Logic for Linguists;
Eight Lectures in the
Michaelmas Term 2017

Thomas Forster

October 11, 2017

Contents

font information has semantics

adjectives in English absolutely have to be in this order: opinion-size-age-shape-colour-origin-material-purpose Noun. So you can have a lovely little old rectangular green French silver whittling knife. But if you mess with that word order in the slightest you'll sound like a maniac. It's an odd thing that every English speaker uses that list, but almost none of us could write it out. And as size comes before colour, green great dragons can't exist.

Notice that in the formal language for chemistry the occurence of 'N' inside 'Ni' or 'Na' cannot be seen ... 'Ni' is a single symbol, as is 'Na'.
semantics of flashing car headlights
in some languages men and women speak different languages

A man is watching a dog lick its bollocks.

> "I wish i could do that".
>
> "Give him a biscuit, he'll probably let you".

We sometimes use the expression 'positional notations' for notations where the meaning of a symbol depends on where it is.

> Give orange me give eat orange me eat orange give me eat orange give me you

said Nim Chimpski.
The meaning is clear but there is no syntax!

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 which i do not use here)

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, too, like to make the point that words do not have magical powers.

Chapter 1

Introduction

One of the [many] 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. Try googling "The Linguistic Turn".

(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 indeed even *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 traffic 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 he's 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

Examples here.(Semantics?)

but are completely intractable when asked about natural languages.

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. But i am not going to assume you have.

Examples here

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.

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
Both literal and metaphorical meaning	literal meaning only.

Also, traditionally formal logic has concerned itself only with statements (what the compscis call *Booleans*) rather than commands or questions, or performatives.

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, [14]

[1] Apparently this terminology goes back to [**?**] or do i mean [18]?.

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 100-odd years ago. 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.

Chapter 2

Some Basics

ordered pairs (which we need for lambda calculus)

`true` and `false` are the two truth-values. '$\top$' and '$\bot$' are the two reserved propositional letters that always denote/evaluate-to `true` and `false`.

2.1 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*.

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.

2.2 Discrete Maths for Linguists

This is a highly concentrated version of `www.dpmms.cam.ac.uk/~tf/cam_only/discretelectures.pdf` and that file is recommended to those who are entirely free of mathsangst.

transitivity, antisymmetry, etc, equivalence relations and congruence relations.

No entity without identity. The basic entities of a science are equivalence classes of phenomena under indistinguishability.

e.g., phonemes as equivalence classes.

one can be substituted for the other without changing *meaning*? or one can be substituted for the other without changing *wellformedness*?

Are these the same?

Collection of congruence relations parametrised by the class of languages

Disjoint unions??

More notation

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**.

$|X|$ is the cardinality of the set X.

More Ideas

type-token,
use-mention,
language-metalanguage
de re and *de dicto*
analytic/synthetic

Talk over this. Details in chchlectures.pdf or discretelectures.tex. How much do we copy over?

2.3 Language/Metalanguage

2.4 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'.

Haddocks' Eyes

As so often the standard example is from [**?**].

> [. . .] 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 or strings of words—'The agèd, agèd man'. . . . Even so, it does seem clear that the White Knight has got it wrong. At the very least: if the name of the song really *is* 'The agèd agèd man' (as he says) then clearly Alice was right to say that was what the song was called. Granted, it might have more names than just that one—'Ways and means' for example—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 [**?**])—burst into song. These infelicities must be deliberate (Carroll does not make elementary mistakes like that), and one wonders whether or not the White Knight realises he is getting it wrong . . . is he an old fool and nothing more? Or is he a paid-up party to a conspiracy to make the reader's reading experience a nightmare? The Alice books are one long nightmare, and perhaps not just for Alice.

Alphabet soup

People complain that they don't want their food to be full of E-numbers. What they mean is that they don't want it to be full of the things denoted by the E-numbers.[1]

[1] Mind you E-300 is Vitamin C and there's nothing wrong with *that*!

Some Good Advice

> Q: Why should you never fall in love with a tennis player?
>
> A: Because ‘love’ means ‘nothing’ to them.

Apple Crumble

“Put cream on the apple crumble”
“But there isn’t any cream!”
“Then put ‘cream’ on the shopping list!”

‘Think’

> “If I were asked to put my advice to a young man in one word, Prestwick, do you know what that word would be?”
>
> “No” said Sir Prestwick.
>
> “ ‘Think’, Prestwick, ‘Think’ ”.
>
> “I don’t know, R.V. ‘Detail’?”
>
> “No, Prestwick, ‘Think’.”
>
> “Er, ‘Courage’?”
>
> “No! ‘Think’!”
>
> “I give up, R.V., ‘Boldness’?”
>
> “For heavan’s sake, Prestwick, what is the matter with you? ‘*Think*’!”
>
> “ ‘Integrity’? ‘Loyalty’? ‘Leadership’?”
>
> “ ‘Think’, Prestwick! ‘Think’, ‘Think’, ‘Think’ ‘Think’!”

Michael Frayn: *The Tin Men.* Frayn has a degree in Philosophy.

Ramsey for Breakfast

In the following example F.P. Ramsey[2] 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’.

[2] You will be hearing more of this chap.

The Deaf Judge

JUDGE (*to PRISONER*): Do you have anything to say before I pass sentence?

PRISONER: Nothing

JUDGE (*to COUNSEL* : Did your Client say anything?

COUNSEL: ‘Nothing’ my Lord.

JUDGE: Funny . . . I could have sworn I saw his lips move. . .

The N-word

One standard way of creating a [token of a] name for a word is to take a token of it and put single quotes either side of it—as indeed we have been doing above . . . “My client said ‘nothing’ my lord”. Naturally in these circumstances one wants to say that the word in question is mentioned not used. However there are circumstances in which it is felt (by some) that the word enclosed in single quotes is, nevertheless, in some sense, being used. This tends to happen when the word being mentioned is heavily taboo-ed and has to be kept at barge-pole length, in particular with words that we are not permitted to *use*. Such words can of course still be *mentioned*; after all, how can one tell a new user of the language that they are not to use the word without somehow denoting it—that is to say, mentioning it? If you mention a word you need a name for it, but a name that we use to mention it cannot be one obtained by putting single quotes round it. There is a slang American word for disparagingly denoting people with black skins; it has six letters and begins with ‘n’. I can allude to this fact, and mention the word in so doing—as i have in fact just done. But were i to mention it by enclosing a token of it in single quotes the sky would probably land on my head. My defence would be that i am not using the word, but mentioning it. I’m not going to risk it, co’s i want a quiet life. Call me a coward: i plead guilty as charged.

This is tied up with all sorts of complex and interesting issues in philosophy of language which you will have to come to grips with in the fullness of time. You could try googling ‘referential opacity’ if you want to read ahead.

The predicament of speakers in situations where the mentioned word is heavily taboo-ed is put to good comedic effect in Scene Five of *The Life of Brian* `http://montypython.50webs.com/scripts/Life_of_Brian/5.htm` where Matthias, son of Deuteronomy of Gath, is to be stoned to death for *using* the name of Jehovah. In his defence he *mentions* the name:

MATTHIAS: Look. I—I’d had a lovely supper, and all I said to my wife was, ‘That piece of halibut was good enough for Jehovah.’

So: there are some words that one wishes to mention—if at all—only by using only those names of it that do not contain embedded occurrences of it—embedded within single quotes for example. *Prima facie* there is a question about how a word can acquire other safe names in this way, and there is presumably a literature on this question ... but i don't know any of it.

Fun on a Train

The use-mention distinction is a rich source of jokes. One of my favourites is the joke about the compartment in the commuter train, where the passengers have travelled together so often that they have long since all told all the jokes they know, and have been reduced to the extremity of 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 tactfully 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 that was one they hadn't heard before.[3]

Notice that bus "numbers" are typically numerals not numbers. Not long ago, needing a number 7 bus to go home, I hopped on a bus that had the string '007' on the front. It turned out to be an entirely different route! Maybe this confusion in people's minds is one reason why this service is now to be discontinued.[4]

A good text to read on the use-mention distinction is the first six paragraphs (that is, up to about p. 37) of chapter 1 of Quine's [**?**].

Related to the use-mention distinction is the error of attributing powers of an object to representations of that object. I tend to think that this is a use-mention confusion. But perhaps it's a deliberate device, and not a confusion at all. So do we want to stop people attributing to representations powers that strictly belong to the things being represented? Wouldn't that spoil a lot of fun? Perhaps, but on the other hand it might help us understand the fun better. There was once a famous English stand-up comic by the name of *Les Dawson* who (did mother-in-law jokes but also) had a routine which involved playing the piano *very badly.* I think Les Dawson must in fact have been quite a good pianist: if you want a sharp act that involves playing the piano as badly as he seemed to be playing it you really have to know what you are doing[5]. The moral is that perhaps you only experience the full *frisson* to be had from use-mention confusion once you understand the use-mention distinction properly.

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,

[3] For sophisticates: this is a joke about *dereferencing*.

[4] But it's obvious anyway that bus numbers are not numbers but rather strings. Otherwise how could we have a bus with a "number" like '7A'?

[5] Wikipædia confirms this: apparently he was an accomplished pianist.

but the numerals we associate with each number change. Thus the numerals '`XI`', 'B', '11', '13' '1011' all represent the same number.[6]

Hotter Temperatures on the Way

No! Hotter *weather* on the way. Temperatures are not hot. They are numbers. Numbers are big or small, or lucky or unlucky.

Rockets

> *Missing: Number of children fleeing care in Cambridgeshire rockets*
>
> Cambridge News, 17:44, 21 Feb 2017
> `http://www.cambridge-news.co.uk/news/cambridge-news/missing-number-children-fleeing-care-1263`

The [7] story is not that there are children fleeing their care home by Cambridgeshire rocket—concerning tho' that is (they *must* be in a hurry); the story is rather that the number of such children has been mislaid:

> $|\{x : \text{child}(x) \wedge x$ is fleeing care in a Cambridgeshire rocket$\}|$
> has been mislaid.

The person in charge of data capture has left it on a train. Or in a rocket, perhaps.

[6] Miniexercise: What is that number, and under which systems do those numerals represent it?

[7] Thank you, Ted Harding!

Chapter 3

Languages, Automata and the Chomsky Hierarchy

We can study syntax shorn of meaning.

New stuff to fit in

Introduction

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

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 the two players each 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, The best way to present machines is through pictures.

which is a character from an input alphabet.

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 $\ldots$.

Clearly any snakes-and-ladders board can be represented by a FSA over the alphabet $\{1, 2, 3, 4, 5, 6\}$ with fewer than 100 states and precisely one accepting state. (*Fewer* than 100. . . ? Reflect that no square that is the point of embarkation for a ladder- or snake-trip can ever be occupied, so it's not really a state.)

3.0.1 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.

Beware mathmospeak

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!) Of course in linguistics the asterisk is used to flag a nonstandard or defective word.

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!

So 'string' corresponds to [english] sentence not to [english] word.

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

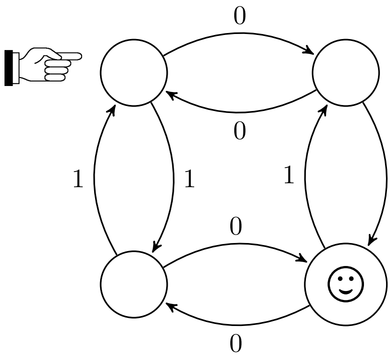
If w and u are strings then wu is w with u stuck on the end, and uw is u with w stuck on the end. Naturally $|uw| = |wu| = |w| + |u|$. ϵ is just the empty string (which we met on page 25) so ϵw—the empty string with w concatenated on the end—is just w. So ww, www and $wwwww$ are respectively the result 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.

3.0.2 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.



You start at the state indicated by the finger, and move from one state to another by following the labelled arrows—the labels on the arrows are letters from the input alphabet. If you land on a state (in this case there is only one) decorated with a smiley you **accept** the string. (And this is a term of art)

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 arrows missing.

- 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.

3.0.3 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 *a*s. The corresponding machine has three states: the first means "I have seen an even number of *a*s" (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 *a*s followed by a greater number of *b*s.

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.

3.0.4 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 24. This is all explained in the new Discrete Mathematics notes.

3.0.5 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.

state transition table here

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 $w_1(w_2)^n w_3$ 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, $x(w')^n y$ is also accepted by $\mathfrak{M}$. That's four blocks of quantifiers: a lot of quantifier alternations!!

They don't know this expression

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.

3.0.6 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.

3.0.7 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 31) 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 $\{a^k b a^k : k \geq 0\}$.

3.0.8 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.

3.1 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, $KKK \ldots$

Exercise 1. *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 2. *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 26 (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 31 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

3.1.1 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 27, 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 34, 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).

3.1.2 More about bombs

3.1.2.1 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.

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 35 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 34.

3.1.2.2 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 $a^kb^ka^{k+1}b^k$.

3.1.3 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 outstanding open left brackets then the regular expression for the machine that can cope with as many as $n + 1$ outstanding open left brackets is $N^*|(0(N^*)1)^*$.

3.2 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

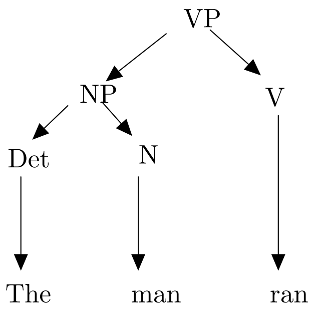
Subject followed by **Verb** followed by **Object**.

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	→	`many`
determiner	→	`the`
determiner	→	`a`
determiner	→	`some`
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, 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

$\mathrm{S} \rightarrow a\mathrm{S}$
$\mathrm{S} \rightarrow \epsilon$

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

$\mathrm{S} \rightarrow a\mathrm{S}a$
$\mathrm{S} \rightarrow b\mathrm{S}b$
$\mathrm{S} \rightarrow \epsilon$

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*

$$N \rightarrow TN'$$

or

$$N \rightarrow T$$

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 that in a context-free grammar the production rules that tell us what a nonterminal can be rewritten to do not consult the context in which the nonterminal lives.

Should Say something about how the nonterminals correspond to states of the automaton

This long quantifier prefix is worth making a fuss about to the linguists

3.2.1 Exercises

Exercise 3.
Provide a context-free grammar for regular expressions over the alphabet $\{a, b\}$

3.3 Actual Real, Live Regular Languages

3.3.1 Some semantics for a regular language

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. The language of strings for notating natural numbers consists of any string from this alphabet that doesn't start with a '0'. This language is clearly regular.

Let's have a grammar for it, and some rule-to-rule semantics

A Grammar for base-10 notation for Natural Numbers:

`singlecharacter` → 0|1|2|3|4|5|6|7|8|9

`nonzero` → 1|2|3|4|5|6|7|8|9

`expression` → `nonzero tailexpression`

`tailexpression` → `tailexpression singlecharacter`| ϵ

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).[1]

You might prefer your numbers to be written out with commas to make them easier to read: '1,000,000' instead of '1000000'. This is slightly more complicated...

It seems that most if not all natural languages are context-free

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 en-

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

tirely 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.

3.4 Some semantics for a regular language

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. The language of strings for notating natural numbers consists of any string from this alphabet that doesn't start with a '0'. This language is clearly regular.

Let's have a grammar for it, and some rule-to-rule semantics

A Grammar for base-10 notation for Natural Numbers:

`singlecharacter` → 0|1|2|3|4|5|6|7|8|9

`nonzero` → 1|2|3|4|5|6|7|8|9

`expression` → `nonzero` `tailexpression`

`tailexpression` → `tailexpression` `singlecharacter`| ϵ

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]

You might prefer your numbers to be written out with commas to make them easier to read: '1,000,000' instead of '1000000'. This is slightly more complicated...

An exercise here!

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

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.

Roman numerals? phonology rules? Coats of arms?

> *Gules, a cross ermine between four lions passant guardant Or, charged with a closed book fesswise of the first, clasped and garnished of thesecond, the clasps to base.*

A description of Oxford University's coat of arms. Presumably in a regular language.

Ambiguous parses; decidability of language equivalence.

No ambiguous parses in regular languages. Why not?

In any natural language, the language of strings of phonemes compliant with the phonological rules of that language form a regular language.

Phonological 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. An exercise here!

[should really say something about this: get a right-regular grammar]

3.4.1 Some semantics for a regular language

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. The language of strings for notating natural numbers consists of any string from this alphabet that doesn't start with a '0'. This language is clearly regular.

Let's have a grammar for it, and some rule-to-rule semantics.

A Grammar for base-10 notation for Natural Numbers:

`singlecharacter` → 0|1|2|3|4|5|6|7|8|9

`nonzero` → 1|2|3|4|5|6|7|8|9

`expression` → `nonzero tailexpression`

`tailexpression` → `tailexpression singlecharacter`| ϵ

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).[3]

You might prefer your numbers to be written out with commas to make them easier to read: '1,000,000' instead of '1000000'. This is slightly more complicated. . .

Vowel harmony?

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.

3.5 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*[4]. 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

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

[4] 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.

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 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!

There seems to be a hiatus here—nondeterministic machines!

Chapter 4

Propositional Logic

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

Where do we make a fuss about our connectives being truth-functional? Here? or in the intension/extension section

A formula (or sentence) is either

- A letter: p, q, r …; or
- The result of putting '$\wedge$', '$\vee$', '$\to$' 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 (context-free) 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 '$\to$' 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 such things *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

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

above and below the line

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, particularly the God example. Talk about $\vee$ and $\wedge$ (‘wedge’) and $\neg$ and truth tables. Talk about rules for connectives. (but not $\to$-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 11.1.1)

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

Exercise 4.

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!)*

   The set of basic disjunctions forms a regular language.

   Exercise: Write a machine that recognises it.

   (Think: what is the alphabet?)

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

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?)

have to be very careful saying things like this

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

4.1 Formal Semantics for Propositional Logic

A language in the sense of the previous chapter is a set of strings, nothing more. However as linguists we are more likely to be interested in languages whose formulæ (expressions, strings, whatever) can be evaluated to a *meaning*.. In this chapter we consider how to describe this process in a formal rigorous way.

However, in order to illustrate the techniques we are going to use we are going to start with semantics for something even simpler than the Propositional Calculus.

These strings (which are compound numerals) evaluate to numbers; the expressions of Propositional Calculus evaluate to truth values.

PC is not a single language but a family of languages. Each of these languages contains variables and the variables have to be told what to evaluate to. Thus semantics for a propositional language is that process that tells us how to evaluate a complex expression on being told how to evaluate the propositional variables that appear within it. A **valuation** [for a given 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 `sat` between valuations and complex expressions. We do this by recursion or (as you would say) *compositionally*.

Definition 2.

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

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

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

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

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

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 $\texttt{sat}(v, \phi)$ is $\texttt{sat}(v, \psi_1) \wedge \texttt{sat}(v, \psi_2)$"

or perhaps

$$\texttt{sat}(v, \psi_1 \wedge \psi_2) \text{ is } \texttt{sat}(v, \psi_1) \wedge \texttt{sat}(v, \psi_2) \tag{4.1}$$

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

$$\texttt{sat}(v, \psi_1 \wedge \psi_2) \tag{4.2}$$

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 4.1 would be OK if we write it as

$$\texttt{sat}(v, \ulcorner\psi_1 \wedge \psi_2\urcorner) \text{ is } \texttt{sat}(v, \psi_1) \wedge \texttt{sat}(v, \psi_2) \tag{4.3}$$

Corner quotes were first developed in [26]. 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

$$\psi_1 \texttt{ and } \psi_2 \qquad \text{(A)}$$

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.

> [Actually let's introduce here a bit of semi-standard notation: instead of writing '$\texttt{sat}(\phi, v)$' we'll write '$[[\phi]]_v$' or perhaps just '$[[\phi]]$' if v is understood or doesn't matter.]

Might want to put this somewhere else

4.1.1 Envoi

Those last few paragraphs were really only for linguistic sophisticates.

The crucially important moral that I want you to take away from this discussion of easy cases of semantics is *the way in which the semantics of a recursively defined language is driven, item by item, by the clauses that make up the definition.* Every way of making new expressions from old has a corresponding semantical rule. The phrase *rule-to-rule semantics* is sometimes used. Definition 2 looks tedious and obsessional and too trivial to be of any interest but it exhibits the fundamental features of importance that you need to master. [namely..?]

4.2 Confluence, and Eager and Lazy Evaluation

4.2.1 Confluence

The recursive definition of the satisfaction relation between valuations and complex formulæ 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. The recursive definition tells us uniquely what the answer must be but it doesn't tell us uniquely how to calculate it. 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.* It wasn't until linguists and computer scientists took an interest in these matters that anyone thought the details of the calculation could be interesting. And those details are actually very interesting, very interesting indeed. They introduce us to two new ideas, or perhaps three depending on how you individuate ideas. The first idea is *confluence*.

In calculating the truth-value of ϕ according to a valuation v for some complex expression ϕ we start with the truth-values (according to v) of the propositional letters inside ϕ and use the recursion to calculate the truth-values-according-to-v of ever larger subformulæ of ϕ until we reach ϕ itself. If two subformulæ are $p \wedge q$ and $s \wedge t$ there is nothing to say that we should compute the truth-value-according-to-v of $p \wedge q$ before we compute the truth-value-according-to-v of $s \wedge t$ or the other way round. It clearly doesn't make any difference. Neither calculation interferes with the other. The process that the two calculations are part of is said to be *confluent*. We will see manifold examples of confluent processes later: it is an important idea.

When we see λ-calculus in chapter 10 we will meet a major theorem (the Church-Rosser theorem) that says that a certain simplification process (called β-reduction) is confluent.

But we could end with a *non*-example. The process of compiling a guest list for a party is not reliably confluent. At any stage in the compilation you may add to the growing list anyone who is on speaking terms with everyone on the list so far. But Arthur and Bertha might both be on speaking terms with everyone on the list l but not with each other!

4.2.2 Eager and Lazy Evaluation

The other idea is that of *evaluation strategy*. The way of calculating $\mathtt{sat}(\phi, v)$ 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.

In some sense we are doing the same things in these two settings, of eager and lazy evaluation. But there's a sense in which we are not doing the same things, in that there are more moves we are allowed to make in the lazy case than in the eager case. Nevertheless

(i) Both projects are confluent, and

(ii) The two projects give the same answer in the end.

I'm not offering to prove these allegations. I'm hoping they are obvious, or at least plausible.

However the situation is more complicated if the evaluation functions we are processing are not total. Then it is no longer true that lazy and eager evaluation give the same answer. Suppose we want the truth-value of $A \vee B$, and we know that A is true. Lazy evaluation will infer that $A \vee B$ is true; however eager evaluation will not say that $A \vee B$ is true until it ascertains $[[B]]$, the truth-value of B. This means that if, for any reason, $[[B]]$ is undefined then eager evaluation will never tell us that $A \vee B$ is true. notation!!

To get realistic examples of situations where our valuations are partial functions we have to go to a rather more mathematical setting—the setting, in fact, where these ideas of eager-and-lazy first arose, namely in connection with languages whose expressions evaluate to numbers or other data objects. For example:

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

No point in calculating all three of $[[x \geq 0]]$, $f(x)$ and $g(x)$ and then deciding which of $f(x)$ and $g(x)$ to output. 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.

The standard examples of cases where lazy evaluation is something you have to think about are mathematical—like the above only worse—but I came across this illustration in the course of my EEG work when I had a patient with HHT—*hereditary hæmorrhagic telangiectasia*. You don't need to know what it is—indeed I didn't know what it was either, and I had to look it up! This is what I found.

There are four diagnostic criteria, otherwise known as the *Curaçāo criteria*, named after the country in which was held the meeting that formally defined hereditary hæmorrhagic telangiectasia. Any patient who ticks at least two of the following boxes has HHT.

1. Spontaneous recurrent epistaxis;
2. Multiple teleangiectasias on typical locations;
3. Proven visceral arterio-venous malformation (lung, liver, brain, spine);
4. First-degree family member with HHT.

(For current purposes you do not need to know what any of *epistaxis, teleangiectasias, visceral* etc mean!)

That is to say, you want to verify the disjunction

$$(1 \wedge 2) \vee (1 \wedge 3) \vee (1 \wedge 4) \vee (2 \wedge 3) \vee (2 \wedge 4) \vee (3 \wedge 4)$$

Clearly your evaluation strategy is not going to be the eager strategy of evaluating all the disjuncts first! That would result in a task that would never halt.[2] Clearly the sensible thing to do is to try 1, 2 and 3 first and then if two of them succeed you don't need to check 4; if only one of them succeeds then you need to check 4; if none of 1, 2 and 3 succeed then you don't need to check 4!

Lazy evaluation is an important concept in the semantics of ordinary language, but that's not beco's natural language has feedback formulæ, but rather because an expression in spoken natural language is not presented to the hearer as a complete static object, which can then be contemplated the way we contemplated $A \wedge (B \vee (C \wedge D))$ above—in its entirely.

4.2.3 Sorites

I take it you are all familiar with the Sorites paradox?

How is it that we are ever able to get definite answers to vague questions? Any theory of vague predicates must explain how this can happen. Because we do get definite answers.

One of my pet hunches is that consideration of evaluation strategies may hold the key to understanding the semantics of vague predicates. A vague predicate $V(\ )$ typically has a large family of non-vague *sharpenings* so that $V(x)$ is a disjunction of lots of sharpenings $V(x) \longleftrightarrow V_1(x) \vee V_2(x) \vee \ldots V_n(x)$.

For example, the vague concept of **adult** is a disjunction of (among others) the non-vague predicates

Starting compulsory schooling	5
Criminal responsibility	12
Making binding contracts (Scotland), girls	12
Baby sitting	14
Being lent a shotgun, to use without certificate on owner's premisses	14
Making binding contracts (Scotland), boys	14
Entering licensed premisses	14
Being given a shotgun, if holding certificate	15
Stopping compulsory schooling	16
Buying cigarettes	16
Heterosexual intercourse	16
Driving motorcycle	16
Marriage without parental consent (Scotland)	16
Buying/consuming cider/perry on licensed premisses	16
Driving car	17
Buying or hiring shotgun or ammunition	17
Marriage without parental consent (England)	18

[2] Or, at least, could never be completed: Your parents are first-degree family members, and although many people have parents living, and many have grandparents living, nobody has *all* their ancestors living!

Voting	18
Making binding contracts (England)	18
Buying or consuming alcohol on licenced premisses	18
Homosexual intercourse (male)	21
Adopting a child	21

The table comes from [20].

One obtains a determinate (non-vague) answer to a question “Does x bear this [vague] predicate by evaluating the disjunction lazily.

To summarise:

- Lazy evaluation differs from eager evaluation in that it allows more reductions, more inferences;
- Typically (e.g. in propositional logic) both sets of rules are confluent;
- If the valuations are total functions then lazy and eager evaluation give the same answers;
- If the valuations are not total (allowing a third outcome, `fail` or `crash`) then lazy and eager evaluation can disagree. In this setting we do need to treat (i) the cases of failure that are revealed only at the end of time differently from (ii) failures that are revealed in this life by a loud bang and smoke and lights going out;
- Notice that the difference between lazy and eager has no bite if the language for which we are providing semantics is regular!
- Streams?

4.3 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 10. I shall put them on the blackboard but not expect you to master them.

4.3.1 The Rules of Natural Deduction

In the following table we see that for each connective we have two rules: one to introduce the connective and one to eliminate it. These two rules are called the **introduction rule** and the **elimination rule** for that connective.

Richard Bornat calls the elimination rules “use” rules because the elimination rule for a connective $\mathcal{C}$ tells us how to **use** the information wrapped up in a formula whose principal connective is $\mathcal{C}$.

(The idea that everything there is to know about a connective can be captured by an elimination rule plus an introduction rule has the same rather operationalist flavour possessed by the various *meaning is use* doctrines one encounters in philosophy of language. In this particular form it goes back to Prawitz, and possibly to Gentzen.)

references?

The rules tell us how to exploit the information contained in a formula. (Some of these rules come in two parts.)

Introduction Rules	Elimination Rules
$\vee$-int: $\frac{A}{A\vee B}$; $\frac{B}{A\vee B}$;	$\boxed{\vee\text{-elim ???}}$
$\wedge$-int: $\frac{A \quad B}{A\wedge B}$;	$\wedge$-elim: $\frac{A\wedge B}{A}$; $\frac{A\wedge B}{B}$
$\boxed{\to\text{-int ???}}$	$\to$-elim: $\frac{A \quad A\to B}{B}$

You will notice the division into two columns. You will also notice the two *lacunæ*: for the moment there is no $\vee$-use rule and no $\to$-int rule.

Some of these rules look a bit daunting so let's start by cutting our teeth on some easy ones.

Exercise 5.

1. *Using just the two rules for $\wedge$, the rule for $\vee$-introduction and $\to$-elimination see what you can do with each of the following sets of formulæ:*[3]

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

2. *Deduce C from $(A \vee B) \to C$ and A;*
   Deduce B from $(A \to B) \to A$ and $A \to B$;
   Deduce R from P, $P \to (Q \to R)$ and $P \to Q$;

You will probably notice in doing these questions that you use one of your assumptions more than once, and indeed that you have to *write it down* more than once (= write down more than one token!) This is particularly likely to happen with $A \wedge B$. If you need to infer both of A and B then you will have to write out '$A \wedge B$' *twice*—once for each application of $\wedge$-elimination. (And

[3] Warning: in some cases the answer might be "nothing!".

of course you are allowed to use an assumption as often as you like. If it is a sunny tuesday you might use $\wedge$-elimination to infer that it is sunny so you can go for a walk in the botanics, but that doesn't relieve you of the obligation of inferring that it is tuesday and that you need to go to your 11 o'clock lecture.)

If you try writing down only one token you will find that you want your sheet of paper to be made of lots of plaited ribbons. Ugh. How so? Well, if you want to infer both A and B from $A \wedge B$ and you want to write '$A \wedge B$' only once, you will find yourself writing '$\frac{A \wedge B}{A \quad B}$' and then building proofs downward from the token of the 'A' on the lower line and also from the 'B' on the lower line. They might rejoin later on. Hence the plaiting.

Now we can introduce a new rule, the *ex falso sequitur quodlibet.*
Ex falso sequitur quodlibet; $\frac{\bot}{A}$
Double negation $\frac{\neg\neg A}{A}$

better explain at some point why it is truth-preserving

Is this the right place for double neg? aka classical contradiction.

The Latin expression *ex falso* ... means: "From the `false` follows whatever you like".

The two rules of *ex falso* and *double negation* are the only rules that specifically mention negation. Recall that $\neg B$ is logically equivalent to $B \to \bot$, so the inference

$$\frac{A \qquad \neg A}{\bot} \tag{4.4}$$

—which *looks* like a new rule—is merely an instance of $\to$-elimination.

The rule of $\to$-introduction

The time has now come to make friends with the rule of $\to$-introduction. Recalling what introduction rules do, you can se that the $\to$-introduction rule will be a rule that tells you how to prove things of the form $A \to B$. Well how, in real life, do you prove "if A then B"? Well, you assume A and deduce B from it. What could be simpler!? Let's have an illustration. We already know how to deduce $A \vee C$ from A (we use $\vee$-introduction) so we should be able to prove $A \to (A \vee C)$.

$$\frac{A}{A \vee C}\ \vee\text{-int} \tag{4.5}$$

So we just put '$A \to (A \vee C)$' on the end ... ?

$$\cfrac{\cfrac{A}{A \vee C}\ \vee\text{-int}}{A \to (A \vee C)} \tag{4.6}$$

That's pretty obviously the right thing to do, but for one thing. The last proof has $A \to (A \vee C)$ as its last line (which is good) but it has A as a live premiss. We assumed A in order to deduce $A \vee C$, but although the truth of $A \vee C$ relied on the truth of A, the truth of $A \to (A \vee C)$ does not rely on the truth of A. (It's a tautology, after all.) We need to record this fact somehow. The point is that, in going from a deduction-of-$A \vee C$-from-A to a proof-of-$A \to (A \vee C)$, we have somehow *used up* the assumption A. We record the fact that it has been used up by putting square brackets round it, and putting a pointer from where the assumption A was made to the line where it was used up.

$$\dfrac{\dfrac{[A]^1}{A \vee C}\,\vee\text{-int}}{A \to (A \vee C)}\,\to\text{-int (1)} \tag{4.7}$$

N.B.: in →-introduction you don't have to cancel all occurrences of the premiss: it is perfectly all right to cancel only some of them .

The rule of ∨-elimination

"they will either contradict the Koran, in which case they are heresy, or they will agree with it, so they are superfluous."

Here is an example, useful to those of you who fry your brains doing sudoku.

	3	8						
	1	6		4		9	7	
4		7	1					6
		2	8		7			5
	5			1			8	
8			4			2		
7		5			1	8		4
	4	3		5		7	1	
						6		

There is a '5' in the top right-hand box—somewhere. But in which row? The '5' in the top left-hand box must be in the first column, and in one of the top two rows. The '5' in the fourth column must be in one of the two top cells. (It cannot be in the fifth row because there is already a '5' there, and it cannot be in the last three rows because that box already has a '5' in it.) So the '5' in the middle box on the top must be in the first column, and in one of the top two rows. These two '5's must of course be in different rows. So where is the '5' in the rightmost of the three top boxes? Either the '5' in the left box is on the first row and the '5' in the middle box is on the second row or the 5 in the middle box is in the first row and the '5' in the left box is in the second row. We don't know which of the possibilities is the true one, but it doesn't matter: either way the '5' in the rightmost box must be in the bottom (third) row.

Need detailed explanation of ∨-elim here

The Identity Rule

Finally we need the identity rule:

$$\frac{A\ B\ C\ldots}{A} \tag{4.8}$$

(where the list of extra premisses may be empty) which records the fact that we can deduce A from A. Not very informative, one might think, but it turns out to be useful. After all, how else would one obtain a proof of the undoubted tautology $A \to (B \to A)$, otherwise known as 'K'? One could do something like

$$\dfrac{\dfrac{\dfrac{\dfrac{[A]^2 \qquad [B]^1}{A \wedge B}\ \wedge\text{-int}}{A}\ \wedge\text{-elim}}{B \to A}\ \to\text{-int (1)}}{A \to (B \to A)}\ \to\text{-int (2)} \tag{4.9}$$

but that is grotesque: it uses a couple of rules for a connective that doesn't even appear in the formula being proved! The obvious thing to do is

$$\dfrac{\dfrac{\dfrac{[A]^2 \qquad [B]^1}{A}\ \text{identity rule}}{B \to A}\ \to\text{-int (1)}}{A \to (B \to A)}\ \to\text{-int (2)} \tag{4.10}$$

If we take seriously the observation above concerning the rule of →-introduction—namely that you are not required to cancel *every* occurrence of an assumption—then you conclude that you are at liberty to cancel *none* of them, and that suggests that you can cancel assumptions that aren't there—then we will not need this rule. This means we can write proofs like 4.11 below. To my taste, it seems less bizarre to discard assumptions than it is to cancel assumptions that aren't there, so I prefer 4.10 to 4.11. It's a matter of taste.

$$\dfrac{\dfrac{[A]^1}{B \to A}\ \to\text{-int}}{A \to (B \to A)}\ \to\text{-int (1)} \tag{4.11}$$

It is customary to connect the several occurrences of a single formula at introductions (it may be introduced several times) with its occurrences at elimination by means of superscripts. Square brackets are placed around eliminated formulæ, as in the formula displayed above.

There are funny logics where you are not allowed to use an assumption more than once: in these **resource logics** assumptions are like sums of money. This also gives us another illustration of the difference between an argument (as in logic) and a debate (as in rhetoric). In rhetoric it may happen that a point—even a good point—can be usefully made only once ... in an ambush perhaps. One such logic is **Linear Logic**, and it has been alleged that it could be useful in linguistics. However I am not going to treat it here.

Do some very simple illustrations of compound proofs here

4.3.2 What do the rules *mean*??

One way in towards an understanding of what the rules do is to dwell on the point made by my friend Richard Bornat that elimination rules are **use** rules:

The rule of →-elimination

The rule of →-elimination tells you how to use the information wrapped up in '$A \to B$'. '$A \to B$' informs us that if A, then B. So the way to use the information is to find yourself in a situation where A holds. You might not be in such a situation, and if you aren't you might have to assume A with a view to using it up later—somehow. We will say more about this.

The rule of ∨-elimination

The rule of ∨-elimination tells you how to **use** the information in '$A \vee B$'. If you are given $A \vee B$, how are you to make use of this information without knowing which of A and B is true? Well, **if** you know you can deduce C from A, and you ALSO know that you can deduce C from B, **then** as soon as you are told $A \vee B$ you can deduce C. One could think of the rule of ∨-elimination as a function that takes (1) $A \vee B$, (2) a proof of C from A and (3) a proof of C from B, and returns a proof of C from $A \vee B$. This will come in useful on page **??**.

There is a more general form of ∨-elimination:

$$\dfrac{\begin{array}{ccccc} [A_1]^1 & [A_2]^1 & \ldots & [A_n]^1 & \\ \vdots & \vdots & & \vdots & \\ C & C & & C & A_1 \vee A_2 \vee \ldots A_n \end{array}}{C}\ \vee\text{-elim (1)} \tag{4.12}$$

where we can cancel more than one assumption. That is to say we have a *set* $\{A_1 \ldots A_n\}$ of assumptions, and the rule accepts as input a list of proofs of C: one proof from A_1, one proof from A_2, and so on up to A_n. It also accepts the disjunction $A_1 \vee \ldots A_n$ of the set $\{A_1 \ldots A_n\}$ of assumptions, and it outputs a proof of C.

The rule of ∨-elimination is a hard one to grasp so do not panic if you don't get it immediately. However, you should persist until you do. Some of the challenges in the exercise which follows require it.

Exercise 6.
Deduce $P \to R$ *from* $P \to (Q \to R)$ *and* $P \to Q$*;*
Deduce $(A \to B) \to B$ *from* A*;*
Deduce C *from* A *and* $((A \to B) \to B) \to C$*;*
Deduce $\neg P$ *from* $\neg(Q \to P)$*;*
Deduce A *from* $B \vee C$*,* $B \to A$ *and* $C \to A$*;*
Deduce $\neg A$ *from* $\neg(A \vee B)$*;*

Deduce Q *from* P *and* $\neg P \vee Q$*;*
Deduce Q *from* $\neg(Q \to P)$*.*

Will need to cut this back heavily

Chapter 5

Predicate (first-order) Logic

5.1 Towards First-Order Logic

Drill down, but how far? Lloyd Reinhardt's story.

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 whole 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—after all, 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.

5.2 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 67 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 [18]. (See p **??**.)

> 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 7. *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.]

5.3 The Syntax of First-order Logic

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

Binders!

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

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

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:

5.3.1 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* . . . .

5.3.2 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;
$(\forall x)(\exists y)L(x, y)$	Everybody loves someone

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

5.3.3 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.

5.4 Warning: Scope ambiguities

Perhaps this could be moved into a section called something like 'subtleties of evaluation'

"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

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

All Frenchmen are not racist

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!

Worth making the point that resolution of scope ambiguities by rearrangement in this way is a part of natural-language semantics but is not part of the semantics of formal logics. Another way in which life is easier for logicians than it is for linguists!

5.5 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*.

5.6 Some exercises to get you started

Exercise 8.
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 9. *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 10. *These involve nested quantifiers and dyadic predicates*
Match up the formulæ on the left with their English equivalents on the right.

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

EXERCISE 11. *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 12. *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.*

5.7 Transitive, reflexive etc

Armed with this new language we can characterise some important properties: (Ideally we would have dealt with this stuff in section 2.2 but at that stage we didn't have the notaton)

- 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.

5.8 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

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

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

5.9 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.

Logicians dislike and distrust second order logic for various reasons that need not concern linguists. If you are using logic in a descriptive way rather than a normative way, so that you are looking to it to furnish a regimented version of ordinary language of the kind that linguists study, then you will definitely need second order logic: Kasia's example "swimming is healthy" certainly looks like a wff from a second-order language. (Tho' you could gloss it as: people who swim are *ceteris paribus* healthier than people who don't.)

5.9.1 Many-sorted

Difference between two-sorted and second-order. Many-sorted.

(possibly copy stuff from part II logic lecture notes)

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.

Possibly say more about this

5.10 Semantics for first-order logic

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. It's all 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 is too slow) that I won't even get round to lecturing it.

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 $\mathfrak{FRAKTUR}$ 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 50. We will need it, but we have to do some preparatory work first.

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

5.10.1 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.

5.10.2 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**.

5.10.3 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 51. type-token

5.10.4 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'. [...]

5.10.5 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.

5.11 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.

Exercise 13. *Let L be a regular language over an alphabet Σ_2 and let Σ_1 be a subset of Σ_2. Let $L \upharpoonright \Sigma_1$ be the set of all strings $w' \in \Sigma_1^*$ such that there is $w \in L$ where w' is the result of deleting from w all characters in $\Sigma_2 \setminus \Sigma_1$. [Why is this not the same as $L \cap \Sigma_1^*$?]*

> *(i) Show that $L \upharpoonright \Sigma_1$ is regular.*
>
> *(ii) Deduce that the language of (wellformed) propositional formulæ (over some fixed alphabet) is not 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 16.

> "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.**

Chapter 6

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 77) and `ind`; more complex syntactic types are built up from them.

Somewhere make the point that nouns and verbs are of the same syntactic type. Binders: set abstracts, λ-terms and ϵ-terms. What is the syntactic type of a binder?

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 the propositional letters 'p' and 'q' and suchlike you met in chapter 4 and all the complex formulæ built up from them by means of the connectives are of syntactic type `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`)

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`.

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* (= thing of type `ind -> bool`) not two *statements* (= thing of type `bool`) so it is clearly—in this case—of syntactic type

`(ind -> bool)` × `(ind -> bool) -> (ind -> bool)`.

Actually, without *tooo* much striving, one can find analogous examples where Fred has been replaced by a tuple:

> William and Kate are married and are launching a new aircraft-carrier

in which 'and' is of type

$$\texttt{((ind}^2\texttt{ -> bool)} \times \texttt{(ind}^2\texttt{ -> bool)) -> (ind}^2\texttt{ -> bool)}$$

So, really, 'and' and its analogues are of ["polymorphic"] type

$$\texttt{((ind}^n\texttt{ -> bool)} \times \texttt{(ind}^n\texttt{ -> bool)) -> (ind}^n\texttt{ -> bool)}$$

where n can be any whole number. What happens if $n = 0$? The expression simplifies to `bool` × `bool -> bool`.

What about 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 free variables then the result of whacking a single quantifier on the front has $n - 1$ free variables, so really a quantifier can have syntactic type
$\texttt{(ind}^n\texttt{ -> bool) -> (ind}^{(n-1)}\texttt{ -> bool)}$, for any n.

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

What about determiners? A determiner is a thing that takes a predicate symbol and returns a quantifier. For example '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

$$\texttt{(ind -> bool) -> (ind}^n\texttt{ -> bool) -> (ind}^m\texttt{ -> bool).}$$

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’.

Adverbial Modifiers and Modal Operators

Adverbial modifiers can clearly be of syntactic type

$$\texttt{(ind}^{n}\ \ \texttt{-> bool) -> (ind}^{n}\ \texttt{-> bool)}$$

for any $n > 0$.

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

$$\texttt{(ind}^{n}\ \ \texttt{-> bool) -> (ind}^{n}\ \texttt{-> bool)}$$

for any $n \geq 0$.

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

Adverbs lead us straight to the next chapter.

Chapter 7

Modal Logic

Modern mathematical logic has founded itself on truth-functional connectives, and this restriction has proved very fruitful. Initially one reason why it was fruitful was that nobody had any clue how to do semantics for non-truth-functional (intensional) connectives. The obvious examples of such intensional connectives are modal operators. Linguists know all about modal verbs! Philosophers were interested in the possibility of intensional logics because they wanted to reason about necessity and contingency: *Neccessarily* $2+2=4$ (written '$\Box(2+2=4)$') and its dual: $\Diamond$. There was 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 connectives $\Box$ and $\Diamond$ are supposed to capture.

The symbols '$\Box$' and '$\Diamond$' were draughted in to stand for 'neccessarily' and 'possibly'—two notions which modern logic inherited from mediæval philosophy.

The modal operators '$\Box$' and '$\Diamond$' can be applied to formulæ with free variables in them and so can be of syntactic type

$$\texttt{(ind}^n\ \texttt{-> bool) -> (ind}^n\ \texttt{-> bool)}$$

for any $n \geq 0$.

Natural languages have lots of lexical items of this flavour; we call them *adverbs*. 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*.

Adverbs are not truth-functional (The truth value of "Balbus loves Julia very much" does not depend merely on the truth-value of "Balbus loves Julia")

That all changed in 1957[1], when Kripke fully spelled out the technique we are going to see in this chapter.

[1] The prehistory is contentious, let us say.

7.1 Possible World Semantics

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

In section 4.1 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. That is to say, the definition of $sat(v, \phi)$ makes no reference to any valuation other than v. The key thought is that if you compel the definition of $sat(v, \phi)$ to consult valuations other than v then you will get a much richer semantics. In this new setting we call the valuations *worlds* and we have an “accessibility” relation between the worlds.

DEFINITION 3. *A* **Possible World Model** $\mathfrak{M}$ *has several components:*

- *There is a collection of* **worlds** *and a binary relation of satisfaction between worlds and formulæ, written ‘*$W \models \phi$*’;*
- *There is a binary relation* R *of* **accessibility** *between the worlds; if* $R(W_1, W_2)$ *we say* W_1 *can* **see** W_2*.*
- *Each world may have inhabitants, and we may stipulate* $W \models \phi(\vec{x})$ *for atomic* ϕ *and tuples* $\vec{x}$ *of inhabitants of* W*.*
- *Finally there is a* **designated** *(or ‘actual’ or ‘root’) world* W_0^M*. We say that* $\mathfrak{M}$ *satisfies* ϕ *if* $W_0 \models \phi$*.*

Some comments:

• Notice that it is **only when** ϕ **is atomic** that we are free to stipulate that $W \models \phi(\vec{x})$. Furthermore, altho’ declining-to-stipulate-that-$W \models \phi(\vec{x})$ is equivalent to a stipulation that $W \not\models \phi(\vec{x})$ (as long as ϕ is atomic) it does not amount to a stipulation that $W \models \neg\phi(\vec{x})$. All will become clear below!

• It is a side-effect of our definitions that $W \models \bot$ never holds. We write this as $W \not\models \bot$. We cannot declare this explicitly—as Aristotle said, a definition cannot be negative—but it works out that way’

• I wouldn’t stake my life on it but I think we generally take it that our worlds are never empty: every world has at least one inhabitant. However there is emphatically 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. Thus the apparatus allows us to stipulate that a particular object might have ϕ in one world but not in another.

Armed with this, we can sex up the various clauses in the definition of the satisfaction relation. Some remain unaltered:

- $W \models A \wedge B$ iff $W \models A$ and $W \models B$;

- $W \models A \vee B$ iff $W \models A$ or $W \models B$;
- $W \models (\exists x)A(x)$ iff there is an x in W such that $W \models A(x)$.

but some now make use of the accessibility relation. Then we can give rules like

- $W \models \Box A$ iff every $W' \models A$ for every W' s.t. $R(W, W')$;
- $W \models \Diamond A$ iff $W' \models A$ for at least one W' s.t. $R(W, W')$;
- $W \models A \to B$ iff every W' such that $R(W, W')$ that $\models A$ also $\models B$;
- $W \models \neg A$ iff there is no W' such that $R(W, W')$ and $W' \models A$;
- $W \models (\forall x)A(x)$ iff for all $W' \, R \, W$ and all x in W', $W' \models A(x)$.

The apparatus of possible world semantics puts **no restrictions whatever** on what properties the accessibility relation R 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. As remarked above, this was one of the problems the invention of this semantics was supposed to solve.

The reader might wish to check that There is an arrow in here

Exercise 14. *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 5.7.

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.

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.

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.

7.2 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 '$\longrightarrow$' 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

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

This illustration uses a constructive arrow

then it is not the case that

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

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 metalanguage, even though it might perhaps not be allowed in the object language. [depends what we are trying to model]

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 $\neg(A \wedge B)$). After all, that principle might not be good in the logic we are trying to model.

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**.)

(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

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.

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

7.2.2 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. We can 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 W can see no world that believes A. So $W \models \neg A$ and thus $W \models (A \vee \neg A)$. So W believes the law of excluded middle.

ators are trivial if there is only one world

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.

Chapter 8

Enhanced syntax

8.1 Quantifiers

We know about $\exists$ and $\forall$. They are not the only ones! The-king-of-France is another one. In Mathematical Logic we consider also "for all but finitely many" written '$\forall_\infty$' and its dual $\exists_\infty$ "there are at least infinitely many".

Mathematically we think of a quantifier Q over a domain D as a set of subsets of D, so that $(Qx)\phi(x)$ is saying that $\{x : \phi(x)\} \in Q$. Thus, to illustrate, $\exists$ is the set of all nonempty subsets of D, and $\forall$ is the singleton $\{D\}$ of D. Also $\exists_\infty$ is the set of all infinite subsets of D, and $\forall$ is the collection of subsets of D with finite complement ("cofinite").

8.1.1 Predicate Modifiers

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, which will always point to a partial order. 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.

Tho' we don't really think of these things as part of the language. Sort this out!

"x is the King of France" according to Russell's analysis is

$$K(x) \wedge (\forall y)(K(y) \to y = x)$$

So "x is the coolest" will be

$$(\forall z)(\text{cooler-than}(x, z)) \wedge (\forall y)((\forall z)(\text{cooler-than}(y, z) \to y = x)).$$

Another predicate modifier is *too*.

No woman can be too thin or too rich.

We will not consider them further.

8.2 epsilon terms

They were invented by Hilbert for reasons that need not concern us. The syntax is that $(\epsilon x)\phi(x)$ is an object of type `ind`, so that ϵ is a binder (like a quantifier or the set-forming brackets $\{,\}$) of syntactic type (`ind` $\to$ `bool`) $\to$ `ind`. The idea is that $(\epsilon x)\phi(x)$ is a thing about which one knows only that it bears the property ϕ as long as there is something that bears ϕ. If there is nothing that is ϕ then we know nothing about it at all. The word 'generic' comes to mind: $(\epsilon x)\phi(x)$ is a bit like a generic thing-that-is-ϕ, except that it exists even if there is nothing that is ϕ. This genericity makes it attractive to linguists of a certain stamp contemplating assertions like "a whale is a mammal". You might think that this could be formalised using ϵ terms as

$$\text{Mammal}((\epsilon x)(\text{whale}(x)))$$

but that doesn't work. The only thing one knows about the generic whale ("the epsilon whale") is that it is a whale if there are any. If there aren't any whales then all bets are off, and if there are any whales then it is one and has all those properties that all whales have, whatever they are. OK, if all whales are mammals and there are whales, then the epsilon whale is a whale, and therefore a mammal. But that's only beco's *all* whales are mammals. Is it a toothed whale or a baleeen whale? There's no telling.

The original reason for Hilbert to be interested in them was that a language qith ϵ-terms has the same expressive power as a language with the existential and the universal quantifier. This is because

$(\exists x)\text{whale}(x)$ is the same as $\text{whale}((\epsilon x)\text{whale}(x))$;

and

$(\forall x)\text{whale}(x)$ is the same as $\text{whale}((\epsilon x)\neg\text{whale}(x))$.

Quite how useful is this ability to get rid of quantifiers is anyone's guess. It is a matter of record that there are people who think that ϵ calculus is useful, and you may fall in with them.

Plurals. see [22].

Chapter 9

Game Semantics

We use `\mathfrak` font letters to range over structures, with the corresponding upper case Roman letter for the carrier set: thus A is the carrier set of $\mathfrak{A}$.

The reader is assumed to know what combinatorial games are, what strategies are, and so on. Usually one considers the possibility of nondeterministic strategies, but in this setting all our strategies will be deterministic.

9.1 Hintikka Games

Hintikka games can give us semantics for any first-order language that contains relation symbols of any arity, function symbols of any arity, individual constants and propositional constants, the connectives $\vee$ and $\wedge$, and quantifiers and $\bot$ but—for the moment—no other connectives. $\mathfrak{M}$ is a structure with carrier set M.

There are two players, **True** and **False**. **True** is female and **False** is male. They play $G(\phi, \mathfrak{M})$ as follows:[1]

Definition 4. *How to play the game $G(\phi, \mathfrak{M})$:*

> *If ϕ is $\psi_1 \wedge \psi_2$ they play $G(\psi_1, \mathfrak{M})$ and $G(\psi_2, \mathfrak{M})$, and* **True** *must win both, otherwise she loses;*
>
> *if ϕ is $\psi_1 \vee \psi_2$ they play $G(\psi_1, \mathfrak{M})$ and $G(\psi_2, \mathfrak{M})$, and* **True** *must win at least one, otherwise she loses;*
>
> *if ϕ is $\exists x \psi(x)$ they play all the games $G(\psi(m), \mathfrak{M})$ for all $m \in \mathfrak{M}$, and* **True** *must win at least one of them, or else she loses;*
>
> *if ϕ is $\forall x \psi(x)$ they play all the games $G(\psi(m), \mathfrak{M})$ for all $m \in \mathfrak{M}$, and* **True** *must win all of them, or else she loses;*

[1] We will sometimes write '$G(\phi)$' where $\mathfrak{M}$ is obvious from context or is irrelevant.

> *If we have $\bot$ as a propositional constant in the language we need the rule: if ϕ is $\bot$* **False** *wins.*
>
> *If ϕ is atomic or negatomic*[2], **True** *wins if $\mathfrak{M} \models \phi$ and* **False** *wins otherwise. (The match referees have access to the diagram of $\mathfrak{M}$.)*

In the two clauses for the quantifiers the games are not, strictly speaking, $G(\psi(m), \mathfrak{M})$, but $G(\psi(x), \mathfrak{M}[a])$, where what i really mean (and almost certainly haven't notated properly) is the Hintikka game for $\psi(x)$ played over the expansion $\mathfrak{M}[m]$ of $\mathfrak{M}$ obtained by giving a name to the element $m \in M$ and forcing 'x' to point to m.

The reader will notice that a play of $G(\phi, \mathfrak{M})$ has no moves, and that therefore there is only one strategy (the empty strategy) for each player! This makes it easy to prove by structural induction on formulæ the proposition that

Remark 1. $\mathfrak{M} \models \phi$ *iff* **True** *has a winning strategy in $G(\phi, \mathfrak{M})$.*

This is a notational triviality, and I have set up the game in this way in order to make the proof of the remark obvious. These games are normally presented quite differently, where instead of the players playing all possible games, they choose which games to play. So we change the relevant clauses to

> If ϕ is $\psi_1 \wedge \psi_2$ **False** picks a conjunct ψ_i and they play $G(\psi_i, \mathfrak{M})$;
>
> if ϕ is $\psi_1 \vee \psi_2$ **True** picks a disjunct ψ_i and they play $G(\psi_i, \mathfrak{M})$;
>
> if ϕ is $\exists x \psi(x)$ **True** picks m from M and they play $G(\psi(m), \mathfrak{M})$;
>
> if ϕ is $\forall x \psi(x)$ **False** picks m from M and they play $G(\psi(m), \mathfrak{M})$;
>
> if ϕ is atomic or negatomic, **True** wins if $\mathfrak{M} \models \phi$ and **False** wins otherwise.

and this is the version I will use, if only to secure conformity with standard usage.

The difference between the two presentations is significant. My treatment in terms of simultaneous plays makes the proof of remark 1 a deflationary triviality, provable by an induction on the subformula relation. To prove the same result for the usual definition requires an appeal to the axiom of choice, and we feel that the entry of the axiom of choice onto the stage at this point is an artefact of the presentation and makes an unwelcome distraction. Part of the project is to understand how little skolemisation has to do with the axiom of choice.

Semantics by means of Hintikka games is just the ordinary recursive semantics spiced up and made to look different. The difference is that game semantics can be sensibly invoked in settings where the usual recursive semantics cannot be applied, for example the branching quantifier language and languages with illfounded subformula relations. There are various interesting applications that we will not cover here, but which I cannot forbear to mention.

[2] '$\bot$' is of course an atomic formula.

9.2 Applications to other languages

9.2.1 Negation

There are various ways of dealing with negation but the subsequent development of these ideas is not affected in any way that I can see by any choice we make on how to treat negation, so i will ignore it.

Why is this safe? Beco's we can—in a way that preserves logical equivalence—*import* negation so that negation signs appear attached only to atomic formulæ.

One standard way of including negation in this treatment is to give the players **rôles**, and they swap rôles whenever they encounter a negation sign. Various people have had this idea but it seems to have been first published by Neil Tennant [**?**].[3]

9.2.2 Monotone quantifiers

A monotone quantifier over $\mathfrak{M}$ is simply an upward-closed subset of $\mathcal{P}(M)$, where M is the carrier set of $\mathfrak{M}$. The enhancement to deal with monotone quantifiers appears to be due to Aczel. (I learnt it from Aczel: [1].) $Qx.\psi(x)$ says simply that $\{x : \psi(x)\} \in Q$ where Q is the upward-closed subset of $\mathcal{P}(M)$ corresponding to Q. We add to the recursion the clause

> If ϕ is $(Qx)\psi(x)$ player **True** picks $X \in Q$, player **False** picks $a \in X$ and they play $G(\psi(a), \mathfrak{M})$.

This enhancement by Aczel is designed for the usual context where players make moves. However it can be modified to work in the context of definition 4. If ϕ is $(Qx)\psi(x)$ they play simultaneously all the games $(G, \psi, q, \mathfrak{M})$ for all $q \in Q$ and **True** has to win one of them. What is the game $(G, \psi, q, \mathfrak{M})$? It is the game that **True** and **False** play by playing simultaneously all the games $G(\psi(a), \mathfrak{M})$ for $a \in q$, and **False** has to win them all.)

9.2.3 Possible World Semantics

If we want a constructive or modal treatment we can easily provide one. For example **True** and **False** play $G(\Box\phi, W)$ by playing simultaneously all the games $G(\phi, W')$ for all worlds W' accessible from W—and **True** of course has to win them all.

If we want semantics for a constructive logic...

True and **False** play $G((\forall x)\phi(x), W)$ by playing simultaneously all the games $G(\phi(m), W')$ for all worlds W' accessible from W and all $m \in W'$—and **True** of course has to win them all.

[3] Miniexercise: explain why this is not the same as saying that $G(\neg\phi, \mathfrak{M})$ is *misère* $G(\phi, \mathfrak{M})$. Neil writes "You should also look at my treatment of the intuitionistic case in my paper 'Language Games and Intuitionism'. Two other papers are my reply to Hintikka in the Nordic JPL and my piece for the volume on Williamson on knowledge. I'll send you the PDFs of these too."

True and **False** play $G((\phi \to \psi), W)$ by playing simultaneously, for all worlds W' accessible from W, all the games $G(\phi, W')$ and $G(\psi, W')$ where **True** has to win either $G(\psi, W')$ or the game $G(\phi, W')$ modified by the players swapping rôles.

9.2.4 Feedback formulæ and Fixed-point Logic

Many years ago (in the 1970s) Aczel considered the possibility of applying this semantics to what he called *feedback formulæ*, formulæ which had themselves as proper subformulæ. Nothing in the definition of this game relies for its legitimacy on the subformula relation on the language being wellfounded. If it is illfounded the game can have plays of length ω. But that is allowed! As far as I know Aczel never published his work on this, but I would imagine that someone must have noticed that this kind of game semantics can be given for fixpoint logics. There is presumably a literature on this, but I don't know it.

Chase this up

(We saw feedback formulæ in section **??**: the definition of HHT is a feedback formula.)

9.3 Branching Quantifiers

Branching quantifier formulæ (Henkin [4]) look like[4]

$$\left(\begin{array}{c} \forall x \exists y \\ \forall z \exists w \end{array}\right)(A(x, y, z, w)) \qquad (\phi)$$

These branching quantifier formulæ are problematic, and we see this even with the simplest possible cases. Suppose we have a matrix game in mind; let $A(x, y)$ be the payoff of row x played against column y. Consider the formula

$$\left(\begin{array}{c} \forall y \\ \exists x \end{array}\right)(A(x, y) \geq k) \qquad (9.1)$$

and the Hintikka game played over it. It is a game of imperfect information beco's it has simultaneous moves. The result is that neither player has a winning strategy. If we want to say that the formula is true iff **True** has a winning strategy in the Hintikka game and that the formula is false iff **False** has a winning strategy in the Hintikka game then we have to concede that there is a truth-value gap. Formula 9.1 looks like both

$(\forall y)(\exists x)(A(x, y) \geq k)$ and $(\exists x)(\forall y)(A(x, y) \geq k)$,

but is not equivalent to either.

[4] A message from (the late) Graham Solomon: Lloyd Humberstone sketched a compositional semantics for branching quantifiers, in a critical notice of Hintikka's The Game of Language, in [7]. Tom Patton [8] rejected Humberstone's proposal. Have there been any other, more recent, published proposals for compositional semantics for branching quantifiers?

Henkin [4] saw from day one that formulæ with branching quantifiers had great expressive power. For example this formula says there are as many things that are F as there are things that are G:

$$\begin{pmatrix} \forall a \exists b' \\ \forall b \exists a' \end{pmatrix} ((a = a' \longleftrightarrow b = b') \wedge (F(a) \to G(b')) \wedge (G(b) \to F(a'))) \tag{9.2}$$

A slight modification of this gives a way of saying that there are infinitely many things that are F. You invent a constant and say that it has F, and then say there is a bijection between the extension of F and the extension of $F \setminus \{a\}$. That is to say, just add a new constant 'c' and—in formula 9.2—replace '$G(x)$' with '$F(x) \wedge x \neq c$'.

Each branch in the quantifier prefix is $\forall^* \exists^*$, and even with such comparatively modest complexity we get strength beyond first-order. It turns out that, for our purposes, $\forall^* \exists^*$ is all we need. The analysis we develop below is applicable to formulæ with $\forall^* \exists^* \forall^* \exists *$ prefixes (and beyond) among their branches, but such formulæ are not required.

The syntax of ϕ looks dead cute, and it's pretty clear that the formula ought to mean something like: "for all x and for all z there are y (depending only on x) and w (depending only on z) such that ...". The obvious way to capture this meaning is by means of a sort of pseudo-skolemisation obtaining

$$(\forall x)(\forall z)A(x, f(y), z, g(w)) \tag{$psk(\phi)$}$$

and it should be clear how to do this in the general case. Let us make this a definition.

Definition 5.

1. *The* **pseudo-skolemisation** *of a branching-quantifier formula is the $\forall^*$ sentence obtained by first replacing every existentially quantified variable v by a term of the form: function letter of arity n applied to the n universally quantified variables in whose scope v appears, and then deleting the existential quantifiers.*
2. *A branching-quantifier formula ψ is* `true` *in a structure $\mathfrak{M}$ iff $\mathfrak{M}$ can be expanded*[5] *by Skolem functions to a structure in which the pseudo-skolemisation of ψ is* `true`.[6]

This gives us an immediate proof of the old result ([**?**], [**?**]) that every branching quantifier formula is equivalent to a Σ^2_1 formula. There is a converse too, and we will get round to it later (theorem **??**.)

[5] beware: this is model-theoretic jargon. It means adding new gadgets to the structure not new elements to it.

[6] It has to be admitted that this definition has the meaning we want it to have only if the Axiom of Choice is globally true, but this is probably less of a worry for my audience than it is for me. I am much struck by the fact that the axiom of choice is not needed for the proof that skolemisation preserves satisfiability.

9.3.1 Teams

The Skolemisation suggests a Hintikka game treatment of this that makes **True** and **False** into *teams* of two players. Each team has a player for each row of the quantifier prefix; the prefix has two rows, so we have two players. For a discussion of the possible genesis of this idea see section **??**. We think of **True** as being a team composed of two players, and **False** similarly. Each player is allowed to respond only to the moves made by the player on the other team that they are *marking*—as it were. That is to say, team **False** has a player $\mathbf{False}_x$ whose job is to instantiate the x variables: he is marked by player $\mathbf{True}_y$ whose job it is to find witnesses for the y variables. And of course team **False** also has a player $\mathbf{False}_z$ whose job is to instantiate the z variables: he is marked by player $\mathbf{True}_w$ whose job it is to find witnesses for the w variables. [7]

Consider the predicament of player $\mathbf{True}_y$ in team **True** who has been given the job of picking witnesses for the variable 'y'. Her job is to mark the player $\mathbf{False}_x$ in team **False** whose job it is to instantiate 'x'. She is not allowed to take any account of the moves made by the other member $\mathbf{False}_z$ of team **False**. Of course her comrade-in-arms $\mathbf{True}_w$ is told of all moves made by $\mathbf{False}_z$, so one way to think of this is as $\mathbf{True}_y$ and $\mathbf{True}_w$ being required to operate a system of chinese walls. They can agree on a strategy in advance, and they share a common purpose, but they must not share information.

How are we, the referees, to detect any breaches of these rules? One obvious thing we can do is to get the two teams to play the game repeatedly, and check that $\mathbf{True}_y$'s move depends solely on $\mathbf{False}_x$'s moves. That is to say, we blow a whistle if we find that $\mathbf{False}_x$ has repeated a move made in an earlier play of the game but $\mathbf{True}_y$'s the second time is not the same as her reply the first time.

We should say something about how this means that team **True** have to be playing deterministiaclly, o/w they might be accused of cheating

[7] Of course the general branching-quantifier prefix is not a set of rows, but an arbitrary partial order, so that the members of the two teams are not indexed by *rows* as if they were rugby forwards but rather by maximal chains through the poset.

Chapter 10

Curry-Howard

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 '→', '∧', '∨' etc. as symbols for operations on sets as well as on formulæ. The ambiguity we will see in the use of '$A \to B$' 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 '$A \to B$' 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 '$A \to B$' 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.

10.1 Decorating Formulæ

10.1.1 The rule of →-elimination

Consider the rule of →-elimination

$$\frac{A \qquad A \to B}{B} \to\text{-elim} \tag{10.1}$$

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 $A \to B$ then I have a B". So what might an $A \to B$ (a member of $A \to B$) be? Clearly $A \to B$ must be the set of functions that give you a member of B when fed a member of A. Thus we can decorate 10.1 to obtain

$$\frac{a : A \qquad f : A \to B}{f(a) : B} \to\text{-elim} \tag{10.2}$$

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 →-int later. For the moment we will look at the rules for ∧.

10.2 Rules for ∧

10.2.1 The rule of ∧-introduction

Consider the rule of ∧-introduction:

$$\frac{A \qquad B}{A \wedge B} \wedge\text{-int} \tag{10.3}$$

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. 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 10.3 is

$$\frac{a : A \qquad b : B}{\langle a, b\rangle : A \times B} \wedge\text{-int} \tag{10.4}$$

Say something about how we use × 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.

10.2.2 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.

[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.

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

$A\times B$ 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.)

10.2.1 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

Resuscitate this?

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

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 $\{\texttt{pink}\}$ 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

[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.

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

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 $\to$-introduction) and to treat it properly we need to think about:

10.3 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.

10.4 Rules for $\wedge$

10.4.1 The rule of $\wedge$-elimination

$$\frac{A \wedge B}{B}\ \wedge\text{-elim} \tag{10.5}$$

We decorate the premiss with a variable:

$$\frac{x : A \wedge B}{B}\ \wedge\text{-elim} \tag{10.6}$$

...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`'.

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

[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.

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

$$\frac{A \wedge B}{A}\ \wedge\text{-elim} \tag{10.7}$$

to obtain

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

10.4.2 The rule of ∧-introduction

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

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

10.5 Rules for →

7.2.2.1 The rule of →-introduction

Here is a simple proof using →-introduction.

$$\cfrac{\cfrac{[A \to B]^1 \qquad A}{B}\ \to\text{-elim}}{(A \to B) \to B}\ \to\text{-int (1)} \tag{10.8}$$

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 '$f(x)$':

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

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 $f(x)$.

[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 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

$$\dfrac{\dfrac{f : [A \to B]^1 \qquad x : A}{f(x) : B}\ {\to}\text{-elim}}{\lambda f.f(x) : (A \to B) \to B}\ {\to}\text{-int (1)} \tag{10.9}$$

Thus, in general, an application of →-introduction will gobble up the proof

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

and emit the proof

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

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

10.6 Rules for ∨

We've discussed ∨-introduction but not ∨-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 15. *Go back and look at the proofs that you wrote up in answer to exercise 5, and decorate those that do not use '∨'.*

10.7 Remaining Rules

10.7.1 Identity Rule

Here is a very simple application of the identity rule.

See [27]: 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 $\underline{\text{K}}$onstant. (See section **??**.)

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

10.7.2 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*.

10.7.3 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 $A \to \perp$ is empty. So $(A \to \perp) \to \perp$ is the set of functions from the empty set to the empty set and is nonempty—being the singleton of the empty function—so $((A \to \perp) \to \perp) \to A$ is the set of functions from the singleton of the empty function to a nonempty set and is sort-of isomorphic to A. empty.

So $((A \to \perp) \to \perp) \to A$ is not reliably inhabited, in the sense that it's inhabited but not uniformly. 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 should definitely pursue.

Chapter 11

Appendices

11.1 Notes to Chapter one

11.1.1 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.

11.2 Notes to Chapter 5

11.2.1 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 **??**.)

$$(\exists x_1 \ldots x_n)(\forall y)(\bigvee_{i \leq n} y = x_i) \tag{11.1}$$

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

$$(\exists x_1 \ldots x_3)(\forall y)(\bigvee_{i \leq 3} y = x_i) \tag{11.2}$$

is an abbreviation of

$$(\exists x_1 x_2 x_3)(\forall y)(y = x_1 \vee y = x_2 \vee y = x_3) \tag{11.3}$$

(Notice that formula 11.2.1 isn't actually *second*-order either, because the dodgy variables are not ranging over subsets of the domain.)

11.3 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 [13]. 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 satisfactory concept of algorithm see [10]. 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 [21]).[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] Aczel, P. Quantifiers, Games and Inductive definitions. Proceedings of the third Scandinavian Logic Symposium Studies in etc vol 82 N-H pp 1-14.

[2] Jon Barwise “Branching quantifiers in English”, Journal of Philosophical Logic **8** (1979) p 47–80.

[3] Barwise, Jon. “Some applications of Henkin quantifiers”. Israel J of Maths **25** 1976 pp 47–63.

[4] Leon Henkin “Some Remarks on Infinitely Long Formulas”, Infinitistic Methods, Proceedings of the Symposium on Foundations of Mathematics, Warsaw, 1959.

[5] Hintikka K.J.J. “Logic, Language Games and Information’ O.U.P. Clarendon Press 1973

[6] Hintikka K.J.J. “Is”, semantical games and semantical relativity”. JPL **8** (1979) pp 433–468.

[7] Lloyd Humberstone, Critical notice of Hintikka’s The Game of Language, MIND **96** (1987): 99–107.

[8] Tom Patton “On Humberstone’s semantics for branching quantifiers” MIND **98** (1989): 429–433.

[9] Allwood, Andersson and Dahl, Logic in Linguistics CUP 1977

[10] Andreas Blass, Nachum Dershowitz and Yuri Gurevich. When are two algorithms the same? Bulletin of Symbolic Logic **15** (june 2009) pp 145–168.

[11] J.L. Borges Labyrinths [publ’n data??]

[12] J.L. Borges *Siete Noches*

[13] Church, A. The calculi of λ-conversion. Princeton 1941

[14] Harold Davenport The Higher Arithmetic, 7th edn, Cambridge University Press, 1999.

[15] Intro to natural language semantics Henriëtte de Swart CSLI

[16] Wilfrid Hodges Logic Penguin

[17] John E Hopcroft and Jeffrey D Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley

[18] Donald Kalish and Richard Montague: Logic: techniques of formal reasoning. Harcourt Brace 1964

[19] 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`

[20] Anne McLaren: Where to draw the line? Proceeding of the Royal Institution **56** (1984) pp 101–121.

[21] Roger Penrose: The Emperor's New Mind

[22] "Plural Logic" Alex Oliver and Timothy Smiley. 352 pp, 978-0-19-957042-3 Oxford University Press 2013

[23] `http://www.cl.cam.ac.uk/Teaching/2002/RLFA/reglfa.ps.gz`

[24] Quine, Predicate functor Logic in Selected Logic papers.

[25] Quine, Ontological remarks on the propositional calculus in Selected Logic Papers 265–272.

[26] Quine, W.V. (1962) Mathematical Logic (revised edition) Harper torchbooks 1962

[27] Scott D.S. Semantical Archæology, a parable. In: Harman and Davidson eds, Semantics of Natural Languages. Reidel 1972 pp 666–674.

[28] Arto Salomaa: Computation and Automata. Encyclopædia of mathematics and its applications. CUP 1985

[29] Peter Smith: An introduction to Formal Logic CUP.

[30] Alfred Tarski and Steven Givant. "A formulation of Set Theory without variables". AMS colloquium publication **41** 1987.

[31] Tversky and Kahnemann "Extensional versus Intuitive reasoning" Psychological Review **90** 293–315.

[32] Tymoczko and Henle. Sweet Reason: A Field Guide to Modern Logic. Birkhäuser 2000 ISBN