Natural Language Processing: part 1 of lecture notes

2003, 8 Lectures
Ann Copestake (aac@cl.cam.ac.uk)
http://www.cl.cam.ac.uk/users/aac/

Lecture Synopsis

Aims

This course aims to introduce the fundamental techniques of natural language processing, to develop an understanding of the limits of those techniques and of current research issues, and evaluate some current and potential applications.

- **Introduction.** Brief history of NLP research, current applications, generic NLP system architecture, knowledge-based *versus* probabilistic approaches.
- **Finite state techniques.** Inflectional and derivational morphology, finite-state automata in NLP, finite-state transducers.
- **Prediction and part-of-speech tagging.** Corpora, simple N-grams, word prediction, stochastic tagging, evaluating system performance.
- **Parsing and generation I.** Generative grammar, context-free grammars, parsing and generation with context-free grammars, weights and probabilities.
- **Parsing and generation II.** Constraint-based grammar, unification, simple compositional semantics.
- **Lexical semantics.** Semantic relations, WordNet, word senses, word sense disambiguation.
- **Discourse.** Anaphora resolution, discourse relations.
- **Applications.** Machine translation, email response, spoken dialogue systems.

Objectives

At the end of the course students should:

- be able to describe the architecture of and basic design for a generic NLP system 'shell'
- be able to discuss the current and likely future performance of several NLP applications, such as machine translation and email response
- be able to briefly describe a fundamental technique for processing language for several subtasks, such as morphological analysis, syntactic parsing, word sense disambiguation etc (as indicated in the lecture synopses).
- understand how these techniques draw on and relate to other areas of (theoretical) computer science, such as formal language theory, formal semantics of programming languages, or theorem proving.

NLP is a large and multidisciplinary field, so this course can only provide a very general introduction. The first lecture is designed to give an overview of the main subareas and a very brief idea of the main applications and the methodologies which have been employed. The history of NLP is briefly discussed as a way of putting this into perspective. The next six lectures describe some of the main subareas in more detail. The organisation is roughly based on increased 'depth' of processing, starting with relatively surface-oriented techniques and progressing to considering meaning of sentences and meaning of utterances in context. Each lecture will consider the subarea as a whole and then go on to describe one or more sample algorithms which tackle particular problems. The algorithms have been chosen because they are relatively straightforward to describe and because they illustrate a specific technique which has been shown to be useful, but the idea is to exemplify an approach, not to give a detailed survey (which would be impossible in the time available). However, other approaches will sometimes be discussed briefly. The final lecture brings the preceding material together in order to describe the state of the art in three sample applications.

There are various themes running throughout the lectures. One theme is the connection to linguistics and the tension that sometimes exists between the predominant view in theoretical linguistics and the approaches adopted within NLP. A somewhat related theme is the distinction between knowledge-based and probabilistic approaches. Evaluation will be discussed in the context of the different algorithms.

Because NLP is such a large area, there are many topics that aren't touched on at all in these lectures. Speech recognition and speech synthesis is almost totally ignored. Information retrieval and information extraction are the topic of a separate course given by Simone Teufel, for which this course is a prerequisite.

Since this is the first time I've given this course, the handout has been largely written from scratch and no doubt contains many typos, bugs, infelicities and incomprehensible bits. Feedback would be greatly appreciated.

Recommended Reading

Background:

Pinker, S., *The Language Instinct*, Penguin, 1994.

This is a thought-provoking and sometimes controversial 'popular' introduction to linguistics. Although the NLP lectures don't assume any exposure to linguistics, the course will be easier to follow if students have some idea of the linguistic notion of a grammar, for instance.

Recommended Book:

Jurafsky, Daniel and James Martin, *Speech and Language Processing*, Prentice-Hall, 2000 (referenced as J&M throughout this handout).

Study and Supervision Guide

The handouts and lectures should contain enough information to enable students to adequately answer the exam questions, but the handout is not intended to substitute for a textbook. In most cases, J&M go into a considable amount of further detail: rather than put lots of suggestions for further reading in the handout, in general I have assumed that students will look at J&M, and then follow up the references in there if they are interested. The notes at the end of each lecture give details of the sections of J&M that are relevant and details of any discrepancies with these notes. Manning and Schütze, 'Foundations of Statistical Natural Language Processing', MIT Press, 1999, is also recommended for further reading for the statistical aspects, especially word sense disambiguation.

Supervisors ought to familiarize themselves with the relevant parts of Jurafsky and Martin (see notes at the end of each lecture). However, good students should find it quite easy to come up with questions that the supervisors (and the lecturer) can't answer! Language is like that ...

Generally I'm taking a rather informal/example-based approach to concepts such as finite-state automata, context-free grammars etc. Part II students should have already got the formal background that enables them to understand the application to NLP. Diploma and Part II (General) students may not have covered all these concepts before, but the expectation is that the examples are straightforward enough so that this won't matter too much.

This course inevitably assumes some very basic linguistic knowledge, such as the distinction between the major parts of speech. It introduces some linguistic concepts that won't be familiar to all students: since I'll have to go through these quickly, reading the first few chapters of an introductory linguistics textbook may help students understand the material. The idea is to introduce just enough linguistics to motivate the approaches used within NLP rather than to teach the linguistics for its own sake. Exam questions won't rely on students remembering the details of any specific linguistic phenomenon.

Although these notes are almost completely new, prior year exam questions are still generally appropriate.

Of course, I'll be happy to try and answer questions about the course or more general NLP questions, preferably by email.

1 Lecture 1: Introduction to NLP

The aim of this lecture is to give students some idea of the objectives of NLP. The main subareas of NLP will be introduced, especially those which will be discussed in more detail in the rest of the course. There will be a preliminary discussion of the main problems involved in language processing by means of examples taken from NLP applications. This lecture also introduces some methodological distinctions and puts the applications and methodology into some historical context.

1.1 What is NLP?

Natural language processing (NLP) can be defined as the automatic (or semi-automatic) processing of human language. The term 'NLP' is sometimes used rather more narrowly than that, often excluding information retrieval and sometimes even excluding machine translation. NLP is sometimes contrasted with 'computational linguistics', with NLP being thought of as more applied. Nowadays, alternative terms are often preferred, like 'Language Technology' or 'Language Engineering'. Language is often used in contrast with speech (e.g., Speech and Language Technology). But I'm going to simply refer to NLP and use the term broadly.

NLP is essentially multidisciplinary: it is closely related to linguistics (although the extent to which NLP overtly draws on linguistic theory varies considerably). It also has links to research in cognitive science, psychology, philosophy and maths (especially logic). Within CS, it relates to formal language theory, compiler techniques, theorem proving, machine learning and human-computer interaction. Of course it is also related to AI, though nowadays it's not generally thought of as part of AI.

1.2 Some linguistic terminology

The course is organised so that there are six lectures corresponding to different NLP subareas, moving from relatively 'shallow' processing to areas which involve meaning and connections with the real world. These subareas loosely correspond to some of the standard subdivisions of linguistics:

1. Morphology: the structure of words. For instance, *unusually* can be thought of as composed of a prefix *un-*, a stem *usual*, and an affix *-ly*. *composed* is *compose* plus the inflectional affix *-ed*: a spelling rule means we end up with *composed* rather than *composeed*. Morphology will be discussed in lecture 2.

2. Syntax: the way words are used to form phrases. e.g., it is part of English syntax that a determiner such as *the* will come before a noun, and also that determiners are obligatory with certain singular nouns. Formal and computational aspects of syntax will be discussed in lectures 3, 4 and 5.

3. Semantics. Compositional semantics is the construction of meaning (generally expressed as logic) based on syntax. Compositional semantics is discussed in lecture 5. This is contrasted to lexical semantics, i.e., the meaning of individual words, which is discussed in lecture 6.

4. Pragmatics: meaning in context. This will come into lecture 7, although linguistics and NLP generally have very different perspectives here.

1.3 Why is language processing difficult?

Consider trying to build a system that would answer email sent by customers to a retailer selling laptops and accessories via the Internet. This might be expected to handle queries such as the following:

- Has my order number 4291 been shipped yet?
- Is FD5 compatible with a 505G?
- What is the speed of the 505G?

Assume the query is to be evaluated against a database containing product and order information, with relations such as the following:

ORDER

Order number	Date ordered	Date shipped
4290	2/2/02	2/2/02
4291	2/2/02	2/2/02
4292	2/2/02	

USER: Has my order number 4291 been shipped yet?
DB QUERY: order(number=4291,date_shipped=?)
RESPONSE TO USER: Order number 4291 was shipped on 2/2/02

It might look quite easy to write patterns for these queries, but very similar strings can mean very different things, while very different strings can mean much the same thing. 1 and 2 below look very similar but mean something completely different, while 2 and 3 look very different but mean much the same thing.

1. How fast is the 505G?
2. How fast will my 505G arrive?
3. Please tell me when I can expect the 505G I ordered.

While some tasks in NLP can be done adequately without having any sort of account of meaning, others require that we can construct detailed representations which will reflect the underlying meaning rather than the superficial string.

In fact, in natural languages (as opposed to programming languages), ambiguity is ubiquitous, so exactly the same string might mean different things. For instance in the query:

> Do you sell Sony laptops and disk drives?

the user may or may not be asking about Sony disk drives. We'll see lots of examples of different types of ambiguity in these lectures.

Often humans have knowledge of the world which resolves a possible ambiguity, probably without the speaker or hearer even being aware that there is a potential ambiguity.[1] But hand-coding such knowledge in NLP applications has turned out to be impossibly hard to do for more than very limited domains: the term *AI-complete* is sometimes used (by analogy to NP-complete), meaning that we'd have to solve the entire problem of representing the world and acquiring world knowledge. The term AI-complete is intended somewhat jokingly, but conveys what's probably the most important guiding principle in current NLP: we're looking for applications which don't require AI-complete solutions: i.e., ones where we can work with very limited domains or approximate full world knowledge knowledge by relatively simple techniques.

1.4 Some NLP applications

The following list is not complete, but useful systems have been built for:

- spelling and grammar checking
- optical character recognition (OCR)
- screen readers for blind and partially sighted users

[1] I'll use 'hearer' generally to mean the person who is on the receiving end, regardless of the modality of the language transmission: i.e., regardless of whether it's spoken, signed or written. Similarly, I'll use *speaker* for the person generating the speech, text etc and *utterance* to mean the speech or text itself. This is the standard linguistic terminology, which recognises that spoken language is primary and text is a later development.

- augmentative and alternative communication (i.e., systems to aid people who have difficulty communicating because of disability)
- machine aided translation (i.e., systems which help a human translator, e.g., by storing translations of phrases and providing online dictionaries integrated with word processors, etc)
- lexicographers' tools
- information retrieval
- document classification (filtering, routing)
- document clustering
- information extraction
- question answering
- summarization
- text segmentation
- exam marking
- report generation (possibly multilingual)
- machine translation
- natural language interfaces to databases
- email understanding
- dialogue systems

Several of these applications are discussed briefly below. Roughly speaking, they are ordered according to the complexity of the language technology required. The applications towards the top of the list can be seen simply as aids to human users, while those at the bottom are perceived as agents in their own right. Perfect performance on any of these applications would be AI-complete, but perfection isn't necessary for utility: in many cases, useful versions of these applications had been built by the late 70s. Commercial success has often been harder to achieve, however.

1.5 Spelling and grammar checking

All spelling checkers can flag words which aren't in a dictionary.

(1) * The neccesary steps are obvious.

(2) The necessary steps are obvious.

If the user can expand the dictionary, or if the language has complex productive morphology (see §2.1), then a simple list of words isn't enough to do this and some morphological processing is needed.[2]

More subtle cases involve words which are correct in isolation, but not in context. Syntax could sort some of these cases out. For instance, possessive *its* generally has to be followed by a noun or an adjective noun combination etc[3]

(3) * Its a fair exchange.

[2]Note the use of * ('star') above: this notation is used in linguistics to indicate a sentence which is judged (by the author, at least) to be incorrect. ? is generally used for a sentence which is questionable, or at least doesn't have the intended interpretation. # is used for a pragmatically anomalous sentence.

[3]The linguistic term I'll use for this is Nbar, properly written $\overline{\text{N}}$. Roughly speaking, an NBar is a noun with all its modifers (adjectives etc) but without a determiner.)

(4) It's a fair exchange.

(5) * The dog came into the room, it's tail wagging.

(6) The dog came into the room, its tail wagging.

But, it sometimes isn't locally clear whether something is an Nbar or not: e.g. *fair* is ambiguous between a noun and an adjective.

(7) * 'Its fair', was all Kim said.

(8) 'It's fair', was all Kim said.

(9) * Every village has an annual fair, except Kimbolton: it's fair is held twice a year.

(10) Every village has an annual fair, except Kimbolton: its fair is held twice a year.

The most elaborate spelling/grammar checkers can get some of these cases right, but none are anywhere near perfect.

Spelling correction can require a form of *word sense disambiguation*:

(11) # The tree's bows were heavy with snow.

(12) The tree's boughs were heavy with snow.

Getting this right requires an association between *tree* and *bough*. In the past, attempts might have been made to hand-code this in terms of general knowledge of trees and their parts. But these days machine learning techniques are generally used to derive word associations from corpora:[4] this can be seen as a substitute for the fully detailed world knowledge, but may actually be a more realistic model of how humans do word sense disambiguation. However, commercial systems don't (yet) do this systematically.

Simple subject verb agreement can be checked automatically:

(13) * My friend were unhappy.

But this isn't as straightforward as it may seem:

(14) A number of my friends were unhappy.

(15) The number of my friends who were unhappy was amazing.

(16) My family were unhappy.

Whether the last example is grammatical or not depends on your dialect of English: it is grammatical for most British English speakers, but not for many Americans.

Checking punctuation can be hard (even AI-complete):
BBC News Online, 3 October, 2001

> Students at Cambridge University, who come from less affluent backgrounds, are being offered up to 1,000 a year under a bursary scheme.

Thissentence contains a *non-restrictive relative clause*: a form of parenthetical comment. The sentence implies that most/all students at Cambridge come from less affluent backgrounds. What the reporter probably meant was a restrictive relative:

> Students at Cambridge University who come from less affluent backgrounds are being offered up to 1,000 a year under a bursary scheme.

If you use a word processor with a spelling and grammar checker, try looking at its treatment of agreement and some of these other phenomena. If possible, try switching settings between British English and American English.

[4] A *corpus* is a body of text that has been collected for some purpose, see §3.1.

1.6 Information retrieval, information extraction and question answering

Information retrieval involves returning a set of documents in response to a user query: Internet search engines are a form of IR. However, one change from classical IR is that Internet search now uses techniques that rank documents according to how many links there are to them (e.g., Google's PageRank) as well as the presence of search terms.

Information extraction involves trying to discover specific information from a set of documents. The information required can be described as a template. For instance, for company joint ventures, the template might have slots for the companies, the dates, the products, the amount of money involved. The slot fillers are generally strings.

Question answering attempts to find a specific answer to a specific question from a set of documents, or at least a short piece of text that contains the answer.

(17) What is the capital of France?
Paris has been the French capital for many centuries.

There are some question-answering systems on the Web, but most use very basic techniques. For instance, Ask Jeeves relies on a fairly large staff of people who search the web to find pages which are answers to potential questions. The system performs very limited manipulation on the input to map to a known question. The same basic technique is used in many online help systems.

1.7 Machine translation

MT work started in the US in the early fifties, concentrating on Russian to English. A prototype system was publicly demonstrated in 1954 (remember that the first electronic computer had only been built a few years before that). MT funded got drastically cut in the US in the mid-60s and ceased to be academically respectable in some places, but Systran was providing useful translations by the late 60s. Systran is still going (updating it over the years is an amazing feat of software engineering): Systran now powers AltaVista's BabelFish

`http://world.altavista.com/`

and many other translation services on the web.

Until the 80s, the utility of general purpose MT systems was severely limited by the fact that text was not available in electronic form: Systran used teams of skilled typists to input Russian documents.

Systran and similar systems are not a substitute for human translation: they are useful because they allow people to get an idea of what a document is about, and maybe decide whether it is interesting enough to get translated properly. This is much more relevant now that documents etc are available on the Web. Bad translation is also, apparently, good for chatrooms.

Spoken language translation is viable for limited domains: research systems include Verbmobil, SLT and CSTAR.

1.8 Natural language interfaces and dialogue systems

Natural language interfaces were the 'classic' NLP problem in the 70s and 80s. LUNAR is the classic example of a natural language interface to a database (NLID): its database concerned lunar rock samples brought back from the Apollo missions. LUNAR is described by Woods (1978) (but note most of the work was done several years earlier): it was capable of translating elaborate natural language expressions into database queries.

SHRDLU (Winograd, 1973) was a system capable of participating in a dialogue about a microworld (the blocks world) and manipulating this world according to commands issued in English by the user. SHRDLU had a big impact on the perception of NLP at the time since it seemed to show that computers could actually 'understand' language: the impossibility of scaling up from the microworld was not realized.

LUNAR and SHRDLU both exploited the limitations of one particular domain to make the natural language understanding problem tractable, particularly with respect to ambiguity. To take a trivial example, if you know your database is about lunar rock, you don't need to consider the music or movement senses of *rock* when you're analysing a query.

There have been many advances in NLP since these systems were built: systems have become much easier to build, and somewhat easier to use, but they still haven't become ubiquitous. Natural Language interfaces to databases were

commercially available in the late 1970s, but largely died out by the 1990s: porting to new databases and especially to new domains requires very specialist skills and is essentially too expensive (automatic porting was attempted but never successfully developed). Users generaly prefered graphical interfaces when these became available. Speech input would make natural language interfaces much more useful: unfortunately, speaker-independent speech recognition still isn't good enough for even 1970s scale NLP to work well. Techniques for dealing with misrecognised data have proved hard to develop. In many ways, current commercially-deployed spoken dialogue systems are using pre-SHRDLU technology.

1.9 Some more history

Before the 1970s, most NLP researchers were concentrating on MT as an application (see above). NLP was a very early application of CS and started about the same time as Chomsky was publishing his first major works in formal linguistics (Chomskyan linguistics quickly became dominant, especially in the US). In the 1950s and early 1960s, ideas about formal grammar were being worked out in linguistics and algorithms for parsing natural language were being developed at the same time as algorithms for parsing programming languages. However, most linguists were uninterested in NLP and the approach that Chomsky developed turned out to be only somewhat indirectly useful for NLP.

NLP in the 1970s and first half of the 1980s was predominantly based on a paradigm where extensive linguistic and real-world knowledge was hand-coded. There was controversy about how much linguistic knowledge was necessary for processing, with some researchers downplaying syntax, in particular, in favour of world knowledge. NLP researchers were very much part of the AI community (especially in the US and the UK), and the debate that went on in AI about the use of logic vs other meaning representations ('neat' vs 'scruffy') also affected NLP. By the 1980s, several linguistic formalisms had appeared which were fully formally grounded and reasonably computationally tractable, and the linguistic/logical paradigm in NLP was firmly established. Unfortunately, this didn't lead to many useful systems, partly because many of the difficult problems (disambiguation etc) were seen as somebody else's job (and mainstream AI was not developing adequate knowledge representation techniques) and partly because most researchers were concentrating on the 'agent-like' applications and neglecting the user aids. Although the symbolic, linguistically-based systems sometimes worked quite well as NLIDs, they proved to be of little use when it came to processing less restricted text, for applications such as IE. It also became apparent that lexical acquisition was a serious bottleneck for serious development of such systems.

Statistical NLP became the most common paradigm in the 1990s, at least in the research community. Speech recognition had demonstrated that simple statistical techniques worked, given enough training data. NLP systems were built which required very limited hand-coded knowledge, apart from initial training material. Most applications were much shallower than the earlier NLIDs, but the switch to statistical NLP coincided with a change in US funding, which started to emphasize speech-based interfaces and IE. There was also a general realization of the importance of serious evaluation and of reporting results in a way that could be reproduced by other researchers. US funding emphasized competitions with specific tasks and supplied test material, which encouraged this, although there was a downside in that some of the techniques developed were very task-specific. It should be emphasised that there had been computational work on corpora for many years (much of it by linguists): it became much easier to do corpus work by the late 1980s as disk space became cheap and machine-readable text became ubiquitous. Despite the shift in research emphaisis to statistical approaches, most commercial systems remained primarily based on hand-coded linguistic information.

More recently the symbolic/statistical split has become less pronounced, since most researchers are interested in both.[5] There is considerable emphasis on machine learning in general, including machine learning for symbolic processing. Linguistically-based NLP has made something of a comeback, with increasing availability of open source resources, and the realisation that at least some of the classic statistical techniques seem to be reaching limits on performance, especially because of difficulties in adapting to new types of text. However, the modern linguistically-based approaches are making use of machine learning and statistical processing. The dotcom boom and bust has considerably affected NLP, but it's too early to say what the long-term implications are. The ubiquity of the Internet has certainly changed the space of interesting NLP applications, and the vast amount of text available can potentially be exploited, especially for statistical techniques.

[5] At least, there are only a few researchers who avoid statistical techniques as a matter of principle and all statistical systems have a symbolic component!

1.10 Generic 'deep' NLP application architecture

Many NLP applications can be adequately implemented with relatively shallow processing. For instance, spelling checking only requires a word list and simple morphology to be useful. I'll use the term 'deep' NLP for systems that build a meaning representation (or an elaborate syntactic representation), which is generally agreed to be required for applications such as NLIDs, email question answering and good MT.

The most important principle in building a successful NLP system is modularity. NLP systems are often big software engineering projects — success requires that systems can be improved incrementally.

The input to an NLP system could be speech or text. It could also be gesture (multimodal input or perhaps a Sign Language). The output might be non-linguistic, but most systems need to give some sort of feedback to the user, even if they are simply performing some action (issuing a ticket, paying a bill, etc). However, often the feedback can be very formulaic.

There's general agreement that the following system components can be described semi-independently, although assumptions about the detailed nature of the interfaces between them differ. Not all systems have all of these components:

- input preprocessing: speech recogniser or text preprocessor (non-trivial in languages like Chinese or for highly structured text for any language) or gesture recogniser. Such system might themselves be very complex, but I won't discuss them in this course — we'll assume that the input to the main NLP component is segmented text.
- morphological analysis: this is relatively well-understood for the most common languages that NLP has considered, but is complicated for many languages (e.g., Turkish, Basque).
- part of speech tagging: not an essential part of most deep processing systems, but sometimes used as a way of cutting down parser search space.
- parsing: this includes syntax and compositional semantics, which are sometimes treated as separate components
- disambiguation: this can be done as part of parsing, or (partially) left to a later phase
- context module: this maintains information about the context, for anaphora resolution, for instance.
- text planning: the part of language generation that's concerned with deciding what meaning to convey (I won't discuss this in this course)
- tactical generation: converts meaning representations to strings. This may use the same grammar and lexicon[6] as the parser.
- morphological generation: as with morphological analysis, this is relatively straightforward for English.
- output processing: text-to-speech, text formatter, etc. As with input processing, this may be complex, but for now we'll assume that we're outputting simple text.

Application specific components, for instance:

1. For NL interfaces, email answering and so on, we need an interface between semantic representation (expressed as some form of logic, for instance) and the underlying knowledge base.
2. For MT based on *transfer*, we need a component that maps between semantic representations.

It is also very important to distinguish between the knowledge sources and the programs that use them. For instance, a morphological analyser has access to a lexicon and a set of morphological rules: the morphological generator might share these knowledge sources. The lexicon for the morphology system may be the same as the lexicon for the parser and generator.

Other things might be required in order to construct the standard components and knowledge sources:

[6]The term *lexicon* is generally used for the part of the NLP system that contains dictionary-like information — i.e. information about individual words.

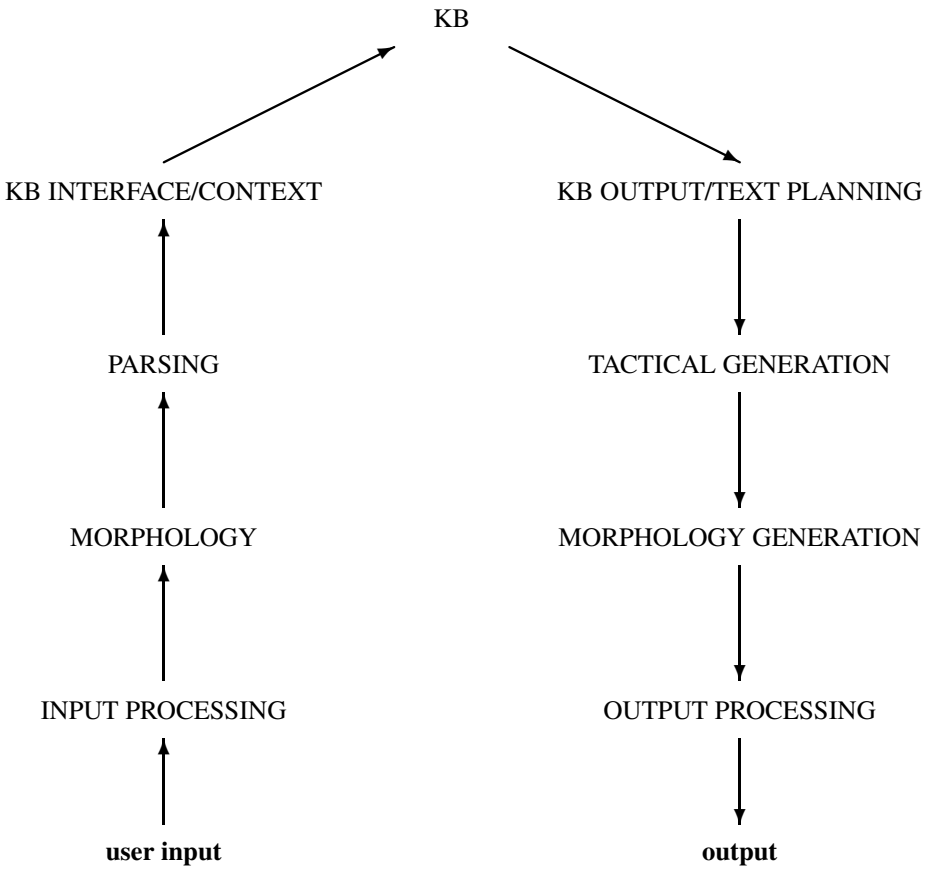
- lexicon acquisition
- grammar acquisition
- acquisition of statistical information

For a component to be a true module, it obviously needs a well-defined set of interfaces. What's less obvious is that it needs its own evaluation strategy and test suites: developers need to be able to work somewhat independently.

In principle, at least, components are *reusable* in various ways: for instance, a parser could be used with multiple grammars, the same grammar can be processed by different parsers and generators, a parser/grammar combination could be used in MT or in a natural language interface. However, for a variety of reasons, it is not easy to reuse components like this, and generally a lot of work is required for each new application, even if it's based on an existing grammar or the grammar is automatically acquired.

We can draw schematic diagrams for applications showing how the modules fit together.

1.11 Natural language interface to a knowledge base



In such systems, the context module generally gets included as part of the KB interface because the discourse state is quite simple, and contextual resolution is domain specific. Similarly, there's often no elaborate text planning requirement, though this depends very much on the KB and type of queries involved.

1.12 General comments

- Even 'simple' NLP applications, such as spelling checkers, need complex knowledge sources for some problems.
- Applications cannot be 100% perfect, because full real world knowledge is not possible.
- Applications that are less than 100% perfect can be useful (humans aren't 100% perfect anyway).
- Applications that aid humans are much easier to construct than applications which replace humans. It is difficult to make the limitations of systems which accept speech or language obvious to naive human users.
- NLP interfaces are nearly always competing with a non-language based approach.
- Currently nearly all applications either do relatively shallow processing on arbitrary input or deep processing on narrow domains. MT can be domain-specific to varying extents: MT on arbitrary text isn't very good, but has some applications.
- Limited domain systems require extensive and expensive expertise to port. Research that relies on extensive hand-coding of knowledge for small domains is now generally regarded as a dead-end, though reusable hand-coding is a different matter.
- The development of NLP has mainly been driven by hardware and software advances, and societal and infrastructure changes, not by great new ideas. Improvements in NLP techniques are generally incremental rather than revolutionary.

2 Lecture 2: Morphology and finite-state techniques

This lecture starts with a brief discussion of morphology, concentrating mainly on English morphology. The concept of a lexicon in an NLP system is discussed with respect to morphological processing. Spelling rules are introduced and the use of finite state transducers to implement spelling rules is explained. The lecture concludes with a brief overview of some other uses of finite state techniques in NLP.

2.1 A very brief and simplified introduction to morphology

Morphology concerns the structure of words. Words are assumed to be made up of *morpheme*s, which are the minimal information carrying unit. Morphemes which can only occur in conjunction with other morphemes are *affix*es: words are made up of a stem (more than one in the case of compounds) and zero or more affixes. For instance, *dog* is a stem which may occur with the plural suffix *+s* i.e., *dogs*. English only has suffixes (affixes which come after a stem) and prefixes (which come before the stem — in English these are limited to derivational morphology), but other languages have *infixes* (affixes which occur inside the stem) and circumfixes (affixes which go around a stem). For instance, Arabic has stems (root forms) such as *k_t_b*, which are combined with infixes to form words (e.g., *kataba*, he wrote; *kotob*, books). Some English irregular verbs show a relic of inflection by infixation (e.g. *sing*, *sang*, *sung*) but this process is no longer *productive* (i.e., it won't apply to any new words, such as *ping*).[7]

2.2 Inflectional vs derivational morphology

Inflectional and derivational morphology can be distinguished, although the dividing line isn't always sharp. The distinction is of some importance in NLP, since it means different representation techniques may be appropriate. Inflectional morphology can be thought of as setting values of slots in some *paradigm*. Inflectional morphology concerns properties such as tense, aspect, number, person, gender, and case, although not all languages code all of these: English, for instance, has very little morphological marking of case and gender. Derivational affixes, such as *un-*, *re-*, *anti-* etc, have a broader range of semantic possibilities and don't fit into neat paradigms. Inflectional affixes may be combined (though not in English). However, there are always obvious limits to this, since once all the possible slot values are 'set', nothing else can happen. In contrast, there are no obvious limitations on the number of derivational affixes (*antidisestablishmentarianism*, *antidisestablishmentarianismization*) and they may even be applied recursively (*antiantimissile*). In some languages, such as Inuit, derivational morphology is often used where English would use adjectival modification or other syntactic means. This leads to very long 'words' occurring naturally and is presumably responsible for the claim that 'Eskimo' has hundreds of words for snow.

Inflectional morphology is generally close to fully productive, in the sense that a word of a particular class will generally show all the possible inflections although the actual affix used may vary. For instance, an English verb will have a present tense form, a 3rd person singular present tense form, a past participle and a passive participle (the latter two being the same for regular verbs). This will also apply to any new words which enter the language: e.g., *text* as a verb — *texts*, *texted*. Derivational morphology is less productive and the classes of words to which an affix applies is less clearcut. For instance, the suffix *-ee* is relatively productive (*textee* sounds plausible, for instance), but doesn't apply to all verbs (?*snoree*, ?*jogee*, ?*dropee*). Derivational affixes may change the part of speech of a word (e.g., *-ise*/*-ize* converts nouns into verbs: *plural*, *pluralise*). However, there are also examples of what is sometimes called *zero derivation*, where a similar effect is observed without an affix: e.g. *tango*, *waltz* etc are words which are basically nouns but can be used as verbs.

Stems and affixes can be individually ambiguous. There is also potential for ambiguity in how a word form is split into morphemes. For instance, *unionised* could be *union -ise -ed* or (in chemistry) *un- ion -ise -ed*. This sort of structural ambiguity isn't nearly as common in English morphology as in syntax, however. Note that *un- ion* is not a possible form (because *un-* can't attach to a noun). Furthermore, although there is a prefix *un-* that can attach to verbs, it nearly always denotes a reversal of a process (e.g., *untie*), whereas the *un-* that attaches to adjectives means 'not', which is the meaning in the case of *un- ion -ise -ed*. Hence the internal structure of *un- ion -ise -ed* has to be *(un- ((ion -ise) -ed))*.

[7] Arguably, though, spoken English has one productive infixation process, exemplified by *absobloodylutely*.

2.3 Spelling rules

English morphology is essentially concatenative: i.e., we can think of words as a sequence of prefixes, stems and suffixes. Some words have irregular morphology and their inflectional forms simply have to be listed. However, in other cases, there are regular phonological or spelling changes associated with affixation. For instance, the suffix *-s* is pronounced differently when it is added to a stem which ends in *s*, *x* or *z* and the spelling reflects this with the addition of an *e* (*boxes* etc). For the purposes of this course, we'll just talk about spelling effects rather than phonological effects: these effects can be captured by *spelling rules* (also known as *orthographic rules*).

English spelling rules can be described independently of the particular stems and affixes involved, simply in terms of the affix boundary. The 'e-insertion' rule can be described as follows:

$$\varepsilon \rightarrow \mathrm{e} / \left\{ \begin{array}{c} \mathrm{s} \\ \mathrm{x} \\ \mathrm{z} \end{array} \right\} \hat{} \_ s$$

In such rules, the mapping is always given from the 'underlying' form to the surface form, the mapping is shown to the left of the slash and the context to the right, with the _ indicating the position in question. ε is used for the empty string and ^ for the affix boundary. This particular rule is read as saying that the empty string maps to 'e' in the context where it is preceded by an s,x, or z and followed by an affix boundary and an s. For instance, this maps *box^s* to *boxes*. This rule might look as though it is written in a context sensitive grammar formalism, but actually we'll see in §2.7 that it corresponds to a finite state transducer. Because the rule is independent of the particular affix, it applies equally to the plural form of nouns and the 3rd person singular present form of verbs. Other spelling rules in English include consonant doubling (e.g., *rat*, *ratted*, though note, not **auditted*) and y/ie conversion (*party*, *parties*).

2.4 Applications of morphological processing

It is possible to use a *full-form lexicon* for English NLP: i.e., to list all the inflected forms and to treat derivational morphology as non-productive. However, when a new word has to be treated (generally because the application is expanded but in principle because a new word has entered the language) it is redundant to have to specify (or learn) the inflected forms as well as the stem, since the vast majority of words in English have regular morphology. So a full-form lexicon is best regarded as a form of compilation. Many other languages have many more inflectional forms, which increases the need to do morphological analysis rather than full-form listing.

IR systems use *stemming* rather than full morphological analysis. For IR, what is required is to relate forms, not to analyse them compositionally, and this can most easily be achieved by reducing all morphologically complex forms to a canonical form. Although this is referred to as stemming, the canonical form may not be the linguistic stem. The most commonly used algorithm is the *Porter stemmer*, which uses a series of simple rules to strip endings (see J&M, section 3.4) without the need for a lexicon. However, stemming does not necessarily help IR. Search engines sometimes do inflectional morphology, but this can be dangerous. For instance, one search engine searches for *corpus* as well as *corpora* when given the latter as input, resulting in a large number of spurious results involving *Corpus Christi* and similar terms.

In most NLP applications, however, morphological analysis is a precursor to some form of parsing. In this case, the requirement is to analyse the form into a stem and affixes so that the necessary syntactic (and possibly semantic) information can be associated with it. Morphological analysis is often called *lemmatization*. For instance, for the part of speech tagging application which we will discuss in the next lecture, *mugged* would be assigned a part of speech tag which indicates it is a verb, though *mug* is ambiguous between verb and noun. For full parsing, as discussed in lectures 4 and 5, we'll need more detailed syntactic and semantic information. Morphological generation takes a stem and some syntactic information and returns the correct form. For some applications, there is a requirement that morphological processing is *bidirectional*: that is, can be used for analysis and generation. The finite state transducers we will look at below have this property.

2.5 Lexical requirements for morphological processing

There are three sorts of lexical information that are needed for full, high precision morphological processing:

- affixes, plus the associated information conveyed by the affix
- irregular forms, with associated information similar to that for affixes
- stems with syntactic categories (plus more detailed information if derivational morphology is to be treated as productive)

On approach to an affix lexicon is for it to consist of a pairing of affix and some encoding of the syntactic/semantic effect of the affix.[8] For instance, consider the following fragment of a suffix lexicon (we can assume there is a separate lexicon for prefixes):

```
ed PAST_VERB
ed PSP_VERB
s PLURAL_NOUN
```

Here `PAST_VERB`, `PSP_VERB` and `PLURAL_NOUN` are abbreviations for some bundle of syntactic/semantic information: we'll discuss this briefly in §2.1.

A lexicon of irregular forms is also needed. One approach is for this to just be a triple consisting of inflected form, 'affix information' and stem, where 'affix information' corresponds to whatever encoding is used for the regular affix. For instance:

```
began PAST_VERB begin
begun PSP_VERB begin
```

Note that this information can be used for generation as well as analysis, as can the affix lexicon.

In most cases, English irregular forms are the same for all senses of a word. For instance, *ran* is the past of *run* whether we are talking about athletes, politicians or noses. This argues for associating irregularity with particular word forms rather than particular senses, especially since compounds also tend to follow the irregular spelling, even non-productively formed ones (e.g., the plural of *dormouse* is *dormice*). However, there are exceptions: e.g., *The washing was hung/*hanged out to dry* vs *the murderer was hanged*.

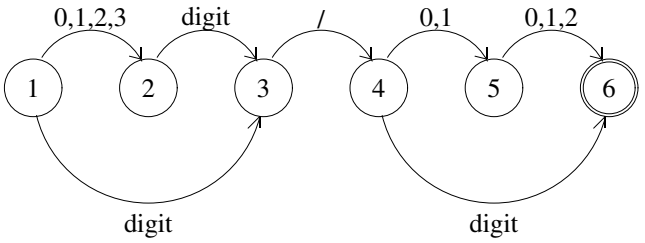
Morphological analysers also generally have access to a lexicon of regular stems. This is needed for high precision: e.g. to avoid analysing *corpus* as *corpu -s* we need to know that there isn't a word *corpu*. There are also cases where historically a word was derived, but where the base form is no longer found in the language: we can avoid analysing *unkempt* as *un- kempt*, for instance, simply by not having *kempt* in the stem lexicon. Ideally this lexicon should have syntactic information: for instance, *feed* could be *fee -ed*, but since *fee* is a noun rather than a verb, this isn't a possible analysis. However, in the approach we'll assume, the morphological analyser is split into two stages. The first of these only concerns morpheme forms and returns both *fee -ed* and *feed* given the input *feed*. A second stage which is closely coupled to the syntactic analysis then rules out *fee -ed* because the affix and stem syntactic information are not compatible (see §2.1 for one approach to this).

If morphology was purely concatenative, it would be very simple to write an algorithm to split off affixes. Spelling rules complicate this somewhat: in fact, it's still possible to do a reasonable job for English with adhoc code, but a cleaner and more general approach is to use finite state techniques.

2.6 Finite state automata for recognition

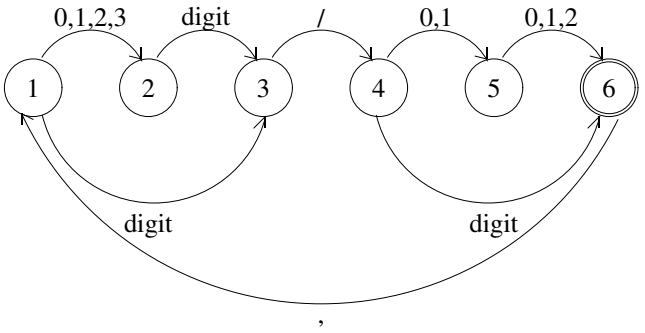
The approach to spelling rules that we'll describe involves the use of finite state trasnducers (FSTs). Rather than jumping straight into this, we'll briefly consider the simpler finite state automata and how they can be used in a simple recogniser. Suppose we want to recognize dates (just day and month pairs) written in the format day/month. The day and the month may be expressed as one or two digits (e.g. 11/2, 1/12 etc). This format corresponds to the following simple FSA, where each character corresponds to one transition:

[8] J&M describe an alternative approach which is to make the syntactic information correspond to a level in a finite state transducer. However, at least for English, this considerably complicates the transducers.



Accept states are shown with a double circle. This is a non-deterministic FSA: for instance, an input starting with the digit 3 will move the FSA to both state 2 and state 3. This corresponds to a *local ambiguity*: i.e., one that will be resolved by subsequent context. By convention, there must be no 'left over' characters when the system is in the final state.

To make this a bit more interesting, suppose we want to recognise a comma-separated list of such dates. The FSA, shown below, now has a cycle and can accept a sequence of indefinite length (note that this is iteration and not full recursion, however).

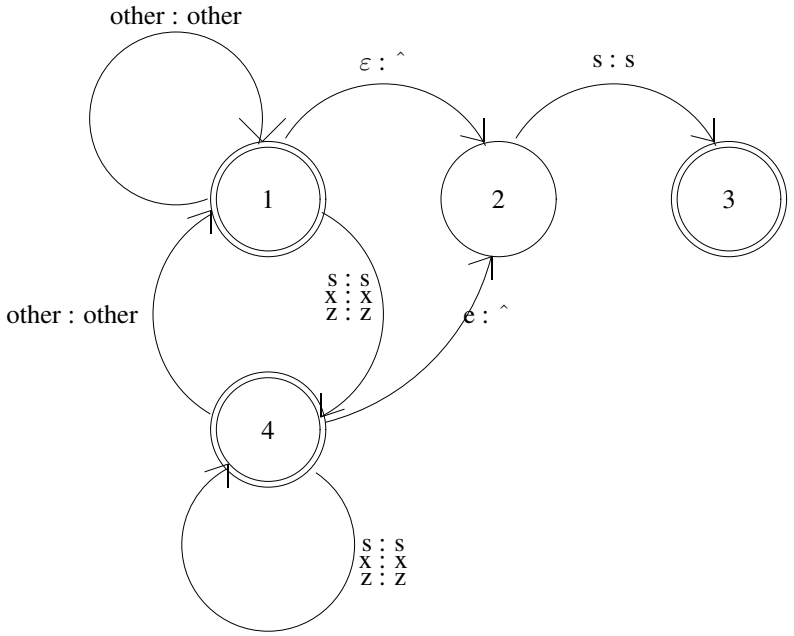


2.7 Finite state transducers

FSAs can be used to recognise particular patterns, but don't, by themselves, allow for any analysis of word forms. Hence for morphology, we use finite state transducers (FSTs) which allow the surface structure to be mapped into the list of morphemes. FSTs are useful for both analysis and generation, since the mapping is bidirectional. This approach is known as *two-level morphology*.

To illustrate two-level morphology, consider the following FST, which recognises the affix *-s* allowing for environments corresponding to the e-insertion spelling rule shown in §2.3.[9]

[9] Actually, I've simplified this slightly so the correspondance to the spelling rule is not exact: J&M give a more complex transducer which is an accurate reflection of the spelling rule.



Transducers map between two representations, so each transition corresponds to a pair of characters. As with the spelling rule, we use the special character 'ε' to correspond to the empty character and '^' to correspond to an affix boundary. The abbreviation 'other : other' means that any character not mentioned specifically in the FST maps to itself. For instance, with this FST, 'dogs' maps to 'd o g ^ s', 'f o x e s' maps to 'f o x ^ s' and 'b u z z e s' maps to 'b u z z ^ s'. When the transducer is run in analysis mode, this means the system can detect an affix boundary (and hence look up the stem and the affix in the appropriate lexicons). In generation mode, it can construct the correct string. This FST is non-deterministic.

Similar FSTs can be written for the other spelling rules for English (although to do consonant doubling correctly, information about stress and syllable boundaries is required and there are also differences between British and American spelling conventions which complicate matters). Morphology systems are usually implemented so that there is one FST per spelling rule and these operate in parallel.

One issue with this use of FSTs is that they do not allow for any internal structure of the wordform. For instance, we can produce a set of FSTs which will result in *unionised* being mapped into *un^ion^ise^ed*, but as we've seen, the affixes actually have to be applied in the right order and this isn't modelled by the FSA.

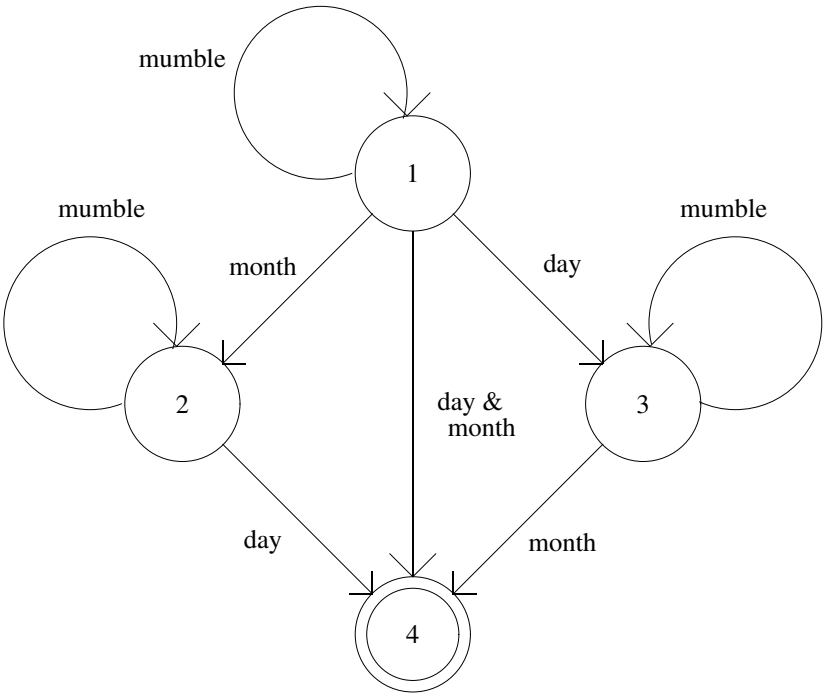
2.8 Some other uses of finite state techniques in NLP

- Grammars for simple spoken dialogue systems. Finite state techniques are not adequate to model grammars of natural languages: we'll discuss this a little in §4.11. However, for very simple spoken dialogue systems, a finite-state grammar may be adequate. More complex grammars can be written as CFGs and compiled into finite state approximations.
- Partial grammars for named entity recognition (briefly discussed in §4.11).
- Dialogue models for spoken dialogue systems (SDS). SDS use dialogue models for a variety of purposes: including controlling the way that the information acquired from the user is instantiated (e.g., the slots that are filled in an underlying database) and limiting the vocabulary to achieve higher recognition rates. FSAs can be used to record possible transitions between states in a simple dialogue. For instance, consider the problem of obtaining a date expressed as a day and a month from a user. There are four possible states, corresponding to the user input recognised so far:

    1. No information. System prompts for month and day.

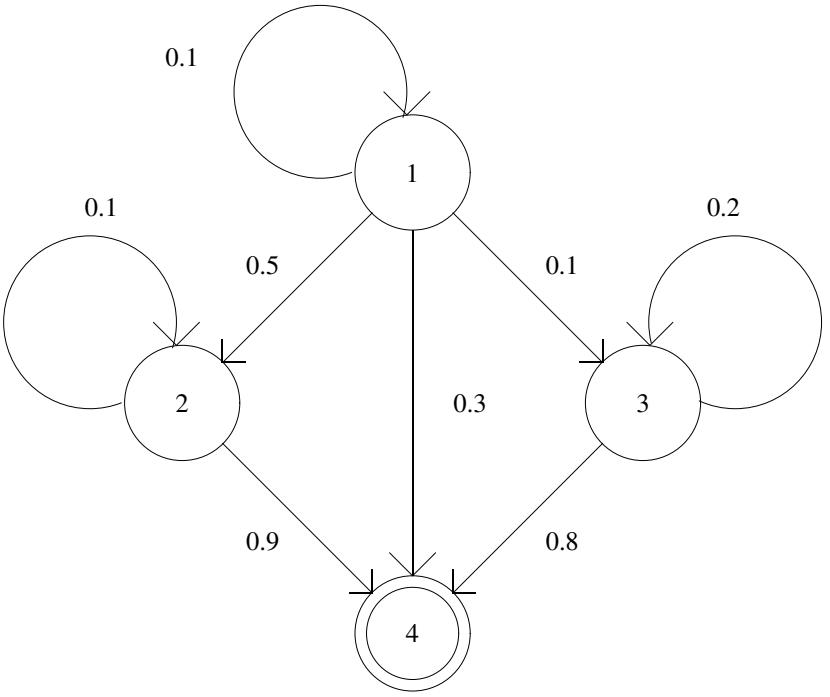
2. Month only is known. System prompts for day.
3. Day only is known. System prompts for month.
4. Month and day known.

The FSA is shown below. The loops that stay in a single state correspond to user responses that aren't recognised as containing the required information (*mumble* is the term generally used for an unrecognised input).



2.9 Probabilistic FSAs

In many cases, it is useful to augment the FSA with information about transition probabilities. For instance, in the SDS system described above, it is more likely that a user will specify a month alone than a day alone. A probabilistic FSA for the SDS is shown below. Note that the probabilities on the outgoing arcs from each state must sum to 1.



2.10 Further reading

Chapters 2 and 3 of J&M. Much of Chapter 2 should be familiar from other courses in the CST (at least to Part II students). Chapter 3 uses more elaborate transducers than I've discussed.

3 Lecture 3: Prediction and part-of-speech tagging

This lecture introduces some simple statistical techniques and illustrates their use in NLP for prediction of words and part-of-speech categories. It starts with a discussion of corpora, then introduces word prediction. Word prediction can be seen as a way of (crudely) modelling some syntactic information (i.e., word order). Similar statistical techniques can also be used to discover parts of speech for uses of words in a corpus. The lecture concludes with some discussion of evaluation.

3.1 Corpora

A *corpus* (corpora is the plural) is simply a body of text that has been collected for some purpose. A *balanced corpus* contains texts which represent different genres (newspapers, fiction, textbooks, parliamentary reports, cooking recipes, scientific papers etc etc): early examples were the Brown corpus (US English) and the Lancaster-Oslo-Bergen (LOB) corpus (British English) which are each about 1 million words: the more recent British National Corpus (BNC) contains approx 100 million words and includes 20 million words of spoken English. Corpora are important for many types of linguistic research, although Chomskyan linguists have tended to dismiss their use in favour of reliance on intuitive judgements. Corpora are essential for much modern NLP research, though NLP researchers have often used newspaper text (particularly the Wall Street Journal) rather than balanced corpora.

Distributed corpora are often annotated in some way: the most important type of annotation for NLP is part-of-speech tagging (POS tagging), which we'll discuss further below.

Corpora may also be collected for a specific task. For instance, when implementing an email answering application, it is essential to collect samples of representative emails. For interface applications in particular, collecting a corpus requires a simulation of the actual application: generally this is done by a *Wizard of Oz* experiment, where a human pretends to be a computer.

Corpora are needed in NLP for two reasons. Firstly, we have to evaluate algorithms on real language: corpora are required for this purpose for any style of NLP. Secondly, corpora provide the data source for many machine-learning approaches.

3.2 Prediction

The essential idea of prediction is that, given a sequence of words, we want to determine what's most likely to come next. There are a number of reasons to want to do this: the most important is as a form of *language modelling* for automatic speech recognition. Speech recognisers cannot accurately determine a word from the sound signal for that word alone, and they cannot reliably tell where each word starts and finishes.[10] So the most probable word is chosen on the basis of the language model, which predicts the most likely word, given the prior context. The language models which are currently most effective work on the basis of *N-grams* (a type of *Markov chain*), where the sequence of the prior $n-1$ words is used to predict the next. Trigram models use the preceeding 2 words, bigram models the preceeding word and unigram models use no context at all, but simply work on the basis of individual word probabilities. Bigrams are discussed below, though I won't go into details of exactly how they are used in speech recognition.

Word prediction is also useful in communication aids: i.e., systems for people who can't speak because of some form of disability. People who use text-to-speech systems to talk because of a non-linguistic disability usually have some form of general motor impairment which also restricts their ability to type at normal rates (stroke, ALS, cerebral palsy etc). Often they use alternative input devices, such as adapted keyboards, puffer switches, mouth sticks or eye trackers. Generally such users can only construct text at a few words a minute, which is too slow for anything like normal communication to be possible (normal speech is around 150 words per minute). As a partial aid, a word prediction system is sometimes helpful: this gives a list of candidate words that changes as the initial letters are entered by the user. The user chooses the desired word from a menu when it appears. The main difficulty with using statistical prediction models in such applications is in finding enough data: to be useful, the model really has to be trained on an individual speaker's output, but of course very little of this is likely to be available.

[10] In fact, although humans are better at doing this than speech recognisers, we also need context to recognise words, especially words like *the* and *a*.

Prediction is important in estimation of entropy, including estimations of the entropy of English. The notion of entropy is important in language modelling because it gives a metric for the difficulty of the prediction problem. For instance, speech recognition is much easier in situations where the speaker is only saying two easily distinguishable words than when the vocabulary is unlimited: measurements of entropy can quantify this, but won't be discussed further in this course.

Other applications for prediction include optical character recognition (OCR), spelling correction and text segmentation for languages such as Chinese, which are conventionally written without explicit word boundaries. Some approaches to word sense disambiguation, to be discussed in lecture 6, can also be treated as a form of prediction.

3.3 bigrams

A bigram model assigns a probability to a word based on the previous word: i.e. $P(w_n|w_{n-1})$ where w_n is the nth word in some string. The probability of some string of words $P(W_1^n)$ is thus approximated by the product of these conditional probabilities:

$$P(W_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

For example, suppose we have the following tiny corpus of utterances:

good morning
good afternoon
good afternoon
it is very good
it is good

We'll use the symbol ⟨s⟩ to indicate the start of an utterance, so the corpus really looks like:

⟨s⟩ good morning ⟨s⟩ good afternoon ⟨s⟩ good afternoon ⟨s⟩ it is very good ⟨s⟩ it is good ⟨s⟩

The bigram probabilities are given as

$$\frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

i.e. the count of a particular bigram, normalized by dividing by the total number of bigrams starting with the same word (which is equivalent to the total number of occurances of that word, except in the case of the last token, a complication which can be ignored for a reasonable size of corpus).

sequence	count	bigram probability
<s>	5	
<s> good	3	.6
<s> it	2	.4
good	5	
good morning	1	.2
good afternoon	2	.4
good <s>	2	.4
morning	1	
morning <s>	1	1
afternoon	2	
afternoon <s>	2	1
it is	2	1
is very	1	.5
is good	1	.5
very good	1	1

This yields a probability of 0.24 for the string '⟨s⟩ good ⟨s⟩' which is the highest probability utterance that we can construct on the basis of the bigrams from this corpus, if we impose the constraint that an utterance must begin with ⟨s⟩ and end with ⟨s⟩.

For application to communication aids, we are simply concerned with predicting the next word: once the user has made their choice, the word can't be changed. For speech recognition, the N-gram approach is applied to maximize the likelihood of a sequence of words, hence we're looking to find the most likely sequence overall. Notice that we can regard bigrams as comprising a simple deterministic weighted FSA. The *Viterbi algorithm*, an efficient method of applying N-grams in speech recognition and other applications, is usually described in terms of an FSA.

The probability of '⟨s⟩ very good' based on this corpus is 0, since the conditional probability of 'very' given '⟨s⟩' is 0 since we haven't found any examples of this in the training data. In general, this is problematic because we will never have enough data to ensure that we will see all possible events and so we don't want to rule out unseen events entirely. To allow for *sparse data* we have to use *smoothing*, which simply means that we make some assumption about the 'real' probability of unseen or very infrequently seen events and distribute that probability appropriately. A common approach is simply to add one to all counts: this is *add-one smoothing* which is not sound theoretically, but is simple to implement. A better approach in the case of bigrams is to *backoff* to the unigram probabilities: i.e., to distribute the unseen probability mass so that it is proportional to the unigram probabilities. This sort of estimation is extremely important to get good results from N-gram techniques, but we won't discuss the details in this course.

3.4 Part of speech tagging

Prediction techniques can be used for word classes, rather than just individual words. One important application is to part-of-speech tagging (POS tagging), where the words in a corpus are associated with a tag indicating some syntactic information that applies to that particular use of the word. For instance, consider the example sentence below:

> They can fish.

This has two readings: one (the most likely) about ability to fish and other about putting fish in cans. *fish* is ambiguous between a singular noun, plural noun and a verb, while *can* is ambiguous between singular noun, verb (the 'put in cans' use) and modal verb. However, *they* is unambiguously a pronoun. (I am ignoring some less likely possibilities, such as proper names.) These distinctions can be indicated by POS tags:

```
they PNP
can VM0 VVB VVI NN1
fish NN1 NN2 VVB VVI
```

There are several standard tagsets used in corpora and in POS tagging experiments. The one I'm using for the examples in this lecture is CLAWS 5 (C5) which is given in full in appendix C in J&M. The meaning of the tags above is:

```
NN1 singular noun
NN2 plural noun
PNP personal pronoun
VM0 modal auxiliary verb
VVB base form of verb (except infinitive)
VVI infinitive form of verb (i.e. occurs with 'to')
```

A POS tagger resolves the lexical ambiguities to give the most likely set of tags for the sentence. In this case, the right tagging is likely to be:

> They_PNP can_VM0 fish_VVB ._PUN

Note the tag for the full stop: punctuation is treated as unambiguous. POS tagging can be regarded as a form of very basic word sense disambiguation.

The other syntactically possible reading is:

> They_PNP can_VVB fish_NN2 ._PUN

However, POS taggers (unlike full parsers) don't attempt to produce globally coherent analyses. Thus a POS tagger might return:

> They_PNP can_VM0 fish_NN2 ._PUN

despite the fact that this doesn't correspond to a possible reading of the sentence.

POS tagging is useful as a way of annotating a corpus because it makes it easier to extract some types of information (for linguistic research or NLP experiments). It also acts as a basis for more complex forms of annotation. Named entity recognisers (discussed in lecture 4) are generally run on POS-tagged data. POS taggers are sometimes run as preprocessors to full parsing, since this can cut down the search space to be considered by the parser.

3.5 Stochastic POS tagging

One form of POS tagging applies the N-gram technique that we saw above, but in this case it applies to the POS tags rather than the individual words. The most common approaches depend on a small amount of manually tagged *training data* from which POS N-grams can be extracted.[11] I'll illustrate this with respect to another trivial corpus:

> They used to can fish in those towns. But now few people fish there.

This might be tagged as follows:

```
They_PNP used_VVD to_TO0 can_VVI fish_NN2 in_PRP those_DT0 towns_NN2 ._PUN
But_CJC now_AV0 few_DT0 people_NN2 fish_VVB in_PRP these_DT0 areas_NN2 ._PUN
```

This yields the following counts and probabilities:

```
sequence       count      bigram probability

AV0              1
AV0 DT0          1                1

CJC              1                1
CJC AV0          1

DT0              3
DT0 NN2          3                1

NN2              4
NN2 PRP          1                0.25
NN2 PUN          2                0.5
NN2 VVB          1                0.25

PNP              1
PNP VVD          1                1

PRP              1
PRP DT0          2                1

PUN              1
PUN CJC          1                1

TO0              1
```

[11] It is possible to build POS taggers that work without a hand-tagged corpus, but they don't perform as well as a system trained on even a 1,000 word corpus which can be tagged in a few hours. Furthermore, these algorithms still require a lexicon which associates possible tags with words.

```
TO0 VVI          1                1

VVB              1
VVB PRP          1                1

VVD              1
VVD TO0          1                1

VVI              1
VVI NN2          1                1
```

We can also obtain a lexicon from the tagged data:

```
word   tag  count

they   PNP    1
used   VVD    1
to     TO0    1
can    VVI    1
fish   NN2    1
       VVB    1
in     PRP    2
those  DT0    1
towns  NN2    1
.      PUN    1
but    CJC    1
now    AV0    1
few    DT0    1
people NN2    1
these  DT0    1
areas  NN2    1
```

The idea of stochastic POS tagging is that the tag can be assigned based on consideration of the lexical probability (how likely it is that the word has that tag), plus the sequence of prior tags. For a bigram model, we only look at a single previous tag. This is slightly more complicated than the word prediction case because we have to take into account both words and tags.

We are trying to estimate the probability of a sequence of tags given a sequence of words: $P(T|W)$. By Bayes theorem:

$$P(T|W) = \frac{P(T)P(W|T)}{P(W)}$$

Since we're looking at assigning tags to a particular sequence of words, $P(W)$ is constant, so for a relative measure of probability we can use:

$$P(T|W) = P(T)P(W|T)$$

We now have to estimate $P(T)$ and $P(W|T)$. If we make the bigram assumption, $P(T)$ is approximated by $P(t_i|t_{i-1})$ — i.e., the probability of some tag given the immediately preceeding tag. We approximate $P(W|T)$ as $P(w_i|t_i)$. These values can be estimated from the corpus frequencies.

Note that we end up multiplying $P(t_i|t_{i-1})$ with $P(w_i|t_i)$ (the probability of the word given the tag) rather than $P(t_i|w_i)$ (the probability of the tag given the word). For instance, if we're trying to choose between the tags NN2 and VVB for *fish* in the sentence *they fish*, we calculate $P(\text{NN2}_i|\text{PNP}_{i-1})$, $P(\text{fish}_i|\text{NN2}_i)$, $P(\text{VVB}_i|\text{PNP}_{i-1})$ and $P(\text{fish}_i|\text{VVB}_i)$.

In fact, POS taggers generally use trigrams rather than bigrams — the relevant equations are given in J&M, page 306. As with word prediction, backoff and smoothing are crucial for reasonable performance.

When a POS tagger sees a word which was not in its training data, we need some way of assigning possible tags to the word. One approach is simply to use all possible *open class* tags, with probabilities based on the unigram probabilities

of those tags. Open class words are ones for which we can never give a complete list for a living language, since words are always being added: i.e., verbs, nouns, adjectives and adverbs. The rest are considered closed class. A better approach is to use a morphological analyser to restrict this set: e.g., words ending in *-ed* are likely to be VVD (simple past) or VVN (past participle), but can't be VVG (-ing form).

3.6 Evaluation of POS tagging

POS tagging algorithms are evaluated in terms of percentage of correct tags. The standard assumption is that every word should be tagged with exactly one tag, which is scored as correct or incorrect: there are no marks for near misses. Generally there are some words which can be tagged in only one way, so are automatically counted as correct. Punctuation is generally given an unambiguous tag. Therefore the success rates of over 95% which are generally quoted for POS tagging are a little misleading: the baseline of choosing the most common tag based on the training set often gives 90% accuracy. Some POS taggers returns multiple tags in cases where more than one tag has a similar probability.

It is worth noting that increasing the size of the tagset does not necessarily result in decreased performance: this depends on whether the tags that are added can generally be assigned unambiguously or not. Potentially, adding more fine-grained tags could increase performance. For instance, suppose we wanted to distinguish between present tense verbs accoring to whether they were 1st, 2nd or 3rd person. With the C5 tagset, and the stochastic tagger described, this would be impossible to do with high accuracy, because all pronouns are tagged PRP, hence they provide is no discriminating power. On the other hand, if we tagged *I* and *we* as PRP1, *you* as PRP2 and so on, the N-gram approach would allow some discrimination. In general, predicting on the basis of classes means we have less of a sparse data problem than when predicting on the basis of words, but we also lose discriminating power. There is also something of a tradeoff between the utility of a set of tags and their usefulness in POS tagging. For instance, C5 assigns separate tags for the different forms of *be*, which is redundant for many purposes, but helps make distinctions between other tags in tagging models where the context is given by a tag sequence alone (i.e., rather than considering words prior to the current one).

POS tagging exemplifies some general issues in NLP evaluation:

Training data and test data The assumption in NLP is always that a system should work on novel data, therefore test data must be kept unseen.

For machine learning approaches, such as stochastic POS tagging, the usual technique is to spilt a data set into 90% training and 10% test data. Care needs to be taken that the test data is representative.

For an approach that relies on significant hand-coding, the test data should be literally unseen by the researchers. Development cycles involve looking at some initial data, developing the algorithm, testing on unseen data, revising the algorithm and testing on a new batch of data. The seen data is kept for regression testing.

Baselines Evaluation should be reported with respect to a baseline, which is normally what could be achieved with a very basic approach, given the same training data. For instance, the baseline for POS tagging with training data is to choose the most common tag for a particular word on the basis of the training data (and to simply choose the most frequent tag of all for unseen words).

Ceiling It is often useful to try and compute some sort of ceiling for the performance of an application. This is usually taken to be human performance on that task, where the ceiling is the percentage agreement found between two annotators (*interannotator agreement*). Fot POS tagging, this has been reported as 96% (which makes existing POS taggers look impressive). However this raises lots of questions: relatively untrained human annotators working independently often have quite low agreement, but trained annotators discussing results can achieve much higher performance (approaching 100% for POS tagging). Human performance varies considerably between individuals. In any case, human performance may not be a realistic ceiling on relatively unnatural tasks, such as POS tagging.

Error analysis The error rate on a particular problem will be distributed very unevenly. For instance, a POS tagger will never confuse the tag PUN with the tag VVN (past participle), but might confuse VVN with AJ0 (adjective) because there's a systematic ambiguity for many forms (e.g., *given*). For a particular application, some errors

may be more important than others. For instance, if one is looking for relatively low frequency cases of denominal verbs (that is verbs derived from nouns — e.g., *canoe*, *tango*, *fork* used as verbs), then POS tagging is not directly useful in general, because a verbal use without a characteristic affix is likely to be mistagged. This makes POS-tagging less useful for lexicographers, who are often specifically interested in finding examples of unusual word uses. Similarly, in text categorization, some errors are more important than others: e.g. treating an incoming order for an expensive product as junk email is a much worse error than the converse.

Reproducibility If at all possible, evaluation should be done on a generally available corpus so that other researchers can replicate the experiments.

3.7 Further reading

This lecture has skimmed over material that is covered in several chapters of J&M. See 5.9 for the Viterbi algorithm, Chapter 6 for N-grams (especially 6.3, 6.4 and 6.7), 7.1-7.3 for speech recognition and Chapter 8 on POS tagging.

4 Lecture 4: Parsing and generation I

In this lecture, we'll discuss syntax in a way which is much closer to the standard notions in formal linguistics than POS-tagging is. To start with, we'll briefly motivate the idea of a generative grammar in linguistics, review the notion of a context-free grammar and then show a context-free grammar for a tiny fragment of English. We'll then show how context free grammars can be used to implement generators and parsers, and discuss chart parsing, which allows efficient processing of strings containing a high degree of ambiguity. Finally we'll briefly touch on probabilistic context-free approaches.

4.1 Generative grammar

Since Chomsky's work in the 1950s, much work in formal linguistics has been concerned with the notion of a *generative grammar* — i.e., a formally specified grammar that can generate all and only the acceptable sentences of a natural language. It's important to realise that nobody has actually written such a grammar for any natural language or even come close to doing so: what most linguists are really interested in is the principles that underly such grammars, especially to the extent that they apply to all natural languages. NLP researchers, on the other hand, are at least sometimes interested in actually building and using large-scale detailed grammars.

The formalisms which are of interest to us for modelling syntax assume assign internal structure to the strings of a language, which can be represented by bracketing. We already saw some evidence of this in derivational morphology (the unionised example), but here we are concerned with the structure of phrases. For instance, the sentence:

> the dog slept

can be bracketed

> ((the (big dog)) slept)

The phrase, *big dog*, is an example of a *constituent* (i.e. something that is enclosed in a pair of brackets): *the big dog* is also a constituent, but *the big* is not. Constituent structure is generally justified by arguments about substitution which I won't go into here: J&M discuss this briefly, but see an introductory syntax book for a full discussion. In this course, I will simply give bracketed structures and hope that the constituents make sense intuitively, rather than trying to justify them.

Two grammars are said to be *weakly-equivalent* if they generate the same strings. Two grammars are *strongly-equivalent* if they assign the same bracketings to all strings they generate.

In most, but not all, approaches, the internal structures are given labels. For instance, *the big dog* is a *noun phrase* (abbreviated NP), *slept*, *slept in the park* and *licked Sandy* are *verb phrase*s (VPs). The labels such as NP and VP correspond to non-terminal symbols in a grammar. In this lecture, we'll discuss the use of simple context-free grammars for language description, moving onto a more expressive formalism in lecture 5.

4.2 Context free grammars

The idea of a context-free grammar (CFG) should be familiar from formal language theory. A CFG has four components, described here as they apply to grammars of natural languages:

1. a set of non-terminal symbols (e.g., S, VP), conventionally written in uppercase;
2. a set of terminal symbols (i.e., the words), conventionally written in lowercase;
3. a set of rules (productions), where the left hand side (the mother) is a single non-terminal and the right hand side is a sequence of one or more non-terminal or terminal symbols (the daughters);
4. a start symbol, conventionally S, which is a member of the set of non-terminal symbols.

The formal description of a CFG generally allows productions with an empty righthandside (e.g., Det $\rightarrow \varepsilon$). It is convenient to exclude these however, since they complicate parsing algorithms, and a weakly-equivalent grammar can always be constructed that disallows such *empty productions*.

A grammar in which all nonterminal daughters are the leftmost daughter in a rule (i.e., where all rules are of the form $X \rightarrow Ya*$), is said to be *left-associative*. A grammar where all the nonterminals are rightmost is *right-associative*. Such grammars are weakly-equivalent to regular grammars (i.e., grammars that can be implemented by FSAs), but natural languages seem to require more expressive power than this (see §4.11).

4.3 A simple CFG for a fragment of English

The following tiny fragment is intended to illustrate some of the properties of CFGs so that we can discuss parsing and generation. It has some serious deficiencies as a representation of even this fragment, which we'll ignore for now, though we'll discuss some of them in lecture 5.

```
S -> NP VP
VP -> VP PP
VP -> V
VP -> V NP
VP -> V VP
NP -> NP PP
PP -> P NP
;;; lexicon
V -> can
V -> fish
NP -> fish
NP -> rivers
NP -> pools
NP -> December
NP -> Scotland
NP -> it
NP -> they
P -> in
```

The rules with terminal symbols on the RHS correspond to the lexicon. Here and below, comments are preceeded by `;;;`

Here are some strings which this grammar generates, along with their bracketings:

they fish
(S (NP they) (VP (V fish)))

they can fish
(S (NP they) (VP (V can) (VP (V fish))))
;;; the modal verb 'are able to' reading
(S (NP they) (VP (V can) (NP fish)))
;;; the less plausible, put fish in cans, reading

they fish in rivers
(S (NP they) (VP (VP (V fish)) (PP (P in) (NP rivers))))

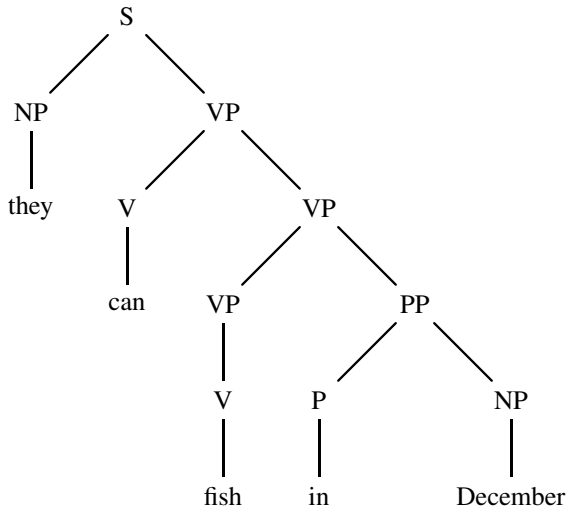
they fish in rivers in December
(S (NP they) (VP (VP (V fish)) (PP (P in) (NP (NP rivers) (PP (P in) (NP December))))))
;;; i.e. the implausible reading where the rivers are in December
;;; (cf rivers in Scotland)
(S (NP they) (VP (VP (VP (V fish)) (PP (P in) (NP (NP rivers)))) (PP (P in) (NP December))))
;;; i.e. the fishing is done in December

One important thing to notice about these examples is that there's lots of potential for ambiguity. In the *they can fish* example, this is due to *lexical ambiguity* (it arises from the dual lexical entries of *can* and *fish*), but the last example demonstrates purely *structural ambiguity*. In this case, the ambiguity arises from the two possible *attachments* of the prepositional phrase (PP) *in December*: it can attach to the NP (*rivers*) or to the VP. These attachments correspond to different semantics, as indicated by the glosses. PP attachment ambiguities are a major headache in parsing, since sequences of four or more PPs are common in real texts and the number of readings increases as the Catalan series, which is exponential. Other phenomena have similar properties: for instance, compound nouns (e.g. *long-stay car park shuttle bus*).

Notice that *fish* could have been entered in the lexicon directly as a VP, but that this would cause problems if we were doing derivational morphology, because we want to say that suffixes like *-ed* apply to Vs. Making *rivers* etc NPs rather than nouns is a simplification I've adopted here to keep the grammar smaller.

4.4 Parse trees

Parse trees are equivalent to bracketed structures, but are easier to read for complex cases. A parse tree and bracketed structure for one reading of *they can fish in December* is shown below. The correspondence should be obvious.



(S (NP they) (VP (V can) (VP (VP (V fish)) (PP (P in) (NP December)))))

4.5 Using a grammar as a random generator

The following simple algorithm illustrates how a grammar can be used to generate sentences.

Expand cat *category sentence-record*:
Let *possibilities* be a set containing all lexical items which match *category* and all rules with left-hand side *category*
If *possibilities* is empty,
then fail
else
    Randomly select a possibility *chosen* from *possibilities*
    If *chosen* is lexical,
    then append it to *sentence-record*
    else **expand cat** on each rhs category in *chosen* (left to right) with the updated *sentence-record*
        return *sentence-record*

For instance:

Expand cat S ()

possibilities = `S -> NP VP`
chosen = `S -> NP VP`

Expand cat NP ()
possibilities = it, they, fish
chosen = fish
sentence-record = (fish)
Expand cat VP (fish)
possibilities = `VP -> V, VP -> V VP, VP -> V NP`
chosen = `VP -> V`

Expand cat V (fish)
possibilities = fish, can
chosen = fish
sentence-record = (fish fish)

Obviously, the strings generated could be arbitrarily long. If in this naive generation algorithm, we explored all the search space rather than randomly selecting a possible expansion, the algorithm wouldn't terminate.

Real generation operates from semantic representations, which aren't encoded in this grammar, so in what follows we'll concentrate on describing parsing algorithms instead. However, it's important to realise that CFGs are, in principle, bidirectional.

4.6 Chart parsing

In order to parse with reasonable efficiency, we need to keep a record of the rules that we have applied so that we don't have to backtrack and redo work that we've done before. This works for parsing with CFGs because the rules are independent of their context: a VP can always expand as a V and an NP regardless of whether or not it was preceeded by an NP or a V, for instance. (In some cases we may be able to apply techniques that look at the context to cut down search space, because we can tell that a particular rule application is never going to be part of a sentence, but this is strictly a filter: we're never going to get incorrect results by reusing partial structures.) This record keeping strategy is an application of dynamic programming which is used in processing formal languages too. In NLP the data structure used for recording partial results is generally known as a *chart* and algorithms for parsing using such structures are referred to as *chart parsing*.

A chart is a list of *edges*. In the simplest version of chart parsing, each edge records a rule application and has the following structure:

[*id*,*left_vertex*, *right_vertex*,*mother_category*, *daughters*]

A vertex is an integer representing a point in the input string, as illustrated below:

```
. they . can . fish .
0      1     2      3
```

mother_category refers to the rule that has been applied to create the edge. *daughters* is a list of the edges that acted as the daughters for this particular rule application: it is there purely for record keeping so that the output of parsing can be a labelled bracketing.

For instance, the following edges would be among those found on the chart after a complete parse of *they can fish* according to the grammar given above (id numbering is arbitrary):

```
id  left   right  mother   daughters
3    1       2      V        (can)
4    2       3      NP       (fish)
5    2       3      V        (fish)
6    2       3      VP       (5)
7    1       3      VP       (3 5)
8    1       3      VP       (3 4)
```

The daughters for the terminal rule applications are simply the input word strings. Note that local ambiguities correspond to situations where a particular span has more than one associated edge. We'll see below that we can *pack* structures so that we never have two edges with the same category and the same span, but we'll ignore this for the moment (see §4.8). Also, in this chart we're only recording complete rule applications: this is *passive* chart parsing. The more efficient *active* chart is discussed below, in §4.9.

4.7 A bottom-up passive chart parser

The following pseudo-code sketch is for a very simple chart parser. The main function is **Add new edge** which is called for each word in the input going left to right. **Add new edge** recursively scans backwards looking for other daughters.

Parse:
  Initialize the chart (i.e., clear previous results)
  For each word *word* in the input sentence, let *from* be the left vertex, *to* be the right vertex and *daughters* be (*word*)
    For each category *category* that is lexically associated with *word*
      Add new edge *from*, *to*, *category*, *daughters*
  Output results for all spanning edges
  (i.e., ones that cover the entire input and which have a mother corresponding to the root category)

Add new edge *from*, *to*, *category*, *daughters*:
  Put edge in chart: [*id*,*from*,*to*, *category*,*daughters*]
  For each *rule* in the grammar of form *lhs* -> $cat_1 \ldots cat_{n-1}$,*category*
    Find sets of contiguous edges [id_1,$from_1$,to_1, cat_1,$daughters_1$] ... [id_{n-1},$from_{n-1}$,*from*, cat_{n-1},$daughters_{n-1}$]
    (such that $to_1 = from_2$ etc)
    For each set of edges, **Add new edge** $from_1$, *to*, *lhs*, ($id_1 \ldots id$)

Notice that this means that the grammar rules are indexed by their rightmost category, and that the edges in the chart must be indexed by their *to* vertex (because we scan backward from the rightmost category). Consider:

```
. they . can . fish .
0      1     2      3
```

The following diagram shows the chart edges as they are constructed in order (when there is a choice, taking rules in a priority order according to the order they appear in the grammar):

id	left	right	mother	daughters
1	0	1	NP	(they)
2	1	2	V	(can)
3	1	2	VP	(2)
4	0	2	S	(1 3)
5	2	3	V	(fish)
6	2	3	VP	(5)
7	1	3	VP	(2 6)
8	0	3	S	(1 7)
9	2	3	NP	(fish)
10	1	3	VP	(2 9)
11	0	3	S	(1 10)

The spanning edges are 11 and 8: the output routine to give bracketed parses simply outputs a left bracket, outputs the category, recurses through each of the daughters and then outputs a right bracket. So, for instance, the output from edge 11 is:

```
(S (NP they) (VP (V can) (NP fish)))
```

4.8 Packing

The algorithm given above is exponential in the case where there are an exponential number of parses. The body of the algorithm can be modified so that it runs in cubic time, though producing the output is still exponential. The modification is simply to change the daughters value on an edge to be a set of lists of daughters and to make an equality check before adding an edge so we don't add one that's equivalent to an existing one. That is, if we are about to add an edge:

[*id*,*left_vertex*, *right_vertex*,*mother_category*, *daughters*]

and there is an existing edge:

[*id-old*,*left_vertex*, *right_vertex*,*mother_category*, *daughters-old*]

we simply modify the old edge to record the new daughters:

[*id-old*,*left_vertex*, *right_vertex*,*mother_category*, *daughters-old* ⊔ *daughters*]

There is no need to recurse with this edge, because we couldn't get any new results.
For the example above, everything proceeds as before up to edge 9:

```
id  left   right  mother   daughters
1    0       1      NP       {(they)}
2    1       2      V        {(can)}
3    1       2      VP       {(2)}
4    0       2      S        {(1 3)}
5    2       3      V        {(fish)}
6    2       3      VP       {(5)}
7    1       3      VP       {(2 6)}
8    0       3      S        {(1 7)}
9    2       3      NP       {(fish)}
```

However, rather than add edge 10, which would be:

```
10   1       3      VP       (2 9)
```

we match this with edge 7, and simply add the new daughters to that.

```
7    1       3      VP       {(2 6), (2 9)}
```

The algorithm then terminates. We only have one spanning edge (edge 8) but the display routine is more complex because we have to consider the alternative sets of daughters for edge 7. (You should go through this to convince yourself that the same results are obtained as before.) Although in this case, the amount of processing saved is small, the effects are much more important with longer sentences (consider *he believes they can fish*, for instance).

4.9 Active chart parsing

A more minor efficiency improvement is obtained by storing the results of partial rule applications. This is *active* chart parsing, so called because the partial edges are considered to be active: i.e. they 'want' more input to make them complete. An active edge records the input it expects as well as the daughters it has already seen. For instance, with an active chart parser, we might have the following edges after seeing *they*:

```
id  left   right  mother    expected      daughters
1    0       1      NP                     {(they)}
2    0       1      S          VP          {(1 ?)}
```

The daughter marked as ? will be instantiated by the edge corresponding to the VP when it is found.

4.10 Ordering the search space

In the pseudo-code above, the order of addition of edges to the chart was determined by the recursion. In general, chart parsers make use of an *agenda* of edges, so that the next edges to be operated on are the ones that are first on the agenda. Different parsing algorithms can be implemented by making this agenda a stack or a queue, for instance.

So far, we've considered *bottom up* parsing: an alternative is *top down* parsing, where the initial edges are given by the rules whose mother corresponds to the start symbol.

Some efficency improvments can be obtained by ordering the search space appropriately, though which version is most efficient depends on properties of the individual grammar. However, the most important reason to use an explicit agenda is when we are returning parses in some sort of priority order, corresponding to weights on different grammar rules or lexical entries.

Weights can be manually assigned to rules and lexical entries in a manually constructed grammar. However, in the last decade, a lot of work has been done on automatically acquiring probabilities from a corpus annotated with trees (a *treebank*), either as part of a general process of automatic grammar acquisition, or as automatically acquired additions to a manually constructed grammar. Probabilistic CFGs (PCFGs) can be defined quite straightforwardly, if the assumption is made that the probabilities of rules and lexical entries are independent of one another (of course this assumption is not correct, but the orderings given seem to work quite well in practice). The importance of this is that we rarely want to return all parses in a real application, but instead we want to return those which are top-ranked: i.e., the most likely parses. This is especially true when we consider that realistic grammars can easily return many thousands of parses for sentences of quite moderate length (20 words or so). If edges are prioritized by probability, very low priority edges can be completely excluded from consideration if there is a cut-off such that we can be reasonably certain that no edges with a lower priority than the cut-off will contribute to the highest-ranked parse. Limiting the number of analyses under consideration is known as *beam search* (the analogy is that we're looking within a beam of light, corresponding to the highest probability edges). Beam search is linear rather than exponential or cubic. Just as importantly, a good priority ordering from a parser reduces the amount of work that has to be done to filter the results by whatever system is processing the parser's output.

4.11 Why can't we use FSAs to model the syntax of natural languages?

In this lecture, we started using CFGs. This raises the question of why we need this more expressive (and hence computationally expensive) formalism, rather than modelling syntax with FSAs. One reason is that the syntax of natural languages cannot be described by an FSA, even in principle, due to the presence of *centre-embedding*, i.e. structures which map to:

$$A \rightarrow \alpha A \beta$$

and which generate grammars of the form $a^n b^n$. For instance:

the students the police arrested complained

has a centre-embedded structure. However, humans have difficulty processing more than two levels of embedding:

? the students the police the journalists criticised arrested complained

If the recursion is finite (no matter how deep), then the strings of the language can be generated by an FSA. So it's not entirely clear whether formally an FSA might not suffice.

There's a fairly extensive discussion of these issues in J&M , but there are two essential points for our purposes:

1. Grammars written using finite state techniques alone are very highly redundant, which makes them very difficult to build and maintain.
2. Without internal structure, we can't build up good semantic representations.

Hence the use of more powerful formalisms: in the next lecture, we'll discuss the inadequacies of simple CFGs from a similar perspective.

However, FSAs are very useful for partial grammars which don't require full recursion. In particular, for information extraction, we need to recognise *named entities*: e.g. Professor Smith, IBM, 101 Dalmations, the White House, the Alps and so on. Although NPs are in general recursive (*the man who likes the dog which bites postmen*), relative clauses are not generally part of named entities. Also the internal structure of the names is unimportant for IE. Hence FSAs can be used, with sequences such as 'title surname', 'DT0 PNP' etc

CFGs can be automatically compiled into approximately equivalent FSAs by putting bounds on the recursion. This is particularly important in speech recognition engines.

4.12 Further reading

This lecture has covered material which J&M discuss in chapters 9 and 10, though we also touched on PCFGs (covered in their chapter 12) and issues of language complexity which they discuss in chapter 13. J&M's discussion covers the Earley algorithm, which can be thought of as a form of active top-down chart parsing. I chose to concentrate on bottom-up parsing in this lecture, mainly because I find it easier to describe, but also because it is easier to see how to extend this to PCFGs. Bottom-up parsing also seems to have better practical performance with the sort of grammars we'll look at in lecture 5.

There are a large number of introductory linguistics textbooks which cover elementary syntax and discuss concepts such as constituency. For instance, students could usefully look at the first five chapters of Tallerman (1998):

Tallerman, Maggie, *Understanding Syntax*, Arnold, London, 1998

An alternative would be the first two chapters of Sag and Wasow (1999) — copies should be in the Computer Laboratory library. This has a narrower focus than most other syntax books, but covers a much more detailed grammar fragment. The later chapters (particularly 3 and 4) are relevant for lecture 5.

Sag, Ivan A. and Thomas Wasow, *Syntactic Theory — a formal introduction*, CSLI Publications, Stanford, CA, USA, 1999