the second means "I have seen an odd number of states" and the third is an error state (a scowlie) to which you go if you receive any character other than an a;

(b) is the set of strings consisting of any number of copies of ab concatenated together. The corresponding machine has three states. The start state (which is also the unique accepting state)...

(c) contains those strings consisting of as followed by a greater number of bs.

3. A burglar alarm has a keypad with the digits $0, 1, 2, 3, 4, 5, 6, 7, 8$ and 9 on it. It is de-activated by any sequence of numbers that ends with the four characters $1, 9, 8$, and 3, in that order. Once deactivated it remains de-activated.

   Represent the burglar alarm as a finite state machine and supply a state diagram of it. Supply also a regular expression and a context-free grammar that capture the set of strings that deactivate the burglar alarm.

2.0.6 The Thought-experiment

We live in a finite world, and all our machines are finite, so regular languages are very important to us, since they are the languages that are recognised by

finite state machines. It's important to be able to spot (I nearly wrote 'recognise' there!) when a language is regular and when it isn't. Here is a thought-experiment that can help.

I am in a darkened room, whose sole feature of interest (since it has neither drinks cabinet nor coffee-making facilities) is a wee hatch through which somebody every now and then throws at me a character from the alphabet Σ. My only task is to say "yes" if the string of characters that I have had thrown at me so far is a member of L and "no" if it isn't (and these answers have to be correct!)

After a while the lack of coffee and a drinks cabinet becomes a bit much for me so I request a drinks break. At this point I need an understudy, and it is going to be you. Your task is to take over where I left off: that is, to continue to answer correctly "yes" or "no" depending on whether or not the string of characters that *we* (first I and then you) have been monitoring all morning is a member of L.

> **What information do you want me to hand on to you when I go off for my drinks break? Can we devise in advance a form that I fill in and hand on to you when I go off duty? That is to say, what are the parameters whose values I need to track? How many values can each parameter take? How much space do I require in order to store those values?**

Let us try some examples

1. Let L be the set of strings over the alphabet $\{a, b\}$ which have the same number of as as bs. What do you want me to tell you?

   `Click here to submit`

   Answer:

   > Clearly you don't need to know the number of as and the number of bs I've received so far but you do need to know the difference between these two quantities. Although this is only a single parameter there is no finite bound on its values (even though the value is always finite!) and it can take infinitely many values. so I cannot bound in advance the number of bits I need if I am to give you this information. L is not regular.

2. Let L be the set of strings over the alphabet $\{a, b\}$ which have an even number of as and an even number of bs. What do you want me to tell you?

   `click here to submit`

Answer:

All you need to know is whether the number of as is even or odd and whether the number of bs is even or odd. That's two parameters, each of which can take one of two values. That's two bits of information, and four states.

3. Let L be the set of strings over the alphabet $\{a, b\}$ where the number of as is divisible by 3 and so is number of bs. What do you want me to tell you?

   `click here to submit`

Answer:

It isn't sufficient to know whether or not the number of as is divisible by 3 (and the number of bs similarly): you need to know the number of as and bs mod 3. But it's still a finite amount of information: there are two parameters we have to keep track of, each of which can have one of three values. as far as we are concerned in this situation, you and I, there are only nine states.

4. $\{a^n b^n : n \in \mathbb{N}\}$ This is the language of strings consisting of any number of as followed by the same number of bs. ('n' is a variable!)

   `click here to submit`

Answer: Clearly you need to count the a's. This number can get arbitrarily large, so you would need infinitely many states ("I have seen one a"; "I have seen two a's" ...)

5. The "matching-brackets language": the set of strings of left and right brackets '(' and ')' that "match". (You know what I mean!)

   `click here to submit`

Answer: You have to keep track of how many left brackets you've opened, and this number cannot be bounded in advance.

Equivalence-from-the-point-of-view-of-L must be a congruence relation for each of the $|\Sigma|$ operations "stick w on the end" (one for each $w \in \Sigma$). We saw these on page 32. This is all explained in the new Discrete Mathematics notes.

2.0.7 The Pumping Lemma

The pumping lemma starts off as a straightforward application of the pigeonhole principle. If a machine $\mathfrak{M}$ has n states, and accepts even one string that is of length greater than n, then in the course of reading (and ultimately accepting)

that string it must have visited one of its states—s, say—twice. (At least twice: quite possibly more often even than that). This means that if w is a string accepted by $\mathfrak{M}$, and its length is greater than n, then there is a decomposition of w as a concatenation of three strings $w_1w_2w_3$ where w_1 is the string of characters that takes it from the start state to the state s; w_2 is a string that takes it from s on a round trip back to s; and w_3 is a string that takes it from s on to an accepting state.

This in turn tells us that $\mathfrak{M}$—having accepted $w_1w_2w_3$—must also accept $w_1(w_2)^nw_3$ for any n. $\mathfrak{M}$ is a finite state machine and although it "knows" at any one moment which state it is *in at that moment* it has no recollection of its history, no recollection of how it got into that state nor of how often it has been in that state before.

Thus we have proved

> **The Pumping Lemma**
>
> If a finite state machine $\mathfrak{M}$ has n states, and w is a string of length $> n$ that is accepted by $\mathfrak{M}$, then there is a decomposition of w as a concatenation of three strings $w_1w_2w_3$ such that $\mathfrak{M}$ also accepts all strings of the form $w_1(w_2)^nw_3$ for any n.

It does *not* imply that if w is a string of length $> n$ that is accepted by $\mathfrak{M}$, and $w_1w_2w_3$ is *any old* decomposition of w as a concatenation of three strings then $\mathfrak{M}$ also accepts all strings of the form $w_1(w_2)^nw_3$ for any n: it says merely that *there is at least one* such decomposition.

You need to be very careful when attempting to state the pumping lemma clearly, as it has so many alternations of quantifiers: There is a number s (actually the number of states in the machine) which is so large that for all strings w with $|w| > s$ that are accepted by $\mathfrak{M}$ there are substrings x, w', and y of w, so that $w = xw'y$ such that for all n, $x(w')^ny$ is also accepted by $\mathfrak{M}$. That's four blocks of quantifiers: a lot of quantifier alternations!!

The pumping lemma is very useful for proving that languages aren't regular.

In order to determine whether a language is regular or not you need to form a hunch and back it. Either guess that it is regular and then find a machine that recognises it or form the hunch that it isn't and then use the pumping lemma. How do you form the hunch? Use the thought experiment. If the thought experiment tells you: "a finite amount of information" you immediately know it's a finite-state machine, and if you think about it, it becomes clear what the machine is. What do we do if the thought-experiment tells you that you need infinitely many states (beco's there appears to be no bound on the amount of information you might need to maintain)? This is where the pumping lemma comes into play. You use it to build *bombs*.

Bombs?! Read on.

2.0.8 Bombs

Let L be a language, and suppose that although L is not regular, there is nevertheless someone who claims to have a machine $\mathfrak{M}$ that recognises it: i.e.,

they claim to have a machine that accepts members of L *and nothing else*. This person is the **Spiv**. This machine is fraudulent of course, but how do we prove it? What we need is a *bomb*.

> A **bomb** (for $\mathfrak{M}$) is a string that is either (i) a member of L not accepted by $\mathfrak{M}$ or (more usually) (ii) a string accepted by $\mathfrak{M}$ that isn't in L. Either way, it is a certificate of fraudulence of the machine $\mathfrak{M}$, and therefore something that **explodes** those fraudulent claims.

How do we find bombs? This is where the pumping lemma comes in handy. The key to designing a bomb is feeding $\mathfrak{M}$ a string w from L whose length is greater than the number of states of $\mathfrak{M}$. $\mathfrak{M}$ accepts w. $\mathfrak{M}$ must have gone through a loop in so doing. Now we ascertain what substring w' of w sent $\mathfrak{M}$ through the loop, and we insert lots of extra copies of that substring next to the one copy already there *and we know that the new "pumped" string will also be accepted by* $\mathfrak{M}$. With any luck it won't be in L, and so it will be a bomb. The key idea here is that *the machine has no memory of what has happened to it beyond what is encoded by it being in one state rather than another.* So it cannot tell how often it has been though a loop. *We*—the bystanders—know how often it has been sent through a loop but the machine itself has no idea.

Examples are always a help, so let us consider some actual challenges to the bomb-maker.

2.0.9 One-step refutations using bombs

The language $\{a^n b^n : n \in \mathbb{N}\}$ is not regular

Suppose the Spiv shows us $\mathfrak{M}$, a finite state machine that—according to him—recognises the language $\{a^n b^n : n \geq 0\}$. The thought-experiment tells us immediately that this language is not regular (this was example 4 on page 39) so we embark on the search for a bomb with high expectations of success. In the first instance the only information we require of the Spiv is the number of states of $\mathfrak{M}$, though we will be back later for some information about the state diagram. For the moment let k be any number such that $2k$ is larger than the number of states of $\mathfrak{M}$. Think about the string $a^k b^k$. What does $\mathfrak{M}$ do when fed $a^k b^k$? It must go through a loop of course, because we have made k so large that it has to. In going through the loop it is reading a substring w of $a^k b^k$. What does the substring consist of? We don't know the exact answer to this, but we can narrow it down to one of the three following possibilities, and in each case we can design a bomb.

1. w consists entirely of as. Then we can take our bomb to be the string obtained from $a^k b^k$ by putting n copies of w instead of just one. Using our notation to the full, we can notate this string $a^{(k-|w|)}a^{(|w|\cdot n)}b^k$

   Explain why this is correct. `click here to submit`

Answer: Well, w consists of $|w|$ a's. Remove w from $a^k b^k$ and put in n copies of what you've taken out. Then stick on the end all the b's you had. $\mathfrak{M}$ will accept this string, but this string contains more as than bs, and $\mathfrak{M}$ shouldn't have accepted it.

2. w consists entirely of bs. Then our bomb will be the string obtained from $a^k b^k$ by putting in several copies of w instead of just one. $\mathfrak{M}$ will accept this string, but this string contains more bs than as, and $\mathfrak{M}$ shouldn't have accepted it. Exercise: how do we notate this bomb, by analogy with the bomb in the previous case? `click here to submit`

Answer: $a^k b^{(k-|w|)} b^{(|w|\cdot n)}$

3. w consists of some as followed by some bs. In this case, when we insert n copies of w to obtain our bomb, we compel $\mathfrak{M}$ to accept a string that contains some as followed by some bs and then some as again (and then some bs). But—by saying that $\mathfrak{M}$ recognised $\{a^n b^n : n \in \mathbb{N}\}$—the Spiv implicitly assured us that the machine would not accept any string containing as *after* bs. Exercise: how do we notate this bomb, by analogy with the bomb in the previous case? `click here to submit`

Answer: This one is a bit tricky. Suppose w is $a^x b^y$ then the bomb can be $a^{(k-x)} w^n b^{(k-y)}$

So we know we are going to be able to make a bomb whatever w is. However, if we are required to actually exhibit a bomb we will have to require the Spiv to tell us what w is.

The language $\{a^k b a^k : k \geq 0\}$ is not regular

Suppose the Spiv turns up with $\mathfrak{M}$, a finite state machine that is alleged to recognise the language $\{a^k b a^k : k \geq 0\}$. Notice that every string in this language is a palindrome (a string that is the same read backwards as read forwards). We will show that $\mathfrak{M}$ will accept some strings that aren't palindromes, and therefore doesn't recognise $\{a^k b a^k : k \geq 0\}$.

As before, we ask the Spiv for the number of states the machine has, and get the answer m, say. Let n be any number bigger than m and consider what happens when we give $\mathfrak{M}$ the string $a^n b a^n$. This will send $\mathfrak{M}$ through a loop. That is to say, this string $a^n b a^n$ has a substring within it which corresponds to the machine's passage through the loop. Call this string w. Now, since we have force-fed the machine n a's, and it has fewer than n states, it follows that w consists entirely of a's. Now we take our string $a^n b a^n$ and modify it by replacing the substring w by lots of copies of itself. Any number of copies will do, as long as it's more than one. This modified string (namely $a^{(k+|w|)} b a^k$) is our bomb. Thus $\mathfrak{M}$ accepts our bomb, thereby demonstrating—as desired—that it doesn't recognise $\{a^k b a^k : k \geq 0\}$.

2.0.10 A few more corollaries

1. $\mathfrak{M}$, if it accepts anything at all, will accept a string of length less than $|\mathfrak{M}|$.

2. If $\mathfrak{M}$ accepts even one string that has more characters than $\mathfrak{M}$ has states will accept arbitrarily long strings.

The Pumping Lemma is a wonderful illustration of the power of **The Pigeonhole Principle**. If you have $n+1$ pigeons and only n pigeonholes to put them in the at least one pigeonhole will have more than one pigeon in it. The pigeonhole principle sounds too obvious to be worth noting, but the pumping lemma shows that it is very fertile.

2.1 Operations on machines and languages

Languages are sets, and there are operations one can perform on them simply in virtue of their being sets. If K and L are languages, so obviously are $K \cup L$, $K \cap L$ and $K \setminus L$. There is one further operation that we need which is defined only because languages are sets of strings and there are operations we can perform on strings, specifically concatenation. This concatenation of strings gives us a notion of concatenation of languages. $KL = \{wu : w \in K \wedge u \in L\}$, and the reader can probably guess what K^* is going to be. It's the union of K, KK, KKK ...

Exercise 3. *If $K = \{aa, ab, bc\}$ and $L = \{bb, ac, ab\}$ what are (i) KL, (ii) LK, (iii) KK, (iv) LL?* `click here to submit`

Answer:
$KL = \{aabb, aaac, aaab, abbb, abac, abab, bcbb, bcac, bcab\}$;
$LK = \{bbaa, bbab, bbbc, acaa, acab, acbc, abaa, abab, abbc\}$;
$KK = \{aaaa, aaab, aabc, abaa, abab, abbc, bcaa, bcab, bcbc\}$;
$LL = \{bbbb, bbac, bbab, acbb, acac, acab, abbb, abac, abab\}$;

Exercise 4. *Can you express $|KL|$ in terms of $|K|$ and $|L|$?*

`click here to submit`

Answer: You want to say $|K| \cdot |L|$, don't you? But if K and L are both $\{a, aa\}$ then $KL = \{aa, aaa, aaaa\}$ with three elements not four, because aaa is generated in two ways not one. All you can say is that $|KL| \leq |K| \cdot |L|$ Contrast this with $|wu| = |w| + |u|$ on page 34 (overloading of vertical bars to mean both size of a set and length of a string).

The following fact is fundamental.

If K and L are regular languages so are $K \cap L$, $K \setminus L$, KL and K^*.

So every language you can obtain from regular languages by means of any of the operations we have just seen is likewise regular.

Let's talk through this result.

$K \cap L$

The thought-experiment shows very clearly that the intersection of two regular languages is regular : if I have a machine $\mathfrak{M}_1$ that recognises L_1 and a machine $\mathfrak{M}_2$ that recognises L_2 the obvious thing to do is to run them in parallel, giving each new incoming character to both machines and accept a string if they both accept it. Using the imagery of the thought-experiment, the clipboard of information that I hand on to you when I go for my coffee break has become a pair of clipboards, one for $\mathfrak{M}_1$ and one for $\mathfrak{M}_2$.

But although this makes it clear that the intersection of two regular languages is regular, it doesn't make clear what the machine is that recognises the intersection. We want to cook up a machine $\mathfrak{M}_3$ such that we can see running-$\mathfrak{M}_1$-and-$\mathfrak{M}_2$-in-parallel as merely running $\mathfrak{M}_3$. $\mathfrak{M}_3$ is a sort of composite of $\mathfrak{M}_1$ and $\mathfrak{M}_2$. What are the states of this new composite machine?

The way to see this is to recall from page 39 the idea that a state of a machine is a state-of-knowledge about the string-seen-so-far. What state-of-knowledge is encoded by a state of the composite machine? Obviously a state of the composite machine must encode your knowledge of the states of the two machines that have been composed to make the new (composite) machine. So states of the composite machine are ordered pairs of states of $\mathfrak{M}_1$ and $\mathfrak{M}_2$.

What is the state transition function for the new machine? Suppose $\mathfrak{M}_1$ has a transition function δ_1 and $\mathfrak{M}_2$ has a transition function δ_2, then the new machine has the transition function that takes the new state $\langle s_1, s_2 \rangle$ and a character c and returns the new state $\langle \delta_1(s_1, c), \delta_2(s_2, c) \rangle$.

$K \cap L$ and $K \setminus L$

The proofs that a union or difference of two regular languages is regular is precisely analogous.

People sometimes talk of the *complement* of a language L. L is a language over an alphabet Σ, and its complement, relative to Σ, is $\Sigma^* \setminus L$. It's easy to see that the complement of a regular language L is regular: if we have a machine $\mathfrak{M}$ that recognises L, then we can obtain from it a machine that recognises the complement of L just by turning all accepting states into non-accepting states and *vice versa*.

(A common mistake is to assume that every subset of a regular language is regular. I think there must be a temptation to assume that a subset of a language recognised by $\mathfrak{M}$ will be recognised by a version of $\mathfrak{M}$ obtained by throwing away some accepting states.)

KL

It's not obvious that the concatenation of two regular languages is regular, but it's plausible. We will explain this later. For the moment we will take it as read and press ahead. This leads us to

2.1.1 Regular Expressions

A regular expression is a formula of a special kind. Regular expressions provide a notation for regular languages. We can declare them in BNF (Backus-Naur form).

We define the class of regular expressions over an alphabet Σ recursively as follows.

1. Any element of Σ standing by itself is a regular expression;
2. If A is a regular expressions, so is A^*;
3. If A and B are regular expressions, so is AB;
4. If A and B are regular expressions, so is $A|B$.

For example, for any character a, a^* is a regular expression.

The idea then is that regular expressions built up in this way from characters in an alphabet Σ will somehow point to regular languages $\subseteq \Sigma^*$. Now we are going to recall the $L()$ notation which we used on page 35, where $L(\mathfrak{M})$ is the language recognised by $\mathfrak{M}$, and overload it to notate a way of getting languages from regular expressions. We do this by recursion using the clauses above.

1. If a is a character from Σ (and thus by clause 1, a regular expression) then $L(a) = \{a\}$.
2. $L(A|B) = L(A) \cup L(B)$.
3. $L(AB) = L(A)L(B)$. This looks a bit cryptic, but actually makes perfect sense. $L(A)$ and $L(B)$ are languages. If K_1 and K_2 are languages, you know what K_1K_2 is from page 43, so you know what $L(A)L(B)$ is!

   $L(A^*)$ is the set $L(A) \cup L(AA) \cup L(AAA) \ldots = \bigcup_{n \in \mathbb{N}} A^n$. A^n is naturally $AAAA \ldots A$ (n times).

2.1.2 More about bombs

2.1.2.1 Palindromes do not form a regular language

You may recall that a palindrome is a string that is the same read backwards or forwards. If you ignore the spaces and the punctuation then the strings 'Madam, I'm Adam' and 'A man, a plan, a canal—Panama!" are palindromes.

(Even better: A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag

again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a peon, a canal—Panama!!)

The thought-experiment swiftly persuades us that the set of palindromes over an alphabet Σ is not regular (unless Σ contains only one character of course!). After all—as you will have found by looking first at "Madam, I'm Adam" and then the two longer examples—to check whether or not a string is a palindrome one finds oneself making several passes through it, and having to compare things that are arbitrarily far apart.

Let L be the language of palindromes over $\{a, b\}$. It isn't regular, but there is no obvious bomb. However, if L were regular then so too would be the language $L \cap L(a^*ba^*)$. (We established on page 43 that the intersection of two regular languages is regular.) This new language is just the language $\{a^kba^k : n \in \mathbb{N}\}$ that we saw on page 42.

2.1.2.2 The language $\{ww : w \in \Sigma^*\}$ is not regular

I don't see how to use a bomb to show that $\{ww : w \in \Sigma^*\}$ is not regular, though it's obvious from the thought-experiment. However, we do know that the language $L(a^*b^*a^*b^*)$ is regular so if our candidate were regular so too would be the language $\{ww : w \in \Sigma^*\} \cap L(a^*b^*a^*b^*)$. Now this language is $\{a^nb^ma^nb^m : m, n \in \mathbb{N}\}$ and it isn't hard to find bombs to explode machines purporting to recognise this language. You might like to complete the proof by finding such a bomb.

`click here to submit`

Answer:
A machine with k states that purports to recognise this language can be exploded by the bomb $a^kb^ka^{k+1}b^k$.

2.1.3 Some more exercises

1. Is every finite language regular?
2. The "reverse" of a regular language is regular: if L is regular, so is $\{w^{-1} : w \in L\}$ where w^{-1} is w written backwards.
3. You know that you cannot build a finite state machine that recognises the matching bracket language. However you can build a machine that accepts all and only those strings of matching brackets where the number of outstanding opened left brackets never exceeds three.

   Find a regular expression for the language accepted by this machine. (I suggest that, in order not to drive yourself crazy, you write '0' instead of the left bracket and '1' instead of the right bracket!!)

   `Click here to submit`

I think the answer is:
$(0(0(01)^*1)^*1)^*$

How about the same for a machine than can cope with as many as five outstanding brackets?

`Click here to submit`

Or six? Or seven? This becomes clear once you think about it. If N is the regular expression for the machine that can cope with as many as n oustanding open left brackets then the regular expression for the machine that can cope with as many as $n+1$ oustanding open left brackets is $N^*|(0(N^*)1)^*$

2.2 Grammars

So far we have encountered two ways of thinking about regular languages: (i) through finite state machines; (ii) through regular expressions. These approaches have their roots in the study of machines, rather than—as you had probably been expecting—the study of natural languages. The third approach, you will be relieved to hear, is one that has its roots in the study of natural languages after all.

Many years ago (when I was at school) children were taught *parsing*. We were told things like the following. Sentences break down into

Subject followed by **Verb** followed by **Object**.

Or perhaps they break down into

Noun Phrase followed by **Verb** followed by **Noun Phrase**.

These constituents break down in turn: noun phrases being composed of **determiner** followed by **adjective** followed by **noun**.

This breaking-down process is important. The idea is that the way in which we assemble a sentence from the parts into which we have broken it down will tell us how to recover the meaning of a sentence from the meanings of its constituents.

If we start off with an alphabet

$$\Sigma = \{\texttt{dog}, \texttt{cat}, \texttt{the}, \texttt{some}, \texttt{walk}, \texttt{swim}, \texttt{all}, \texttt{some} \ldots\}$$

then rules like

sentence → Subject verb object

and

Noun phrase → determiner adjective noun

have the potential, once equipped with further rules like

determiner → `the a some many`
verb → `swim`
verb → `walk`
noun → `dog`
noun → `cat`

to generate words in a language $L \subseteq \Sigma^*$. This time 'language' really does mean something like 'language' in an ordinary sense (and 'word' now means something like sentence'). But the "words" that we generate are actually things that in an ordinary context would be called *sentences*. And this time we think of 'dog' not as a string but as a character from an alphabet, don't we!

The languages that we have seen earlier in this coursework can be generated by rules like this. For example

$S \to aS$
$S \to \epsilon$

generates every string in the language $L(a^*)$ and a set of rules like

$S \to aSa$
$S \to bSb$
$S \to \epsilon$

generates the language of palindromes over the alphabet $\Sigma = \{a, b\}$.

These bundles of rules are called **grammars** and the rules in each bundle are called **productions**.

There are two sorts of characters that appear in productions. There are **nonterminals** which appear on the left of productions (and sometimes on the right). These are things like 'noun' and 'verb'. **Terminals** are the characters from the alphabet of the language we are trying to build, and they only ever appear on the right-hand-side of the production. Examples here are '`swim`', '`walk`', '`cat`' and '`dog`'.

Notice that there is a grammar that generates the language of palindromes over $\Sigma = \{a, b\}$ even though this language is not regular. Grammars that generate regular languages have special features that mark them off. One can ascertain what these features are by reverse-engineering the definition of regular language from regular expressions or finite state machines, but we might as well just give it straight off.

Definition 1. *A* **Regular Grammar** *is one in which all productions are of the form*

$$N \to TN'$$

or

$$N \to T$$

Where N and N' are nonterminals and T is a string of terminals.

The other illustrations i have given are of grammars not having this restriction. They are context-free. Reason for this nomenclature is state transition table here

2.2.1 Exercises

Exercise 5. *Provide a context-free grammar for regular expressions over the alphabet* $\{a, b\}$

2.2.2 Pushdown Automata

We have characterised context-free languages in terms of grammars, but they can also be characterised in terms of machines. There is a special kind—a special class—of machines which we call **push-down automata** (or"PDA" for short) that are related to context-free languages in the same way that finite state machines are related to regular languages. Just as a language is regular iff there is a finite-state machine that recognises it (accepts all the strings in it and accepts no other strings), so a language is context-free iff there is a pushdown automaton that recognises it.

So what is a push-down automaton? One way in to this is to first reflect on what extra bells and whistles a finite state machine must be given if it is to recognise a context-free language such as the matching-brackets language, and to then engineer those bells and whistles into the machine. The thought-experiment comes in handy here. What does the thought-experiment tell you about the matching backets language? Run the experiment and you will find that to crack the matching-brackets language it would be really nice to have a *stack*[2]. Every time you see a left bracket you push it onto the stack and every time you see a right-bracket you pop a left-bracket off the stack; whenever the stack is empty you are in an accepting state. If you ever find yourself trying to pop a bracket off an empty stack you know that things have gone permanently wrong so you shunt yourself into a scowlie. What I need to hand over to you when I go off for my coffee is the stack (or the scowlie).

Let us now try to formalise the idea of a machine-with a stack. Our way in is to start off by thinking of a PDA as a finite state machine which we are going to upgrade, so we have the idea of start state and accepting state as before. The stack of course contains a string of characters. For reasons of hygiene we will tend to assume that the alphabet of characters that go on the stack (the "pushdown alphabet") is disjoint from the alphabet that the context free language is drawn from. Clearly if there is to be any point in having a stack at all then the machine is going to have to read it, and this entails immediately that the transition function of the machine has not *two* arguments (as the transition function of a finite state machine does, namely the old state and the character being read) but *three*, with the novel third argument being the character at the top of the stack. And of course the transition function under this new arrangement not only tells you what state the machine will go into, but what

[2]A stack of course is one of those things-with-springs that plates in cafeterias sit on, so the top plate is always the same height no matter how many plates there are in it—within reason.

to do with the stack—namely push onto it a word (possibly empty) from the stack alphabet.

(This is a bit confusing: the transition function in the new scheme of things takes an input which is an ordered triple of the contents-of-the-stack, the new character that is being fed it by the user, and the current state; the value is a pair of a state and the new contents-of-the-stack. However the transition function doesn't look at the whole of the contents of the stack but only the top element. Specifically this means that the new stack can differ from the old in only very limited ways. The new stack is always the old stack with the top element replaced by a string (possibly empty) of characters from the stack alphabet.)

If you are happy with this description you might now like to try designing a PDA that accepts the matching bracket language. (Hint: it has only three states: (i) accepting, (ii) wait-and-see, and (iii) dead!)

click here to submit

Answer: The reason why we need a stack is to keep track of the number of unmatched left brackets we have accumulated. We don't need it for anything else so the stack alphabet has only one character (which might as well be a left bracket. I know we said the alphabets should be kept disjoint for reasons of hygiene but...!) The PDA starts in the accepting state with an empty stack. Transition rules are as follows:

1. If you are in the dead state, stay there whatever happens.
2. If you are in the accepting state
   (a) if you read a right bracket go to the dead state;
   (b) if you read a left-bracket push it onto the stack and go to the wait-and-see state;
3. If you are in the wait-and-see state then
   (a) if you read a left bracket push it on the stack and remain in the wait-and-see state;
   (b) if you read a right bracket pop the top character off the stack and stay in the wait-and-see state unless the stack is now empty, in which case go to the accepting state.

Notice that if you are in the wait-and-see state then the stack is not empty, so we don't have to define the transition function for the case when the stack is not empty. Similarly when we are in the accepting state the stack must be empty. These two facts are not hard to see, but are a wee bit tricky to prove. You would have to prove by induction on the length n of possible input strings that for all strings s of length n when the machine has read s then if it's in the wait-and-see state then the stack is not empty and if it is in the accepting state then the stack is empty.
The PDA you have just written is a deterministic PDA.

There is a slight complication in that PDA's are nondeterministic, so the parallel is with regular languages being recognised by nondeterministic finite automata rather than FSAs... But we haven't dealt with nondeterministic machines yet. So we'd better deal with them at once!

Chapter 3

Propositional Logic

So let's start by looking at a very simple logical language: the language of propositional logic.

> A formula (or sentence) is either
>
> - A letter: p, q, r ...; or
> - The result of putting '$\wedge$', '$\vee$', '$\rightarrow$' between two formulæ or a '$\neg$' in front of a formula'
>
> Nothing else is a formula.

Actually, while we are about it, we may as well provide for this toy language the corresponding toy example of a grammar for it:

$S \to (S \wedge S)$

$S \to (S \vee S)$

$S \to (S \to S)$

$S \to \neg(S)$

beware of the double use of '$\rightarrow$' in the third line! use $\to$ Talk over the slide

This is not so much a language as a skeleton of a language. A natural language comes equipped with meanings for all its words: for linguists the meanings of the words are part of the language. Here only the meanings of the mathematical-looking symbols '$\wedge$' etc. are determined. They are **reserved words** (see p. 13). Grammatically they are what you linguists would probably call *conjunctions* things like 'and' and 'or'. In formal logic we call them *connectives*—beco's we use the word 'conjunction' for something else. (Another annoying example of two communities using divergent notations for the same thing!)

In contrast the letters 'p', 'q', 'r' etc have no internal structure and do not come equipped with any meaning. They are dummies or variables, and they

stand for *statements* in the sense that the intended use of this syntax is to replace the letters with things like 'Daisy is a cow' that have truth-values: that is, are true or false.

You could, if you wanted, think of this language as like a natural language where we know the grammar and some of the parts of speech (the conjunctions), but where we do not know the meanings of any of the other items in the lexicon.

People use this language to formalise ordinary language arguments. We use $\vee$ for 'or' and '$\wedge$' for 'and'.

Let's have some illustrations.

"If Anna can cancan or Kant can't cant, then Greville will cavil vilely. If Greville will cavil vilely, Will won't want. But Will will want. Therefore, Kant can cant."

We abbreviate

Anna can cancan	to	A
Kant can cant	to	K
Greville will cavill vilely	to	G
Will will want	to	W

Then in the new language I am introducing we can write this as

$$\frac{(A \vee \neg K) \rightarrow G; \quad G \rightarrow \neg W}{K}$$

It looks as if there are two stages of semantics: one at which you decide what complex expressions the letters are dummies for, and another at which you give truth-values to those complex expressions. It becomes a language in *your* sense once you replace the p and q etc by complex expressions.

Of course historically the purpose of the invention of propositional logic was to capture the structure of arguments, and that isn't really what we as linguists are trying to do.

Origin of the terminology **propositional**. Euclid. In propositional logic we are not trying to capture commands, merely assertions.

Use examples from Kalisch and Montague. Talk about $\vee$ and $\wedge$ ('wedge') and $\neg$ and truth tables. Talk about rules for connectives. (but not $\rightarrow$-int or $\vee$-elim!). The connectives are part of the **Logical Vocabulary** (whose meaning is fixed).

Then talk about how some connectives are more extensional than others and how we are interested in extensional connectives in the first instance. Not $\Box$!

Truth tables.

and, **because**, **despite** are intensions that all have the same extension.

Very hard to capture **implies** with an extensional logic. Say something about the material conditional at this point, but very briefly. (It's covered in appendix 8.1.1)

Must talk about Disjunctive Normal Form. DNF prepare us for possible world semantics.

Exercise 6.

1. *Propositional letters are p, p', p'', p''' .... This is a regular language.*

   Exercise: Write a machine that recognises it.

   (Think: what is the alphabet?)

   Make somewhere the point that this $'$ operation has no semantics

2. *A literal is either a propositional letter, or a propositional letter preceded by a '$\neg$'.*

   The set of literals forms a regular language.

   Exercise: Write a machine that recognises it.

   (Think: what is the alphabet?)

3. *A basic disjunction is a string like $p \vee \neg q \vee r$, namely a string of literals separated by '$\vee$'. (For the sake of simplicity we overlook the fact that no literal may occur twice!)*

   Even if the language is infinite! Extended notion of 'regular' here

   The set of basic disjunctions forms a regular language.

   Exercise: Write a machine that recognises it.

   (Think: what is the alphabet?)

4. *A formula in CNF is a string of basic disjunctions separated by $\wedge$, in the way that a basic disjunction is a set of literals separated by '$\vee$'.*[1]

   The set of formulæ in CNF forms a regular language.

   Exercise: Write a machine that recognises it.

   (Think: what is the alphabet?)

Write context-free grammars for conjunctive normal form and disjunctive normal; form.

`click here to submit`

[1] You might think that we need to wrap up each basic disjunction in a pair of matching brackets. In general this is true, but here we can get away without doing it.

A Grammar for CNF:

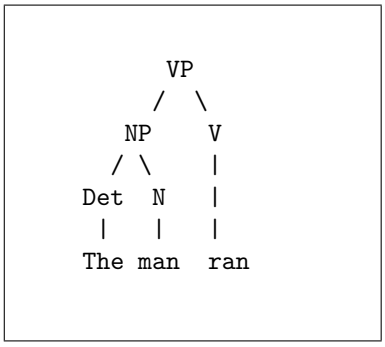
Formula $\rightarrow$ conjunct $\wedge$ Formula
Formula $\rightarrow$ ϵ
conjunct $\rightarrow$ literal $\vee$ conjunct
conjunct $\rightarrow$ ϵ
literal $\rightarrow$ atomic
literal $\rightarrow$ negatomic
atomic $\rightarrow$ $p, q, r \ldots$
negatomic $\rightarrow$ $\neg$ atomic

Notice how in this case the production rules have genuine natural semantic meaning. Notice also that the grammar is not regular.

A tautology is something whose truth table has nothing but 1s in its main column.

3.1 Formal Semantics for Propositional Logic

Just we we can do parse trees for English:



. . . we can do parse trees for propositional logic. And we find that the semantics for the formulæ of propositional logic are driven by parse trees in the exact same way.

We start with a simple—indeed almost trivial—example to illustrate the techniques we are going to use.

Ordinary modern ("arabic" but actually indian) notation for natural numbers admits strings from the following alphabet: {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}. Notice the quotation marks: the alphabet consists of the *symbols* not of the numbers they name, so when I am mentioning the symbols I use their *names*, which I obtain by putting single quotation marks round tokens of those same symbols.

A wff of this language is any string not beginning with a '0'. So here is the semantics:

The meaning of '0' is the number zero;
The meaning of '1' is the number one;
The meaning of '2' is the number two;
The meaning of '3' is the number three;
The meaning of '4' is the number four;
The meaning of '5' is the number five;
The meaning of '6' is the number six;
The meaning of '7' is the number seven;
The meaning of '8' is the number eight;
The meaning of '9' is the number nine.

The meaning of a string (thought of as a list) that consists of a string s with a character x stuck on the end is: (ten times the meaning of s) + (the meaning of x).[2]

We see here in microcosm many of the features we see in more sophisticated cases. We are defining a function that takes pieces of syntax (things that are entirely innocent of meaning) and gives out meanings, and does so in a systematic way that takes account of the way in which we parse the syntax.

The key to not getting lost in this enterprise is to bear in mind that the expressions of propositional logic are built up from atomic formulæ (letters) whose meaning is not reserved: they can be anything: *Anna can cancan* or *Kant can't cant*, but the symbols in the logical vocabulary—'$\wedge$', '$\vee$' and so on—emphatically *are* reserved.

A valuation [for propositional language] is a function that assigns truth-values (not meanings!) to the primitive letters of that language. We will use the letter 'v' to range over valuations. Now we define a satisfaction relation $\mathtt{sat}$ between valuations and complex expressions.

Definition 2.

A complex expression ϕ might be a propositional letter and—if it is—then $\mathtt{sat}(v, \phi)$ is just $v(\phi)$, the result of applying v to ϕ;

If ϕ is the conjunction of ψ_1 and ψ_2 then $\mathtt{sat}(v, \phi)$ is $\mathtt{sat}(v, \psi_1) \wedge \mathtt{sat}(v, \psi_2)$;

If ϕ is the disjunction of ψ_1 and ψ_2 then $\mathtt{sat}(v, \phi)$ is $\mathtt{sat}(v, \psi_1) \vee \mathtt{sat}(v, \psi_2)$;

If ϕ is the conditional whose antecedent is ψ_1 and whose consequent is ψ_2 then $\mathtt{sat}(v, \phi)$ is $\mathtt{sat}(v, \psi_1) \rightarrow \mathtt{sat}(v, \psi_2)$;

If ϕ is the negation of ψ_1 then $\mathtt{sat}(v, \phi)$ is $\neg\mathtt{sat}(v, \psi_1)$;

If ϕ is the biconditional whose two immediate subformulæ are ψ_1 and ψ_2 then $\mathtt{sat}(v, \phi)$ is $\mathtt{sat}(v, \psi_1) \longleftrightarrow \mathtt{sat}(v, \psi_2)$.

Notice that here i am using the letters 'ϕ' and 'ψ_1' and 'ψ_2' as variables that range over formulæ, as in the form of words "If ϕ is the conjunction of ψ_1 and ψ_2 then ...". They are not abbreviations of formulæ. There is a temptation to write things like

"If ϕ is $\psi_1 \wedge \psi_2$ then $\mathtt{sat}(v, \phi)$ is $\mathtt{sat}(v, \psi_1) \wedge \mathtt{sat}(v, \psi_2)$"

[2] Observe here that 'x' is a variable that takes as its values the characters '0', '1' ....

or perhaps

$$\texttt{sat}(v, \psi_1 \wedge \psi_2) \text{ is } \texttt{sat}(v, \psi_1) \wedge \texttt{sat}(v, \psi_2) \tag{3.1}$$

Now although our fault-tolerant pattern matching enables us to see immediately what is intended, the pattern matching does, indeed, need to be fault-tolerant. (In fact it corrects the fault so quickly that we tend not to notice the processing that is going on.)

In an expression like '$\texttt{sat}(v, \phi)$' the 'ϕ' has to be a name of a formula, as we noted above, not an abbreviation for a formula. But then how are we to make sense of

$$\texttt{sat}(v, \psi_1 \wedge \psi_2) \tag{3.2}$$

The string '$\psi_1 \wedge \psi_2$' has to be the name of formula. Now you don't have to be The Brain of Britain to work out that it has got to be the name of whatever formula it is that we get by putting a '$\wedge$' between the two formulæ named by 'ψ_1' and 'ψ_2'—and this is what your fault-tolerant pattern-matching wetware (supplied by Brain-Of-Britain) will tell you. But we started off by making a fuss about the fact that names have no internal structure, and now we suddenly find ourselves wanting names to have internal structure after all!

In fact there is a way of making sense of this, and that is to use the cunning device of *corner quotes* to create an environment wherein compounds of names of formulæ (composed with connectives) name composites (composed by means of those same connectives) of the formulæ named..

That is to say, we have a kind of `environment` command that creates an environment within which [deep breath]

> *contructors applied to* **pointers** *to objects*
>
> construct
>
> **pointers** *to the objects thereby constructed.*

So 3.1 would be OK if we write it as

$$\texttt{sat}(v, \ulcorner \psi_1 \wedge \psi_2 \urcorner) \text{ is } \texttt{sat}(v, \psi_1) \wedge \texttt{sat}(v, \psi_2) \tag{3.3}$$

Corner quotes were first developed in [21]. See pp 33-37. An alternative way of proceding that does not make use of corner quotes is instead to use an entirely new suite of symbols—as it might be '`and`' and '`or`' and so on, and setting up links between them and the connectives '$\wedge$' and so on in the object language so that—for example

$$\psi_1 \texttt{ and } \psi_2 \tag{A}$$

is the conjunction of ψ_1 and ψ_2. The only drawback to this is the need to conjure up an entire suite of symbols, all related suggestively to the connectives they are supposed to name. Here one runs up against the fact that any symbols that are suitably suggestive will also be laden with associations from their other

uses, and these associations may not be helpful. Suppose we were to use an ampersand instead of '`and`'; then the fact that it is elsewhere used instead of '$\wedge$' might cause the reader to assume it is just a synonym for '$\wedge$'. There is no easy way through.

3.2 Eager and Lazy Evaluation

The recursive definition of `sat` in the previous section gives us a way of determining what truth-value a formula receives under a valuation. Start with what the valuation does to the propositional letters (the leaves of the parse tree) and work up the tree. Traditionally the formal logic that grew up in the 20th century took no interest in how things like $\texttt{sat}(\phi, v)$ were actually *calculated*. The recursive definition tells us uniquely what the answer must be but it doesn't tell us uniquely how to calculate it.

The way of calculating $\texttt{sat}(\phi, v)$ that we have just seen (start with what the valuation does to the propositional letters—the leaves of the parse tree—and work up the tree) is called **Eager evaluation** also known as **Strict evaluation**. But there are other ways of calculating that will give the same answer. One of them is the beguilingly named **Lazy evaluation** which we will now describe.

Consider the project of filling out a truth-table for the formula $A \wedge (B \vee (C \wedge D))$. One can observe immediately that any valuation (row of the truth-table) that makes 'A' false will make the whole formula false:

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0		0		0		0
0		0		0		1
0		0		1		0
0		0		1		1
0		1		0		0
0		1		0		1
0		1		1		0
0		1		1		1
1		0		0		0
1		0		0		1
1		0		1		0
1		0		1		1
1		1		0		0
1		1		0		1
1		1		1		0
1		1		1		1

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0	0	0		0		0
0	0	0		0		1
0	0	0		1		0
0	0	0		1		1
0	0	1		0		0
0	0	1		0		1
0	0	1		1		0
0	0	1		1		1
1		0		0		0
1		0		0		1
1		0		1		0
1		0		1		1
1		1		0		0
1		1		0		1
1		1		1		0
1		1		1		1

Now, in the remaining cases we can observe that any valuation that makes 'B' true will make the whole formula true.:

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0	0	0		0		0
0	0	0		0		1
0	0	0		1		0
0	0	0		1		1
0	0	1		0		0
0	0	1		0		1
0	0	1		1		0
0	0	1		1		1
1		0		0		0
1		0		0		1
1		0		1		0
1		0		1		1
1		1	1	0		0
1		1	1	0		1
1		1	1	1		0
1		1	1	1		1

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0	0	0		0		0
0	0	0		0		1
0	0	0		1		0
0	0	0		1		1
0	0	1		0		0
0	0	1		0		1
0	0	1		1		0
0	0	1		1		1
1		0		0		0
1		0		0		1
1		0		1		0
1		0		1		1
1	1	1	1	0		0
1	1	1	1	0		1
1	1	1	1	1		0
1	1	1	1	1		1

In the remaining cases any valuation that makes 'C' false will make the whole formula false.

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0	0	0		0		0
0	0	0		0		1
0	0	0		1		0
0	0	0		1		1
0	0	1		0		0
0	0	1		0		1
0	0	1		1		0
0	0	1		1		1
1	0	0	0	0	0	0
1	0	0	0	0	0	1
1		0		1		0
1		0		1		1
1	1	1	1	0		0
1	1	1	1	0		1
1	1	1	1	1		0
1	1	1	1	1		1

A	$\wedge$	$(B$	$\vee$	$(C$	$\wedge$	$D))$
0	0	0		0		0
0	0	0		0		1
0	0	0		1		0
0	0	0		1		1
0	0	1		0		0
0	0	1		0		1
0	0	1		1		0
0	0	1		1		1
1	0	0	0	0	0	0
1	0	0	0	0	0	1
1	0	0	0	1	0	0^*
1	1	0	1	1	1	1^*
1	1	1	1	0		0
1	1	1	1	0		1
1	1	1	1	1		0
1	1	1	1	1		1

The starred '0^*' and '1^*' are the only cases where we actually have to look at the truth-value of D.

These illustrations concern evaluation in languages whose expressions evaluate to truth-values. The idea originally arose in connection with langauges whose expressions evaluate to numbers or other data objects.

`if` $x \geq 0$ `then` $f(x)$ `else` $g(x)$.

Of course you evaluate lazily. No point in calculating both $f(x)$ and $g(x)$ when you are clearly going to need only one of them! First you evaluate x to see whether it is above or below 0 and then you do whichever of $f(x)$ and $g(x)$ that it turns out you need. Indeed it might not be merely *wasteful* to calculate both $f(x)$ and $g(x)$; it might not even be *possible*. It could be the case that the calculation of the one you don't need does not ever terminate, so you sit in a loop for ever.

Notice also in this connection that it might not be neccessary to *completely* evaluate x to ascertain which way to jump. If x is presented to me as a double-precision decimal number I have 12 decimal places to evaluate, but I will know already after evaluating the first of them whether x is positive or negative.

(The difference between functional and declarative programming languages is that a functional programming language allows you to define a function, whereas a declarative one allows you to tell the computer how to evaluate it.

3.3 Validity and Inference

Not of particular interest to linguists, tho' of great interest in Logic and philosophy.

We will need to know about the rules of inference when we come to Curry-Howard in chapter 5. I shall put them on the blackboard but not expect you to learn how to use them. This material is all covered *in extenso* in chchlectures.pdf, so i shall not put it here. This is one of those bullet-points i warned you about!

Exercise 7. *Abbreviate "Jack arrives late for lunch" etc etc., to single letters, and use these abbreviations to formalise the arguments below. (To keep things simple you can ignore the tenses!)*

1. *If Jill arrives late for lunch, she will be cross with Jack. Jack will arrive late. Therefore Jill will be cross with Jack.*

2. *If Jill arrives late for lunch, Jack will be cross with her. Jill will arrive late. Therefore Jill will be cross with Jack.*

3. *If Jill arrives late for lunch, Jack will be cross with her. Jack will arrive late. Therefore Jill will be cross with Jack.*

4. *If Jack arrives late for lunch, Jill will be cross with him. Jack will arrive late. Therefore Jill will be cross with Jack.*

5. *If George is guilty he'll be reluctant to answer questions; George is reluctant to answer questions. Therefore George is guilty.*

6. *If George is broke he won't be able to buy lunch; George is broke. Therefore George will not be able to buy lunch.*

7. *Assuming that the lectures are dull, if the text is not readable then Alfred will not pass.*

8. *If Logic is difficult Alfred will pass only if he concentrates.*

9. *If Alfred studies, then he receives good marks. If he does not study, then he enjoys college. If he does not receive good marks then he does not enjoy college. [Therefore Alfred receives good marks]*

10. *If Herbert can take the flat only if he divorces his wife then he should think twice. If Herbert keeps Fido, then he cannot take the flat. Herbert's wife insists on keeping Fido. If Herbert does not keep Fido then he will divorce his wife—at least if she insists on keeping Fido. [Therefore Herbert should think twice]*

11. *If Herbert grows rich, then he can take the flat. If he divorces his wife he will not receive his inheritance. Herbert will grow rich if he receives his inheritance. Herbert can take the flat only if he divorces his wife.*

12. *If God exists then He is omnipotent. If God exists then He is omniscient. If God exists then He is benevolent. If God can prevent evil then—if He knows that evil exists—then He is not benevolent if He does not prevent it. If God is omnipotent, then He can prevent evil. If God is omniscient then He knows that evil exists if it does indeed exist. Evil does not exist if God prevents it. Evil exists. [Therefore God does not exist]*

Must say something very briefly about rules of inference at this stage, if only to prepare us for Curry-Howard.

Chapter 4

Predicate (first-order) Logic

4.1 Towards First-Order Logic

The following puzzle comes from Lewis Carroll.

> Dix, Lang, Cole, Barry and Mill are five friends who dine together regularly. They agree on the following rules about which of the two condiments—salt and mustard—they are to have with their beef. (For some reason they always have beef?!)
>
> Formalise the following.
>
> 1. If Barry takes salt, then either Cole or Lang takes only *one* of the two condiments, salt and mustard (and *vice versa*). If he takes mustard then either Dix takes neither condiment or Mill takes both (and *vice versa*).
> 2. If Cole takes salt, then either Barry takes only *one* condiment, or Mill takes neither (and *vice versa*). If he takes mustard then either Dix or Lang takes both (and *vice versa*).
> 3. If Dix takes salt, then either Barry takes neither condiment or Cole takes both (and *vice versa*). If he takes mustard then either Lang or Mill takes neither (and *vice versa*).
> 4. If Lang takes salt, then either Barry or Dix takes only *one* condiment (and *vice versa*). If he takes mustard then either Cole or Mill takes neither (and *vice versa*).
> 5. If Mill takes salt, then either Barry or Lang takes both condiments (and *vice versa*). If he takes mustard then either Cole or Dix takes only one (and *vice versa*).

As I say, this puzzle comes from Lewis Carroll. The task he sets is to ascertain whether or not these conditions can in fact be met. I do not know the answer, and it would involve a lot of hand-calculation—which of course is the point! I

don't suppose for a moment that you want to crunch it out (I haven't done it and I have no intention of doing it—I have a life) but it's a good idea to think a bit about some of the preparatory work.

The way to do this would be to create a number of propositional letters, one each to abbreviate each of the assorted assertions "Barry takes salt", "Mill takes mustard" and so on. How many propositional letters will there be? Obviously 10, co's you can count them: each propositional letter corresponds to a choice of one of {Dix, Lang, Cole, Barry, Mill}, and one choice of {salt, mustard} and $2 \times 5 = 10$. We could use propositional letters 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x' and 'y'. This is probably what you have done. But notice that using ten different letters—*mere* letters—in this way fails to capture certain relations that hold between them. Suppose they were arranged like:

'p': Barry takes salt	'u': Barry takes mustard
'q': Mill takes salt	'v': Mill takes mustard
'r': Cole takes salt	'w': Cole takes mustard
's': Lang takes salt	'x': Lang takes mustard
't': Dix takes salt	'y': Dix takes mustard

Then we see that two things in the same row are related to each other in a way that they aren't related to things in other rows; ditto things in the same column. This subtle information cannot be read off just from the letters 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x' and 'y' themselves. That is to say, there is *internal structure* to the propositions "Mill takes salt" etc, that is not captured by reducing each to one letter.

The time has come to do something about this.

A first step would be to replace all of 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x' and 'y' by things like 'ds' and 'bm' which will mean 'Dix takes salt' and 'Barry takes mustard'. Then we can build truth-tables and do other kinds of hand-calculation as before, this time with the aid of a few mnemonics. If we do this, the new things like 'bm' are really just propositional letters as before, but slightly bigger ones. The internal structure is visible to *us*—*we* know that 'ds' is really short for 'Dix takes salt' but it is not visible to the logic. The logic regards 'ds' as a single propositional letter. To do this satisfactorily we must do it in a way that makes the internal structure explicit.

4.2 First-order Logic

What we need is **Predicate Logic**. It's also called **First-Order Logic** and sometimes **Predicate Calculus**. In this new pastime we don't just use suggestive mnemonic symbols for propositional letters but we open up the old propositional letters that we had, and find that they have internal structure. "Romeo loves Juliet" will be represented not by a single letter 'p' but by something with suggestive internal structure like $L(r, j)$. We use capital Roman letters as

predicate symbols (also known as **relation** symbols). In this case the letter 'L' is a *binary* relation symbol, co's it relates *two* things. The 'r' and the 'j' are **arguments** to the relation symbol. They are **constants** that denote the things that are related to each other by the (meaning of the) relation symbol.

The obvious way to apply this to Lewis Carroll's problem on page 61 is to have a two-place predicate letter 'T', and symbols 'd', 'l', 'm', 'b' and 'c' for Dix, Lang, Mill, Barry and Cole, respectively. I am going to write them in lower case beco's we keep upper case letters for predicates—relation symbols. And we'd better have two constants for the condiments salt and mustard: 's' for salt and—oops!—can't use 'm' for mustard co's we've already used that letter for Mill! Let's use 'u'. So, instead of 'p' and 'q' or even 'ds' etc we have:

'$T(b, s)$': Barry takes salt	'$T(b, u)$': Barry takes mustard
'$T(m, s)$': Mill takes salt	'$T(m, u)$': Mill takes mustard
'$T(c, s)$': Cole takes salt	'$T(c, u)$': Cole takes mustard
'$T(l, s)$': Lang takes salt	'$T(l, u)$': Lang takes mustard
'$T(d, s)$': Dix takes salt	'$T(d, u)$': Dix takes mustard

And now the symbolism we are using makes it clear what it is that two things in the same row have in common, and what it is that two things in the same column have in common.

I have used here a convention that you always write the relation symbol first, and then put its arguments after it, enclosed within parentheses: we don't write '$m\,T\,s$'. However identity is a special case and we do write "Hesperus = Phosphorous" (the two ancient names for the evening star and the morning star) and when we write the relation symbol between its two arguments we say we are using **infix** notation. (Infix notation only makes sense if you have two arguments not three: If you had three aguments where would you put the relation symbol if not at the front?)

Here's an exercise I found in [8]. (See p 60.)

> If Herbert can take the flat only if he divorces his wife then he should think twice. If Herbert keeps Fido, then he cannot take the flat. Herbert's wife insists on keeping Fido. If Herbert does not keep Fido then he will divorce his wife—at least if she insists on keeping Fido.

You will need constant names 'h' for Herbert, 'f' for Fido, and 'w' for the wife. You will also need a few binary relation symbols: K for *k*eeps, as in "Herbert keeps Fido". Some things might leave you undecided. Do you want to have a binary relation symbol 'T' for *t*akes, as in $T(h, f)$ meaning "Herbert takes the flat"? If you do you will need a constant symbol 'f' to denote the flat. Or would you rather go for a unary relation symbol 'TF' to be applied to Herbert? No-one else is conjectured to take the flat after all ... If you are undecided between these, all it means is that you have discovered the wonderful flexibility of predicate calculus.

Rule of thumb: We use Capital Letters for *properties* and *relations*; on the whole we use small letters for *things*. (We do tend to use small letters for functions too). The capital letters are called **relational symbols** or **predicate letters** and the lower case letters are called **constants**.

Surely these are a bit hard, at this stage?

Exercise 8. *Formalise the following, using a lexicon of your choice*

1. *Romeo loves Juliet; Juliet loves Romeo.*
2. *Balbus loves Julia. Julia does not love Balbus. What a pity.* [1]
3. *Fido sits on the sofa; Herbert sits on the chair.*
4. *Fido sits on Herbert.*
5. *If Fido sits on Herbert and Herbert is sitting on the chair then Fido is sitting on the chair.*
6. *The sofa sits on Herbert. [just because something is absurd doesn't mean it can't be said!]*
7. *Alfred drinks more whisky than Herbert; Herbert drinks more whisky than Mary.*
8. *John scratches Mary's back. Mary scratches her own back.*

   [A binary relation can hold between a thing and itself. It doesn't have to relate two distinct things.]

4.3 The Syntax of First-order Logic

Explain the gadgetry: constants, individual variables, predicate letters and function letters. Boolean connectives, then quantifiers.

Linguists will probably be able to understand the concept of a **free variable** and a **binder** Then you can stick them with Curry-Howard.

All the apparatus for constructing formulæ in propositional logic works too in this new context: If A and B are formulæ so are $A \vee B$, $A \wedge B$, $\neg A$ and so on. However we now have new ways of creating formulæ, new gadgets which we had better spell out:

There is really an abuse of notation here: we should use quasi-quotes ...

4.3.1 Constants and variables

Constants tend to be lower-case letters at the start of the Roman alphabet ('a', 'b' ...) and variables tend to be lower-case letters at the end of the alphabet ('x', 'y', 'z' ...). Since we tend to run out of letters we often enrich them with subscripts to obtain a larger supply: 'x_1' etc.

[1] I found this in a latin primer: *Balbus amat Juliam; Julia non amat Balbum* ....

4.3.2 Predicate letters

These are upper-case letters from the Roman alphabet, usually from the early part: 'F' 'G' .... They are called *predicate* letters because they arise from a programme of formalising reasoning about predicates and predication. '$F(x, y)$' could have arisen from 'x is fighting y'. Each predicate letter has a particular number of terms that it expects; this is the **arity** of the letter. 'loves' has arity 2 (it is binary) 'sits-on' is binary too. If we feed it the correct number of terms—so we have an expression like $F(x, y)$—we call the result an **atomic formula.**

The equality symbol '=' is a very special predicate letter: you are not allowed to reinterpret it the way you can reinterpret other predicate letters. (The Information Technology fraternity say of strings that cannot be assigned meanings by the user that they are **reserved**). It is said to be **part of the logical vocabulary**. The equality symbol '=' is the only relation symbol that is reserved. In this respect it behaves like '$\wedge$' and '$\forall$' and the connectives, all of which are reserved in this sense.

Unary predicates have one argument, **binary** predicates have two; n-**ary** have n. Similarly functions.

Atomic formulæ can be treated the way we treated literals in propositional logic: we can combine them together by using '$\wedge$' '$\vee$' and the other connectives.

lots of illustrations here please

Finally we can **bind** variables with **quantifiers**. There are two: $\exists$ and $\forall$. We can write things like

$(\forall x)F(x)$: Everything is a frog;
$(\forall x)(\forall y)L(x, y)$ Everybody loves everyone

To save space we might write this second thing as

$$(\forall xy)L(x, y)$$

The syntax for quantifiers is variable-preceded-by quantifier enclosed in brackets, followed by stuff inside brackets:

$$(\exists x)(\ldots) \text{ and } (\forall y)(\ldots)$$

We sometimes omit the pair of brackets to the right of the quantifier when no ambiguity is caused thereby.

The difference between variables and constants is that you can bind variables with quantifiers, but you can't bind constants. The meaning of a constant is fixed.

... free

complete this explanation; quantifiers are connectives too

For example, in a formula like

$$(\forall x)(F(x) \to G(x))$$

the letter 'x' is a variable: you can tell because it is bound by the universal quantifier. The letter 'F' is not a variable, but a predicate letter. It is not bound

by a quantifier, and cannot be: the syntax forbids it. In a first-order language you are not allowed to treat predicate letters as variables: you may not bind them with quantifiers. Binding predicate letters with quantifiers (treating them as variables) is the tell-tale sign of **second-order** Logic.

We also have

4.3.3 Function letters

These are lower-case Roman letters, typically 'f', 'g', 'h' .... We apply them to variables and constants, and this gives us **terms**: $f(x)$, $g(a, y)$ and suchlike. In fact we can even apply them to terms: $f(g(a, y))$, $g(f(g(a, y), x))$ and so on. So a term is either a variable or a constant or something built up from variables-and-constants by means of function letters. What is a function? That is, what sort of thing do we try to capture with function letters? We have seen an example: *father-of* is a function: you have precisely one father; *son-of* is not a function. Some people have more than one, or even none at all.

4.4 Warning: Scope ambiguities

"All that glisters is not gold"

is not

$$(\forall x)(\text{glisters}(x) \to \neg\text{gold}(x))$$

and

"All is not lost"

is not

$$(\forall x)(\neg\text{lost}(x))$$

The difference is called a matter of **scope**. 'Scope'? The point is that in "$(\forall x)(\neg \ldots)$" the "scope" of the '$\forall x$' is the whole formula whereas in the '$\neg(\forall x)(\ldots)$ it isn't.

It is a curious fact that humans using ordinary language can be very casual about getting the bits of the sentence they are constructing in the right order so that each bit has the right scope. We often say things that we don't literally mean. On the receiving end, when trying to read things like $(\forall x)(\exists y)(x$ loves $y)$ and $(\exists y)(\forall x)(x$ loves $y)$, people often get into tangles because they try to resolve their uncertainty about the scope of the quantifiers by looking at the overall meaning of the sentence rather than by just checking to see which order they are in!

4.5 First-person and third-person

Natural languages have these wonderful gadgets like 'I' and 'you'. These connect the denotation of the expressions in the language to the *users* of the language. This has the effect that if A is a formula that contains one of these pronouns then different tokens of A will have different meanings! This is completely unheard-of in the languages of formal logic: it's formula *types* that the semantics gives meanings to, not formula-*tokens*. Another difference between formal languages and natural languages is that the users of formal languages (us!) do not belong to the world described by the expressions in those languages. (Or at least if we do then the semantics has no way of expressing this fact.) Formal languages do have *variables*, and variables function grammatically like pronouns, but the pronouns they resemble are *third person* pronouns not first- or second-person pronouns. This is connected with their use in science: no first- or second-person perspective in science. This is because science is agent/observer-invariant. Connected to *objectivity*. The languages that people use/discuss in Formal Logic do not deal in any way with speech acts/formula tokens: only with the types of which they are tokens.

Along the same lines one can observe that in the formal languages of logic there is no *tense* or *aspect* or *mood*.

4.6 Some exercises to get you started

Exercise 9.
Render the following fragments of English into predicate calculus, using a lexicon of your choice.

This first bunch involve monadic predicates only and no nested quantifiers.

1. *Every good boy deserves favour; George is a good boy. Therefore George deserves favour.*
2. *All cows eat grass; Daisy eats grass. Therefore Daisy is a cow.*
3. *Socrates is a man; all men are mortal. Therefore Socrates is mortal.*
4. *Daisy is a cow; all cows eat grass. Therefore Daisy eats grass.*
5. *Daisy is a cow; all cows are mad. Therefore Daisy is mad.*
6. *No thieves are honest; some dishonest people are found out. Therefore Some thieves are found out.*
7. *No muffins are wholesome; all puffy food is unwholesome. Therefore all muffins are puffy.*
8. *No birds except peacocks are proud of their tails; some birds that are proud of their tails cannot sing. Therefore some peacocks cannot sing.*

9. *A wise man walks on his feet; an unwise man on his hands. Therefore no man walks on both.*

10. *No fossil can be crossed in love; an oyster may be crossed in love. Therefore oysters are not fossils.*

11. *All who are anxious to learn work hard; some of these students work hard. Therefore some of these students are anxious to learn.*

12. *His songs never last an hour. A song that lasts an hour is tedious. Therefore his songs are never tedious.*

13. *Some lessons are difficult; what is difficult needs attention. Therefore some lessons need attention.*

14. *All humans are mammals; all mammals are warm blooded. Therefore all humans are warm-blooded.*

15. *Warmth relieves pain; nothing that does not relieve pain is useful in toothache. Therefore warmth is useful in toothache.*

16. *Guilty people are reluctant to answer questions;*

17. *Louis is the King of France; all Kings of France are bald. Therefore Louis is bald;*

18. *Anyone who plays Aussie rules runs 20km in 90 minutes; anyone who can run 20km in 90 minutes is a serious athlete; you have to be a thug to play Aussie rules. Therefore at least some serious athletes are thugs.*

Exercise 10. *Render the following into Predicate calculus, using a lexicon of your choice. These involve nestings of more than one quantifier, polyadic predicate letters, equality and even function letters.*

1. *Anyone who has forgiven at least one person is a saint.*

2. *Nobody in the logic class is cleverer than everybody in the history class.*

3. *Everyone likes Mary—except Mary herself.*

4. *Jane saw a bear, and Roger saw one too.*

5. *Jane saw a bear and Roger saw it too.*

6. *Some students are not taught by every teacher;*

7. *No student has the same teacher for every subject.*

8. *Everybody loves my baby, but my baby loves nobody but me.*

Exercise 11. *These involve nested quantifiers and dyadic predicates*
Match up the formulæ on the left with their English equivalents on the right.

(i) $(\forall x)(\exists y)(x \text{ loves } y)$	*(a) Everyone loves someone*
(ii) $(\forall y)(\exists x)(x \text{ loves } y)$	*(b) There is someone everyone loves*
(iii) $(\exists y)(\forall x)(x \text{ loves } y)$	*(c) There is someone that loves everyone*
(iv) $(\exists x)(\forall y)(x \text{ loves } y)$	*(d) Everyone is loved by someone*

Exercise 12. *Render the following pieces of English into Predicate calculus, using a lexicon of your choice.*

1. *Everyone who loves is loved;*
2. *Everyone loves a lover;*
3. *The enemy of an enemy is a friend*
4. *The friend of an enemy is an enemy*
5. *Any friend of George's is a friend of mine*
6. *Jack and Jill have at least two friends in common*
7. *Two people who love the same person do not love each other.*
8. *None but the brave deserve the fair.*
9. *If there is anyone in the residences with measles then anyone who has a friend in the residences will need a measles jab.*
10. *No two people are separated by more than six steps of aquaintanceship.*

This next batch involves nested quantifiers and dyadic predicates and equality.

Exercise 13. *Render the following pieces of English into Predicate calculus, using a lexicon of your choice.*

1. *There are two islands in New Zealand;*
2. *There are* three[2] *islands in New Zealand;*
3. *tf knows (at least) two pop stars;*

   (You must resist the temptation to express this as a relation between tf and a plural object consisting of two pop stars coalesced into a kind of plural object like Jeff Goldblum and the Fly. You will need to use '=', the symbol for equality.)
4. *You are loved only if you yourself love someone [other than yourself!];*
5. *God will destroy the city unless there are (at least) two righteous men in it;*

[2]The third is Stewart Island

6. *There is at most one king of France;*
7. *I know no more than two pop stars;*
8. *There is precisely one king of France;*
9. *I know three FRS's and one of them is bald;*
10. *Brothers and sisters have I none; this man's father is my father's son.*
11. ** Anyone who is between a rock and a hard place is also between a hard place and a rock.*

4.7 Transitive, reflexive etc

Armed with this new language we can characterise some important properties:

- A relation R is **transitive** if $\forall x \forall y \forall z((R(x,y) \wedge R(y,z)) \rightarrow R(x,z))$
- A relation R is **symmetrical** if $\forall x \forall y(R(x,y) \longleftrightarrow R(y,x))$
- $(\forall x)(R(x,x))$ says that R is **reflexive**; and $(\forall x)(\neg R(x,x))$ says that R is **irreflexive**.
- A relation that is transitive, reflexive and symmetrical is an **equivalence relation**.

The binary relation "full sibling of" is symmetric, and so is the binary relation "half-sibling of". However, "full sibling of" is transitive whereas "half-sibling of" is not.

4.8 Russell's Theory of Descriptions

'There is precisely one King of France and he is bald' can be captured satisfactorily in predicate calculus/first-order logic by anyone who has done the preceding exercises. We get

$$(\exists x)((K(x) \wedge (\forall y)(K(y) \rightarrow y = x) \wedge B(x))) \qquad \text{(A)}$$

Is the formulation we arrive at the same as what we would get if we were to try to capture (B)?

"The King of France is bald" (B)

Well, if (A) holds then the unique thing that is King of France and is bald certainly sounds as if it is going to be *the* King of France, and it is bald, and so if (A) is true then the King of France is bald. What about the converse (or rather its contrapositive)? If (A) is false, must it be false that the King of France is bald? It might be that (A) is false because there is more than one King of France. In those circumstances one might want to suspend judgement on

(B) on the grounds that we don't yet know which of the two prospective Kings of France is the real one, and one of them might be bald. Indeed they might *both* be bald. Or we might simply feel that we can't properly use expressions like "the King of France" at all unless we know that there is precisely one. If there isn't precisely one then allegations about the King of France simply lack truth-value—or so it is felt.

What's going on here is that we are trying to add to our language a new quantifier, a thing like '$\forall$' or '$\exists$'—which we could write '$(Qx)(\ldots)$ so that '$(Qx)(F(x))$' is true precisely when the King of France has the property F. The question is: can we translate expressions that *do* contain this new quantifier into expressions that *do not* contain it? The answer depends on what truth-value you attribute to (B) when there is no King of France. If you think that (B) is false in these circumstances then you may well be willing to accept (A) as a translation of it, but you won't if you think that (B) lacks truth-value.

Is this the first place where we talk about translations?

If you think that (A) is the correct formalisation of (B), and that in general you analyse "The F is G" as

$$(\exists x)((F(x) \wedge (\forall y)(F(y) \rightarrow y = x) \wedge G(x))) \qquad \text{(C)}$$

then you are a subscriber to **Russell's theory of descriptions.**

4.9 First-order and second-order

We need to be clear right from the outset about the difference between first-order and second-order. In first-order languages predicate letters and function letters cannot be variables. The idea is that the variables range only over individual inhabitants of the structures we consider, not over sets of them or properties of them. This idea—put like that—is clearly a semantic idea. However it can be (and must be!) given a purely syntactic description.

In propositional logic every wellformed expression is something which will evaluate to a truth-value: to `true` or to `false`. These things are called **booleans** so we say that every wellformed formula of propositional logic is of type `bool`.

Explain this idiom

In first order logic it is as if we have looked inside the propositional letters 'p', 'q' etc. that were the things that evaluate to `true` or to `false`, and have discovered that the letter—as it might be—'p' actually, on closer inspection, turned out to be '$F(x, y)$'. To know the truth-value of this formula we have to know what objects the variables 'x' and 'y' point to, and what binary relation the letter 'F' represents.

First-order logic extends propositional logic in another way too. Blah quantifiers.

Many-sorted

If you think the universe consists of only one kind of stuff then you will have only one domain of stuff for your variables to range over. If you think the universe has two kinds of stuff (for example, you might think that there are two kinds of

stuff: the mental and the physical) then you might want two domains for your variables to range over.

If you are a cartesian dualist trying to formulate a theory of mind in first-order logic you would want to have variables of two *sorts*: for mental and for physical entities.

4.10 Semantics for first-order logic

This section is admittedly a bit scary and if you suffer from *mathsangst* you can skip it, beco's it isn't strictly neccessary for any material that comes later. However, it is fairly central, and if this were an examinable course I would insist on you coming to grips with it.

It may even be (if our progress to too slow) that I won't even get round to lecturing it.

In this section we develop the ideas of truth and validity (which we first saw in the case of propositional logic) in the rather more complex setting of predicate logic.

We are going to say what it is for a formula to be **true** in a structure. We will achieve this by doing something rather more general. What we will give is—for each language $\mathcal{L}$—a definition of what it is *for a formula of $\mathcal{L}$ to be true in a structure*. Semantics is a relation not so much between an expression and a structure as between a *language* and a structure. [Slogan: semantics for an expression cannot be done in isolation.]

We know what expressions are, so what is a structure? It's a set with knobs on. You needn't be alarmed here by the sudden appearance of the word 'set'. You don't need to know any fancy set theory to understand what is going on. The set in question is called the *carrier set*, or *domain*. One custom in mathematics is to denote structures with characters in uppercase FRAKTUR font, typically with an '$\mathfrak{M}$'.

The obvious examples of structures arise in mathematics and can be misleading and in any case are not really suitable for our expository purposes here. We can start off with the idea that a structure is a set-with-knobs on. Here is a simple example that cannot mislead anyone.

The carrier set is the set {Beethoven, Handel, Domenico Scarlatti} and the knobs are (well, *is* rather than *are* because there is only one knob in this case) the binary relation *is-the-favourite-composer-of*. We would obtain a different structure by adding a second relation: *is-older-than* perhaps.

If we are to make sense of the idea of an expression being true in a structure then the structure must have things in it to match the various gadgets in the language to which the expression belongs. If the expression contains a two-place relation symbol 'loves' then the structure must have a binary relation on it to correspond. This information is laid down in the **signature**. The signature of the structure in the composers example above has one binary relation symbol

and three constant symbols; the signature of set theory is equality plus one binary predicate; the signature of the language of first-order Peano arithmetic has slots for one unary function symbol, one nullary function symbol (or constant) and equality.

Let's have some illustrations, at least situations where the idea of a signature is useful.

- Cricket and baseball resemble each other in a way that cricket and tennis do not. One might say that cricket and baseball have the same signature. Well, more or less! They can be described by giving different values to the same set of parameters.
- It has been said that a French farce is a play with four characters, two doors and one bed. This *aperçu* is best expressed by using the concept of signature.
- Perhaps when you were little you bought mail-order kitsets that you assembled into things. When your mail-order kitset arrives, somewhere buried in the polystyrene chips you have a piece of paper (the "manifest") that tells you how many objects you have of each kind, but it does not tell you what to do with them. Loosely, the manifest is the *signature* in this example. Instructions on what you do with the objects come with the *axioms* (instructions for assembly).
- Recipes correspond to theories: lists of ingredients to signatures.

Structure	**Signature**	**Axioms**
French Farce	4 chars, 2 doors 1 bed	Plot
Dish	Ingredients	recipe
Kitset	list of contents	Instructions for assembly
cricket/baseball	innings, catches, etc	rules
tennis/table tennis		

It is now time to tackle the semantics of first-order logic. This time the function we define from the syntax will give back not meaning but truth-values. Still, the machinery is very similar.

We will need the idea of *valuation* from the semantics of propositional logic, definition 2 page 55. We will need it, but we have to do some preparatory work first.

We start with the idea of a **structure for a language**.

4.10.1 The Domain

The first thing to settle is what our universe of discourse is to be. In technical jargon, we have to decide what the variables in our language are going to range over. The things the variables range over are the things that we deem to exist in the piece of semantical theatre we are embarking on. The universe of discourse

is often referred to as the *domain* or as the *carrier set*. The locution 'carrier set' alludes to the fact that the domain *carries* the semantics—everything is built on it. Let us use the capital Roman letter 'D' to denote the domain.

4.10.2 Interpretations

Once we have decided what is is we are going to be talking about, we are in a position to decide what the meanings of the relation and function symbols in our language are to be. If our language contains a binary relation symbol such as '$<$', and we have decided what objects our variables are to range over, then the interpretation of the symbol '$<$' will have to be a binary relation holding between (some or all of) those objects. [close examination of the syntax will have told us that '$<$' is a binary relation symbol rather than a variable or a propositional constant].

The interpretation can/should be thought of as a function that takes pieces of syntax (such as the '$<$' symbol) as arguments and gives back as values suitable bits of the domain D.

The symbol '$=$' is a reserved word: its interpretation must be the equality relation on D. It can never be anything else. What, never? Well, hardly ever. If '$=$' points to anything other than the equality relation on D we say that the interpretation is **nonstandard**.

4.10.3 Assignment Functions

Think of them as the states of a program. (Beware, the connection with the idea of state of a finite state machine, altho' real enough, is pretty tenuous and—at this stage—is merely misleading.)

An assignment function f is a function from variables to elements of D: it tells you what the values (in D) are of the variables of $\mathcal{L}$. Assignment functions are sometimes taken to be partial, sometimes taken to be total. It doesn't much matter ...

How do we write the value of f (the thing in D) that f gives to the variable 'x'? We'd better not write it "$f(x)$", because that expression points to whatever it is to which the function f (the thing pointed to by the letter 'f') sends the thing pointed to by the letter 'x'. That's not what we want. Our function f here does not point to something that acts on things pointed to by the variable 'x'; it points to something that acts on the variable 'x' itself! To make this clear we should really describe the behaviour of an assignment function f by means of a variable that ranges over variables.

Anyway we now need a **satisfaction relation.** This holds between assignment functions and expressions of our language $\mathcal{L}$. To illustrate with an example that is as simple as possible. What would it be for the assignment function f to satisfy the formula '$x < y$'? Well, the interpretation has told us which things in D are related by the interpretation of '$<$'. We say that:

f satisfies '$x < y$' if the thing-to-which-f-sends-the-variable-'x' stands

> in the relation-which-is-the-interpretation-of-'$<$' to the thing-to-which-f-sends-the-variable-'y'.

Similarly for any other atomic formula.

What about compound ("molecular") formulæ? Easy, we define what it is for an assignment function to satisfy a compound formula by a a process known variously as *compositional* (if you are a linguist) or *recursive* (if you are a logician). We have clauses like

> f satisfies the conjunction of two formulæ if and only if it satisfies both conjuncts.
>
> f satisfies the disjunction of two formulæ if and only if it satisfies at least one disjunct.

These two clauses were stated with great care. I did *not* write

> f satisfies $A \wedge B$ if and only if it satisfies A and satisfies B. (1)

To write such a thing would not be legitimate, at least in the absence of certain linguistic conventions which we have not yet set up (and probably won't). For (1) to be squeaky-clean the letters 'A' and 'B' would have to be variables ranging over formulæ. OK, we say, we hereby let 'A' and 'B' be variables ranging over formulæ. But now we have a problem with the symbol '$\wedge$'. Hitherto '$\wedge$' has been a symbol that we put between two formula-tokens to obtain a new formula token. Here it is being put between two variables that range over formulæ. This harks back to the discussion earlier, in the propositional case, and we refer the reader back to that passage. See page 56.

type-token

4.10.4 The quantifiers and the satisfaction relation

The tricky cases involve the two quantifiers. When do we want to say that an assignment function f satisfies an expression like '$(\exists x)A$'? Well, it's going to depend on what f does to the variable 'x'. [...]

4.10.5 Truth (of a formula in a model)

If a formula is satisfied by all assignment functions operating under the rubric of an interpretation we say that the formula is **true**—according to that interpretation.

Notice the way in which the difference between first-order and second order is played out in this process: Things that you can't quantify over are settled at the stage where we define the interpretation; quantifiable variables are dealt with later on, by the assignment functions.

4.11 Expressive Power

The expressive power [of a language] is a very important idea in Logic which is probably of interest to linguists too. It sounds like a circular endeavour—or at least one without a firm foundation—because how is one to say what a language can express except by use of language? However, one must not be discouraged.

Let us start with a very simple observation: *the language of propositional logic is not regular*.

This because if A and B are expressions of the language of propositional logic then so is $\ulcorner(A \vee B)\urcorner$, so that grammatical expressions of this language can have arbitrarily many left-hand brackets open at any one time, so by the pumping lemma the language cannot be regular.

However, the language of propositional logic is context-free. What about the language of first-order logic? Apparently this depends on whether or not one is allowed to reuse variables. Do you want to allow:

$$(\forall x)(F(x) \to G(x)) \wedge (\exists x)(P(x))$$

as wellformed? Or should we insist on replacing the 'x' in one of the conjuncts by a 'y' to make the formula less confusing? I have been told that if you want to forbid the reuse of variables then the resulting grammar is not context-free.

Of slightly more interest in the idea of *semantic closure*. This harks back to the language/metalanguage distinction from page 14.

> "What i am now saying is false"

It is a simple consequence of the liar paradox that no language can completely describe its own semantics. We say: no language can be semantically closed. Natural languages of course *are* semantically closed, but then they allow us to do nasty things like the Liar Paradox which upset logicians and shouldn't be allowed to happen.

If we want to preclude disasters like the Liar Paradox then the semantics for a language has to be provided in a metalanguage. And the semantics for the metalanguage has to be provided in a metametalanguage and so on... transfinitely!

Perhaps say something about **completeness theorems.**

4.12 Enhanced syntax

branching quantifiers, "Independence-friendly logic"

modal operators.

Plurals. see [12].

Predicate modifiers

A predicate modifier is a second-order function letter. They are sometimes called *adverbial* modifiers. For example we might have a predicate modifier $\mathcal{V}$ whose intended meaning is something like "a lot" or "very much', so that if $L(x, y)$ was our formalisation of x *loves* y then $\mathcal{V}(L(x, y))$ means x *loves* y *very much*. In the old grammar books I had at school we were taught that adjectives had three forms: *simple* ("cool") *comparative* ("cooler") and *superlative* ("coolest"). These could be represented in higher order logic by two predicate modifiers. The '**-er**' (comparative) modifier takes a one-place predicate letter and returns a two-place predicate letter. The '**-est**' (superlative) operator takes a one-place predicate letter and returns another one-place predicate letter. (In fact, by using Russell's theory of descriptions we can see how to define the superlative predicate in terms of the comparative. This makes a useful exercise.)

Another predicate modifier is *too*.

> No woman can be too thin or too rich.

We will not consider them further.

4.13 Perhaps a brief look at game semantics

This will almost certainly not be written up. You'll just have to stay awake during the lecture!

Chapter 5

Curry-Howard

(One could say quite a lot about type-distinctions in artificial languages and in natural languages.)

Must introduce the slang expression "propositions-as-types"

The Curry-Howard trick is to exploit the possibility of using the letters 'A', 'B' *etc.* to be dummies not just for propositions but for sets. This means reading the symbols '$\rightarrow$', '$\wedge$', '$\vee$' etc. as symbols for operations on sets as well as on formulæ. The ambiguity we will see in the use of '$A \rightarrow B$' is quite different from the ambiguity arising from the two uses of the word 'tank'. Those two uses are completely unrelated. In contrast the two uses of the arrow in '$A \rightarrow B$' have a deep and meaningful relationship. The result is a kind of cosmic pun. Here is the simplest case.

Altho' we use it as a formula in propositional logic, the expression '$A \rightarrow B$' is used by various mathematical communities to denote the set of all functions from A to B. To understand this usage you don't really need to have decided whether your functions are to be functions-in-intension or functions-in-extension; either will do. The ideas in play here work quite well at an informal level. A function from A to B is a thing such that when you give it a member of A it gives you back a member of B.

5.1 Decorating Formulæ

5.1.1 The rule of $\rightarrow$-elimination

Consider the rule of $\rightarrow$-elimination

$$\frac{A \qquad A \rightarrow B}{B} \rightarrow\text{-elim} \tag{5.1}$$

If we are to think of A and B as sets then this will say something like "If I have an A (abbreviation of "if i have a member of the set A") and an $A \rightarrow B$ then I have a B". So what might an $A \rightarrow B$ (a member of $A \rightarrow B$) be? Clearly $A \rightarrow B$ must be the set of functions that give you a member of B when fed a member of A. Thus we can decorate 5.1 to obtain

$$\frac{a : A \qquad f : A \to B}{f(a) : B} \to\text{-elim} \tag{5.2}$$

which says something like: "If a is in A and f takes As to Bs then $f(a)$ is a B.[1] This gives us an alternative reading of the arrow: '$A \to B$' can now be read ambiguously as either the conditional "if A then B" (where A and B are propositions) or as a notation for the set of all functions that take members of A and give members of B as output (where A and B are sets).

These new letters preceding the colon sign are **decorations**. The idea of Curry-Howard is that we can decorate *entire proofs*—not just individual formulæ—in a uniform and informative manner.

We will deal with $\to$-int later. For the moment we will look at the rules for $\wedge$.

5.2 Rules for $\wedge$

5.2.1 The rule of $\wedge$-introduction

Consider the rule of $\wedge$-introduction:

$$\frac{A \qquad B}{A \wedge B} \wedge\text{-int} \tag{5.1}$$

If I have an A and a B then I have a . . . ? thing that is both A and B? No. If I have one apple and I have one banana then I don't have a thing that is both an apple and a banana; what I do have is a sort of plural object that I suppose is a pair of an apple and a banana. (By the way I hope you are relaxed about having compound objects like this in your world. Better start your breathing exercises *now*.) The thing we want is called an **ordered pair**: $\langle a, b\rangle$ is the ordered pair of a and b. So the decorated version of 5.1 is

$$\frac{a : A \qquad b : B}{\langle a, b\rangle : A \times B} \wedge\text{-int} \tag{5.2}$$

Say something about how we use $\times$ here . . .

What is the ordered pair of a and b? It might be a kind of funny plural object, like the object consisting of all the people in this room, but it's safest to be entirely *operationalist* about it: all you know about ordered pairs is that there is a way of putting them together and a way of undoing the putting-together, so you can recover the components. Asking for any further information about what they *are* is not cool: they are what they do. Be doo be doo. That's operationalism for you.

[1] So why not write this as '$a \in A$' if it means that a is a member of A? There are various reasons, some of them cultural, but certainly one is that here one tends to think of the denotations of the capital letters 'A' and 'B' and so on as predicates rather than sets.

5.2.2 The rule of ∧-elimination

If you can do them up, you can undo them: if I have a pair-of-an-A-and-a-B then I have an A and I have a B.

$$\frac{\langle a, b\rangle : A \wedge B}{a : A} \qquad\qquad \frac{\langle a, b\rangle : A \wedge B}{b : B}$$

$A \times B$ is the set $\{\langle a, b\rangle : a \in A \wedge b \in B\}$ of[2] pairs whose first components are in A and whose second components are in B. $A \times B$ is the **Cartesian product** of A and B.

(Do not forget that it's $A \times B$ not $A \cap B$ that we want. A thing in $A \cap B$ is a thing that is both an A and a B: it's not a pair of things one of which is an A and the other a B; remember the apples and bananas above.)

5.2.1 Rules for ∨

To make sense of the rules for ∨ we need a different gadget.

$$\frac{A}{A \vee B} \qquad\qquad \frac{B}{A \vee B}$$

If I have a thing that is an A, then I certainly have a thing that is either an A or a B—namely the thing I started with. And in fact I know which of A and B it is—it's an A. Similarly If I have a thing that is a B, then I certainly have a thing that is either an A or a B—namely the thing I started with. And in fact I know which of A and B it is—it's a B.

Just as we have cartesian product to correspond with ∧, we have **disjoint union** to correspond with ∨. This is not like the ordinary union you may remember from school maths. You can't tell by looking at a member of $A \cup B$ whether it got in there by being a member of A or by being a member of B. After all, if $A \cup B$ is $\{1, 2, 3\}$ it could have been that A was $\{1, 2\}$ and B was $\{2, 3\}$, or the other way round. Or it might have been that A was $\{2\}$ and B was $\{1, 3\}$. Or they could both have been $\{1, 2, 3\}$! We can't tell. However, with disjoint union you *can* tell.

To make sense of disjoint union we need to rekindle the idea of a *copy* from section **??**. The disjoint union $A \sqcup B$ of A and B is obtained by making copies of everything in A and marking them with wee flecks of *pink* paint and making copies of everything in B and marking them with wee flecks of *blue* paint, then putting them all in a set. We can put this slightly more formally, now that we have the concept of an ordered pair: $A \sqcup B$ is

$$(A \times \{\texttt{pink}\}) \quad \cup \quad (B \times \{\texttt{blue}\}),$$

where `pink` and `blue` are two arbitrary labels.

(Check that you are happy with the notation: $A \times \{\texttt{pink}\}$ is the set of all ordered pairs whose first component is in A and whose second component is in

[2] If you are less than 100% happy about this curly bracket notation have a look at the discrete mathematics material on my home page.

$\{\texttt{pink}\}$ which is the singleton of[3] `pink`, which is to say whose second component *is* `pink`. Do not ever confuse any object x with the set $\{x\}$—the set whose sole member is x! So an element of $A \times \{\texttt{pink}\}$ is an ordered pair whose first component is in A and whose second component is `pink`. We can think of such an ordered pair as an object from A labelled with a pink fleck.)

Say something about $A \sqcup B = B \sqcup A$

$\vee$-introduction now says:

$$\frac{a : A}{\langle a, \texttt{pink}\rangle : A \sqcup B} \qquad \frac{b : B}{\langle b, \texttt{blue}\rangle : A \sqcup B}$$

$\vee$-elimination is an action-at-a-distance rule (like $\rightarrow$-introduction) and to treat it properly we need to think about:

5.3 Propagating Decorations

The first rule of decorating is to decorate each assumption with a variable, a thing with no syntactic structure: a single symbol.[4] This is an easy thing to remember, and it helps guide the beginner in understanding the rest of the gadgetry. Pin it to the wall:

Decorate each assumption with a variable!

How are you to decorate formulæ that are not assumptions? You can work that out by checking what rules they are the outputs of. We will discover through some examples what extra gadgetry we need to sensibly extend decorations beyond assumptions to the rest of a proof.

5.4 Rules for $\wedge$

5.4.1 The rule of $\wedge$-elimination

$$\frac{A \wedge B}{B}\ \wedge\text{-elim} \tag{5.1}$$

We decorate the premiss with a variable:

$$\frac{x : A \wedge B}{B}\ \wedge\text{-elim} \tag{5.2}$$

. . . but how do we decorate the conclusion? Well, x must be an ordered pair of something in A with something in B. What we want is the second component of x, which will be a thing in B as desired. So we need a gadget that when we give it an ordered pair, gives us its second component. Let's write this '`snd`'.

[3] The singleton of x is the set whose sole member is x.

[4] You may be wondering what you should do if you want to introduce the same assumption twice. Do you use the same variable? The answer is that if you want to discharge two assumptions with a single application of a rule then the two assumptions must be decorated with the same variable.

$$\frac{x : A \wedge B}{\texttt{snd}(x) : B}$$

By the same token we will need a gadget '`fst`' which gives the first component of an ordered pair so we can decorate[5]

$$\frac{A \wedge B}{A}\ \wedge\text{-elim} \tag{5.3}$$

to obtain

$$\frac{x : A \wedge B}{\texttt{fst}(x) : A}$$

5.4.2 The rule of ∧-introduction

Actually we can put these proofs together and whack an ∧-introduction on the end:

$$\frac{\dfrac{x : A \wedge B}{\texttt{snd}(x) : B} \quad \dfrac{x : A \wedge B}{\texttt{fst}(x) : A}}{\langle \texttt{snd}(x), \texttt{fst}(x) \rangle : B \wedge A}$$

5.5 Rules for →

7.2.2.1 The rule of →-introduction

Here is a simple proof using →-introduction.

$$\frac{\dfrac{[A \to B]^1 \qquad A}{B}\ \to\text{-elim}}{(A \to B) \to B}\ \to\text{-int } (1) \tag{5.1}$$

We decorate the two premisses with single letters (variables): say we use 'f' to decorate '$A \to B$', and 'x' to decorate 'A'. (This is sensible. 'f' is a letter traditionally used to point to functions, and clearly anything in $A \to B$ is going to be a function.) How are we going to decorate 'B'? Well, if x is in A and f is a function that takes things in A and gives things in B then the obvious thing in B that we get is going to be denoted by the decoration '$f(x)$':

$$\frac{\dfrac{f : [A \to B]^1 \quad x : A}{f(x) : B}}{??? : (A \to B) \to B}$$

[5] Agreed: it's shorter to write 'x_1' and 'x_2' than it is to write 'fst(x)' and 'snd(x)' but this would prevent us using 'x_1 and x_2' as variables and in any case I prefer to make explicit the fact that there is a function that extracts components from ordered pairs, rather than having it hidden it away in the notation.

So far so good. But how are we to decorate '$(A \to B) \to B$'? What can the '???' stand for? It must be a notation for a thing (a function) in $(A \to B) \to B$; that is to say, a notation for something that takes a thing in $A \to B$ and returns a thing in B. What might this function be? It is given f and gives back $f(x)$. So we need a notation for a function that, on being given f, returns $f(x)$. (Remember, we decorate all assumptions with variables, and we reach for this notation when we are discharging an assumption so it will always be a variable). We write this

$$\lambda f.f(x)$$

This notation points to the function which, when given f, returns $f(x)$. In general we need a notation for a function that, on being given x, gives back some possibly complex term t. We will write:

$$\lambda x.t$$

for this. Thus we have

$$\dfrac{\dfrac{f : [A \to B]^1 \qquad x : A}{f(x) : B} \to\text{-elim}}{\lambda f.f(x) : (A \to B) \to B} \to\text{-int } (1) \tag{5.2}$$

Thus, in general, an application of $\to$-introduction will gobble up the proof

$$\begin{array}{c} \underline{x : A} \\ \vdots \\ t : B \end{array}$$

and emit the proof

$$\dfrac{\begin{array}{c} \underline{[x : A]} \\ \vdots \\ t : B \end{array}}{\lambda x.t : A \to B}$$

This notation—$\lambda x.t$—for a function that accepts x and returns t is incredibly simple and useful. Almost the only other thing you need to know about it is that if we apply the function $\lambda x.t$ to an input y the output must be the result of substituting 'y' for all the occurrences of 'x' in t. In the literature this result is notated in several ways, for example $[y/x]t$ or $t[y/x]$.

Go over a proof of S at this point

5.6 Rules for $\vee$

We've discussed $\vee$-introduction but not $\vee$-elimination. It's very tricky and—at this stage at least—we don't really need to. It's something to come back to—perhaps!

Exercise 14. *Go back and look at the proofs that you wrote up in answer to exercise* **??**, *and decorate those that do not use '$\vee$'.*

5.7 Remaining Rules

5.7.1 Identity Rule

Here is a very simple application of the identity rule.

See [22]: Semantical Archæology.

$$\dfrac{\dfrac{\dfrac{A \quad B}{B}}{B \to A}}{A \to (B \to A)}$$

Can you think of a function from A to the set of all functions from B to A? If I give you a member a of A, what function from B to A does it suggest to you? Obviously the function that, when given b in B, gives you a.

This gives us the decoration

$$\dfrac{\dfrac{\dfrac{a : A \quad b : B}{b : B}}{\lambda b.a : B \to A}}{\lambda a.(\lambda b.a) : A \to (B \to A)}$$

The function $\lambda a.\lambda b.a$ has a name: K for Konstant. (See section **??**.)

Show how do do this using the option of cancelling non-existent assumptions.

5.7.2 The *ex falso*

The *ex falso sequitur quodlibet* speaks of the propositional constant $\bot$. To correspond to this constant *proposition* we are going to need a constant *set*. The obvious candidate for a set corresponding to $\bot$ is the empty set. Now $\bot \to A$ is a propositional tautology. Can we find a function from the empty set to A which we can specify without knowing anything about A? Yes: the empty function! (You might want to check very carefully that the empty function ticks all the right boxes: is it really the case that whenever we give the empty function a member of the empty set to contemplate it gives us back one and only one answer? Well yes! It has never been known to fail to do this!! Look again at page **??**.) That takes care of $\bot \to A$, the *ex falso*.

5.7.3 Double Negation

What are we to make of $A \to \bot$? Clearly there can be no function from A to the empty set unless A is empty itself. What happens to double negation under this analysis?

$$((A \to \bot) \to \bot) \to A$$

- If A is empty then $A \to \bot$ is the singleton of the empty function and is not empty. So $(A \to \bot) \to \bot$ is the set of functions from a nonempty set to the empty set and is therefore the empty set, so $((A \to \bot) \to \bot) \to A$ is the set of functions from the empty set to the empty set and is therefore the singleton of the empty function, so it is at any rate nonempty.
- However if A is nonempty then $A \to \bot$ is empty. So $(A \to \bot) \to \bot$ is the set of functions from the empty set to the empty set and is nonempty—being the singleton of the empty function—so $((A \to \bot) \to \bot) \to A$ is the set of functions from a nonempty set to the empty set and is therefore empty.

So $((A \to \bot) \to \bot) \to A$ is not reliably inhabited. This is in contrast to all the other truth-table tautologies we have considered. Every other truth-table tautology that we have looked at has a lambda term corresponding to it.

This chapter has been concerned with the relations between the λ-calculus and propositional logic. However the λ-calculus has a life of its own, and—if you can conquer your *mathsangst*—is something you hould definitely pursue.

5.8 Syntactic types

The syntactic type of a piece of syntax is the gadget that tells you what sort of object that piece of syntax evaluates to. A syntactic type is a complex piece [of syntax!] in its own right. We start with two simple syntactic types: `bool` (short for 'boolean': we saw this expression on page 71) and `ind`; more complex syntactic types are built up from them.

A thing of type `ind` will be a thing that always evaluates to an individual. Thus `ind` is the syntactic type of variables and constant symbols.

`bool` is the type of truth-values, so that all the complex formulæ you met in chapter 3, built up from 'p' and 'q' and suchlike by means of the connectives are of syntactic type `bool`.

What about the formulæ of first order logic?

A monadic predicate [symbol] has syntactic type `ind -> bool`.

A dyadic predicate [symbol] has syntactic type `(ind × ind) -> bool`.

The propositional connectives 'and' 'or' etc clearly take two booleans and give back a boolean, so they are all of type `(bool × bool) -> bool`.
(Except of course that ¬, negation, is of type `bool -> bool`)

However, we apply the connectives to predicates as well as booleans.
Not only can we write

It is tuesday and the sun is shining

(in which 'and' is clearly of syntactic type `bool × bool -> bool`) but also

Fred is handsome and charming

in which 'and' is a connective [you linguists would probably say a *conjunction* but don't confuse me] joining two *predicates* not two *statements*, so it is clearly of syntactic type `(ind -> bool)` $\times$ `(ind -> bool) -> (ind -> bool)`.

So, really, they are of type

```
((ind^n -> bool) × (ind^n -> bool)) -> (ind^n -> bool)
```

for any n. What happens if $n = 0$? We get `bool` $\times$ `bool -> bool`.

5.8.1 Adverbial Modifiers and Modal Operators

Adverbial modifiers can clearly be of syntactic type

```
(ind^n  -> bool) -> (ind^n -> bool)
```

for any $n > 0$.

The modal operators '$\Box$' and '$\Diamond$' can be applied to closed formulæ and so can be of syntactic type

```
(ind^n  -> bool) -> (ind^n -> bool)
```

for any $n \geq 0$.

In the degenerate case where $n = 0$ they are of type `bool -> bool`.

5.8.2 Quantifiers

Think about what use you put quantifiers to. You have a complex expression ϕ, with a variable 'x' free in it. This expression is saying something about x, so it is of syntactic type `ind -> bool`. You whack a '$\exists x$' or a '$\forall x$' on the front, getting a thing that has a truth-value, which is to say, is of syntactic type `bool`. This tells us that a quantifier [symbol] is [a piece of syntax] of type `(ind -> bool) -> bool`.

Well, that's an oversimplification. That's what happens if ϕ has only one free variable. If it has n then the result of whacking a quantifier on the front has $n - 1$ free variables, so really a quantifier can have syntactic type `(ind^n  -> bool) -> (ind^(n-1) -> bool)`, for any n.

If $n = 1$ then of course `(ind^n -> bool) -> (ind^(n-1) -> bool)` simplifies to `(ind -> bool) -> bool`. This makes sense: `ind^0 -> bool` should indeed be `bool`: a complex formula with no free variables in it (which is what a thing of syntactic type `ind^0 -> bool` will be) is clearly of syntactic type `bool`.

5.8.3 Determiners

What about determiners? A determiner (I am a logician not a linguist, and logicians tend not to bother with determiners) is a thing that takes a predicate symbol and returns a quantifier. 'the' and 'most' are determiners. Thus [the

denotation of the string] "The man" is a quantifier: something that can be applied to a monadic predicate to give a truth-value—as in 'the man sings'—so it is a quantifier. 'man' is a one-place place predicate symbol, so the syntactic type of the determiner 'the' is

```
(ind -> bool) -> (ind^n  -> bool) -> (ind^m -> bool).
```

Thus:

String	Syntactic Type
the	`(ind -> bool) -> ((ind  -> bool) -> bool))`
man	`ind -> bool`
the man	`(ind  -> bool) -> bool`
sings	`ind -> bool`
the man sings	`bool`

Observe that the type of 'the man' is the result of applying the type of 'the' to the type of 'man'; next we find that the type of 'the man sings' is the result of applying the type of 'the man' to the type of 'sings'.

Observe further that if we do the same parsing to 'most men sing' we get the same

I suppose that this means that in English the word 'all' is a determiner rather than a quantifier, since its typical use is in things like "all men sing' which has the same syntax as 'most men sing'.

what about ϵ terms?

Chapter 6

Possible World Semantics

This should really be called "Multiple world semantics" but the current usage is entrenched.

It's a way of providing semantics for intensional connectives like $\Box$ and $\Diamond$. There are various debates in logic about these connectives but mercifully getting embroiled in them is not a requirement for mastering possible world semantics. Our concern for the moment is merely with the machinery not with the uses to which other people have put it.

Definition 3. *A* **possible world model** $\mathfrak{M}$ *has several components:*

- *There is a collection of* **worlds** *with a binary relation* $\leq$ *between them; If* $W_1 \leq W_2$ *we say* W_1 *can* **see** W_2*.*
- *There is also a binary relation between worlds and formulæ, written '*$W \models \phi$*';*
- *Finally there is a* **designated** *(or 'actual' or 'root') world* W_0^M*.*

We stipulate the following connections between the ingredients:

1. $W \models \bot$ *never holds. We write this as* $W \not\models \bot$*.*
2. $W \models A \wedge B$ *iff* $W \models A$ *and* $W \models B$*;*
3. $W \models A \vee B$ *iff* $W \models A$ *or* $W \models B$*;*
4. $W \models A \to B$ *iff every* $W' \, R \, W$ *that* $\models A$ *also* $\models B$*;*
5. $W \models \neg A$ *iff there is no* $W' \, R \, W$ *such that* $W' \models A$*;*
6. $W \models \Box A$ *iff every* $W' \, R \, W \models A$*;*
7. $W \models \Diamond A$ *iff at least one* $W' \, R \, W \models A$*;*
8. $W \models A \to B$ *iff every* $W' \, R \, W$ *that* $\models A$ *also* $\models B$*;*

$\Box$ and $\Diamond$ are dual: $\neg\Box\neg p$ is the same as $\Diamond p$ and $\neg\Diamond\neg p$ is the same as $\Box p$. '$\forall$' and '$\exists$' are dual in the same way.

(We will deal with the quantifiers later)

Then we say

$$\mathfrak{M} \models A \quad \textit{if} \quad W_0^M \models A$$

4 is a special case of 3: $\neg A$ is just $A \to \bot$, and no world believes $\bot$!

The relation which we here write with a 'R' is the **accessibility** relation between worlds. The apparatus of possible world semantics puts **no restrictions** on what properties the accessibility relation might have. This freedom of manœuvre is very useful beco's it turns out that imposing conditions on the accessibility relation corresponds to enforcing certain modal principles.

The reader might wish to check that if the accessibility relation is (for example)

transitive	then the principle	$\Diamond\Diamond p \to \Diamond p$	holds
reflexive		$\Box p \to p$	holds
symmetrical		$p \to \Box\Diamond p$	holds
empty		$\Box p$	holds

You remember what 'transitive' *etc* mean from section 4.7.

There is quite a lot that can be said along these lines, but we don't need to know the details of it to understand how the machinery works. That said, checking the truth of the four assertions above makes a useful exercise that a beginner should attempt, even if not this very minute and second.

It's probably a good idea to think a bit about how this gadgetry is a generalisation of the semantics for propositional logic that we saw in section 3.1, or—to put it the other way round—how the semantics there is a degenerate case of what we have here. *Worlds* here correspond to *valuations* (or rows of a truth-table) there. In section 3.1 each valuation went on its merry way without reference to any other valuation: if you wanted to know whether a valuation v made a formula ϕ true you had to look at subformulæ of ϕ but you didn't have to look at what any other valuation did to ϕ or to any of its subformulæ. If you think about this a bit you will realise that if you have a possible world model where the the accessibility relation R is the identity relation [so that the only valuation that a valuation v ever has to consult is v itself] then the semantics you get will be the same as in section 3.1. Another thing that happens if R is the identity is that $\Diamond p$, p and $\Box p$ turn out to be the same.

should eventually copy this paragraph to chchlectures.tex

Say something about what $\Diamond$ and $\Box$ might mean.

Chat about quantifier alternation. There is a case for writing out the definitions in a formal language, on the grounds that the quantifier alternation (which bothers a lot of people) can be made clearer by use of a formal language. The advantage of not using a formal language is that it makes the language-metalanguage distinction clearer.

This stuff was invented in order to provide answers to a raft of questions about principles of modal reasoning: $p \to \Box\Box p$? $p \to \Box\Diamond p$? Is $\Diamond\Diamond p$ ever false? Of course the answers will depend on which intensional monadic connective $\Box$ is supposed to capture.

However the gadgetry has outgrown its original application, and you don't have to be interested in modal logic to find possible world semantics useful.

Quantifiers

Definition 3 didn't have rules for the quantifiers.

9 : $W \models (\exists x)A(x)$ iff there is an x in W such that $W \models A(x)$;

10 : $W \models (\forall x)A(x)$ iff for all $W' \geq W$ and all x in W', $W' \models A(x)$.

The rules for the quantifiers assume that worlds don't just believe primitive propositions but also that they have inhabitants. I think we generally take it that our worlds are never empty: every world has at least one inhabitant. However there is no global assumption that all worlds have the same inhabitants. Objects may pop in and out of existence. However we do take the identity relation between inhabitants across possible worlds as a given.

Notice that according to 9 and 10 the quantifiers are not dual.

6.1 Language and Metalanguage again

It is very important to distinguish between the stuff that appears to the left of a '$\models$' sign and that which appears to the right of it. The stuff to the right of the '$\models$' sign belongs to the *object language* and the stuff to the left of the '$\models$' sign belongs to the *metalanguage*. So that we do not lose track of where we are I am going to write '$\to$' for *if–then* in the metalanguage and '`&`' for *and* in the metalanguage instead of '$\wedge$'. And I shall use square brackets instead of round brackets in the metalanguage.

If you do not keep this distinction clear in your mind you will end up making one of the two mistakes below (tho' you are unlikely to make both.)

For example here is a manœuvre that is perfectly legitimate:

If

$$\neg[W \models A \to B]$$

then it is not the case that

$$(\forall W' \geq W)(W' \models A \ \to \ W' \models B)$$

So, in particular,

$$(\exists W' \geq W)(W' \models A \ \& \ \neg(W' \models B))$$

The inference drawn here from $\neg\forall$ to $\exists\neg$ is perfectly all right in the classical metalanguage, even though it's not allowed in the constructive object language. [more detail here]

In contrast it is *not* all right to think that—for example—$W \models \neg A \vee \neg B$ is the same as $W \models \neg(A \wedge B)$ (on the grounds that $\neg A \vee \neg B$ is the same as $\neg(A \wedge B)$). One way of warding off the temptation to do is is to remind ourselves—again—that the aim of the Possible World exercise was to give people who believe in classical logic a way of making sense of the thinking of people who believe in constructive logic. That means that it is not OK to use classical logic in reasoning with/manipulating stuff to the right of a '$\models$' sign.

Another way of warding off the same temptation is to think of the stuff after the '$\models$' sign as stuff that goes on in a fiction. You, the reader of a fiction, know things about the characters in the fiction that they do not know about each other. Just because something is true doesn't mean they know it!! (This is what the literary people call **Dramatic Irony**.)[1]

(This reflection brings with it the thought that reading "$W \models \neg\neg A$" as "W believes not not A" is perhaps not the happiest piece of slang. After all, in circumstances where $W \models \neg\neg A$ there is no suggestion that the fact-that-no-world-$\geq$-W-believes-A is encoded in W in any way at all.)

Could say more about this

Another mistake is to think that we are obliged to use constructive logic in the metalanguage which we are using to discuss constructive logic—to the left of the '$\models$' sign. I suspect it's a widespread error. It may be the same mistake as the mistake of supposing that you have to convert to Christianity to understand what is going on in the heads of Christians. Christians of some stripes would no doubt agree with the assertion that there are bits of it you can't understand until you convert, but I think that is just a mind-game.

Doesn't this duplicate earlier stuff?

We could make it easier for the nervous to discern the difference between the places where it's all right to use classical reasoning (the metalanguage) and the object language (where it isn't) by using different fonts or different alphabets. One could write "For all W" instead of $(\forall W)\ldots$". That would certainly be a useful way of making the point, but once the point has been made, persisting with it looks a bit obsessional: in general people seem to prefer overloading to disambiguation.

6.1.1 A possibly helpful illustration

Let us illustrate with the following variants on the theme of "there is a Magic Sword." All these variants are classically equivalent. The subtle distinctions that the possible worlds semantics enable us to make are very pleasing.

1. $\neg\forall x\neg MS(x)$

[1] Appreciation of the difference between something being true and your interlocutor knowing it is something that autists can have trouble with. Some animals that have "a theory of other minds" (in that they know that their conspecifics might know something) too can have difficulty with this distinction. Humans seem to be able to cope with it from the age of about three

2. $\neg\neg\exists x MS(x)$

3. $\exists x\neg\neg MS(x)$

4. $\exists x MS(x)$

The first two are constructively equivalent as well.

To explain the differences we need the difference between **histories** and **futures.**

- A *future* (from the point of view of a world W) is any world $W' \geq W$.
- A *history* is a string of worlds—an unbounded trajectory through the available futures.

No gaps between worlds...?

$\neg\forall x\neg MS(x)$ and $\neg\neg\exists x MS(x)$ say that every future can see a future in which there is a Magic Sword, even though there might be histories that avoid Magic Swords altogether: *Magic Swords are a permanent possibility: you should never give up hope of finding one.*

How can this be, that every future can see a future in which there is a magic sword but there is a history that contains no magic sword–ever? It could happen like this: each world has precisely two immediate children. If it is a world with a magic sword then those two worlds also have magic swords in them. If it is a world without a magic sword then one of its two children continues swordless, and the other one acquires a sword. We stipulate that the root world contains no magic sword. That way every world can see a world that has a magic sword, and yet there is a history that has no magic swords.

$\exists x\neg\neg MS(x)$ says that every history contains a Magic Sword and moreover the thing which is destined to be a Magic Sword is already here. Perhaps it's still a lump of silver at the moment but it will be a Magic Sword one day.

6.2 Some Useful Short Cuts

6.2.1 Double negation

The first one that comes to mind is $W \models \neg\neg\phi$. This is the same as $(\forall W' \geq W)(\exists W'' \geq W')(W'' \models \phi)$. "Every world that W can see can see a world that believes ϕ". Let's thrash this out by hand.

By clause 5 of definition 3

$$W \models \neg(\neg\phi)$$

iff

$$(\forall W' \geq W)\neg[W' \models \neg\phi] \tag{6.1}$$

Now

$W' \models \neg\phi$ iff $(\forall W'' \geq W')\neg[W'' \models \phi]$ by clause 5 of definition 3 so

$\neg[W' \models \neg\phi]$ is the same as $\neg(\forall W'' \geq W')\neg[W'' \models \phi]$ which is

$$(\exists W'' \geq W')(W'' \models \phi).$$

Substituting this last formula for for '$W' \models \neg\phi$' in (6.1) we obtain

$$(\forall W' \geq W)(\exists W'' \geq W')(W'' \models \phi)$$

6.2.2 If there is only one world then the logic is classical

If $\mathfrak{M}$ contains only one world—W, say—then $\mathfrak{M}$ believes classical logic. Let me illustrate this in two ways:

1. Suppose $\mathfrak{M} \models \neg\neg A$. Then $W \models \neg\neg A$, since W is the root world of $\mathfrak{M}$. If $W \models \neg\neg A$, then for every world $W' \geq W$ there is $W'' \geq W$ that believes A. So in particular there is a world $\geq W$ that believes A. But the only world $\geq W$ is W itself. So $W \models A$. So every world $\geq W$ that believes $\neg\neg A$ also believes A. So $W \models \neg\neg A \to A$.

2. W either believes A or it doesn't. If it believes A then it certainly believes $A \vee \neg A$, so suppose W does not believe A. Then no world that W can see believes A. So $W \models \neg A$ and thus $W \models (A \vee \neg A)$. So W believes the law of excluded middle.

We must show that the logic of quantifiers is classical too

The same arguments can be used even in models with more than one world, if the worlds in question can see only themselves.

6.3 Independence Proofs Using Possible world semantics

6.3.1 Some Worked Examples

Challenge 6.3.1.1: Find a countermodel for $A \vee \neg A$

The first thing to notice is that this formula is a classical (truth-table) tautology. Because of subsection 6.2.2 this means that any countermodel for it must contain more than one world.

The root world W_0 must not believe A and it must not believe $\neg A$. If it cannot see a world that believes A then it will believe $\neg A$, so we will have to arrange for it to see a world that believes A. One will do, so let there be W_1 such that $W_1 \models A$.

picture here

Challenge 6.3.1.2: Find a countermodel for $\neg\neg A \vee \neg A$

The root world W_0 must not believe $\neg\neg A$ and it must not believe $\neg A$. If it cannot see a world that believes A then it will believe $\neg A$, so we will have to arrange for it to see a world that believes A. One will do, so let there be W_1

such that ($W_1 \models A$). It must also not believe $\neg\neg A$. It will believe $\neg\neg A$ as long as every world it can see can see a world that believes A. So there had better be a world it can see that cannot see any world that believes A. This cannot be W_1 because $W_1 \models A$, and it cannot be W_0 itself, since $W_0 \leq W_1$. So there must be a third world W_2 which does not believe A.

Challenge 6.3.1.3: Find a countermodel that satisfies $(A \to B) \to B)$ but does not satisfy $A \vee B$

insert details here

Challenge 6.3.1.4: Find a countermodel for $((A \to B) \to A) \to A$

We've deleted Zarg ...

You may recall from exercise **??** on page **??** that this formula is believed to be false on Planet Zarg. There we had a three-valued truth table. Here we are going to use possible worlds. As before, with $A \vee \neg A$, the formula is a truth-table tautology and so we will need more than one world

Recall that a model $\mathfrak{M}$ satisfies a formula ψ iff the root world of $\mathfrak{M}$ believes ψ: that is what it is for a model to satisfy ψ. Definition!

As usual I shall write 'W_0' for the root world; and will also write '$W \models \psi$' to mean that the world W believes ψ; and $\neg[W \models \psi]$ to mean that W does not believe ψ.

So we know that $\neg[W_0 \models ((A \to B) \to A) \to A]$.

Now the definition of $W \models X \to Y$ is (by definition 3)

$$(\forall W' \geq W)[W' \models X \ \to \ W' \models Y] \tag{6.1}$$

So since

$$\neg[W_0 \models ((A \to B) \to A) \to A]$$

we know that there must be a $W' \geq W_0$ which believes $((A \to B) \to A$ but does not believe A. (In symbols: $(\exists W' \geq W_0)[W' \models ((A \to B) \to A)$ `&` $\neg(W' \models A)]$.) Remember too that in the metalanguage we are allowed to exploit the equivalence of $\neg\forall$ with $\exists\neg$. Now every world can see itself, so might this W' happen to be W_0 itself? No harm in trying...

So, on the assumption that this W' that we need is W_0 itself, we have:

1. $W_0 \models (A \to B) \to A$; and
2. $\neg[W_0 \models A]$.

This is quite informative. Fact (1) tells us that every $W' \geq W_0$ that believes $A \to B$ also believes A. Now one of those W' is W_0 itself (Every world can see itself: remember that $\geq$ is reflexive). Put this together with fact (2) which says that W_0 does not believe A, and we know at once that W_0 cannot believe $A \to B$. How can we arrange for W_0 not to believe $A \to B$? Recall the definition 3 above of $W \models A \to B$. We have to ensure that there is a $W' \geq W_0$ that believes A but does not believe B. This W' cannot be W_0 because W_0 does not believe A. So there must be a *new* world (we always knew there would

be!) visible from W_0 that believes A but does not believe B. (In symbols this is $(\exists W' \geq W_0)[W' \models A \quad \& \quad \neg(W' \models B)]$.)

So our countermodel contains two worlds W_0 and W', with $W_0 \leq W'$. $W' \models A$ but $\neg[W_0 \models A]$, and $\neg[W' \models B]$.

Let's check that this really works. We want

$$\neg[W_0 \models ((A \to B) \to A) \to A]$$

We have to ensure that at least one of the worlds beyond W_0 satisfies $(A \to B) \to A$ but does not satisfy A. W_0 doesn't satisfy A so it will suffice to check that it does satisfy $(A \to B) \to A$. So we have to check (i) that if W_0 satisfies $(A \to B)$ then it also satisfies A and we have to check (ii) that if W' satisfies $(A \to B)$ then it also satisfies A. W' satisfies A so (ii) is taken care of. For (i) we have to check that W_0 does not satisfy $A \to B$. For this we need a world $\geq W_0$ that believes A but does not believe B and W' is such a world.

Challenge 6.3.1.5: Find a model that satisfies $(A \to B) \to B$ but does not satisfy $(B \to A) \to A$

We must have

$$W_0 \models (A \to B) \to B \tag{1}$$

and

$$\neg[W_0 \models (B \to A) \to A] \tag{2}$$

By (2) we must have $W_1 \geq W_0$ such that

$$W_1 \models B \to A \tag{3}$$

but

$$\neg[W_1 \models A] \tag{4}$$

We can now show

$$\neg[W_1 \models A \to B] \tag{5}$$

If (5) were false then $W_1 \models B$ would follow from (1) and then $W_1 \models A$ would follow from (3). (5) now tells us that there is $W_2 \geq W_1$ such that

$$W_2 \models A \tag{6}$$

and

$$\neg[W_2 \models B] \tag{7}$$

From (7) and persistence we infer

$$\neg[W_1 \models B] \tag{8}$$

and

$$\neg[W_0 \models B] \tag{9}$$

Also, (4) tells us

$$\neg[W_0 \models A]. \tag{10}$$

So far we have nothing to tell us that $W_0 \neq W_1$. So perhaps we can get away with having only two worlds W_0 and W_1 with $W_1 \models A$ and W_0 believing nothing.

W_0 believes $(A \to B) \to B$ vacuously: it cannot see a world that believes $A \to B$ so—vacuously—every world that it can see that believes $A \to B$ also believes B. However, every world that it can see believes $(B \to A)$ but it does not believe A itself. That is to say, it can see a world that does not believe A so it can see a world that believes $B \to A$ but does not believe A so it does not believe $(B \to A) \to A$.

6.3.2 Exercises

Exercise 15. *Return to Planet Zarg!*[2]
The truth-tables for Zarg-style connectives are on p **??**.

1. *Write out a truth-table for* $((p \to q) \to q) \to (p \vee q)$.
   (Before you start, ask yourself how many rows this truth-table will have).
2. *Identify a row in which the formula does not take truth-value* 1.
3. *Find a sequent proof for* $((p \to q) \to q) \to (p \vee q)$.

Exercise 16. *Find a model that satisfies* $(p \to q) \to q$ *but does not satisfy* $p \vee q$.

It turns out that Zarg-truth-value 1 means "true in W_0 and in W'"; Zarg-truth-value 2 means "true in W'", and Zarg-truth-value 3 means "true in neither"—where W_0 and W_1 are the two worlds in the countermodel we found for Peirce's law. (Challenge 6.3.1.5)

Exercise 17. *Find a model that satisfies* $p \to q$ *but not* $\neg p \vee q$.

Exercise 18. *Find a model that doesn't satisfy* $p \vee \neg p$. *How many worlds has it got? Does it satisfy* $\neg p \vee \neg\neg p$? *If it does, find one that doesn't satisfy* $\neg p \vee \neg\neg p$.

Exercise 19.
1. *Find a model that satisfies* $A \to (B \vee C)$ *but doesn't satisfy* $(A \to B) \vee (A \to C))$.
2. *Find a model that satisfies* $(A \to B) \wedge (C \to D)$ *but doesn't satisfy* $(A \to D) \vee (B \to C))$[3].

[2] Beware: Zarg is a planet not a possible world!

[3] This is a celebrated illustration of how $\to$ does not capture 'if-then'. Match the antecedntto "If Jones is in Aberdeen then Jones is in Scotland and if Jones is in Delhi then Jones is in India".

3. *Find a model that satisfies* $\neg(A \wedge B)$ *but does not satisfy* $\neg A \vee \neg B$

4. *Find a model that satisfies* $(A \to B) \to B)$ *and* $(B \to A) \to A$ *but does not satisfy* $A \vee B$. *(Check that in the three-valued Zarg world* $((A \to B) \to B) \wedge ((B \to A) \to A)$ *always has the same truth-table as* $A \vee B$*).*

EXERCISE 20. *Find countermodels for:*

1. $(A \to B) \vee (B \to A)$*;*

2. $(\exists x)(\forall y)(F(y) \to F(x))$ *(which is the formula in exercise* **??** *part 1 on page* **??***).*

EXERCISE 21. *Consider the model in which there are two worlds,* W_0 *and* W_1*, with* $W_0 \leq W_1$*.* W_0 *contains various things, all of which it believes to be frogs;* W_1 *contains everything in* W_1 *plus various additional things, none of which it believes to be frogs. Which of the following assertions does this model believe?*

1. $(\forall x)(F(x))$*;*

2. $(\exists x)(\neg F(x))$*;*

3. $\neg\exists x\neg F(x)$*;*

4. $\neg\neg(\exists x)(\neg F(x))$*.*

Chapter 7

Enhanced Syntax

Branching quantifiers “Independence-froendly logic”

Chapter 8

Appendices

8.1 Notes to Chapter one

8.1.1 The Material Conditional

Lots of students dislike the material conditional as an account of implication. The usual cause of this unease is that in some cases a material conditional $p \to q$ evaluates to `true` for what seem to them to be spurious and thoroughly unsatisfactory reasons: namely, that p is false or that q is true. How can q follow from p merely because q happens to be true? The meaning of p might have no bearing on q whatever! Standard illustrations in the literature include

> If Julius Cæsar is Emperor then sea water is salt.

need a few more examples

These example seem odd because we feel that to decide whether or not p implies q we need to know a lot more than the truth-values of p and q.

This unease shows that we have forgotten that we were supposed to be examining a relation between *extensions*, and have carelessly returned to our original endeavour of trying to understand implication between *intensions*. $\wedge$ and $\vee$, too, are relations between intensions but they also make sense applied to extensions. Now if p implies q, what does this tell us about what p and q evaluate to? Well, at the very least, it tells us that p cannot evaluate to `true` when q evaluates to `false`.

Thus we can expect the *extension* corresponding to a conditional to satisfy *modus ponens* at the very least.

How many extensional connectives are there that satisfy *modus ponens*? For a connective C to satisfy *modus ponens* it suffices that in each of the two rows of the truth table for C where p is true, if $p\,C\,q$ is true in that row then q is true too.

p	C	q
1	?	1
0	?	1
1	0	0
0	?	0

We cannot make $p\,C\,q$ true in the third row, because that would cause C to disobey *modus ponens*, but it doesn't matter what we put in the centre column in the three other rows. This leaves eight possibilities:

(1) : $\dfrac{p \quad q}{q}$ (2) : $\dfrac{p \quad p \longleftrightarrow q}{q}$ (3) : $\dfrac{p \quad \neg p}{q}$ (4) : $\dfrac{p \quad p \to q}{q}$

p	C^1	q
1	1	1
1	0	0
0	1	1
0	0	0

p	C^2	q
1	1	1
1	0	0
0	0	1
0	1	0

p	C^3	q
1	0	1
1	0	0
0	1	1
0	1	0

p	C^4	p
1	1	1
1	0	0
0	1	1
0	1	0

(5) : $\dfrac{p \quad \bot}{q}$ (6) : $\dfrac{p \quad p \wedge q}{q}$ (7) : $\dfrac{p \quad \neg p \wedge q}{q}$ (8) : $\dfrac{p \quad \neg p \wedge \neg q}{q}$

p	C^5	q
1	0	1
1	0	0
0	0	1
0	0	0

p	C^6	q
1	1	1
1	0	0
0	0	1
0	0	0

p	C^7	q
1	0	1
1	0	0
0	1	1
0	0	0

p	C^8	p
1	0	1
1	0	0
0	0	1
0	1	0

The horizontal lines should not go all the way across, but be divided into four segments, one for each truth table. I haven't worked out how to make that happen!

. . . obtained from the rule of *modus ponens* by replacing '$p \to q$' by each of the eight extensional binary connectives that satisfy the rule.

(1) will never tell us anything we didn't know before;
(5) we can never use because its major premiss is never true;
(6) is a poor substitute for the rule of "$\wedge$-elimination";
(3),(7) and (8) we will never be able to use if our premisses are consistent.

(2), (4) and (6) are the only sensible rules left. (2) is not what we are after because it is symmetrical in p and q whereas "if p then q" is not. The advantage of (4) is that you can use it whenever you can use (2) or (6). So it's more use!

We had better check that this policy of evaluating $p \to q$ to `true` unless there is a very good reason not to does not get us into trouble. Fortunately, in cases where the conditional is evaluated to `true` *merely* for spurious reasons, then no harm can be done by accepting that evaluation. For consider: if it is evaluated to `true` *merely* because p evaluates to `false`, then we are never going to be able to invoke it (as a major premiss at least), and if it is evaluated to `true` *merely*

because q evaluates to `true`, then if we invoke it as a major premiss, the only thing we can conclude—namely q—is something we knew anyway.

This last paragraph is not intended to be a *justification* of our policy of using only the material conditional: it is merely intended to make it look less unnatural than it otherwise might. The astute reader who spotted that nothing was said there about conditionals as *minor* premisses should not complain. They may wish to ponder the reason for this omission.

8.2 Notes to Chapter 4

8.2.1 Subtleties in the definition of first-order language

The following formula looks like a first-order sentence that says there are at least n distinct things in the universe. (Remember the $\bigvee$ symbol from page ??.)

$$(\exists x_1 \ldots x_n)(\forall y)(\bigvee_{i \leq n} y = x_i) \tag{8.1}$$

But if you are the kind of pedant that does well in Logic you will notice that it isn't a formula of the first-order logic we have just seen because there are variables (the subscripts) ranging over variables! If you put in a concrete actual number for n then what you have is an *abbreviation* of a formula of our first-order language. Thus

$$(\exists x_1 \ldots x_3)(\forall y)(\bigvee_{i \leq 3} y = x_i) \tag{8.2}$$

is an abbreviation of

$$(\exists x_1 x_2 x_3)(\forall y)(y = x_1 \vee y = x_2 \vee y = x_3) \tag{8.3}$$

(Notice that formula 8.2.1 isn't actually *second*-order either, because the dodgy variables are not ranging over subsets of the domain.)

8.3 Church on intension and extension

> *"The foregoing discussion leaves it undetermined under what circumstances two functions shall be considered the same. The most immediate and, from some points of view, the best way to settle this question is to specify that two functions f and g are the same if they have the same range of arguments and, for every element a that belongs to this range, $f(a)$ is the same as $g(a)$. When this is done we shall say that we are dealing with functions in extension.*
>
> *It is possible, however, to allow two functions to be different on the ground that the rule of correspondence is different in meaning in*

> *the two cases although always yielding the same result when applied to any particular argument. When this is done we shall say that we are dealing with functions in intension. The notion of difference in meaning between two rules of correspondence is a vague one, but, in terms of some system of notation, it can be made exact in various ways. We shall not attempt to decide what is the true notion of difference in meaning but shall speak of functions in intension in any case where a more severe criterion of identity is adopted than for functions in extension. There is thus not one notion of function in intension, but many notions; involving various degrees of intensionality".*

Church [4]. p 2.

The intension-extension distinction has proved particularly useful in computer science—specifically in the theory of computable functions, since the distinction between a *program* and the *graph* of a function corresponds neatly to the difference between a function-in-intension and a function-in-extension. Computer Science provides us with perhaps the best-motivated modern illustration. A piece of code that needs to call another function can do it in either of two ways. If the function being called is going to be called often, on a restricted range of arguments, and is hard to compute, then the obvious thing to do is compute the set of values in advance and store them in a look-up table in line in the code. On the other hand if the function to be called is not going to be called very often, and the set of arguments on which it is to be called cannot be determined in advance, and if there is an easy algorithm available to compute it, then the obvious strategy is to write code for that algorithm and call it when needed. In the first case the embedded subordinate function is represented as a function-in-extension, and in the second case as a function-in-intension.

The concept of algorithm seems to be more intensional than function-in-extension but not as intensional as function-in-intension. Different programs can instantiate the same algorithm, and there can be more than one algorithm for computing a function-in-extension. Not clear what the identity criteria for algorithms are. Indeed it has been argued that there can be no satifactory concept of algorithm see [1]. This is particularly unfortunate because of the weight the concept of algorithm is made to bear in some philosophies of mind (or some parodies of philosophy-of-mind ["strong AI"] such as are to be found in [**?**]).[1]

[1] Perhaps that is why is is made to carry that weight! If your sights are set not on devising a true philosophical theory, but merely on cobbling together a philosophical theory that will be hard to refute then a good strategy is to have as a keystone concept one that is so vague that any attack on the theory can be repelled by a fallacy of equivocation. The unclarity in the key concept ensures that the target presented to aspiring refuters is a fuzzy one, so that no refutation is ever conclusive. This is why squids have ink.

Bibliography

[1] Andreas Blass, Nachum Dershowitz and Yuri Gurevich. When are two algorithms the same? Bulletin of Symbolic Logic **15** (june 2009) pp 145–168.

[2] J.L. Borges Labyrinths [publ'n data??]

[3] J.L. Borges *Siete Noches*

[4] Church, A. The calculi of λ-conversion. Princeton 1941

[5] T.E.Forster. The Modal æther. www.dpmms.cam.ac.uk/~tf/modalrealism.ps Bulletin de L'Académie Royal de Belgique, série de la Classe des Sciences **15** 1929 pp 183–8

[6] Wilfrid Hodges Logic Penguin

[7] John E Hopcroft and Jeffrey D Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley

[8] Donald Kalish and Richard Montague: Logic: techniques of formal reasoning. Harcourt Brace 1964

[9] Per Martin-Löf. Truth of a Proposition, Evidence of a Judgement, Validity of a Proof, Synthese **73**, pp.407-420.

[10] Per Martin-Löf. On the meaning of the Logical Constants and the Justifications of the Logical Laws', Nordic Journal of Philosophical Logic 1(1) (1996), pp.11-60.

`http://www.hf.uio.no/ifikk/filosofi/njpl/vol1no1/meaning/meaning.html`

[11] Mendelson, E. (1979) Introduction to Mathematical Logic. (2nd edn) Van Nostrand

[12] "Plural Logic" Alex Oliver and Timothy Smiley. 352 pp, 978-0-19-957042-3 Oxford University Press 2013

[13] `http://www.cl.cam.ac.uk/Teaching/2002/RLFA/reglfa.ps.gz`

[14] Graham Priest Some Priorities of Berkeley, Logic and Reality: Essays on the Legacy of Arthur Prior, ed. B.J.Copeland, Oxford University Press, 1996.

[15] Quine, Mathematical Logic

[16] Arthur Prior The runabout inference ticket. Analysis **21**(1960) pp 38–9. Reprinted in Oxford Readings in Philosophy ed Strawson 1967

[17] Quine, Ontological relativity and Other Essays

[18] Quine, Word and Object

[19] Quine, Predicate functor Logic in Selected Logic papers.

[20] Quine, Ontological remarks on the propositional calculus in Selected Logic Papers 265–272.

[21] Quine, W.V. (1962) Mathematical Logic (revised edition) Harper torchbooks 1962

[22] Scott D.S. Semantical Archæology, a parable. In: Harman and Davidson eds, Semantics of Natural Languages. Reidel 1972 pp 666–674.

[23] Arto Salomaa: Computation and Automata. Encyclopædia of mathematics and its applications. CUP 1985

[24] Peter Smith: An introduction to Formal Logic CUP.

[25] Alfred Tarski and Steven Givant. "A formulation of Set Theory without variables". AMS colloquium publication **41** 1987.

[26] Tversky and Kahnemann "Extensional versus Intuitive reasoning" Psychological Review **90** 293-315.

[27] Tymoczko and Henle. Sweet Reason: A Field Guide to Modern Logic. Birkhäuser 2000 ISBN