

# **Informatics 2A 2015–16**

## Lecture 1

### Introduction and Course Administration

John Longley  
Shay Cohen

22nd September 2015

## **Subject of the course**

The full title of the course is:

**Informatics 2A: Processing Formal and Natural Languages**

The course is about ways of describing, specifying and processing both **computer languages** and **human languages**.

Remarkably, many important ideas and methods are common to both of these — though there are also major differences.

**Lecture 2** will give a overview and roadmap of the intellectual content of the course.

## Course staff

### Lecturers:

- John Longley ([jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk))  
Office hour: Thursdays 10:30–11:30 (from Wk 3), IF 4.11
- Shay Cohen ([scohen@inf.ed.ac.uk](mailto:scohen@inf.ed.ac.uk))  
Office hour: Mondays 11:30–12:30 (from Wk 3), IF 4.26

### Informatics 2 Year Organiser:

- Sharon Goldwater ([sgwater@inf.ed.ac.uk](mailto:sgwater@inf.ed.ac.uk))

### Course Secretary:

- Kendal Reid, ITO ([ito@inf.ed.ac.uk](mailto:ito@inf.ed.ac.uk), Forrest Hill 1.B15)

## Communication mechanisms

**Course website:** <http://www.inf.ed.ac.uk/teaching/courses/inf2a/>  
This is the main anchor point for all course information and material. Bookmark it now!

**Email list:** [inf2a-students@inf.ed.ac.uk](mailto:inf2a-students@inf.ed.ac.uk)

Important administrative announcements (e.g. changes to deadlines) will be posted here.

**Discussion forum:**

For discussion of course content, lectures, assignments etc.

**Course reps:** [ug2-reps@inf.ed.ac.uk](mailto:ug2-reps@inf.ed.ac.uk)

For feedback from you to course staff.

It is your responsibility to check (especially) your email and the website and to stay in touch with what's going on.

## Lectures

Lectures are on **Tuesday, Thursday** and **Friday** afternoons.

- Tuesdays & Fridays: 16:10–17:00, Appleton Tower, LT 5.
- Thursdays: 17:10–18:00, David Hume Tower, Hall C.

The last lecture is a revision lecture on Thursday of Week 11 (3 December).

Lectures will be **videoed**, with videos linked from the webpage.

## Lecture materials

The website contains links to the slides for each lecture.

These links will become live immediately after (or just before) the lecture takes place.

For those who wish to see the material in advance (e.g., students with an adjustment schedule), last year's slides are available via a link at the top of the page.

If you want printed copies of lecture slides, please print them off yourself if you need them, bearing in mind the cost to [the planet](#). (E.g., use the **4up** option.)

## Tutorials

Tutorials for Inf2a start in **Week 3** (beginning Monday 5 Oct). So Tutorial  $n$  happens in Week  $n + 2$ .

Each tutorial will cover material from the previous week's lectures. A *tutorial sheet*, consisting of problems to be discussed in Tutorial  $n$ , will be released (on the course website) on the Friday of Week  $n + 1$ .

You should have already received an email from Kendal Reid via **inf2a-students**, advertising the preliminary allocation of students to tutor groups. If you can't make the time of your allocated group, please email Kendal suggesting some groups you *could* manage. Or if you need to change tutor groups for any other reason, **please let Kendal know** (important!).

*N.B. If you miss two tutorials in a row, your PT will be notified and you will be chased up!*

## **Python and Lab Sessions**

In parallel with the lecture material, you will be learning the programming language (**Python**), and learning to use the associated Natural Language Toolkit (**NLTK**). These skills will be needed for the second assessed course assignment.

This can be done with the help of worksheets, available via the website, which you can work through at the **Lab Session** to which you have been assigned (or a different one, or on your own).

The purposes of the lab sessions are: assistance in learning Python/NLTK; assistance with coursework assignments; additional feedback on assignment 1. **Lab Demonstrators** will be on hand at these sessions to offer help.

Lab sessions start in Week 3.

## Assessed coursework

There will be **two** assessed coursework assignments, carrying equal weight. Each is worth 12.5% of the course mark.

**Assignment 1:** issued Tue 14 Oct, due in Tue 28 Oct, 4pm

**Assignment 2:** issued Fri 7 Nov, due in Fri 28 Nov, 4pm

Both assignments will be computer-based, and are to be submitted online from DICE machines. Assignment 1 is in **Java** and Assignment 2 is in **Python**.

Marked assignments will be available for collection by students **2 weeks** after the submission deadline.

All assessed work must be **your own individual work**.

## **Inf2A exam**

The main exam takes place in December 2014.

The resit is in August 2015.

Exam dates are set by Student Administration, not us. We'll let you know once they are announced.

The exam is pen-and-paper, and lasts 2 hours. It consists of:

- 5 compulsory short questions (10% each), and
- a choice of 2 out of 3 longer questions (25% each).

The total 100% contributes 75% to the course mark.

## Recommended reading

The following textbook is highly recommended for this course and many other Natural Language courses in later years:

- D. Jurafsky and J. Martin, **Speech and Language Processing (2nd edition)**, Prentice-Hall, 2009.

For the formal language side, suitable texts include:

- D. Kozen, **Automata and Computability**, Springer, 2000.
- M. Sipser, **Introduction to the Theory of Computation (3rd edition)**, Cengage Learning, 2012.

Lectures will stick closely to the terminology and notation of the Jurafsky & Martin and Kozen texts.

## Formative feedback

Assessed coursework provides you with **summative feedback** on the course.

*Formative feedback* is feedback on non-assessed parts of the course. This helps your understanding and serves as *feedforward* towards future assessed components (e.g., the exam). Formative feedback provided in Inf2A includes:

- Self-assessment and challenge questions in lectures.
- Feedback from tutors in tutorials.
- Feedback from demonstrators in lab sessions.
- Feedback from lecturers at drop-in office hours.

## Needing help?

- If you are suffering from **personal circumstances** that may be adversely affecting your work, contact your **PT**.
- If you wish to apply for a coursework **deadline extension** (for a good reason!), contact the **Informatics Teaching Organization**, *not* the lecturers. Except in exceptional circumstances, extensions will only be granted if applied for *prior* to the coursework deadline.
- If you are having difficulties **understanding** the course material, possible sources of help are: your **class mates**, the **discussion forum**, your **tutor**, the **lecturers**.
- If you wish to anonymously raise any **issue** about the course material or delivery, contact **ug2-reps@inf.ed.ac.uk**

**Enjoy the course!**

Lecture 2 on Thursday: Overview and roadmap of the intellectual content of the course (JL+SC).

Volunteers for UG2 student reps: please come and add your name to the list.

Any questions?

# Course Roadmap

## Informatics 2A: Lecture 2

John Longley  
Shay Cohen

School of Informatics  
University of Edinburgh  
[jrl,scohen@inf.ed.ac.uk](mailto:jrl,scohen@inf.ed.ac.uk)

24 September 2015

## 1 What Is Inf2a about?

- Formal and natural languages
- The language processing pipeline
- Comparison between FLs and NLs

## 2 Course overview

- Levels of language complexity
- Formal language component
- Natural language component

# Formal and natural languages

This course is about methods for describing, specifying and processing **languages** of various kinds:

- **Formal (computer) languages**, e.g. Java, Haskell, HTML, SQL, Postscript, ...
- **Natural (human) languages**, e.g. English, Greek, Japanese.
- '**Languages**' that represent the **behaviour** of some machine or system. E.g. think about 'communicating' with a vending machine via coin insertions and button presses:

```
insert50p . pressButton1 . deliverMarsBar
```

# A common theoretical core

We'll be focusing on certain theoretical concepts that can be applied to each of the above three domains:

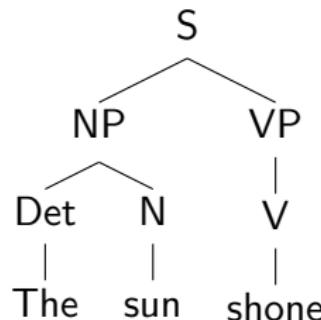
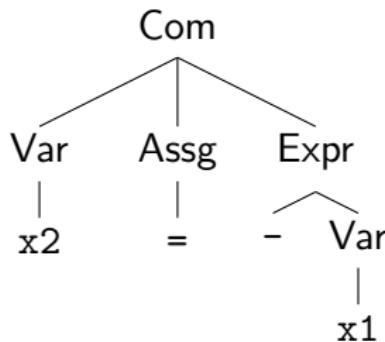
- regular languages
- finite state machines
- context-free languages, syntax trees
- types, compositional semantics

The fact that the same underlying theory can be applied in such diverse contexts suggests that the theory is somehow **fundamental**, and worth learning about!

Mostly, we'll be looking at various aspects of formal languages (mainly AS) and natural languages (mainly JL). As we'll see, there are some important similarities between formal and natural languages — and some important differences.

# Syntax trees: a central concept

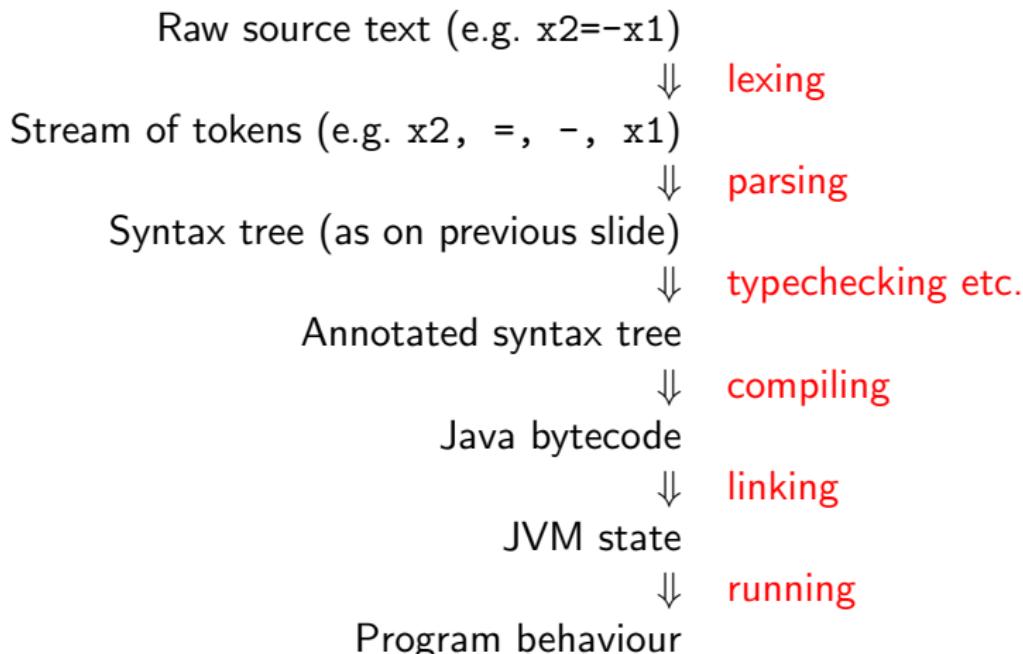
In both FLs and NLs, phrases have **structure** that can be represented via **syntax trees**.



Determining the structure of a phrase is an important first step towards doing other things with it. Much of this course will be about **describing** and **computing** syntax trees for phrases of some given language.

# The language processing 'pipeline' (FL version)

Think about the phases in which a Java program is processed:



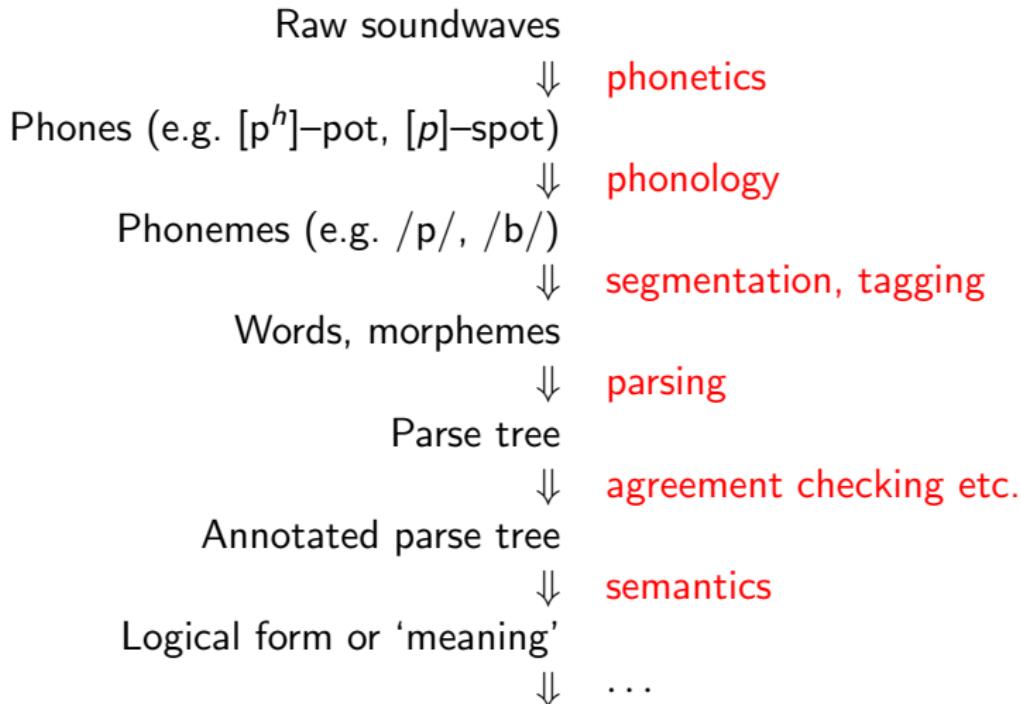
# Language processing for programming languages

In the case of programming languages, the pipeline typically works in a very ‘pure’ way: each phase depends only on the output from the previous phase.

- In this course, we'll be concentrating mainly on the first half of this pipeline: **lexing, parsing, typechecking**. (Especially parsing).
- We'll be looking both at the **theoretical concepts** involved (e.g. what is a syntax tree?)
- And at **algorithms** for the various phases (e.g. how do we construct the syntax tree for a given program)?
- We won't say much about techniques for compilation etc.
- However, we'll briefly touch on how the intended runtime behaviour of programs (i.e. their **semantics**) may be specified.

# The language processing ‘pipeline’ (NL version)

A broadly similar pipeline may be considered e.g. for English:



# Comparison between FLs and NLs

There are close relationships between these two pipelines. However, there are also important differences:

- FLs can be pinned down by a precise definition. NLs are fluid, fuzzy at the edges, and constantly evolving.  
(Oxford Dictionaries Word of the Year 2013: **selfie**. 2014: **vape**.)
- NLs are riddled with **ambiguity** at all levels. This is normally avoidable in FLs.
- For FLs the pipeline is typically ‘pure’. In NLs, information from later stages is sometimes used to resolve ambiguities at earlier stages, e.g.

*Time flies like an arrow.*

*Fruit flies like a banana.*

# Kinds of ambiguity in NL

- **Phonological** ambiguity: e.g. 'an ice lolly' vs. 'a nice lolly'.
- **Lexical** ambiguity: e.g. 'fast' has many senses (as noun, verb, adjective, adverb).
- **Syntactic** ambiguity: e.g. two possible syntax trees for 'complaints about referees multiplying'.
- **Semantic** ambiguity: e.g. 'Please use all available doors when boarding the train'.

## More on the NL pipeline

In the case of natural languages, one could in principle think of the pipeline ...

- either as a model for how an **artificial** speech processing system might be structured,
- or as a proposed (crude) model for what **naturally** goes on in human minds.

In this course, we mostly emphasize the former perspective.

Also, in the NL setting, it's equally sensible to think of running the pipeline backwards: starting with a logical form or 'meaning' and generating a speech utterance to express it. But we won't say much about this in this course.

## Recommended reading

The following textbook is highly recommended for this course and many other Natural Language courses in later years:

- D. Jurafsky and J. Martin, **Speech and Language Processing (2nd edition)**, Prentice-Hall, 2009.

For the formal language side, suitable texts include:

- D. Kozen, **Automata and Computability**, Springer, 2000.
- M. Sipser, **Introduction to the Theory of Computation (3rd edition)**, Cengage Learning, 2012.

Lectures will stick closely to the terminology and notation of the Jurafsky & Martin and Kozen texts.

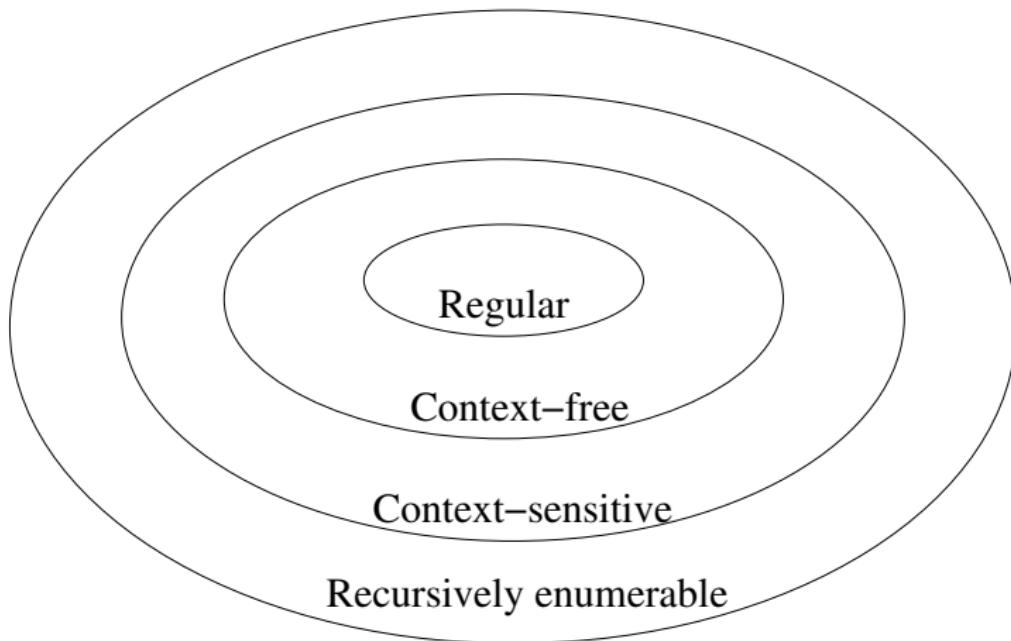
# Levels of language complexity

Some languages / language features are '**more complex**' (harder to describe, harder to process) than others. In fact, we can classify languages on a scale of complexity (the **Chomsky hierarchy**):

- **Regular** languages: those whose phrases can be 'recognized' by a finite state machine (cf. Informatics 1).
- **Context-free** languages. The basic structure of most programming languages, and many aspects of natural languages, can be described at this level.
- **Context-sensitive** languages. Some NLs involve features of this level of complexity.
- **Recursively enumerable** languages: *all* languages that can in principle be defined via mechanical rules.

Roughly speaking, we'll start with regular languages and work our way up the hierarchy. **Context-free** languages get most attention.

# The Chomsky Hierarchy (picture)



# Formal Language component: overview

## Regular languages:

- Definition using finite state machines (as in Inf1A).
- Equivalence of deterministic FSMs, non-deterministic FSMs, regular expressions.
- Applications: pattern matching, lexing, morphology.
- The **pumping lemma**: proving a given language *isn't* regular.

## Context-free languages:

- Context-free grammars, syntax trees.
- The corresponding machines: **pushdown automata**.
- **Parsing**: constructing the syntax tree for a given phrase.
- A parsing algorithm for **LL(1)** languages, in detail.

## Formal Language component: overview (continued)

After a break to cover some NL material, we'll glance briefly at some concepts from further down the pipeline: e.g. **typechecking** and **semantics** for programming languages.

Then we continue up the Chomsky hierarchy:

### Context-sensitive languages:

- Definition, examples.
- Relationship to **linear bounded automata**.

### Recursively enumerable languages:

- **Turing machines**; theoretical limits of what's 'computable in principle'.
- Undecidable problems.

## Natural language component: overview

We'll look at various parts of the NL processing pipeline, concentrating especially on **part-of-speech tagging** and **parsing**, with a little bit on **agreement checking** and **semantics**.

Our main focus is on how to get **computers** to perform these tasks, for applications such as

- speech synthesis
- machine translation
- text summarization and simplification
- (simple) NL dialogue systems.

But there'll also be a couple of lectures on scientific studies of how **we as humans** perform them.

# Natural language component: overview (continued)

Some specific topics:

- **Complexity of human languages:** E.g. whereabouts do human languages sit in the Chomsky hierarchy?
- **Parsing algorithms:** Because NLs differ from FLs in various ways, it turns out that different kinds of parsing algorithms are suitable.
- **Probabilistic versions of FL concepts:** In NL, because of ambiguity, we're typically looking for the **most likely** way of analysing a phrase. For this purpose, probabilistic analogues of e.g. finite state machines or context-free grammars are useful.
- **Use of text corpora:** Rather than building in all the relevant knowledge of the language by hand, we sometimes get a NLP system to 'learn' it for itself from some large sample of pre-existing text.

# Natural language semantics

Consider the sentence:

*Every student has access to a computer.*

The ‘meaning’ of this can be expressed by a logical formula:

$$\forall x. (\text{student}(x) \Rightarrow \exists y. (\text{computer}(y) \wedge \text{hasAccessTo}(x, y)))$$

Or perhaps:

$$\exists y. (\text{computer}(y) \wedge \forall x. (\text{student}(x) \Rightarrow \text{hasAccessTo}(x, y)))$$

**Problem:** how can (either of) these formulae be mechanically generated from a syntax tree for the original sentence? This is what **semantics** is all about.

# The Python programming language



- Invented by Guido van Rossum (pictured)
- Object-oriented programming language (like Java): has classes and objects.
- Dynamic typing (unlike Java). More flexibility but more chance of run-time errors.
- Clear and powerful syntax – very succinct (unlike Java). Especially convenient for [string processing](#).
- Typically driven [interactively](#) via a console session (like Haskell).
- Interfaces to many system calls, libraries, window systems, and other programming languages.

# Natural language processing with Python

## NLTK: Natural Language Toolkit

Developed by Steven Bird, Ewan Klein and Edward Loper; mainly addresses education and research; the book is online:

<http://www.nltk.org>

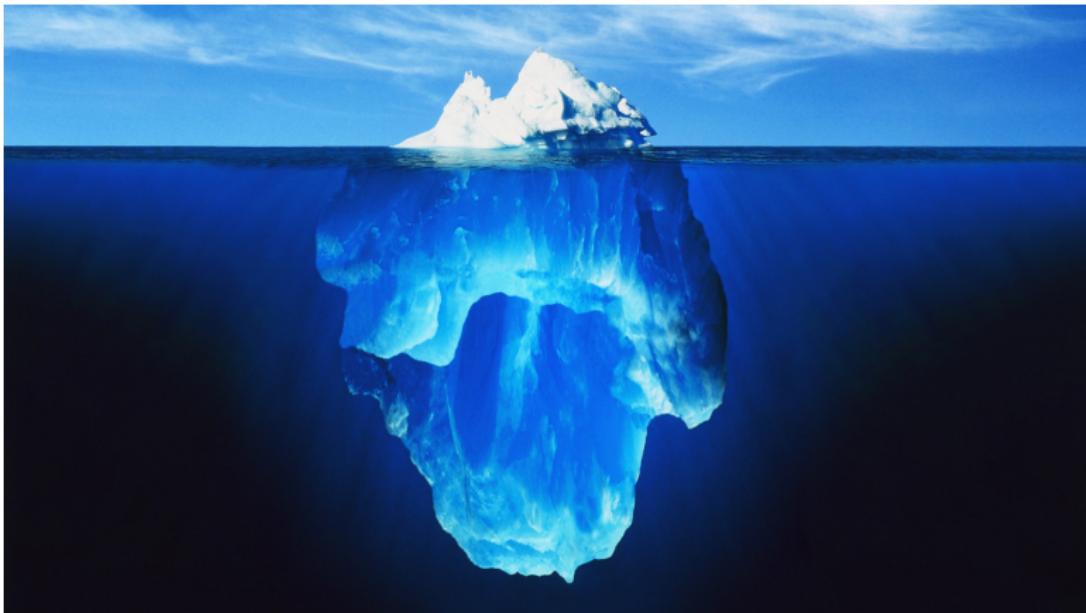
The NLTK provides support for many parts of the NL processing pipeline, e.g.

- Part-of-speech tagging
- Parsing
- Meaning extraction (semantics)

Lab sessions will introduce you to both Python and NLTK.

In Assignment 2, we'll show how one can fit these together to construct a (very simple) **natural language dialogue system**.

# Tip of the Iceberg



# Summary

- What is Inf2a about?
- We will learn about formal and natural languages.
- We will discuss their similarities and differences.
- We will cover finite state machines, context-free grammars, syntax trees, parsing, pos-tagging, ambiguity.
- We will use Python for natural language processing.
- We will have lots of fun!

**Next lecture:** Finite state machines (revision)

**Reading:** Kozen chapter 1, 2; J&M[2nd Ed] chapter 1

# Questions?

# Finite Automata

Informatics 2A: Lecture 3

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

25 September 2015

## 1 Languages and Automata

- What is a ‘language’?
- Finite automata: recap

## 2 Some formal definitions

- Finite automaton
- Regular language
- DFAs and NFAs

## 3 Determinization

- Execution of NFAs
- The subset construction

# Languages and alphabets

Throughout this course, languages will consist of finite sequences of symbols drawn from some given **alphabet**.

An **alphabet**  $\Sigma$  is simply some finite set of *letters* or *symbols* which we treat as 'primitive'. These might be ...

- English letters:  $\Sigma = \{a, b, \dots, z\}$
- Decimal digits:  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ASCII characters:  $\Sigma = \{0, 1, \dots, a, b, \dots, ?, !, \dots\}$
- Programming language 'tokens':  $\Sigma = \{\text{if}, \text{while}, \text{x}, ==, \dots\}$
- Words in (some fragment of) a natural language.
- 'Primitive' actions performable by a machine or system, e.g.  
 $\Sigma = \{\text{insert50p}, \text{pressButton1}, \dots\}$

In toy examples, we'll use simple alphabets like  $\{0, 1\}$  or  $\{a, b, c\}$ .

# What is a 'language'?

A language over an alphabet  $\Sigma$  will consist of finite sequences (**strings**) of elements of  $\Sigma$ . E.g. the following are strings over the alphabet  $\Sigma = \{a, b, c\}$ :

a      b      ab      cab      bacca      cccccccc

There's also the **empty string**, which we usually write as  $\epsilon$ .

A **language** over  $\Sigma$  is simply a (finite or infinite) set of strings over  $\Sigma$ . A string  $s$  is **legal** in the language  $L$  if and only if  $s \in L$ .

We write  $\Sigma^*$  for the set of *all* possible strings over  $\Sigma$ . So a language  $L$  is simply a subset of  $\Sigma^*$ . ( $L \subseteq \Sigma^*$ )

(N.B. This is just a technical definition — any *real* language is obviously much more than this!)

# Ways to define a language

There are many ways in which we might formally define a language:

- Direct mathematical definition, e.g.

$$L_1 = \{a, aa, ab, abbc\}$$

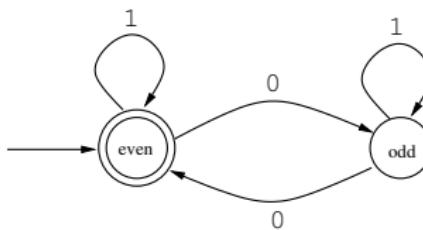
$$L_2 = \{axb \mid x \in \Sigma^*\}$$

$$L_3 = \{a^n b^n \mid n \geq 0\}$$

- Regular expressions (see Lecture 5).
- Formal grammars (see Lecture 8 onwards).
- Specify some **machine** for testing whether a string is legal or not.

The more complex the language, the more complex the machine might need to be. As we shall see, each level in the **Chomsky hierarchy** is correlated with a certain class of machines.

# Finite automata (a.k.a. finite state machines)



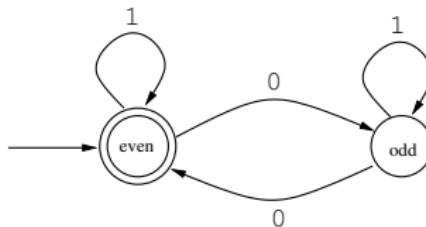
This is an example of a **finite automaton** over  $\Sigma = \{0, 1\}$ .

At any moment, the machine is in one of 2 **states**. From any state, each symbol in  $\Sigma$  determines a 'destination' state we can jump to.

The state marked with the in-arrow is picked out as the **starting state**. So any string in  $\Sigma^*$  gives rise to a sequence of states.

Certain states (with double circles) are designated as **accepting**. We call a string 'legal' if it takes us from the start state to some accepting state. In this way, the machine defines a language  $L \subseteq \Sigma^*$ : the language  $L$  is the **set of all legal strings**.

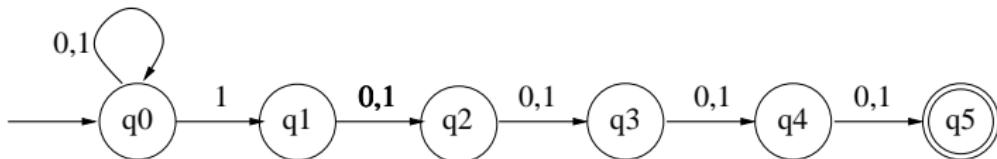
## Self-assessment question



For the finite state machine shown here, which of the following strings are legal (i.e. accepted)?

- ①  $\epsilon$
- ② 11
- ③ 1010
- ④ 1101

More generally, for any current state and any symbol, there may be **zero, one or many** new states we can jump to.



Here there are two transitions for '1' from  $q_0$ , and none from  $q_5$ .

The language associated with the machine is defined to consist of all strings that are accepted under **some** possible execution run.

The language associated with the example machine above is

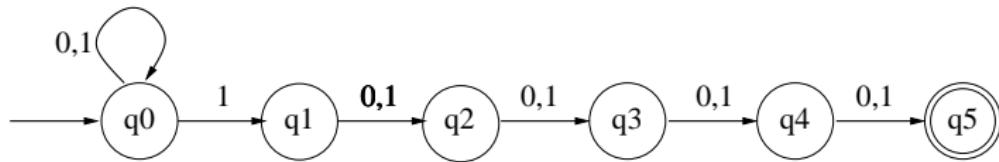
$$\{x \in \Sigma^* \mid \text{the fifth symbol from the end of } x \text{ is } 1\}$$

# Formal definition of finite automaton

Formally, a **finite automaton** with alphabet  $\Sigma$  consists of:

- A finite set  $Q$  of **states**,
- A **transition relation**  $\Delta \subseteq Q \times \Sigma \times Q$ ,
- A set  $S \subseteq Q$  of possible **starting states**.
- A set  $F \subseteq Q$  of **accepting states**.

## Example formal definition



$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$\Delta = \{ (q_0, 0, q_0), (q_0, 1, q_0), (q_0, 1, q_1), (q_1, 0, q_2), \\ (q_1, 1, q_2), (q_2, 0, q_3), (q_2, 1, q_3), (q_3, 0, q_4), \\ (q_3, 1, q_4), (q_4, 0, q_5), (q_4, 1, q_5) \}$$

$$S = \{q_0\}$$

$$F = \{q_5\}$$

# Regular language

Suppose  $M = (Q, \Delta, S, F)$  is a finite automaton with alphabet  $\Sigma$ .

We say that a string  $x \in \Sigma^*$  is **accepted** if there exists a path through the set of states  $Q$ , starting at some state  $s \in S$ , ending at some state  $f \in F$ , with each step taken from the  $\Delta$  relation, and with the path as a whole spelling out the string  $x$ .

This enables us to define the **language accepted by  $M$** :

$$\mathcal{L}(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}$$

We call a language  $L \subseteq \Sigma^*$  **regular** if  $L = \mathcal{L}(M)$  for **some** finite automaton  $M$ .

Regular languages are the subject of lectures 4–7 of the course.

# DFAs and NFAs

A finite automaton with alphabet  $\Sigma$  is **deterministic** if:

- It has exactly one starting state.
- For every state  $q \in Q$  and symbol  $a \in \Sigma$  there is exactly one state  $q'$  for which there exists a transition  $q \xrightarrow{a} q'$  in  $\Delta$ .

The first condition says that  $S$  is a **singleton** set.

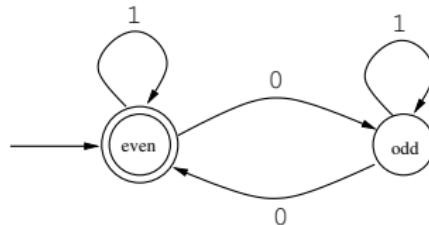
The second condition says that  $\Delta$  specifies a **function**  $Q \times \Sigma \rightarrow Q$ .

Deterministic finite automata are usually abbreviated **DFAs**.

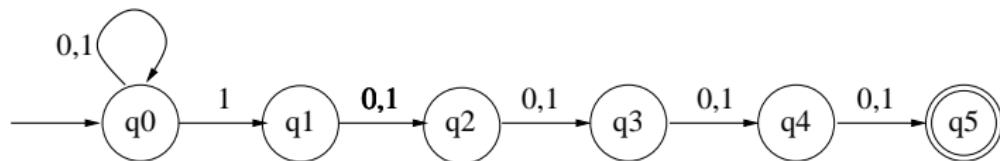
General finite automata are usually called **nondeterministic**, by way of contrast, and abbreviated **NFAs**.

Note that every DFA is an NFA.

# Example



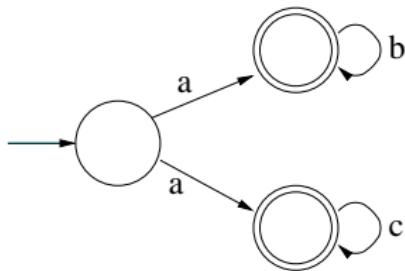
This is a DFA (and hence an NFA).



This is an NFA but not a DFA.

## Challenge question

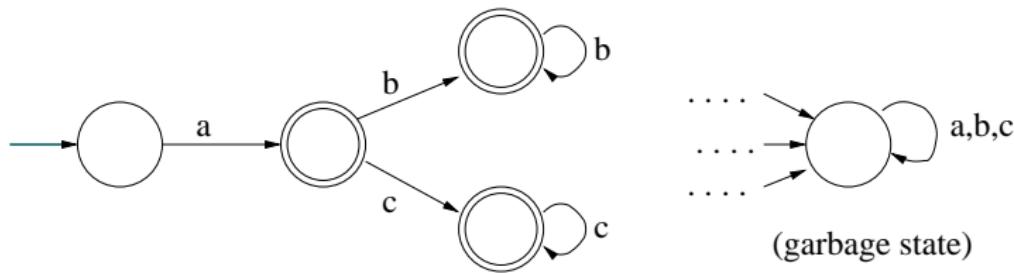
Consider the following NFA over  $\{a, b, c\}$ :



What is the *minimum* number of states of an equivalent DFA?

# Solution

An equivalent DFA must have at least **5 states!**



# Specifying a DFA

Clearly, a **DFA** with alphabet  $\Sigma$  can equivalently be given by:

- A finite set  $Q$  of states,
- A transition **function**  $\delta : Q \times \Sigma \rightarrow Q$ ,
- A **single designated** starting state  $s \in Q$ ,
- A set  $F \subseteq Q$  of accepting states.

Example:

$$Q = \{\text{even, odd}\}$$

$\delta$		0	1
$s$	even	odd	even
	odd	even	odd

$$s = \text{even}$$

$$F = \{\text{even}\}$$

# Running a finite automaton

DFAs are dead easy to implement and efficient to run. We don't need much more than a two-dimensional array for the transition function  $\delta$ . Given an input string  $x$  it is easy to follow the unique path determined by  $x$  and so determine whether or not the DFA accepts  $x$ .

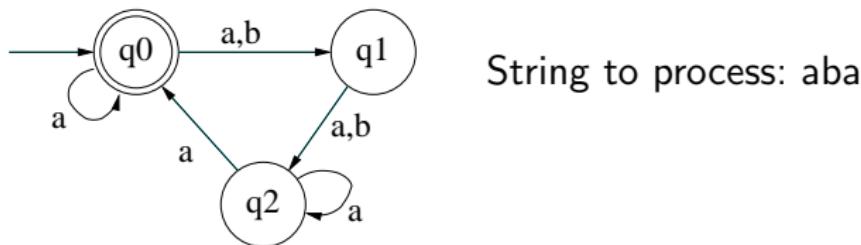
It is by no means so obvious how to run an NFA over an input string  $x$ . How do we prevent ourselves from making incorrect nondeterministic choices?

**Solution:** At each stage in processing the string, keep track of **all** the states the machine **might possibly** be in.

# Executing an NFA: example

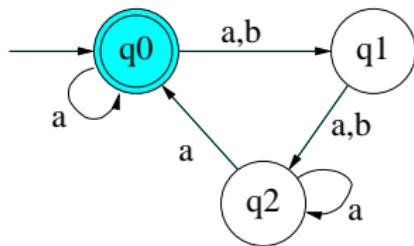
Given an NFA  $N$  over  $\Sigma$  and a string  $x \in \Sigma^*$ , how can we *in practice* decide whether  $x \in \mathcal{L}(N)$ ?

We illustrate with the running example below.



## Stage 0: initial state

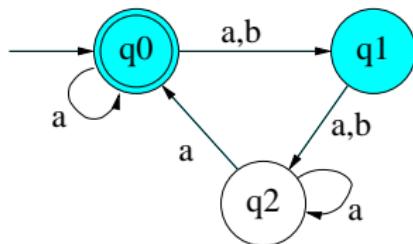
At the start, the NFA *can only be* in the initial state  $q_0$ .



String to process: aba  
Processed so far:  $\epsilon$   
Next symbol: a

## Stage 1: after processing 'a'

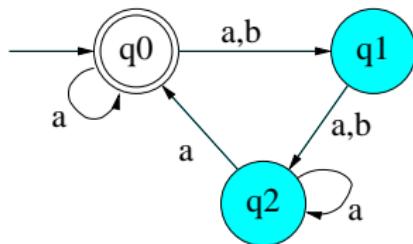
The NFA could now be in either  $q_0$  or  $q_1$ .



String to process: aba  
Processed so far: a  
Next symbol: b

## Stage 2: after processing 'ab'

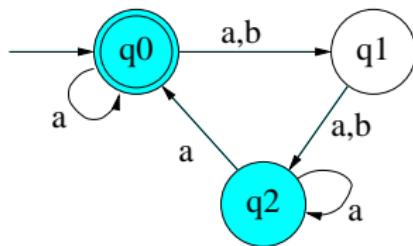
The NFA could now be in either  $q_1$  or  $q_2$ .



String to process: aba  
Processed so far: ab  
Next symbol: a

## Stage 3: final state

The NFA could now be in  $q_2$  or  $q_0$ . (It could have got to  $q_2$  in two different ways, though we don't need to keep track of this.)



String to process: aba  
Processed so far: aba

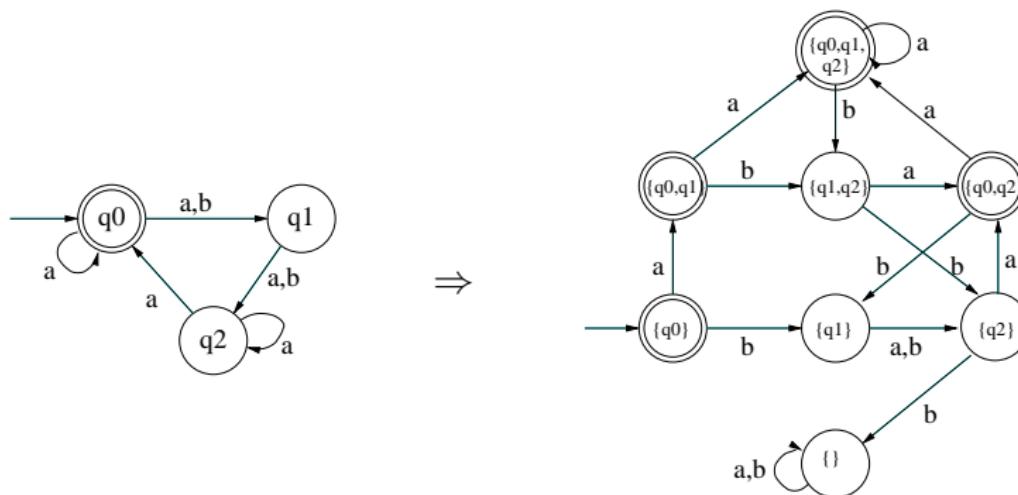
Since we've reached the end of the input string, and the set of possible states includes the accepting state  $q_0$ , we can say that the string aba is accepted by this NFA.

# The key insight

- The process we've just described is a completely **deterministic** process! Given any current set of 'coloured' states, and any input symbol in  $\Sigma$ , there's only one right answer to the question: 'What should the new set of coloured states be?'
- What's more, it's a **finite state** process. A 'state' is simply a choice of 'coloured' states in the original NFA  $N$ . If  $N$  has  $n$  states, there are  $2^n$  such choices.
- This suggests how an NFA with  $n$  states can be converted into an equivalent DFA with  $2^n$  states.

# The subset construction: example

Our 3-state NFA gives rise to a DFA with  $2^3 = 8$  states. The states of this DFA are **subsets** of  $\{q_0, q_1, q_2\}$ .



The accepting states of this DFA are exactly those that *contain* an accepting state of the original NFA.

# The subset construction in general

Given an NFA  $N = (Q, \Delta, S, F)$ , we can define an equivalent DFA  $M = (Q', \delta', s', F')$  (over the same alphabet  $\Sigma$ ) like this:

- $Q'$  is  $2^Q$ , the set of all subsets of  $Q$ . (Also written  $\mathcal{P}(Q)$ .)
- $\delta'(A, u) = \{q' \in Q \mid \exists q \in A. (q, u, q') \in \Delta\}$ . (Set of all states reachable via  $u$  from *some* state in  $A$ .)
- $s' = S$ .
- $F' = \{A \subseteq Q \mid \exists q \in A. q \in F\}$ .

It's then not hard to prove mathematically that  $\mathcal{L}(M) = \mathcal{L}(N)$ .  
(See Kozen for details.)

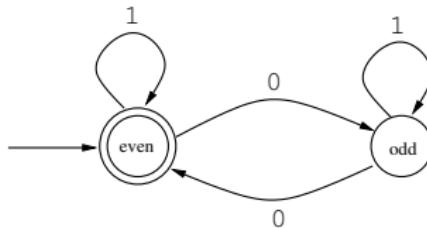
This process is called **determinization**.

# Summary

- We've shown that for any NFA  $N$ , we can construct a DFA  $M$  with the same associated language.
- Since every DFA is also an NFA, the classes of languages recognised by DFAs and by NFAs coincide — these are the **regular languages**.
- Often a language can be specified more concisely by an NFA than by a DFA.
- We can automatically convert an NFA to a DFA, at the risk of an exponential blow-up in the number of states.
- To determine whether a string  $x$  is accepted by an NFA we do not need to construct the entire DFA, but instead we efficiently simulate the execution of the DFA on  $x$  on a step-by-step basis. (This is called **just-in-time** simulation.)

## End-of-lecture question 1

Let  $M$  be the DFA shown earlier:



Give a concise mathematical definition of the language  $\mathcal{L}(M)$ .

Answer:

$$\mathcal{L}(M) = \{x \in \Sigma^* \mid x \text{ contains an even number of 0's}\}$$

## End-of-lecture question 2

Which of these three languages are regular?

$$L_1 = \{a, aa, ab, abbc\}$$

$$L_2 = \{axb \mid x \in \Sigma^*\}$$

$$L_3 = \{a^n b^n \mid n \geq 0\}$$

If regular, can you design an NFA that shows this? What about a DFA?

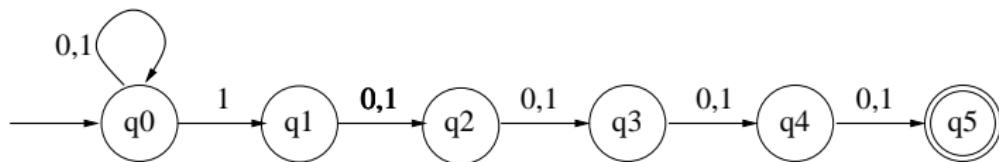
If not regular, can you explain why not?

Answer:  $L_1$  and  $L_2$  are regular. (NFAs and DFAs left as exercises.)

$L_3$  is not regular. We shall see why in lecture 7.

## End-of-lecture challenge question 3

Consider our first example NFA over  $\{0, 1\}$ :



What is the number of states of the smallest DFA that recognises the same language?

Answer given in Lecture 4.

# Reference material

- Kozen chapters 3, 5 and 6.
- J & M section 2.2 (rather brief).

# Constructions on Finite Automata

Informatics 2A: Lecture 4

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

29 September 2015

1 Closure properties of regular languages

- Union
- Intersection
- Complement

2 DFA minimization

- The problem
- An algorithm for minimization

## Recap of Lecture 3

- A language is a set of strings over an alphabet  $\Sigma$ .
- A language is called **regular** if it is recognised by some NFA.
- DFAs are an important subclass of NFAs.
- An NFA with  $n$  states can be **determinized** to an equivalent DFA with  $2^n$  states, using the **subset construction**.
- Therefore the regular languages are exactly the languages recognised by DFAs.

# Union of regular languages

Consider the following little theorem:

*If  $L_1$  and  $L_2$  are regular languages over  $\Sigma$ , so is  $L_1 \cup L_2$ .*

This is **dead easy** to prove using NFAs.

Suppose  $N_1 = (Q_1, \Delta_1, S_1, F_1)$  is an NFA for  $L_1$ , and  $N_2 = (Q_2, \Delta_2, S_2, F_2)$  is an NFA for  $L_2$ .

We may assume  $Q_1 \cap Q_2 = \emptyset$  (just relabel states if not).

Now consider the NFA

$$(Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, S_1 \cup S_2, F_1 \cup F_2)$$

This is just  $N_1$  and  $N_2$  ‘side by side’. Clearly, this NFA recognizes precisely  $L_1 \cup L_2$ .

Number of states =  $|Q_1| + |Q_2|$  — no state explosion!

# Intersection of regular languages

If  $L_1$  and  $L_2$  are regular languages over  $\Sigma$ , so is  $L_1 \cap L_2$ .

Suppose  $N_1 = (Q_1, \Delta_1, S_1, F_1)$  is an NFA for  $L_1$ , and  $N_2 = (Q_2, \Delta_2, S_2, F_2)$  is an NFA for  $L_2$ .

We define a **product** NFA  $(Q', \Delta', S', F')$  by:

$$Q' = Q_1 \times Q_2$$

$$(q, r) \xrightarrow{a} (q', r') \in \Delta' \iff q \xrightarrow{a} q' \in \Delta_1 \text{ and } r \xrightarrow{a} r' \in \Delta_2$$

$$S' = S_1 \times S_2$$

$$F' = F_1 \times F_2$$

Number of states =  $|Q_1| \times |Q_2|$  — a bit more costly than union!

If  $N_1$  and  $N_2$  are DFAs then the product automaton is a DFA too.

# Complement of a regular language

( Recall the set-difference operation,

$$A - B = \{x \in A \mid x \notin B\}$$

where  $A, B$  are sets. )

If  $L$  is a regular language over  $\Sigma$ , then so is  $\Sigma^* - L$ .

Suppose  $N = (Q, \delta, s, F)$  is a DFA for  $L$ .

Then  $(Q, \delta, s, Q - F)$  is a DFA for  $\Sigma^* - L$ . (We simply swap the accepting and rejecting states in  $N$ .)

Number of states =  $|Q|$  — no blow up at all, but we are required to start with a DFA. This in itself has size implications.

The complement construction does not work if  $N$  is not deterministic!

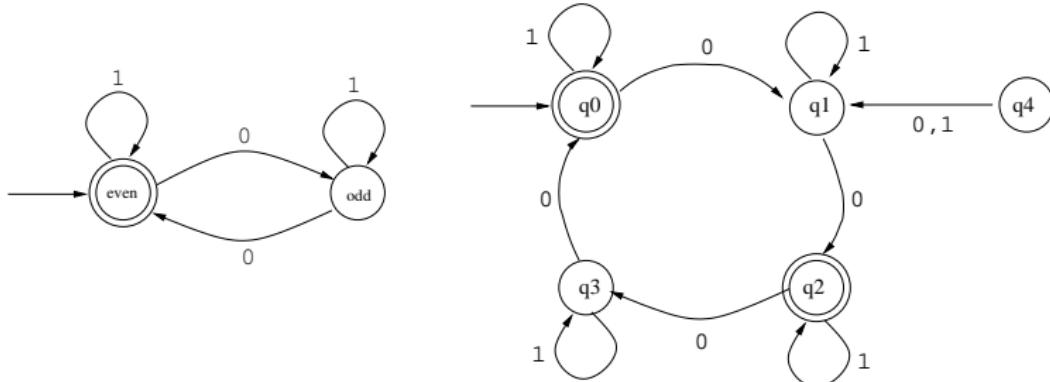
# Closure properties of regular languages

- We've seen that if both  $L_1$  and  $L_2$  are regular languages, then so are:
  - $L_1 \cup L_2$  (union)
  - $L_1 \cap L_2$  (intersection)
  - $\Sigma^* - L_1$  (complement)
- We sometimes express this by saying that regular languages are **closed under** the operations of union, intersection and complementation. ('Closed' used here in the sense of 'self-contained'.)
- Each closure property corresponds to an explicit construction on finite automata. Sometimes this uses NFAs (union), sometimes DFAs (complement), and sometimes the construction works equally well for both NFAs and DFAs (intersection).

# The Minimization Problem

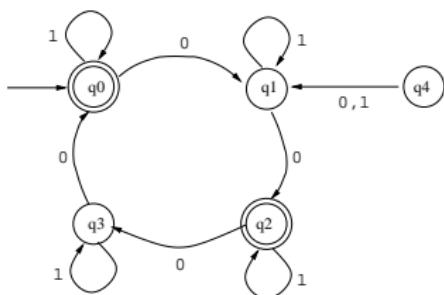
Determinization involves an exponential blow-up in the automaton.  
Is it sometimes possible to reduce the size of the resulting DFA?

Many different DFAs can give rise to the same language, e.g.:



We shall see that there is always a **unique smallest** DFA for a given regular language.

## DFA minimization



We perform the following steps to ‘reduce’  $M$  above:

- Throw away **unreachable** states (in this case,  $q_4$ ).
- Squish together **equivalent** states, i.e. states  $q, q'$  such that: every string accepted starting from  $q$  is accepted starting from  $q'$ , and vice versa. (In this case,  $q_0$  and  $q_2$  are equivalent, as are  $q_1$  and  $q_3$ .)

Let’s write  $\text{Min}(M)$  for the resulting reduced DFA. In this case,  $\text{Min}(M)$  is essentially the two-state machine on the previous slide.

# Properties of minimization

The minimization operation on DFAs enjoys the following properties which characterise the construction:

- $\mathcal{L}(\text{Min}(M)) = \mathcal{L}(M)$ .
- If  $\mathcal{L}(M') = \mathcal{L}(M)$  and  $|M'| \leq |\text{Min}(M)|$  then  $M' \cong \text{Min}(M)$ .

Here  $|M|$  is the number of states of the DFA  $M$ , and  $\cong$  means the two DFAs are **isomorphic**: that is, identical apart from a possible renaming of states.

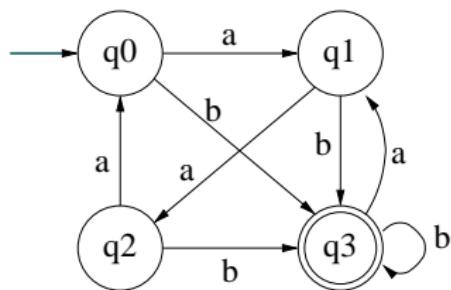
Two consequences of the above are:

- $\text{Min}(M) \cong \text{Min}(M')$  if and only if  $\mathcal{L}(M) = \mathcal{L}(M')$ .
- $\text{Min}(\text{Min}(M)) \cong \text{Min}(M)$ .

For a formal treatment of minimization, see Kozen chapters 13–16.

## Challenge question

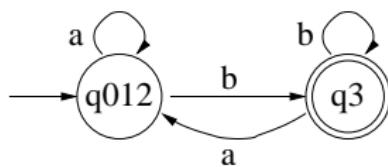
Consider the following DFA over  $\{a, b\}$ .



How many states does the minimized DFA have?

# Solution

The minimized DFA has just **2 states**:



The minimized DFA has been obtained by squishing together states  $q_0$ ,  $q_1$  and  $q_2$ . Clearly  $q_3$  must be kept distinct.

Note that the corresponding language consists of **all strings ending with b**.

## Minimization in practice

Let's look again at our definition of **equivalent states**:

*states  $q, q'$  such that: every string accepted starting from  $q$  is accepted starting from  $q'$ , and vice versa.*

This is fine as an abstract **mathematical** definition of equivalence, but it doesn't seem to give us a way to **compute** which states are equivalent: we'd have to 'check' infinitely many strings  $x \in \Sigma^*$ .

Fortunately, there's an actual **algorithm** for DFA minimization that works in reasonable time.

This is useful in practice: we can specify our DFA in the most convenient way without worrying about its size, then minimize to a more 'compact' DFA to be implemented e.g. in hardware.

# An algorithm for minimization

First eliminate any unreachable states (easy).

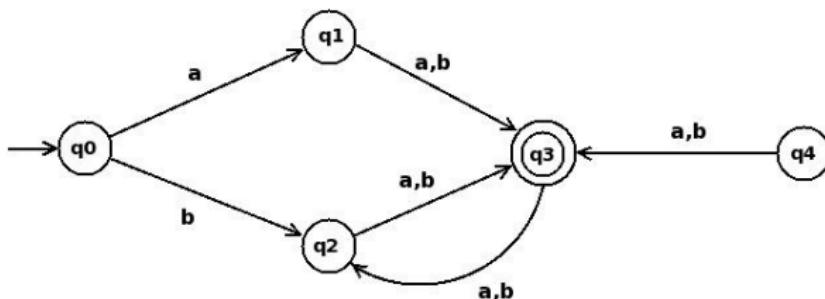
Then create a table of all possible pairs of states  $(p, q)$ , initially **unmarked**. (E.g. a two-dimensional array of booleans, initially set to false.) We **mark** pairs  $(p, q)$  as and when we discover that  $p$  and  $q$  **cannot** be equivalent.

- ① Start by marking all pairs  $(p, q)$  where  $p \in F$  and  $q \notin F$ , or vice versa.
- ② Look for unmarked pairs  $(p, q)$  such that for some  $u \in \Sigma$ , the pair  $(\delta(p, u), \delta(q, u))$  is marked. Then mark  $(p, q)$ .
- ③ Repeat step 2 until no such unmarked pairs remain.

If  $(p, q)$  is still unmarked, can collapse  $p, q$  to a single state.

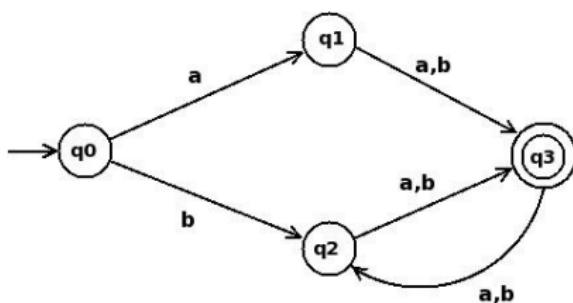
## Illustration of minimization algorithm

Consider the following DFA over  $\{a, b\}$ .



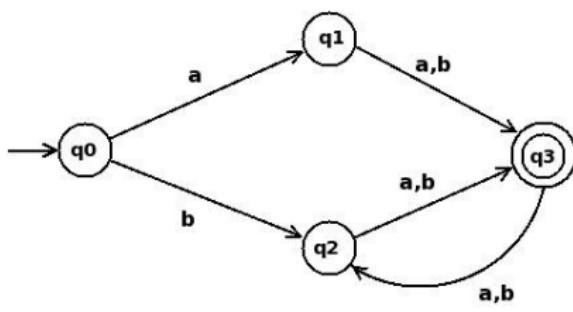
## Illustration of minimization algorithm

After eliminating unreachable states:



## Illustration of minimization algorithm

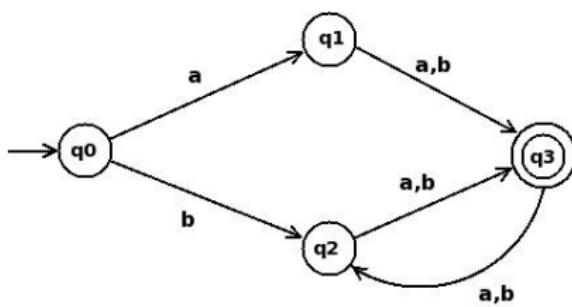
We mark states to be kept distinct using a half matrix:



q0				
q1	.			
q2	.	.		
q3	.	.	.	
	q0	q1	q2	q3

## Illustration of minimization algorithm

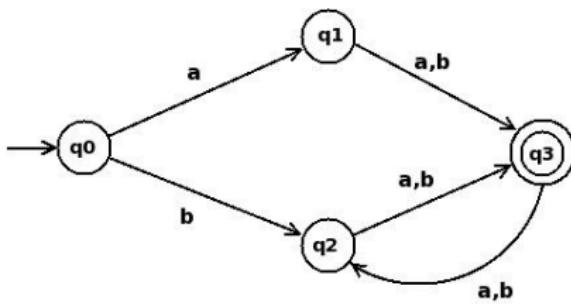
First mark accepting/non-accepting pairs:



q0	.
q1	.
q2	.
q3	✓
q0	q0
q1	q1
q2	q2
q3	q3

## Illustration of minimization algorithm

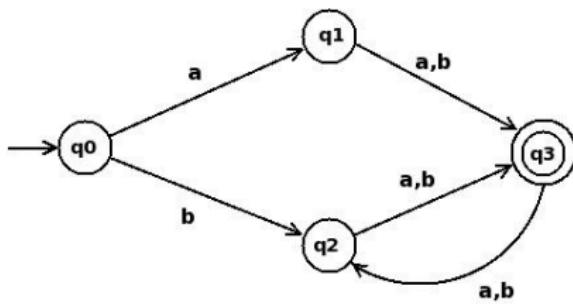
$(q_0, q_1)$  is unmarked,  $q_0 \xrightarrow{a} q_1$ ,  $q_1 \xrightarrow{a} q_3$ , and  $(q_1, q_3)$  is marked.



$q_0$	
$q_1$	.
$q_2$	.
$q_3$	✓
$q_0$	q0
$q_1$	q1
$q_2$	q2
$q_3$	q3

## Illustration of minimization algorithm

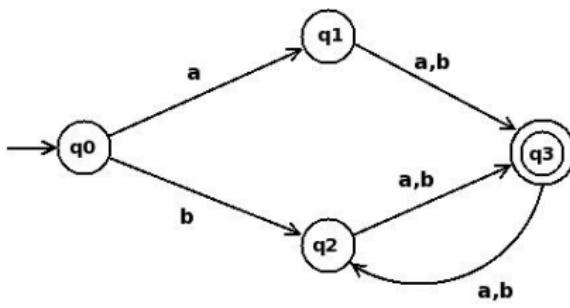
So mark (q0,q1).



q0	✓
q1	.
q2	.
q3	✓
q0	q0
q1	q1
q2	q2
q3	q3

## Illustration of minimization algorithm

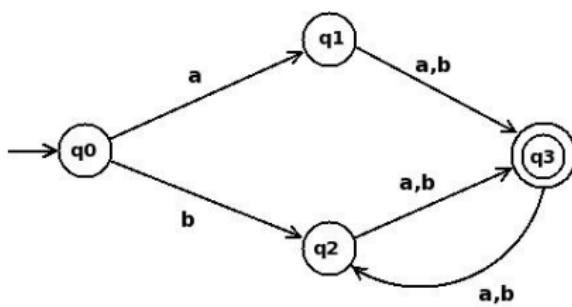
$(q_0, q_2)$  is unmarked,  $q_0 \xrightarrow{a} q_1$ ,  $q_2 \xrightarrow{a} q_3$ , and  $(q_1, q_3)$  is marked.



$q_0$	
$q_1$	✓
$q_2$	.
$q_3$	✓
$q_0$	q0
$q_1$	q1
$q_2$	q2
$q_3$	q3

## Illustration of minimization algorithm

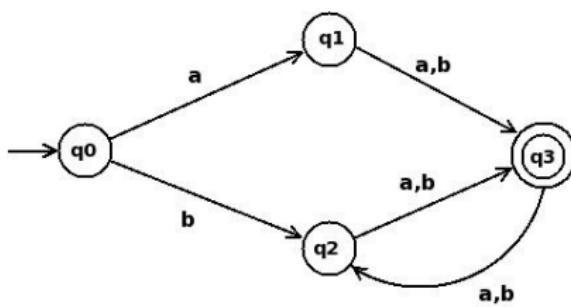
So mark (q0,q2).



q0	
q1	✓
q2	✓ .
q3	✓ ✓ ✓
	q0 q1 q2 q3

## Illustration of minimization algorithm

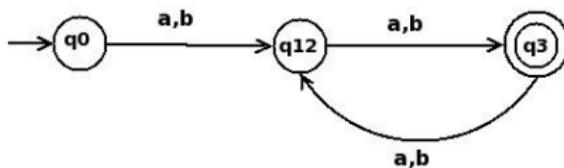
The only remaining unmarked pair  $(q_1, q_2)$  stays unmarked.



q0	
q1	✓
q2	✓ .
q3	✓ ✓ ✓
	q0 q1 q2 q3

# Illustration of minimization algorithm

So obtain mimized DFA by collapsing q1, q2 to a single state.



# Improving determinization

Now we have a minimization algorithm, the following improved determinization procedure is possible.

To determinize an NFA  $M$  with  $n$  states:

- ① Perform the subset construction on  $M$  to produce an equivalent DFA  $N$  with  $2^n$  states.
- ② Perform the minimization algorithm on  $N$  to produce a DFA  $\text{Min}(N)$  with  $\leq 2^n$  states.

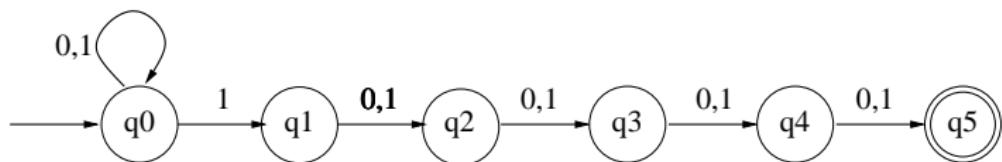
Using this method we are guaranteed to produce the smallest possible DFA equivalent to  $M$ .

In many cases this avoids the exponential state-space blow-up.

In some cases, however, an exponential blow-up is unavoidable.

## End-of-last-lecture challenge question

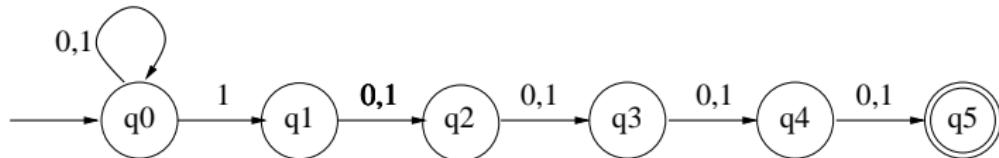
Consider last lecture's example NFA over  $\{0, 1\}$ :



What is the number of states of the smallest DFA that recognises the same language?

# End-of-last-lecture challenge question

Consider last lecture's example NFA over  $\{0, 1\}$ :

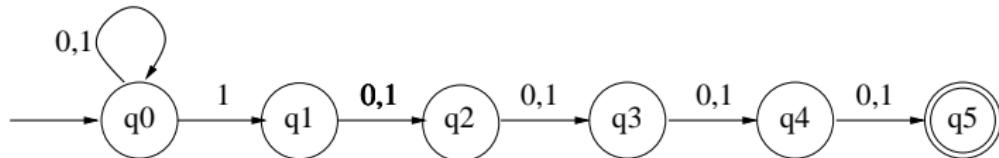


What is the number of states of the smallest DFA that recognises the same language?

**Answer:** The smallest DFA has 32 states.

## End-of-last-lecture challenge question

Consider last lecture's example NFA over  $\{0, 1\}$ :



What is the number of states of the smallest DFA that recognises the same language?

**Answer:** The smallest DFA has 32 states.

More generally, the smallest DFA for the language:

$$\{x \in \Sigma^* \mid \text{the } n\text{-th symbol from the end of } x \text{ is 1}\}$$

has  $2^n$  states. Whereas, there is an NFA with  $n + 1$  states.

## End-of-lecture questions

- ① Show that the construction of a complement automaton (used to prove that regular languages are closed under complementation) does not work for nondeterministic finite automata.

Specifically, find an example of an NFA  $M = (Q, \Delta, S, F)$  for which the NFA  $(Q, \Delta, S, Q - F)$ , obtained by swapping accepting and non-accepting states, does not recognize  $\Sigma^* - \mathcal{L}(M)$ .

**Answer:** For example, over the alphabet  $\Sigma = \{a\}$  consider

$$(\{q_0, q_1\}, \{(q_0, a, q_0), (q_1, a, q_1)\}, \{q_0, q_1\}, \{q_0\})$$

Here  $\{q_0, q_1\}$  is the set of start states, and  $\{q_0\}$  is the set of accepting states. This recognises the full language  $\Sigma^*$ . If accepting and non-accepting states are swapped, the new set of accepting states is  $\{q_1\}$ , and the language recognised is still  $\Sigma^*$ .

## End-of-lecture questions

- ② Suppose  $N_1 = (Q_1, \Delta_1, S_1, F_1)$  is an DFA for  $L_1$ , and  $N_2 = (Q_2, \Delta_2, S_2, F_2)$  is an DFA for  $L_2$ .  
Give a direct construction of a DFA  $N$  such that  $\mathcal{L}(N) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$ .

**Answer:** Use the identity:

$$L_1 \cup L_2 = \Sigma^* - ((\Sigma^* - L_1) \cap (\Sigma^* - L_2))$$

to reduce the construction to a composite of complementation and intersection constructions.

More directly (but equivalently), simply modify the product automaton construction on slide 5 with

$$(q, r) \xrightarrow{a} (q', r') \in \Delta' \iff q \xrightarrow{a} q' \in \Delta_1 \text{ or } r \xrightarrow{a} r' \in \Delta_2$$

# Reading

## Relevant reading:

- Closure properties of regular languages: Kozen chapter 4.
- Minimization: Kozen chapters 13–14.

## Next time:

- Regular expressions and Kleene's Theorem.  
(Kozen chapters 7–9.)

# Regular expressions and Kleene's theorem

## Informatics 2A: Lecture 5

John Longley

School of Informatics  
University of Edinburgh  
[als@inf.ed.ac.uk](mailto:als@inf.ed.ac.uk)

1 October 2015

## 1 More closure properties of regular languages

- Operations on languages
- $\epsilon$ -NFAs
- Closure under concatenation and Kleene star

## 2 Regular expressions

- Regular expressions
- From regular expressions to regular languages

## 3 Kleene's theorem and Kleene algebra

- Kleene's theorem
- Kleene algebra
- From DFAs to regular expressions

## Recap of Lecture 4

- Regular languages are **closed under** union, intersection and complement.
- These closure properties are proved using explicit constructions on finite automata (sometimes using NFAs, sometimes DFAs).
- Every regular language has a unique **minimum** DFA that recognises it.
- An algorithm for **minimizing** a DFA.

# Concatenation

We write  $L_1.L_2$  for the **concatenation** of languages  $L_1$  and  $L_2$ , defined by:

$$L_1.L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

For example, if  $L_1 = \{aaa\}$  and  $L_2 = \{b, c\}$  then  $L_1.L_2$  is the language  $\{aaab, aaac\}$ .

Later we will prove the following closure property.

*If  $L_1$  and  $L_2$  are regular languages then so is  $L_1.L_2$ .*

# Kleene star

We write  $L^*$  for the **Kleene star** of the language  $L$ , defined by:

$$L^* = \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \dots$$

For example, if  $L_3 = \{aaa, b\}$  then  $L_3^*$  contains strings like  $aaaaaa$ ,  $bbbbbb$ ,  $baaaaaabbaaaa$ , etc.

More precisely,  $L_3^*$  contains all strings over  $\{a, b\}$  in which the letter  $a$  always appears in sequences of length some multiple of 3

Later we will prove the following closure property.

*If  $L$  is a regular language then so is  $L^*$ .*

## Self-assessment question

Consider the language over the alphabet  $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language  $L.L$  ?

- ①  $abcabc$       Ans: yes
- ②  $acacac$       Ans: yes
- ③  $abcbcac$       Ans: yes
- ④  $abcbacbc$       Ans: no

## Self-assessment question

Consider the (same) language over the alphabet  $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language  $L^*$  ?

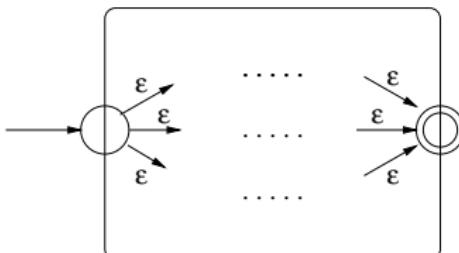
- ①  $\epsilon$       Ans: yes
- ②  $acaca$       Ans: no
- ③  $abcbc$       Ans: yes
- ④  $acacacacac$       Ans: yes

## NFAs with $\epsilon$ -transitions

We can vary the definition of NFA by also allowing transitions labelled with the special symbol  $\epsilon$  (*not* a symbol in  $\Sigma$ ).

The automaton may (but doesn't have to) perform a spontaneous  $\epsilon$ -transition at any time, without reading an input symbol.

This is quite convenient: for instance, we can turn any NFA into an  $\epsilon$ -NFA with just **one start state** and **one accepting state**:



(Add  $\epsilon$ -transitions from new start state to each state in  $S$ , and from each state in  $F$  to new accepting state.)

# Equivalence to ordinary NFAs

Allowing  $\epsilon$ -transitions is just a convenience: it doesn't fundamentally change the power of NFAs.

If  $N = (Q, \Delta, S, F)$  is an  $\epsilon$ -NFA, we can convert  $N$  to an ordinary NFA with the same associated language, by simply 'expanding'  $\Delta$  and  $S$  to allow for silent  $\epsilon$ -transitions.

To achieve this, perform the following steps on  $N$ .

- For every pair of transitions  $q \xrightarrow{a} q'$  (where  $a \in \Sigma$ ) and  $q' \xrightarrow{\epsilon} q''$ , add a new transition  $q \xrightarrow{a} q''$ .
- For every transition  $q \xrightarrow{\epsilon} q'$ , where  $q$  is a start state, make  $q'$  a start state too.

Repeat the two steps above until no further new transitions or new start states can be added.

Finally, remove all  $\epsilon$ -transitions from the  $\epsilon$ -NFA resulting from the above process. This produces the desired NFA.

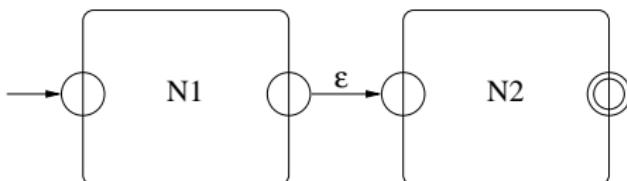
# Closure under concatenation

We use  $\epsilon$ -NFAs to show, as promised, that regular languages are closed under the **concatenation** operation:

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

If  $L_1, L_2$  are any regular languages, choose  $\epsilon$ -NFAs  $N_1, N_2$  that define them. As noted earlier, we can pick  $N_1$  and  $N_2$  to have just one start state and one accepting state.

Now hook up  $N_1$  and  $N_2$  like this:



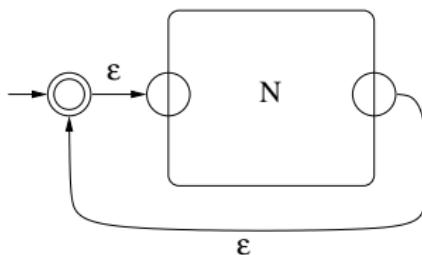
Clearly, this NFA corresponds to the language  $L_1 \cdot L_2$ .

# Closure under Kleene star

Similarly, we can now show that regular languages are closed under the **Kleene star** operation:

$$L^* = \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \dots$$

For suppose  $L$  is represented by an  $\epsilon$ -NFA  $N$  with one start state and one accepting state. Consider the following  $\epsilon$ -NFA:



Clearly, this  $\epsilon$ -NFA corresponds to the language  $L^*$ .

# Regular expressions

We've been looking at ways of specifying regular languages via machines (often presented as **pictures**). But it's very useful for applications to have more **textual** ways of defining languages.

A **regular expression** is a written mathematical expression that defines a language over a given alphabet  $\Sigma$ .

- The **basic** regular expressions are

$$\emptyset \qquad \epsilon \qquad a \text{ (for } a \in \Sigma\text{)}$$

- From these, more complicated regular expressions can be built up by (repeatedly) applying the two binary operations  $+$ ,  $.$  and the unary operation  $*$ . Example:  $(a.b + \epsilon)^* + a$

We use brackets to indicate precedence. In the absence of brackets,  $*$  binds more tightly than  $.$ , which itself binds more tightly than  $+$ .

So  $a + b.a^*$  means  $a + (b.(a^*))$

Also the dot is often omitted:  $ab$  means  $a.b$

# How do regular expressions define languages?

A regular expression is itself just a **written expression**. However, every regular expression  $\alpha$  over  $\Sigma$  can be seen as **defining** an actual **language**  $\mathcal{L}(\alpha) \subseteq \Sigma^*$  in the following way.

- $\mathcal{L}(\emptyset) = \emptyset, \quad \mathcal{L}(\epsilon) = \{\epsilon\}, \quad \mathcal{L}(a) = \{a\}.$
- $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha.\beta) = \mathcal{L}(\alpha) . \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

**Example:**  $a + ba^*$  defines the language  $\{a, b, ba, baa, baaa, \dots\}$ .

The languages defined by  $\emptyset, \epsilon, a$  are obviously **regular**.

What's more, we've seen that regular languages are **closed under** union, concatenation and Kleene star.

This means **every regular expression defines a regular language**.  
(Formal proof by induction on the size of the regular expression.)

## Self-assessment question

Consider (again) the language

$$\{x \in \{0, 1\}^* \mid x \text{ contains an even number of 0's}\}$$

Which of the following regular expressions define the above language?

- ①  $(1^*01^*01^*)^*$       **Ans:** no — 1 does not match expression
- ②  $(1^*01^*0)^*1^*$       **Ans:** yes
- ③  $1^*(01^*0)^*1^*$       **Ans:** no — 00100 does not match expression
- ④  $(1 + 01^*0)^*$       **Ans:** yes

# Kleene's theorem

We've seen that every regular expression defines a regular language.

The major goal of the lecture is to show the converse: **every regular language can be defined by a regular expression**. For this purpose, we introduce **Kleene algebra**: the algebra of regular expressions.

The equivalence between regular languages and expressions is:

## Kleene's theorem

*DFAs and regular expressions give rise to exactly the same class of languages (the regular languages).*

As we've already seen, NFAs (with or without  $\epsilon$ -transitions) also give rise to this class of languages.

So the evidence is mounting that the class of regular languages is mathematically a very 'natural' class to consider.

# Kleene algebra

Regular expressions give a **textual** way of specifying regular languages. This is useful e.g. for communicating regular languages to a computer.

Another benefit: regular expressions can be manipulated using algebraic laws (**Kleene algebra**). For example:

$$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$$

$$\alpha + \emptyset = \alpha$$

$$\alpha(\beta\gamma) = (\alpha\beta)\gamma$$

$$\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$$

$$\emptyset\alpha = \alpha\emptyset = \emptyset$$

$$\alpha + \beta = \beta + \alpha$$

$$\alpha + \alpha = \alpha$$

$$\epsilon\alpha = \alpha\epsilon = \alpha$$

$$(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$$

$$\epsilon + \alpha\alpha^* = \epsilon + \alpha^*\alpha = \alpha^*$$

Often these can be used to **simplify** regular expressions down to more pleasant ones.

# Other reasoning principles

Let's write  $\alpha \leq \beta$  to mean  $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$  (or equivalently  $\alpha + \beta = \beta$ ). Then

$$\begin{aligned}\alpha\gamma + \beta &\leq \gamma \Rightarrow \alpha^*\beta \leq \gamma \\ \beta + \gamma\alpha &\leq \gamma \Rightarrow \beta\alpha^* \leq \gamma\end{aligned}$$

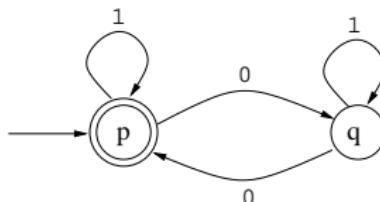
**Arden's rule:** Given an equation of the form  $X = \alpha X + \beta$ , its smallest solution is  $X = \alpha^*\beta$ .

What's more, if  $\epsilon \notin \mathcal{L}(\alpha)$ , this is the *only* solution.

**Beautiful fact:** The rules on this slide and the last form a **complete** set of reasoning principles, in the sense that if  $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$ , then ' $\alpha = \beta$ ' is provable using these rules. (Beyond scope of Inf2A.)

# DFAs to regular expressions

We use an example to show how to convert a DFA to an equivalent regular expression.



For each state  $r$ , let the variable  $X_r$  stand for the set of strings that take us from  $r$  to an accepting state. Then we can write some simultaneous equations:

$$\begin{aligned} X_p &= 1X_p + 0X_q + \epsilon \\ X_q &= 1X_q + 0X_p \end{aligned}$$

# Where do the equations come from?

Consider:

$$X_p = 1X_p + 0X_q + \epsilon$$

This asserts the following.

Any string that takes us from  $p$  to an accepting state is:

- a 1 followed by a string that takes us from  $p$  to an accepting state; or
- a 0 followed by a string that takes us from  $q$  to an accepting state; or
- the empty string.

Note that the empty string is included because  $p$  is an accepting state.

# Solving the equations

We solve the equations by eliminating one variable at a time:

$$X_q = 1^*0X_p \text{ by Arden's rule}$$

$$\begin{aligned} \text{So } X_p &= 1X_p + 01^*0X_p + \epsilon \\ &= (1 + 01^*0)X_p + \epsilon \end{aligned}$$

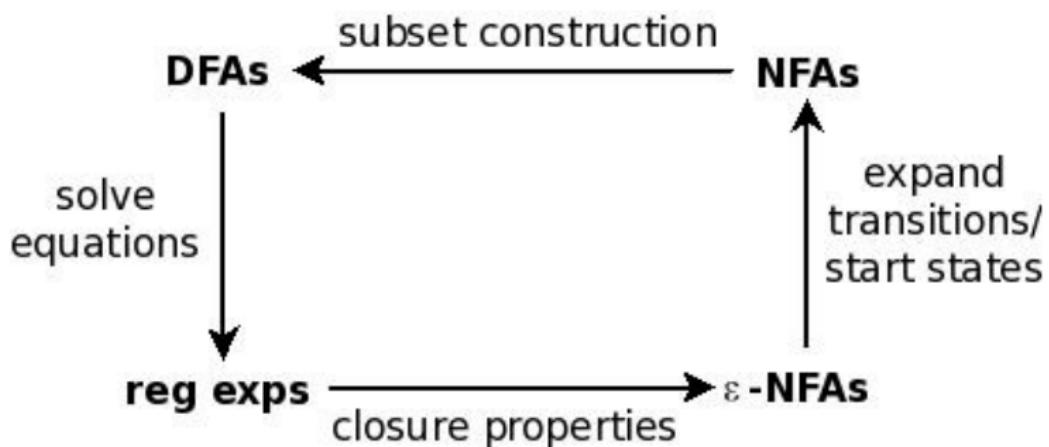
$$\text{So } X_p = (1 + 01^*0)^* \text{ by Arden's rule}$$

Since the start state is  $p$ , the resulting regular expression for  $X_p$  is the one we are seeking. Thus the language recognised by the automaton is:

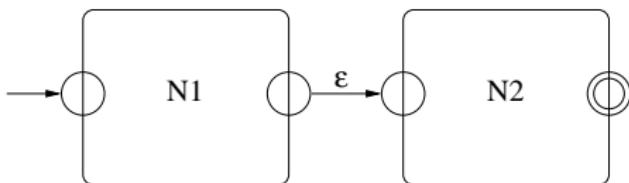
$$(1 + 01^*0)^*$$

The method we have illustrated here, in fact, works for arbitrary NFAs (without  $\epsilon$ -transitions).

# Theory of regular languages: overview



## End-of-lecture question



Suppose the above  $\epsilon$ -NFA defining concatenation is modified by identifying the final state of  $N_1$  with the start state of  $N_2$  (and removing the then-redundant  $\epsilon$ -transition linking the two states).

- ➊ Find a pair of  $\epsilon$ -NFAs,  $N_1$  and  $N_2$ , each with a single start state and single accepting state, for which the modified construction does not recognise  $\mathcal{L}(N_1).\mathcal{L}(N_2)$ .
- ➋ Show that if  $N_1$  has no loops from the accepting state back to itself, then the modified  $\epsilon$ -NFA does recognise  $\mathcal{L}(N_1).\mathcal{L}(N_2)$ .
- ➌ Which construction of an  $\epsilon$ -NFA in this lecture violates the assumption above about  $N_1$ ?

# Reading

## Relevant reading:

- Regular expressions: Kozen chapters 7,8; J & M chapter 2.1.  
(Both texts actually discuss more general ‘patterns’ — see next lecture.)
- From regular expressions to NFAs: Kozen chapter 8; J & M chapter 2.3.
- Kleene algebra: Kozen chapter 9.
- From NFAs to regular expressions: Kozen chapter 9.

**Next time:** Some applications of all this theory.

- Pattern matching
- Lexical analysis

## Appendix: (non-examinable) proof of Kleene's theorem

Given an NFA  $N = (Q, \Delta, S, F)$  (without  $\epsilon$ -transitions), we'll show how to define a regular expression defining the same language as  $N$ .

In fact, to build this up, we'll construct a **three-dimensional array** of regular expressions  $\alpha_{uv}^X$ : one for every  $u \in Q, v \in Q, X \subseteq Q$ .

**Informally**,  $\alpha_{uv}^X$  will define the set of *strings that get us from u to v allowing only intermediate states in X*.

We shall build suitable regular expressions  $\alpha_{u,v}^X$  by working our way from smaller to larger sets  $X$ .

Eventually, the language defined by  $N$  will be given by the **sum** (+) of the languages  $\alpha_{sf}^Q$  for all states  $s \in S$  and  $f \in F$ .

Construction of  $\alpha_{uv}^X$ 

Here's how the regular expressions  $\alpha_{uv}^X$  are built up.

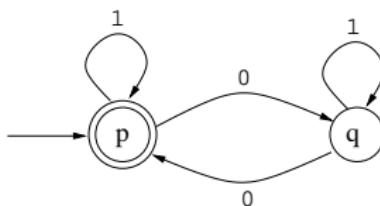
- If  $X = \emptyset$ , let  $a_1, \dots, a_k$  be all the symbols  $a$  such that  $(u, a, v) \in \Delta$ . Two subcases:
  - If  $u \neq v$ , take  $\alpha_{uv}^\emptyset = a_1 + \dots + a_k$
  - If  $u = v$ , take  $\alpha_{uv}^\emptyset = (a_1 + \dots + a_k) + \epsilon$
- Convention: if  $k = 0$ , take ' $a_1 + \dots + a_k$ ' to mean  $\emptyset$ .
- If  $X \neq \emptyset$ , choose any  $q \in X$ , let  $Y = X - \{q\}$ , and define

$$\alpha_{uv}^X = \alpha_{uv}^Y + \alpha_{uq}^Y (\alpha_{qq}^Y)^* \alpha_{qv}^Y$$

Applying these rules repeatedly gives us  $\alpha_{u,v}^X$  for every  $u, v, X$ .

## NFAs to regular expressions: tiny example

Let's revisit our old friend:



Here  $p$  is the only start state and the only accepting state.  
By the rules on the previous slide:

$$\alpha_{p,p}^{\{p,q\}} = \alpha_{p,p}^{\{p\}} + \alpha_{p,q}^{\{p\}} (\alpha_{q,q}^{\{p\}})^* \alpha_{q,p}^{\{p\}}$$

Now by inspection (or by the rules again), we have

$$\begin{array}{ll} \alpha_{p,p}^{\{p\}} = 1^* & \alpha_{p,q}^{\{p\}} = 1^* 0 \\ \alpha_{q,q}^{\{p\}} = 1 + 01^* 0 & \alpha_{q,p}^{\{p\}} = 01^* \end{array}$$

So the required regular expression is

$$1^* + 1^* 0(1 + 01^* 0)^* 01^* \quad (\text{A bit messy!})$$

# Pattern matching and lexing

## Informatics 2A: Lecture 6

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

2 October 2015

## Recap of Lecture 5

- Regular languages are closed under the operations of concatenation and Kleene star.
- This is proved using  $\epsilon$ -NFAs, which can be easily converted to ordinary NFAs.
- Regular expressions provide a textual representation of regular languages.
- Kleene's Theorem: regular expressions define exactly the regular languages.
- The difficult direction of the theorem, that every regular language is defined by some regular expression, can be proved using Kleene algebra to solve a system of simultaneous equations, exploiting Arden's Rule.

# Applications of regular-language machinery (i.e. of finite automata and regular expressions)

Applications of regular expressions encountered in Inf1-DA:

- Specifying the structure of XML documents using DTDs (Document Type Definitions)
- Searching for **concordances** (and more general forms of pattern matching) in a **corpus** (e.g., using CQP).

In this lecture we consider two further applications:

- Pattern matching in UNIX/Linux/Mac OS X, using **grep**
- **Lexing**: the first stage in the formal-language-processing 'pipeline' introduced in Lecture 2 (Course Roadmap).

Another application to **morphological parsing** will be covered in Lecture 14.

# Pattern matching with Grep tools

**Important practical problem:** Search a large file (or batch of files) for strings of a certain form.

Most UNIX/Linux-style systems since the '70s have provided a bunch of utilities for this purpose, known as **Grep** (Global Regular Expression Print).

Extremely useful and powerful in the hands of a practised user.  
Make serious use of the theory of regular languages.

Typical uses:

```
grep "[0-9]*\.[0-9][0-9]" document.txt
-- searches for prices in pounds and pence
```

```
egrep "([^\w\W])he([^\w\W]|$)" document.txt
-- searches for occurrences of the word "the"
```

# grep, egrep, fgrep

There are three related search commands, of increasing generality and correspondingly decreasing speed:

- fgrep searches for one or more **fixed strings**, using an efficient *string matching* algorithm.
- grep searches for strings matching a certain **pattern** (a simple kind of regular expression).
- egrep searches for strings matching an **extended pattern** (these give the full power of regular expressions).

For us, the last of these is the most interesting.

# Syntax of patterns (a selection)

a	Single character
[abc]	Choice of characters
[A-Z]	Any character in ASCII range
[^Ss]	Any character except those given
.	Any single character
^, \$	Beginning, end of line
*	zero or more occurrences of preceding pattern
?	optional occurrence of preceding pattern
+	one or more occurrences of preceding pattern
	choice between two patterns ('union')

(N.B. The last three of these are specific to egrep.)

This kind of syntax is very widely used. In Perl/Python (including NLTK), patterns are delimited by /.../ rather than "...".

# Mathematical versus pattern syntax

Don't be confused by the mismatch between the mathematical syntax for regular expressions (as used, e.g., in Kleene algebra) and the pattern syntax for regular expressions.

The union of two languages is written using  $+$  in mathematical syntax, and  $|$  in pattern syntax.

In pattern syntax,  $+$  is a unary operation representing one or more concatenations of strings satisfying the pattern

mathematical	pattern
$\alpha + \beta$	$\alpha   \beta$
$\alpha\alpha^*$	$\alpha +$

## Self-assessment question

Which of the patterns below are suitable for matching non-negative decimal integers in a written English document?

- ① [0-9]\*
- ② [0-9] +
- ③ 0 | [1-9] [0-9]\*
- ④ 0 | [1-9] [0-9]? [0-9]? (, [0-9] [0-9] [0-9])\*

# How egrep (typically) works

egrep will print all lines containing a match for the given pattern.  
How can it do this efficiently?

- Patterns are clearly regular expressions in disguise.
- So we can convert a pattern into a (smallish) NFA.  
(More precisely, the number of states of the NFA grows linearly in the length of the regular expression.)
- We then run the NFA , using the just-in-time simulation discussed in Lecture 4.  
We **do not** determinize the NFA to construct the full DFA, because of the potential exponential state-space blow-up.

grep can be a bit more efficient, exploiting the fact that there's 'less non-determinism' around in the absence of +, ?, |.

## Challenge question

Regular expressions and the pattern language have operations that correspond to the closure of regular languages under union, concatenation and Kleene star.

However, we have seen other closure properties of regular languages too: closure under **intersection** and **complement**.

**Question:** Why do regular expressions and patterns not include operations for intersection and complement?

# Lexical analysis of formal languages

Another application: **lexical analysis** (a.k.a. **lexing**).

**The problem:** Given a source text in some formal language, split it up into a stream of lexical tokens (or **lexemes**), each classified according to its **lexical class**.

**Example:** In Java,

```
while(count2<=1000)count2+=100
```

would be lexed as

while	(	count2	<=	1000	)
WHILE	LBRACK	IDENT	INFIX-OP	INT-LIT	RBRACK
count2	+=	100			
IDENT	ASS-OP	INT-LIT			

# Lexing in context

- The output of the lexing phase (a stream of tagged lexemes) serves as the input for the **parsing** phase.

For parsing purposes, tokens like 100 and 1000 can be conveniently lumped together in the class of *integer literals*. Wherever 100 can legitimately appear in a Java program, so can 1000.

Keywords of the language (like `while`) and other special symbols (like brackets) typically get a lexical class to themselves.

- Another job of the lexing phase (at least in languages like Java) is to throw away **whitespace** and **comments**.

**Rule of thumb:** Lexeme boundaries are the places where a space could harmlessly be inserted.

# Lexical tokens and regular languages

In most computer language (e.g. Java), the allowable forms of identifiers, integer literals, floating point literals, comments etc. are simple enough to be described by **regular expressions**.

This means we can use the technology of finite automata to produce efficient lexers.

**Even better**, if you're designing a language, you don't actually need to write a lexer yourself!

Just write some regular expressions that define the various lexical classes, and let the machine automatically generate the code for your lexer.

This is the idea behind **lexer generators**, such as the UNIX-based `lex` and the more recent Java-based `jflex`.

# Sample code (from Jflex user guide)

```
Identifier = [:jletter:] [:jletterdigit:]*  
DecIntegerLiteral = 0 | [1-9][0-9]*  
LineTerminator = \r|\n|\r\n  
InputCharacter = [^\r\n]  
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}
```

... and later on ...

```
{"while"}           { return symbol(sym.WHILE); }  
{Identifier}        { return symbol(sym.IDENT); }  
{DecIntegerLiteral} { return symbol(sym.INT_LIT); }  
{"=="}              { return symbol(sym.ASS_OP); }  
{EndOfLineComment}  { }
```

# Nerd question

A correct pattern defining jletterdigit is:

[0-9] | [a-z] | [A-Z]

What is wrong with the pattern below?

[0-9] | [A-z]

- ➊ It does not make sense.
- ➋ It matches the symbol '['.
- ➌ It does not match any letter.
- ➍ No idea.

# Recognizing a lexical token using NFAs

- Build NFAs for our lexical classes  $L_1, \dots, L_k$  in the order listed:  $N_1, \dots, N_k$ .
- Run the the ‘parallel’ automaton  $N_1 \cup \dots \cup N_k$  on some input string  $x$ .
- Choose the *smallest*  $i$  such that we’re in an accepting state of  $N_i$ . Choose class  $L_i$  as the lexical class for  $x$  with *highest priority*.
- Perform the specified *action* for the class  $L_i$  (typically ‘return tagged lexeme’, or ignore).

**Problem:** How do we know when we’ve reached the end of the current lexical token?

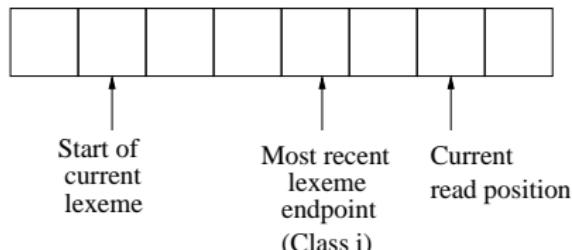
It needn’t be at the *first* point where we enter an accepting state.  
E.g. `i`, `if`, `if2` and `if23` are all valid tokens in Java.

# Principle of longest match

In most computer languages, the convention is that each stage, the *longest possible* lexical token is selected. This is known as the principle of **longest match** (a.k.a. **maximal munch**).

To find the longest lexical token starting from a given point, we'd better run  $N_1 \cup \dots \cup N_k$  until it **expires**, i.e. the set of possible states becomes empty. (Or max lexeme length is exceeded...)

We'd better also keep a note of the *last* point at which we were in an accepting state (and what the top priority lexical class was). So we need to keep track of three positions in the text:



## Lexing: (conclusion)

Once our NFA has expired, we output the string from 'start' to 'most recent end' as a lexical token of class *i*.

We then advance the 'start' pointer to the character after the 'most recent end'... and repeat until the end of the file is reached.

All this is the basis for an efficient lexing procedure (further refinements are of course possible).

In the context of lexing, the same language definition will hopefully be applicable to hundreds of source files. So in contrast to pattern searching, well worth taking some time to 'optimize' our automaton.

## End-of-lecture challenge question

Very often in the process of lexing, as described on slide 17, it occurs that, at the time that the NFA expires, the **most recent lexeme endpoint** is only one character behind the current read position.

**Question:** Think of an example, taken from a real programming language, in which the most recent endpoint lies 2 or more characters behind the current read position.

# Reading

## Relevant reading:

- Pattern matching: J & M chapter 2.1 is good. Also online documentation for grep and the like.
- Lexical analysis: see Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Chapter 3.

Next time: Limitations of regular languages.

# The Pumping Lemma: limitations of regular languages

Informatics 2A: Lecture 7

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

6 October, 2015

## Recap of Lecture 6

- A pattern syntax for regular expressions is used by the `grep/egrep` commands of unix-derived operating systems.
- `grep/egrep` are used to search for occurrences of matching strings in a file system.
- Lexical classes in programming languages are specified as regular languages.
- The `lexing algorithm` runs a parallel NFA in order to find the next `lexeme` using the `principle of longest match`.

# Non-regular languages

We have hinted before that not all languages are regular. E.g.

- The language  $\{a^n b^n \mid n \geq 0\}$ .
- The language of all *well-matched* sequences of brackets (, ).  
N.B. A sequence  $x$  is well-matched if it contains the same number of opening brackets '(' and closing brackets ')', and no initial subsequence  $y$  of  $x$  contains more ')'s than '('s.
- The language of all prefixes of well-matched sequences of brackets (, ). A string  $x$  is in this language if no initial subsequence  $y$  of  $x$  contains more ')'s than '('s.

But **how do we know** these languages aren't regular?

And can we come up with a **general technique** for proving the non-regularity of languages?

# The basic intuition: DFAs can't count!

Consider  $L = \{a^n b^n \mid n \geq 0\}$ . Just suppose, hypothetically, there were some DFA  $M$  with  $\mathcal{L}(M) = L$ .

Suppose furthermore that  $M$  had just processed  $a^n$ , and some continuation  $b^m$  was to follow.

**Intuition:**  $M$  would need to have *counted* the number of  $a$ 's, in order to know how many  $b$ 's to require.

More precisely, let  $q_n$  denote the state of  $M$  after processing  $a^n$ . Then for any  $m \neq n$ , the states  $q_m, q_n$  must be different, since  $b^m$  takes us to an accepting state from  $q_m$ , but not from  $q_n$ .

In other words,  $M$  would need **infinitely many states**, one for each natural number. Contradiction!

## Self-assessment questions

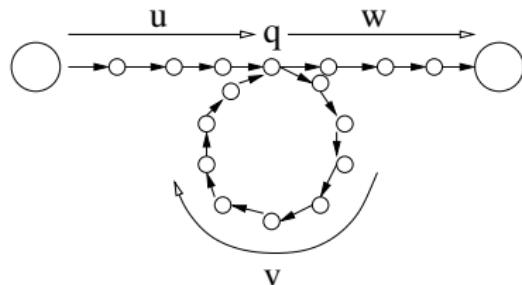
Consider the following languages over  $\{a, b\}$ .

Are they regular or not?

- ① Strings with an odd number of  $a$ 's and an even number of  $b$ 's.
- ② Strings containing strictly more  $a$ 's than  $b$ 's.
- ③ Strings such that  $(\text{no. of } a\text{'s}) \times (\text{no. of } b\text{'s}) \equiv 6 \pmod{24}$

## Loops in DFAs

Let  $M$  be a DFA with  $k$  states. Suppose, starting from any state of  $M$ , we process a string  $y$  of length  $|y| \geq k$ . We then pass through a sequence of  $|y| + 1$  states. So there must be some state  $q$  that's visited *twice or more*:



(Note that  $u$  and  $w$  might be  $\epsilon$ , but  $v$  definitely isn't.)

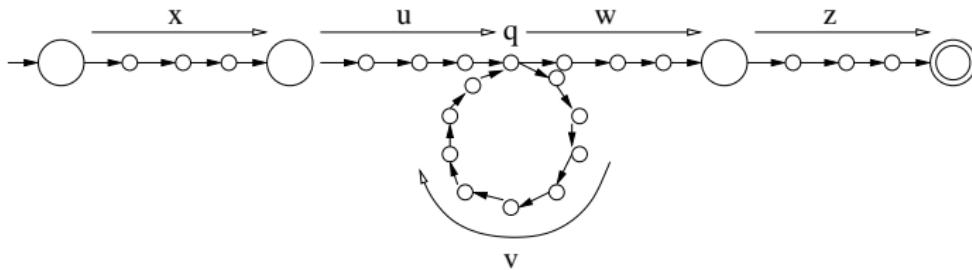
So any string  $y$  with  $|y| \geq k$  can be decomposed as  $uvw$ , where

- $u$  is the prefix of  $y$  that leads to the first visit of  $q$
- $v$  takes us once round the loop from  $q$  to  $q$ ,
- $w$  is whatever is left of  $y$  after  $uv$ .

## A general consequence

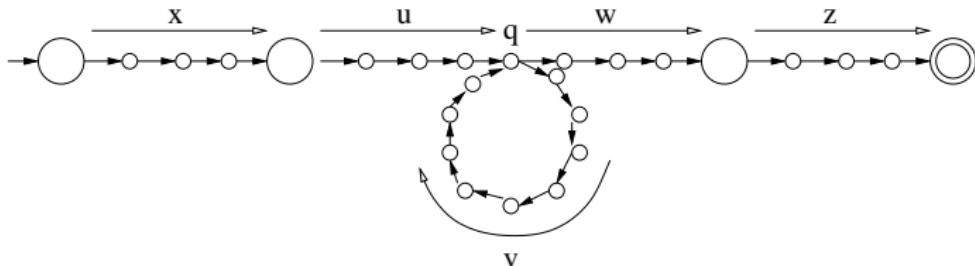
If  $L$  is *any* regular language, we can pick *some* corresponding DFA  $M$ , and it will have some number of states, say  $k$ .

Suppose we run  $M$  on a string  $xyz \in L$ , where  $|y| \geq k$ . There must be at least one state  $q$  visited twice in the course of processing  $y$ :



(There may be other 'revisited states' not indicated here.)

# The idea of 'pumping'



So  $y$  can be decomposed as  $uvw$ , where

- $xu$  takes  $M$  from the initial state to  $q$ ,
- $v \neq \epsilon$  takes  $M$  once round the loop from  $q$  to  $q$ ,
- $wz$  takes  $M$  from  $q$  to an accepting state.

But now  $M$  will be oblivious to whether, or how many times, we go round the  $v$ -loop!

So we can 'pump in' as many copies of the substring  $v$  as we like, knowing that we'll still end in an accepting state.

## The pumping lemma: official form

The pumping lemma basically summarizes what we've just said.

**Pumping Lemma.** Suppose  $L$  is a regular language. Then  $L$  has the following property.

(P) *There exists  $k \geq 0$  such that, for all strings  $x, y, z$  with  $xyz \in L$  and  $|y| \geq k$ , there exist strings  $u, v, w$  such that  $y = uvw$ ,  $v \neq \epsilon$ , and for every  $i \geq 0$  we have  $xuv^iwz \in L$ .*

## The pumping lemma: contrapositive form

Since we want to use the pumping lemma to show a language *isn't* regular, we usually apply it in the following equivalent but back-to-front form.

Suppose  $L$  is a language for which the following property holds:

( $\neg P$ ) For all  $k \geq 0$ , there exist strings  $x, y, z$  with  $xyz \in L$  and  $|y| \geq k$  such that, for every decomposition of  $y$  as  $y = uvw$  where  $v \neq \epsilon$ , there is some  $i \geq 0$  for which  $xuv^iwz \notin L$ .

Then  $L$  is not a regular language.

N.B. The pumping lemma can only be used to show a language *isn't* regular. Showing  $L$  satisfies ( $P$ ) doesn't prove  $L$  is regular!

To show that a language *is* regular, give some DFA or NFA or regular expression that defines it.

## The pumping lemma: a user's guide

So to show some language  $L$  is not regular, it's enough to show that  $L$  satisfies  $(\neg P)$ .

Note that  $(\neg P)$  is quite a complex statement:  $\forall \dots \exists \dots \forall \dots \exists \dots$

We'll look at a simple example first, then offer some advice on the general pattern of argument.

## Example 1

Consider  $L = \{a^n b^n \mid n \geq 0\}$ .

We show that  $L$  satisfies  $(\neg P)$ .

Suppose  $k \geq 0$ . (We can't choose the value of  $k$ . The argument has to work for all numbers.)

Consider the strings  $x = \epsilon$ ,  $y = a^k$ ,  $z = b^k$ . Note that  $xyz \in L$  and  $|y| \geq k$  as required. (We make a cunning choice of  $x, y, z$ .)

Suppose now we're given a decomposition of  $y$  as  $uvw$  with  $v \neq \epsilon$ . (We can't choose the strings  $u, v, w$ . The argument has to work for all possibilities.)

Let  $i = 0$ . (We make a cunning choice of  $i$ .)

Then  $uv^i w = uw = a^l$  for some  $l < k$ . So  $xuv^i wz = a^l b^k \notin L$ .

Thus  $L$  satisfies  $(\neg P)$ , so  $L$  isn't regular.

## Use of pumping lemma: general pattern

On the previous slide, the full argument is in black, whereas the **parenthetical comments in blue** are for pedagogical purposes only.

The comments emphasise the care that is needed in dealing with the quantifiers in the property ( $\neg P$ ). In general:

- You are not allowed to choose the number  $k \geq 0$ . Your argument has to work for every possible value of  $k$ .
- You have to choose the strings  $x, y, z$ , which might depend on  $k$ . You must choose these to satisfy  $xyz \in L$  and  $|y| \geq k$ . Also,  $y$  should be chosen cunningly to '**disallow pumping**' ...
- You are not allowed to choose the strings  $u, v, w$ . Your argument has to work for every possible decomposition of  $y$  as  $uvw$  with  $v \neq \epsilon$ .
- You have to choose the number  $i$  ( $\neq 1$ ) such that  $xuv^iwz \notin L$ . Here  $i$  might depend on all the previous data.

## Example 2

Consider  $L = \{a^{n^2} \mid n \geq 0\}$ .

We show that  $L$  satisfies  $(\neg P)$ :

Suppose  $k \geq 0$ .

Let  $x = a^{k^2-k}$ ,  $y = a^k$ ,  $z = \epsilon$ , so  $xyz = a^{k^2} \in L$ .

Given any splitting of  $y$  as  $uvw$  with  $v \neq \epsilon$ , we have  $1 \leq |v| \leq k$ .

So taking  $i = 2$ , we have  $xuv^2wz = a^n$  where  $k^2 + 1 \leq n \leq k^2 + k$ .

But there are no perfect squares between  $k^2$  and  $k^2 + 2k + 1$ .

So  $n$  isn't a perfect square. Thus  $xuv^2wz \notin L$ .

Thus  $L$  satisfies  $(\neg P)$ , so  $L$  isn't regular.

# Reading and prospectus

This concludes the part of the course on regular languages.

**Relevant reading:** Kozen chapters 11, 12.

Next time, we start on the next level up in the Chomsky hierarchy:  
[context-free languages](#).

# Context-free grammars

Informatics 2A: Lecture 8

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

9 October 2015

## Recap of lecture 7

- Languages that require an ability to count are not regular.
- Examples of this are  $\{a^n b^n \mid n \geq 0\}$  and the language of well-matched sequences of brackets.
- The **pumping lemma** captures a pattern of regularity necessarily present in a regular language.
- When applied in its **contrapositive form** the pumping lemma provides a powerful tool for proving that a given language is not regular.

# Beyond regular languages

Regular languages have significant limitations. (E.g. they can't cope with nesting of brackets).

So we'd like some more powerful means of defining languages.

Today we'll explore a new approach — via **generative grammars** (Chomsky 1952). A language is defined by giving a set of rules capable of 'generating' all the sentences of the language.

The particular kind of generative grammars we'll consider are called **context-free grammars**.

## Context-free grammars: an example

Here is an example **context-free grammar**.

$$\begin{array}{l} \text{Exp} \rightarrow \text{Var} \mid \text{Num} \mid ( \text{Exp} ) \\ \text{Exp} \rightarrow \text{Exp} + \text{Exp} \\ \text{Exp} \rightarrow \text{Exp} * \text{Exp} \\ \text{Var} \rightarrow x \mid y \mid z \\ \text{Num} \rightarrow 0 \mid \dots \mid 9 \end{array}$$

It generates simple **arithmetic expressions** such as

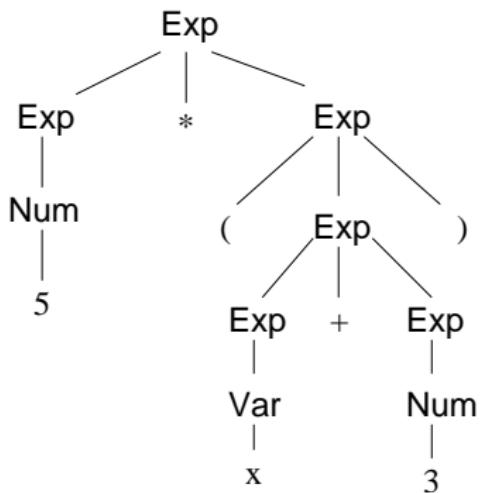
$$6 + 7 \qquad 5 * (x + 3) \qquad x * ((z * 2) + y) \qquad 8 \qquad z$$

The symbols  $+, *, (,), x, y, z, 0, \dots, 9$  are called **terminals**: these form the ultimate constituents of the phrases we generate.

The symbols Exp, Var, Num are called **non-terminals**: they name various kinds of 'sub-phrases'. We designate Exp the **start symbol**.

# Syntax trees

We grow **syntax trees** by repeatedly expanding non-terminal symbols using these rules. E.g.:



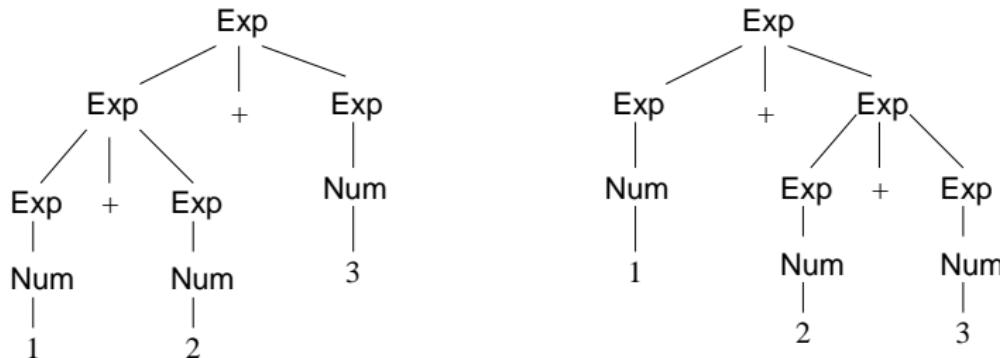
This generates  $5 * (x + 3)$ .

# The language defined by a grammar

By choosing different rules to apply, we can generate infinitely many strings from this grammar.

The **language** generated by the grammar is, by definition, the set of all **strings of terminals** that can be derived from the **start symbol** via such a syntax tree.

Note that strings such as  $1+2+3$  may be generated by more than one tree (**structural ambiguity**):



## Challenge question

How many possible syntax trees are there for the string below?

1 + 2 + 3 + 4

# Derivations

As a more ‘machine-oriented’ alternative to syntax trees, we can think in terms of **derivations** involving (mixed) strings of terminals and non-terminals. E.g.

$$\begin{aligned} \text{Exp} &\Rightarrow \text{Exp} * \text{Exp} \\ &\Rightarrow \text{Num} * \text{Exp} \\ &\Rightarrow \text{Num} * (\text{Exp}) \\ &\Rightarrow \text{Num} * (\text{Exp} + \text{Exp}) \\ &\Rightarrow 5 * (\text{Exp} + \text{Exp}) \\ &\Rightarrow 5 * (\text{Exp} + \text{Num}) \\ &\Rightarrow 5 * (\text{Var} + \text{Exp}) \\ &\Rightarrow 5 * (x + \text{Exp}) \\ &\Rightarrow 5 * (x + 3) \end{aligned}$$

At each stage, we choose one **non-terminal** and expand it using a suitable rule. When there are only **terminals** left, we can stop!

# Multiple derivations

Clearly, any **derivation** can be turned into a **syntax tree**.

However, even when there's only one syntax tree, there might be many derivations for it:

$$\begin{aligned}\text{Exp} &\Rightarrow \text{Exp} + \text{Exp} \\ &\Rightarrow \text{Num} + \text{Exp} \\ &\Rightarrow 1 + \text{Exp} \\ &\Rightarrow 1 + \text{Num} \\ &\Rightarrow 1 + 2\end{aligned}$$

(... a **leftmost** derivation)

$$\begin{aligned}\text{Exp} &\Rightarrow \text{Exp} + \text{Exp} \\ &\Rightarrow \text{Exp} + \text{Num} \\ &\Rightarrow \text{Exp} + 2 \\ &\Rightarrow \text{Num} + 2 \\ &\Rightarrow 1 + 2\end{aligned}$$

(... a **rightmost** derivation)

In the end, it's the **syntax tree** that matters — we don't normally care about the differences between various derivations for it.

However, derivations — especially leftmost and rightmost ones — will play a significant role when we consider **parsing algorithms**.

## Second example: comma-separated lists

Consider lists of (zero or more) alphabetic characters, separated by commas:

$\epsilon$        $a$        $e, d$        $q, w, e, r, t, y$

These can be generated by the following grammar (note the rules with *empty* right hand side).

List  $\rightarrow \epsilon \mid \text{Char Tail}$

Tail  $\rightarrow \epsilon \mid , \text{Char Tail}$

Char  $\rightarrow a \mid \dots \mid z$

Terminals:  $a, \dots, z, ,$

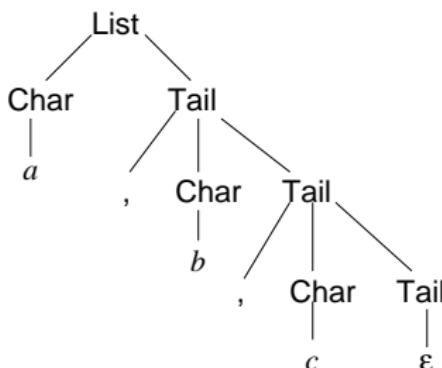
Non-terminals: List, Tail, Char

Start symbol: List

## Syntax trees for comma-separated lists

$$\begin{array}{lcl} \text{List} & \rightarrow & \epsilon \mid \text{Char Tail} \\ \text{Tail} & \rightarrow & \epsilon \mid , \text{Char Tail} \\ \text{Char} & \rightarrow & a \mid \dots \mid z \end{array}$$

Here is the syntax tree for the list  $a, b, c$ :



Notice how we indicate the application of an ' $\epsilon$ -rule'.

## Other examples

- The language  $\{a^n b^n \mid n \geq 0\}$  may be defined by the grammar:

$$S \rightarrow \epsilon \mid aSb$$

- The language of well-matched sequences of brackets ( ) may be defined by

$$S \rightarrow \epsilon \mid SS \mid (S)$$

So both of these are examples of **context-free languages**.

## Context-free grammars: formal definition

A **context-free grammar** (CFG)  $\mathcal{G}$  consists of

- a finite set  $N$  of **non-terminals**,
- a finite set  $\Sigma$  of **terminals**, disjoint from  $N$ ,
- a finite set  $P$  of **productions** of the form  $X \rightarrow \alpha$ , where  $X \in N$ ,  $\alpha \in (N \cup \Sigma)^*$ ,
- a choice of **start symbol**  $S \in N$ .

A **sentential form** is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

**Formal definition:** The set of **sentential forms** derivable from  $\mathcal{G}$  is the smallest set  $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$  such that

- $S \in \mathcal{S}(\mathcal{G})$
- if  $\alpha X \beta \in \mathcal{S}(\mathcal{G})$  and  $X \rightarrow \gamma \in P$ , then  $\alpha \gamma \beta \in \mathcal{S}(\mathcal{G})$ .

The **language** associated with grammar is the set of sentential forms that contain only terminals.

**Formal definition:** The **language** associated with  $\mathcal{G}$  is defined by  $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A **sentential form** is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

**Formal definition:** The set of **sentential forms** derivable from  $\mathcal{G}$  is the smallest set  $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$  such that

- $S \in \mathcal{S}(\mathcal{G})$
- if  $\alpha X \beta \in \mathcal{S}(\mathcal{G})$  and  $X \rightarrow \gamma \in P$ , then  $\alpha \gamma \beta \in \mathcal{S}(\mathcal{G})$ .

The **language** associated with grammar is the set of sentential forms that contain only terminals.

**Formal definition:** The **language** associated with  $\mathcal{G}$  is defined by  $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A language  $L \subseteq \Sigma^*$  is defined to be **context-free** if there exists some CFG  $\mathcal{G}$  such that  $L = \mathcal{L}(\mathcal{G})$ .

## Assorted remarks

- $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  is simply an abbreviation for a bunch of productions  $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$ .
- These grammars are called **context-free** because a rule  $X \rightarrow \alpha$  says that an  $X$  can *always* be expanded to  $\alpha$ , no matter where the  $X$  occurs.  
This contrasts with **context-sensitive** rules, which might allow us to expand  $X$  only in certain contexts, e.g.  $bXc \rightarrow bac$ .
- Broad intuition: context-free languages allow **nesting of structures to arbitrary depth**. E.g. brackets, begin-end blocks, if-then-else statements, subordinate clauses in English, ...

## Arithmetic expressions again

Our earlier grammar for arithmetic expressions was limited in that only single-character variables/numerals were allowed. One could address this problem in either of two ways:

- Add more grammar rules to allow generation of longer variables/numerals, e.g.

$$\begin{aligned} \text{Num} &\rightarrow 0 \mid \text{NonZeroDigit Digits} \\ \text{Digits} &\rightarrow \epsilon \mid \text{Digit Digits} \end{aligned}$$

- Give a separate description of the lexical structure of the language (e.g. using regular expressions), and treat the names of lexical classes (e.g. VAR, NUM) as terminals from the point of view of the CFG. So the CFG will generate strings such as

$$\text{NUM} * (\text{VAR} + \text{NUM})$$

The second option is generally preferable: lexing (using regular expressions) is computationally 'cheaper' than parsing for CFGs.

## A programming language example

Building on our grammar for arithmetic expressions, we can give a CFG for a little programming language, e.g.:

```
stmt → if-stmt | while-stmt | begin-stmt | assg-stmt
if-stmt → if bool-expr then stmt else stmt
while-stmt → while bool-expr do stmt
begin-stmt → begin stmt-list end
stmt-list → stmt | stmt ; stmt-list
assg-stmt → VAR := arith-expr
bool-expr → arith-expr compare-op arith-expr
compare-op → < | > | <= | >= | == | !=
```

Grammars like this (often with ::= in place of →) are standard in computer language reference manuals. This notation is often called **BNF** (Backus-Naur Form).

## A natural language example

Consider the following lexical classes ('parts of speech') in English:

N	nouns	( <i>alien, cat, dog, house, malt, owl, rat, table</i> )
Name	proper names	( <i>Jack, Susan</i> )
TrV	transitive verbs	( <i>admired, ate, built, chased, killed</i> )
LocV	locative verbs	( <i>is, lives, lay</i> )
Prep	prepositions	( <i>in, on, by, under</i> )
Det	determiners	( <i>the, my, some</i> )

Now consider the following productions (start symbol S):

$$\begin{array}{l} S \rightarrow NP\ VP \\ NP \rightarrow this \mid Name \mid Det\ N \mid Det\ N\ RelCI \\ RelCI \rightarrow that\ VP \mid NP\ TrV \\ VP \rightarrow is\ NP \mid TrV\ NP \mid LocV\ Prep\ NP \end{array}$$

## Natural language example in action

Even this modest bunch of rules can generate a rich multitude of English sentences, for example:

- *this is Jack*
  - *some alien ate my owl*
  - *Susan admired the rat that lay under my table*
  - *this is the dog that chased the cat that killed the rat that ate the malt that lay in the house that Jack built*
  - *(???) the malt the rat the cat the dog chased killed ate lay in the house that Jack built*
- (Hard to parse in practice — later we'll see 'why'.)

## Nesting in natural language

Excerpt from Jane Austen, *Mansfield Park*.

Whatever effect Sir Thomas's little harangue might really produce on Mr. Crawford, it raised some awkward sensations in two of the others, two of his most attentive listeners — Miss Crawford and Fanny. One of whom, having never before understood that Thornton was so soon and so completely to be his home, was pondering with downcast eyes on what it would be *not* to see Edmund every day; and the other, startled from the agreeable fancies she had been previously indulging on the strength of her brother's description, no longer able, in the picture she had been forming of a future Thornton, to shut out the church, sink the clergyman, and see only the respectable, elegant, modernized and occasional residence of a man of independent fortune, was considering Sir Thomas, with decided ill-will, as the destroyer of all this, and suffering the more from ...

## More nesting in natural language

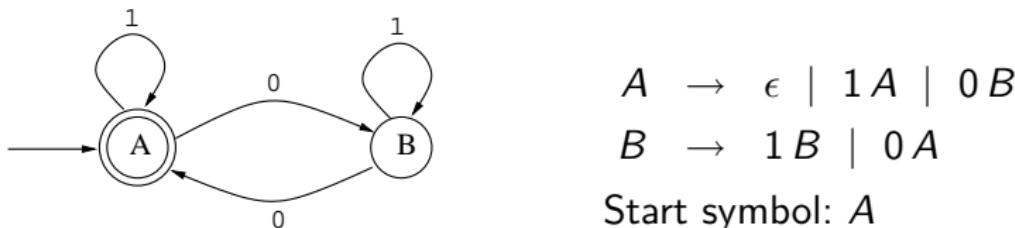
From a recent magazine article (2015):

I am not suggesting that the answer to the immense problems that the huge number of refugees from Syria and from oppression, political, religious or economic, across the globe are causing to countries faced by the initial arrival is an easy one ...

(Canon John Richardson, St. Columba's-by-the-Castle church magazine, Harvest 2015)

# Every regular language is context-free!

We can easily turn a DFA into a CFG, e.g.



- **Terminals** are **input symbols** for the DFA.
- **Non-terminals** are **states** of the DFA.
- **Start symbol** is **initial state**.
- For every **transition**  $X \xrightarrow{a} Y$ , we have a **production**  $X \rightarrow aY$ .
- For every **accepting state**  $X$ , we have a **production**  $X \rightarrow \epsilon$ .

A CFG is called **regular** if all rules are of the form  $X \rightarrow aY$ ,  $X \rightarrow Y$ ,  $X \rightarrow \epsilon$ . The languages definable by regular CFGs are precisely the **regular languages**.

## End-of-lecture self-assessment question

Recall from Slide 11:

$$\begin{aligned}\text{List} &\rightarrow \epsilon \mid \text{Char Tail} \\ \text{Tail} &\rightarrow \epsilon \mid , \text{Char Tail}\end{aligned}$$

Which of the following alternative context-free grammars for List is incorrect in the sense that it defines a different language for List?

- 1:  $\begin{aligned}\text{List} &\rightarrow \epsilon \mid \text{Body Char} \\ \text{Body} &\rightarrow \epsilon \mid \text{Body Char} ,\end{aligned}$
- 2:  $\begin{aligned}\text{List} &\rightarrow \epsilon \mid \text{NonEmpty} \\ \text{NonEmpty} &\rightarrow \text{Char} \mid \text{Char} , \text{NonEmpty}\end{aligned}$
- 3:  $\begin{aligned}\text{List} &\rightarrow \epsilon \mid \text{NonEmpty} \\ \text{NonEmpty} &\rightarrow \text{Char} \mid \text{NonEmpty} , \text{NonEmpty}\end{aligned}$
- 4: They are all correct

# Reading and prospectus

Relevant reading:

- Kozen chapters 19, 20
- Jurafsky & Martin, sections 12.1–12.3

Next time: What kinds of machines (analogous to DFAs or NFAs) correspond to context-free languages?

# Pushdown automata

Informatics 2A: Lecture 9

John Longley

School of Informatics  
University of Edinburgh

[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

13 October 2015

## Recap of lecture 8

- Context-free languages are defined by context-free grammars.
- A grammar generates strings by applying productions starting from a start symbol.
- This produces a derivation: a sequence of sentential forms ending in a string of terminal symbols.
- Every derivation determines a corresponding syntax tree.
- A grammar is structurally ambiguous if there is a string in its language that can be given two or more syntax trees.

# Machines for context-free languages

We've seen that **regular** languages can be defined by (det. or non-det.) **finite** automata.

**Question:** What kinds of machines do **context-free** languages correspond to?

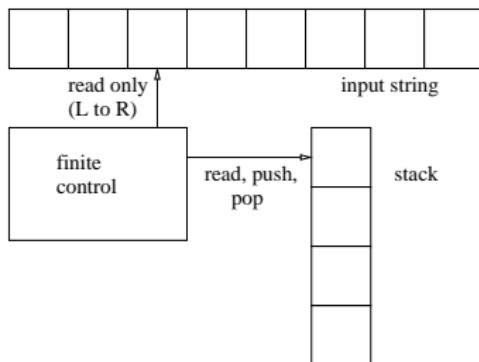
**Motivation:** Seeing how to **recognize** sentences of a CF language will be a step towards seeing how to **parse** such sentences.

**Short answer:** NFAs equipped with unlimited memory in the form of a **stack**.

# Pushdown automata (PDAs)

As in NFAs, imagine a **control unit** with finitely many possible states, equipped with a **read head** that can (only) read the input string from left to right.

Add to this a **stack** with associated operations of **push**, **pop** and **read** the item on the top of the stack.



Transitions can depend on **both** the current input symbol and the current stack item.

Transitions can also cause items to be pushed to and/or popped from the stack.

Note: the machine **can't read a stack item other than the top one** without first popping (and so losing) all items above it.

## Example of a PDA

Consider a PDA with a single state  $q$ , input alphabet  $\Sigma = \{(), ()\}$  and stack alphabet  $\Gamma = \{(), \perp\}$ . Call  $\perp$  the initial stack symbol.

Our PDA has four transitions  $q \rightarrow q$ , labelled as follows:

$$^1 (, \perp : (\perp \quad ^2 (, ( : (( \quad ^3 ), ( : \epsilon \quad ^4 \epsilon, \perp : \epsilon$$

**Meaning.** <sup>1</sup> If current read symbol is ( and current stack symbol is  $\perp$ , may pop the  $\perp$  and replace it with (  $\perp$ . Note that we push  $\perp$  first, then (, so stack grows to the left.

<sup>2</sup> Similarly with ( in place of  $\perp$ .

<sup>3</sup> If current read symbol is ) and current stack symbol is (, may simply pop the (.

<sup>4</sup> If current stack symbol is  $\perp$ , can just pop it.

**Idea:** stack keeps track of currently pending ('). When stack clears, may pop the initial  $\perp$  — this ends the computation.

# Sample execution

<sup>1</sup> (, ⊥ : ( ⊥      <sup>2</sup> (, ( : ((      <sup>3</sup> ), ( : ε      <sup>4</sup> ε, ⊥ : ε

	Unread input	Stack state
	((())	⊥
<sup>1</sup> →	)()	( ⊥
<sup>2</sup> →	)()	(( ⊥
<sup>3</sup> →	()	( ⊥
<sup>2</sup> →	) )	(( ⊥
<sup>3</sup> →	)	( ⊥
<sup>3</sup> →	ε	⊥
<sup>4</sup> →	ε	ε

# Language recognised by example PDA

Our example PDA has a single state  $q$ , input alphabet  $\Sigma = \{(, )\}$ , and stack alphabet  $\Gamma = \{(), \perp\}$  with initial stack symbol  $\perp$ , and transitions  $q \rightarrow q$ ,

$$\begin{array}{llll} 1 \quad (, \perp : (\perp & 2 \quad (, ( : (( & 3 \quad ), ( : \epsilon & 4 \quad \epsilon, \perp : \epsilon \end{array}$$

This machine can empty its stack at the end of the input string if and only if the input string is a well-matched sequence of brackets.

So the PDA acts as a (non-deterministic) recognizer for the language of well-matched sequences of brackets.

# PDAs: formal definition

A (nondeterministic) pushdown automaton (N)PDA  $M$  consists of

- a finite set  $Q$  of control states
- a finite input alphabet  $\Sigma$
- a finite stack alphabet  $\Gamma$  including a start symbol  $\perp$
- a start state  $s \in Q$
- a transition relation  $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$

Note that each individual transition has the form  $((p, a, s), (q, u))$  where  $p, q \in Q$ ,  $a \in \Sigma \cup \{\epsilon\}$ ,  $s \in \Gamma$  and  $u \in \Gamma^*$ . Such a transition is sometimes drawn as:

$$p \xrightarrow{a, s:u} q$$

## Accepting by empty stack

A string  $x \in \Sigma^*$  is **accepted** by  $M$  if there is some run of  $M$  on  $x$ , starting at control state  $s$  with stack  $\perp$ , and finishing (at any control state) with **empty stack** having consumed all of  $x$ .

This definition implements **acceptance by empty stack**.

The language accepted by  $M$  is

$$\mathcal{L}(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$$

## Other acceptance conditions

Other acceptance conditions for PDAs can be found in the literature.

**Example variation:** Equip  $M$  with a set of accepting states, and say a string is accepted if it can trigger a computation ending in an accepting state.

The choice of acceptance condition is not a big deal, For any NPDA of either kind, we can build one of the other kind that accepts the same language. For details see Kozen chapter E.

In Inf2A, we use empty stack as our default acceptance condition for PDAs.

## How powerful are PDAs?

We shall outline the proof of the following theorem:

*A language  $L$  is context-free if and only if there is an N PDA  $M$  such that  $L = \mathcal{L}(M)$ .*

One can also define **deterministic** pushdown automata (DPDAs) in a reasonable way.

;-( Sadly, there's no analogue of the NFA 'powerset construction' for NPDAs. In fact, there are context-free languages that can't be recognized by **any** DPDA (example in Lecture 28).

Since DPDAs allow for efficient processing, this prompts us to focus on 'simple' kinds of CFLs that **can** be recognized by a DPDA. (See next lecture on efficient table-driven parsing.)

# From CFGs to NPDAs

Given a CFG with nonterminals  $N$ , terminals  $\Sigma$ , productions  $P$  and start symbol  $S$ .

Build an NPDA with a **single state**  $q$ , input alphabet  $\Sigma$  and stack alphabet  $N \cup \Sigma$ . Take  $S$  as initial stack symbol.

- For each **production**  $X \rightarrow \alpha$  in  $P$ , include an  **$\epsilon$ -transition**

$$q \xrightarrow{\epsilon, X:\alpha} q$$

- For each **terminal**  $a \in \Sigma$ , include a **transition**

$$q \xrightarrow{a, a:\epsilon} q.$$

**Intuition:** the stack records a ‘guess’ at a sentential form for the part of the input still to come.

## From CFGs to NPDAs: Example

Recall our grammar for arithmetic expressions:

$$\begin{array}{ll} \text{Exp} \rightarrow \text{Var} \mid \text{Num} \mid (\text{Exp}) & \text{Var} \rightarrow x \mid y \mid z \\ \text{Exp} \rightarrow \text{Exp} + \text{Exp} & \text{Num} \rightarrow 0 \mid \dots \mid 9 \\ \text{Exp} \rightarrow \text{Exp} * \text{Exp} & \end{array}$$

Suppose we turn this into an NPDA as above.

What does an accepting run of this NPDA on the input string  
**(x)\*5** look like?

# An accepting run of the NPDA

Consider the input string  $(x)^*5$ .

Our machine can proceed as follows, correctly guessing the rule to apply at each stage.

Transition	Input read	Stack state
	$\epsilon$	Exp
Apply * rule	$\epsilon$	Exp * Exp
Apply () rule	$\epsilon$	( Exp ) * Exp
Match (	(	Exp ) * Exp
Apply Var rule	(	Var ) * Exp
Apply x rule	(	x ) * Exp
Match x	(x)	) * Exp
Match )	(x)	* Exp
Match *	(x)*	Exp
Apply Num rule	(x)*	Num
Apply 5 rule	(x)*	5
Match 5	(x) * 5	stack empty!

At each stage, combining 'input read' and 'stack state' gives us a **sentential form**. In effect, the computation traces out a **leftmost derivation** of  $t$  in  $\mathcal{G}$ . So the computation tells us not just that the string is legal, but how to build a syntax tree for it.

## From NPDA to CFGs: brief sketch

Suppose first  $M$  is an NPDA with just one state  $q$ .

Can turn  $M$  into a CFG: almost the reverse of what we just did.

Use  $M$ 's stack alphabet  $\Gamma$  as the set of nonterminals of the grammar.

General form for transitions of  $M$  is  $((q, a, X), (q, \alpha))$ , where  $a$  and/or  $\alpha$  might be  $\epsilon$ . Turn these into productions  $X \rightarrow a\alpha$ .

Now suppose we have an NPDA  $M$  with many states. Can turn it into an equivalent NPDA with just one state, essentially by storing all 'state information' on the stack.

When pushing multiple stack entries, must nondeterministically 'guess' the intermediate states. For details see Kozen chapter 25.

## Summary

We've seen that NPDAs exactly 'match' CFGs in terms of their power for defining languages. Indeed, a **pushdown store** (stack) gives just the computational power needed to deal with **nesting**.

Accepting computations don't just tell us a string is legal — they 'do parsing' (i.e. tell us how to build a syntax tree).

Problem is that computations here are **non-deterministic**: must correctly 'guess' which production to apply at each stage. If only things were deterministic, we'd have an **efficient** parsing algorithm!

## Summary

We've seen that NPDAs exactly 'match' CFGs in terms of their power for defining languages. Indeed, a **pushdown store** (stack) gives just the computational power needed to deal with **nesting**.

Accepting computations don't just tell us a string is legal — they 'do parsing' (i.e. tell us how to build a syntax tree).

Problem is that computations here are **non-deterministic**: must correctly 'guess' which production to apply at each stage. If only things were deterministic, we'd have an **efficient** parsing algorithm!

**Next**, we'll see how far we can get with special CFGs for which the corresponding PDA *is* deterministic. This is good enough for many computing applications.

## Summary

We've seen that NPDAs exactly 'match' CFGs in terms of their power for defining languages. Indeed, a **pushdown store** (stack) gives just the computational power needed to deal with **nesting**.

Accepting computations don't just tell us a string is legal — they 'do parsing' (i.e. tell us how to build a syntax tree).

Problem is that computations here are **non-deterministic**: must correctly 'guess' which production to apply at each stage. If only things were deterministic, we'd have an **efficient** parsing algorithm!

**Next**, we'll see how far we can get with special CFGs for which the corresponding PDA *is* deterministic. This is good enough for many computing applications.

**Later**, we'll look at 'semi-efficient' parsers that work for *any* CFG.

## Reading and prospectus

### Relevant reading:

- Core material: Kozen chapters 19, 23, 24.
- Further topics: E, 25, F.

Next time, we look at **LL(1) grammars**: a class of relatively 'simple' CFGs for which very efficient parsing is possible.

# LL(1) predictive parsing

## Informatics 2A: Lecture 10

John Longley

School of Informatics  
University of Edinburgh  
[jrl@staffmail.ed.ac.uk](mailto:jrl@staffmail.ed.ac.uk)

15 October 2015

## Recap of lecture 9

- A **pushdown automaton (PDA)** uses **control states** and a **stack** to recognise input strings.
- We considered PDAs whose goal is to **empty the stack** after having read the input string.
- The languages recognised by **nondeterministic pushdown automata (NPDAs)** are exactly the **context-free languages**.
- Different to the situation for finite automata, **deterministic pushdown automata (DPDAs)** are less powerful than NPDAs.  
(The definition of DPDAs is subtle and beyond the scope of Inf2A, though related material will be covered in Tutorial 3.)

## Predictive parsing: first steps

Consider how we would like to parse a program in the little programming language from Slide 17 of Lecture 8.

```
stmt   → if-stmt | while-stmt | begin-stmt | assg-stmt
if-stmt → if bool-expr then stmt else stmt
while-stmt → while bool-expr do stmt
begin-stmt → begin stmt-list end
stmt-list → stmt | stmt ; stmt-list
assg-stmt → VAR := arith-expr
bool-expr → arith-expr compare-op arith-expr
compare-op → < | > | <= | >= | == | !=
```

We read the lexed program token-by-token from the start.

The start symbol of the grammar is **stmt**.

Suppose the first lexical class in the program is **begin**.

From this information, we can tell that the first production in the syntax tree must be

$$\text{stmt} \rightarrow \text{begin-stmt}$$

We thus have to parse the program as **begin-stmt**.

We now see that the next production in the syntax tree has to be

$$\text{begin-stmt} \rightarrow \text{begin stmt-list end}$$

We thus have to parse the full program **begin** ..... as **begin stmt-list end**.

We can thus step over **begin**, and proceed to parse the remaining program ..... as **stmt-list end**, etc.

In the example, the correct production to apply is being determined from just two pieces of information:

- current lexical class
- a predicted nonterminal symbol.

If it is always possible to determine the next production from the above information, the grammar is said to be **LL(1)**.

When a grammar is LL(1), parsing can run **efficiently** and **deterministically**.

As it turns out, in spite of the promising start to parsing on the previous slides, the grammar for our little programming language is **not** LL(1). However, we shall see in Lecture 13 how the grammar can be **repaired** to provide an LL(1) grammar for the same language.

## LL(1) grammars: another intuition

Think about a (one-state) NPDA derived from a CFG  $\mathcal{G}$ .

At each step, our NPDA can ‘see’ two things:

- the current input symbol
- the topmost stack symbol

Roughly speaking,  $\mathcal{G}$  is an **LL(1) grammar** if, just from this information, it’s possible to determine which transition/production to apply next.

Here LL(1) means ‘read input from **Left**, build **Leftmost** derivation, look just **one** symbol ahead’.

**Subtle point:** When doing LL(1) parsing (as opposed to executing an NDPA) we use the input symbol to help us choose a production without consuming the input symbol . . . hence **look ahead**.

## Parse tables

Saying the current input symbol and stack symbol uniquely determine the production means we can draw up a two-dimensional **parse table** telling us which production to apply in any situation.

Consider e.g. the following grammar for well-bracketed sequences:

$$S \rightarrow \epsilon \mid TS \quad T \rightarrow (S)$$

This has the following parse table:

		(	)	\$
		S	$S \rightarrow TS$	$S \rightarrow \epsilon$
		T	$T \rightarrow (S)$	

- **Columns** are labelled by **terminals**, which are the input symbols. We include an extra column for an ‘end-of-input’ marker \$.
- **Rows** are labelled by **nonterminals**.
- **Blank entries** correspond to situations that can never arise in processing a legal input string.

## Predictive parsing with parse tables

Given such a parse table, parsing can be done very efficiently using a stack. The stack (reading downwards) records the predicted sentential form for the remaining part of the input.

- Begin with just start symbol  $S$  on the stack.
- If current input symbol is  $a$  (maybe  $\$$ ), and current stack symbol is a non-terminal  $X$ , look up rule for  $a, X$  in table.  
**[Error if no rule.]** If rule is  $X \rightarrow \beta$ , pop  $X$  and replace with  $\beta$  (pushed right-end-first!)
- If current input symbol is  $a$  and stack symbol is  $a$ , just pop  $a$  from stack and advance input read position.  
**[Error if stack symbol is any other terminal.]**
- Accept if stack empties with  $\$$  as input symbol.  
**[Error if stack empties sooner.]**

## Example of predictive parsing

	(	)	\$
$S$	$S \rightarrow TS$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
$T$	$T \rightarrow (S)$		

Let's use this table to parse the input string  $(( ))$ .

Operation	Remaining input	Stack state
	$(( ))\$$	$S$
Lookup (, $S$	$(( ))\$$	$TS$
Lookup (, $T$	$(( ))\$$	$(S)S$
Match (	$(( ))\$$	$S)S$
Lookup (, $S$	$(( ))\$$	$TS)S$
Lookup (, $T$	$(( ))\$$	$(S)S)S$
Match (	$(( ))\$$	$S)S)S$
Lookup ), $S$	$(( ))\$$	$)S)S$
Match )	$(( ))\$$	$S)S$
Lookup ), $S$	$(( ))\$$	$)S$
Match )	$(( ))\$$	$S$
Lookup \$, $S$	$(( ))\$$	empty stack

(Also easy to build a **syntax tree** as we go along!)

## Self-assessment questions

	(	)	\$
S	$S \rightarrow TS$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
T	$T \rightarrow (S)$		

For each of the following two input strings:

) (

what will go wrong when we try to apply the predictive parsing algorithm?

- ① Blank entry in table encountered
- ② Input symbol (or end marker) doesn't match expected symbol
- ③ Stack empties before end of string reached

## Further remarks

**Slogan:** the parse table entry for  $a, X$  tells us which rule to apply if we're expecting an  $X$  and see an  $a$ .

- Often, the  $a$  will be simply the **first** symbol of the  $X$ -subphrase in question.
- But not always: maybe the  $X$ -subphrase in question is  $\epsilon$ , and the  $a$  belongs to whatever **follows** the  $X$ .  
E.g. in the lookups for  $), S$  on the previous slide, the  $S$  in question turns out to be empty.

Once we've got a parse table for a given grammar  $\mathcal{G}$ , we can parse strings of length  $n$  in  $O(n)$  time (and  $O(n)$  space).

Our algorithm is an example of a **top-down predictive parser**: it works by 'predicting' the form of the remainder of the input, and builds syntax trees a top-down way (i.e. starting from the root). There are other parsers (e.g. LR(1)) that work 'bottom up'.

## LL(1) grammars: formal definition

Suppose  $\mathcal{G}$  is a CFG containing no ‘useless’ nonterminals, i.e.

- every nonterminal appears in some sentential form derived from the start symbol;
- every nonterminal can be expanded to some (possibly empty) string of terminals.

We say  $\mathcal{G}$  is **LL(1)** if for each terminal  $a$  and nonterminal  $X$ , there is some production  $X \rightarrow \alpha$  with the following property:

*If  $b_1 \dots b_n X \gamma$  is a sentential form appearing in a leftmost derivation of some string  $b_1 \dots b_n a c_1 \dots c_m$  ( $n, m \geq 0$ ), the next sentential form appearing in the derivation is necessarily  $b_1 \dots b_n \alpha \gamma$ .*

(Note that if  $a, X$  corresponds to a ‘blank entry’ in the table, *any* production  $X \rightarrow \alpha$  will satisfy this property, because a sentential form  $b_1 \dots b_n X \gamma$  can’t arise.)

## Non-LL(1) grammars

Roughly speaking, a grammar is **by definition** LL(1) if and only if there's a parse table for it. Not all CFGs have this property!

Consider e.g. a different grammar for the same language of well-bracketed sequences:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Suppose we'd set up our initial stack with  $S$ , and we see the input symbol  $($ . What rule should we apply?

- If the input string is  $((()$ , should apply  $S \rightarrow (S)$ .
- If the input string is  $()()$ , should apply  $S \rightarrow SS$ . We can't tell without looking further ahead.

**Put another way:** if we tried to build a parse table for this grammar, the two rules  $S \rightarrow (S)$  and  $S \rightarrow SS$  would be competing for the slot  $(, S$ . So this grammar is **not** LL(1).

## Remaining issues

Easy to see from the definition that any LL(1) grammar will be **unambiguous**: never have two syntax trees for the same string.

- For [computer languages](#), this is fine: normally want to avoid ambiguity anyway.
- For [natural languages](#), ambiguity is a fact of life! So LL(1) grammars are normally inappropriate.

Two outstanding questions . . .

- How can we tell if a grammar is LL(1) — and if it is, how can we construct a parse table? (See [Lecture 11](#).)
- If a grammar isn't LL(1), is there any hope of replacing it by an equivalent one that is? (See [Lecture 13](#).)

# Reading and prospectus

## Relevant reading:

- Some lecture notes from a previous year (covering the same material but with different examples) are available via the course website. (See [Readings](#) column of course schedule.)
- See also Aho, Sethi and Ullman, *Compilers: Principles, Techniques, Tools*, Section 4.4.

# Automatic generation of LL(1) parsers

## Informatics 2A: Lecture 11

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

16 October 2015

## Recap of Lecture 10

- LL(1) predictive parsing reads the input string from left to right, and determines the correct production to apply purely on the basis of two pieces of information: (1) the **current input symbol**, and (2) the **current predicted nonterminal symbol** (which is kept on the head of a stack).
- The parsing algorithm is efficient and deterministic and uses a **parse table** to determine the next production.
- LL(1) parsing is suitable only for **formal languages** with **unambiguous grammars**. Even for such languages, a clever choice of grammar is required for the grammar to be LL(1).  
**(Addendum:** Some formal languages with unambiguous grammars cannot be given an LL(1) grammar at all.)

# Generating parse tables

We've seen that if a grammar  $\mathcal{G}$  happens to be LL(1) — i.e. if it admits a **parse table** — then efficient, deterministic, predictive parsing is possible with the help of a stack.

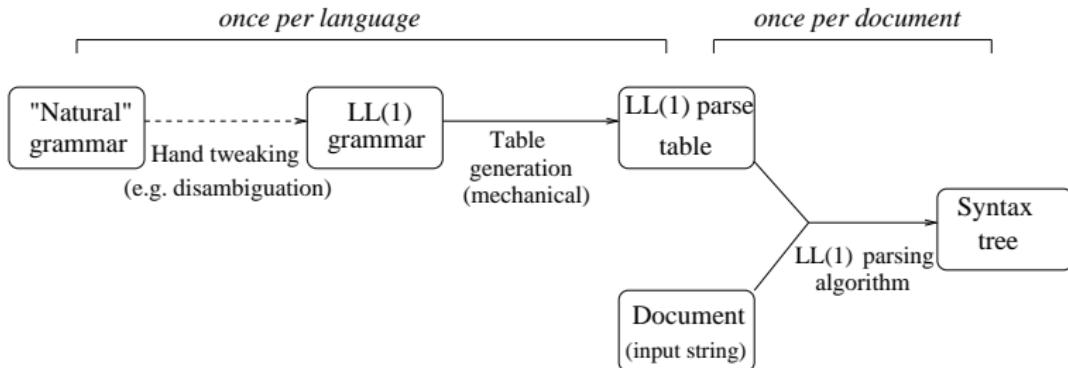
What's more, if  $\mathcal{G}$  is LL(1),  $\mathcal{G}$  is automatically **unambiguous**.

But **how do we tell** whether a grammar is LL(1)? And if it is, **how can we construct a parse table** for it?

For very small grammars, might be able to answer these questions by eye inspection. But for realistic grammars, a systematic method is needed.

In this lecture, we give an **algorithmic procedure** for answering both questions.

# The overall picture



Previous lecture: the **LL(1) parsing algorithm**, which works on a parse table and a particular input string.

This lecture: algorithm for getting from a grammar  $\mathcal{G}$  to a parse table. The algorithm will succeed if  $\mathcal{G}$  is LL(1), or fail if it isn't. (As in previous lecture, assume  $\mathcal{G}$  has no 'useless nonterminals'.)

Next lecture: ways of getting from a grammar to an equivalent LL(1) grammar. (Not always possible, but work quite often.)

# First and Follow sets

Two steps to construct a parse table for a given grammar:

- ① For each nonterminal  $X$ , compute two sets called  $First(X)$  and  $Follow(X)$ , defined as follows:
  - $First(X)$  is the set of all terminals that can appear at the start of a phrase derived from  $X$ .  
[Convention: if  $\epsilon$  can be derived from  $X$ , also include the special symbol  $\epsilon$  in  $First(X)$ .]
  - $Follow(X)$  is the set of all terminals that can appear immediately after  $X$  in some sentential form derived from the start symbol  $S$ .  
[Convention: if  $X$  can appear at the end of some such sentential form, also include the special symbol  $\$$  in  $Follow(X)$ .]
- ② Use these  $First$  and  $Follow$  sets to fill out the parse table.

The first step is somewhat tricky. The second is easier.

## Two self-assessment questions

- $\text{First}(X)$  is the set of all terminals that can appear at the start of a phrase derived from  $X$ .  
[Convention: if  $\epsilon$  can be derived from  $X$ , also include the special symbol  $\epsilon$  in  $\text{First}(X)$ .]

Recall our LL(1) grammar for well-matched bracket sequences:

$$S \rightarrow \epsilon \mid TS \qquad T \rightarrow (S)$$

**Question.** Work out each of the two sets below.

- ①  $\text{First}(T)$
- ②  $\text{First}(S)$

## Two more self-assessment questions

- $\text{Follow}(X)$  is the set of all terminals that can appear immediately after  $X$  in some sentential form derived from the start symbol  $S$ .  
[Convention: if  $X$  can appear at the end of some such sentential form, also include  $\$$  in  $\text{Follow}(X)$ .]

Again consider the same LL(1) grammar:

$$S \rightarrow \epsilon \mid TS \qquad T \rightarrow (S)$$

**Question.** Work out each of the two sets below.

- ①  $\text{Follow}(S)$
- ②  $\text{Follow}(T)$

# First and Follow sets: an example

Look again at our grammar for well-matched bracket sequences:

$$S \rightarrow \epsilon \mid TS \qquad T \rightarrow (S)$$

By inspection, we can see that

$$\text{First}(S) = \{ (, \epsilon \} \quad \text{because an } S \text{ can begin with ( or be empty}$$

$$\text{First}(T) = \{ ( \} \quad \text{because a } T \text{ must begin with (}$$

$$\text{Follow}(S) = \{ ), \$ \} \quad \text{because within a complete phrase, an } S \\ \text{can be followed by ) or appear at the end}$$

$$\text{Follow}(T) = \{ (, ), \$ \} \quad \text{because a } T \text{ can be followed by ( or )} \\ \text{or appear at the end}$$

Later we'll give a systematic method for computing these sets.

Further convention: take  $\text{First}(a) = \{ a \}$  for each **terminal**  $a$ .

# Filling out the parse table

Once we've got these *First* and *Follow* sets, we can fill out the parse table as follows.

For each production  $X \rightarrow \alpha$  of  $\mathcal{G}$  in turn:

- For each terminal  $a$ , if  $\alpha$  'can begin with'  $a$ , insert  $X \rightarrow \alpha$  in row  $X$ , column  $a$ .
- If  $\alpha$  'can be empty', then for each  $b \in \text{Follow}(X)$  (where  $b$  may be  $\$$ ), insert  $X \rightarrow \alpha$  in row  $X$ , column  $b$ .

If doing this leads to **clashes** (i.e. two productions fighting for the same table entry) then **conclude that the grammar is not LL(1)**.

To explain the phrases in **blue**, suppose  $\alpha = x_1 \dots x_n$ , where the  $x_i$  may be terminals or nonterminals.

- $\alpha$  **can be empty** means  $\epsilon \in \text{First}(\alpha)$  for every  $x_i$ .
- $\alpha$  **can begin with**  $a$  means that, for some  $i$ ,  
 $\epsilon \in \text{First}(x_1) \cap \dots \cap \text{First}(x_{i-1})$ , and  $a \in \text{First}(x_i)$ .

## Comments on filling out the parse table

- The case  $\alpha = \epsilon$  is counted as a case in which  $\alpha$  can be empty.  
(This case is implicit in the last slide since  $\alpha = \epsilon$  counts as an instance of  $\alpha = x_1 \dots x_n$  by taking  $n = 0$ , whence the condition " $\epsilon \in \text{First}(x_i)$  for every  $x_i$ " is vacuously true since there are no  $x_i$ .)
- Similarly, we count  $\alpha = x_1 \dots x_n$  with  $a \in \text{First}(x_1)$  as one case in which  $\alpha$  can begin with  $a$ .  
(Again this is implicit in the last slide. The condition  $\epsilon \in \text{First}(x_1) \cap \dots \cap \text{First}(x_{i-1})$  means that  $\epsilon$  is contained in all the sets  $\text{First}(x_1)$ ,  $\text{First}(x_2)$  up to  $\text{First}(x_{i-1})$ . In the case that  $i = 1$ , we consider the sequence  $x_1, \dots, x_{i-1}$  as being empty. Thus the condition " $\epsilon \in \text{First}(x_1) \cap \dots \cap \text{First}(x_{i-1})$ " is again vacuously true. )

## Filling out the parse table: example

$$S \rightarrow \epsilon \mid TS \qquad T \rightarrow (S)$$

$$\begin{array}{ll} First(S) = \{(), \epsilon\} & Follow(S) = \{(), \$\} \\ First(T) = \{()\} & Follow(T) = \{(), \$\} \end{array}$$

Use this information to fill out the [parse table](#):

- $(S)$  can begin with  $($ , so insert  $T \rightarrow (S)$  in entry for  $(, T$ .
- $TS$  can begin with  $($ , so insert  $S \rightarrow TS$  in entry for  $(, S$ .
- $\epsilon$  can be empty, and  $Follow(S) = \{(), \$\}$ , so insert  $S \rightarrow \epsilon$  in entries for  $), S$  and  $\$, S$ .

This gives the parse table we had in the previous lecture:

		(	)	\$
S	$S \rightarrow TS$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	
T	$T \rightarrow (S)$			

## Intermezzo: true or false?

- ① Every LL(1) grammar is context free.
- ② Every context-free language can be presented using an LL(1) grammar.
- ③ Every regular language can be presented using an LL(1) grammar.
- ④ Every LL(1) grammar is unambiguous.
- ⑤ Languages defined by LL(1) grammars can be efficiently parsed.

# Calculating First and Follow sets: preliminary stage

To complete the story, we'd like an algorithm for calculating *First* and *Follow* sets.

Easy first step: compute the set  $E$  of nonterminals that 'can be  $\epsilon$ ':

- ① Start by adding  $X$  to  $E$  whenever  $X \rightarrow \epsilon$  is a production of  $\mathcal{G}$ .
- ② If  $X \rightarrow Y_1 \dots Y_m$  is a production and all  $Y_1, \dots, Y_m$  are already in  $E$ , add  $X$  to  $E$ .
- ③ Repeat step 2 until  $E$  stabilizes.

**Example:** for our grammar of well-matched bracket sequences, we have  $E = \{S\}$ .

# Calculating First sets: the details

- ① Set  $\text{First}(a) = \{a\}$  for each  $a \in \Sigma$ . For each nonterminal  $X$ , initially set  $\text{First}(X)$  to  $\{\epsilon\}$  if  $X \in E$ , or  $\emptyset$  otherwise.
- ② For each production  $X \rightarrow x_1 \dots x_n$  and each  $i \leq n$ , if  $x_1, \dots, x_{i-1} \in E$  and  $a \in \text{First}(x_i)$ , add  $a$  to  $\text{First}(X)$ .
- ③ Repeat step 2 until all  $\text{First}$  sets stabilize.

## Example:

- Start with  $\text{First}(S) = \{\epsilon\}$ ,  $\text{First}(T) = \emptyset$ , etc.
- Consider  $T \rightarrow (S)$  with  $i = 1$ : add  $($  to  $\text{First}(T)$ .
- Now consider  $S \rightarrow TS$  with  $i = 1$ : add  $($  to  $\text{First}(S)$ .
- That's all.

# Calculating Follow sets: the details

- ① Initially set  $\text{Follow}(S) = \{\$\}$  for the start symbol  $S$ , and  $\text{Follow}(X) = \emptyset$  for all other nonterminals  $X$ .
- ② For each production  $X \rightarrow \alpha$ , each splitting of  $\alpha$  as  $\beta Y x_1 \dots x_n$  where  $n \geq 1$ , and each  $i$  with  $x_1, \dots, x_{i-1} \in E$ , add all of  $\text{First}(x_i)$  (excluding  $\epsilon$ ) to  $\text{Follow}(Y)$ .
- ③ For each production  $X \rightarrow \alpha$  and each splitting of  $\alpha$  as  $\beta Y$  or  $\beta Y x_1 \dots x_n$  with  $x_1, \dots, x_n \in E$ , add all of  $\text{Follow}(X)$  to  $\text{Follow}(Y)$ .
- ④ Repeat step 3 until all  $\text{Follow}$  sets stabilize.

## Example:

- Start with  $\text{Follow}(S) = \{\$\}$ ,  $\text{Follow}(T) = \emptyset$ .
- Apply step 2 to  $T \rightarrow (S)$  with  $i = 1$ : add  $)$  to  $\text{Follow}(S)$ .
- Apply step 2 to  $S \rightarrow TS$  with  $i = 1$ : add  $($  to  $\text{Follow}(T)$ .
- Apply step 3 to  $S \rightarrow TS$  with  $n = 1$ : add  $)$  and  $\$$  to  $\text{Follow}(T)$ .
- That's all.

## Parser generators

LL(1) is representative of a bunch of [classes of CFGs](#) that are efficiently parseable. E.g.  $\text{LL}(1) \subset \text{LALR} \subset \text{LR}(1)$ . These involve various tradeoffs of expressive power vs. efficiency/simplicity.

For such languages, a parser can be generated [automatically](#) from a suitable grammar. (E.g. for LL(1), just need parse table plus fixed ‘driver’ for the parsing algorithm.)

So we don’t need to write parsers ourselves — just the grammar! (E.g. one can basically define the syntax of Java in about 7 pages of context-free rules.)

This is the principle behind [parser generators](#) like yacc ('yet another compiler compiler') and java-cup.

# Reading

- Recommended: Some relevant lecture notes (“Note 12” in particular) and a tutorial sheet from previous years are available via the Course Schedule webpage.
- Dragon book: Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Section 4.4.
- Tiger book: Andrew Appel, *Modern Compiler Implementation in (C | Java | ML)*.
- Turtle book: Aho and Ullman, *Foundations of Computer Science*.

# Types and Static Type Checking (Introducing Micro-Haskell)

Informatics 2A: Lecture 12

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

20 October 2015

- 1 Types
- 2 Micro-Haskell: crash course
- 3 MH Types & Abstract Syntax
- 4 Type Checking

So far in the course, we have examined the machinery that, in the case of a programming language, takes us from a program text to a parse tree, via the stages of **lexing** and **parsing**.

This lecture looks at two further stages of the pipeline:

- The parse tree may be converted into an **abstract syntax tree (AST)**, a kind of simplified parse tree which contains just the information needed for further processing.
- The resulting AST may then be subjected to various checks to ensure that certain obvious errors are avoided (**static analysis**). One common form of static analysis is **type-checking**.

After this, the AST will be fed to an **evaluator** or **interpreter** to execute the program—or else to a **compiler** to translate it into executable low-level code.

# Types

Consider the expression

`3 + True`

How is a compiler or interpreter supposed to execute this?

It does not make sense to apply the numerical addition operation to the argument `True`, which is a boolean.

This is an example of a **type error**.

Different programming languages take different approaches to such errors.

## Approaches to type errors

**Laissez faire:** Even if an operation does not make sense for the data its being applied to, just go ahead and apply it to the (binary) machine representation of the data. In some cases this will do something harmful. In other cases it might even be useful.

**Dynamic checking:** At the point during execution at which a type mismatch (between operation and argument) is encountered, raise an error. This gives rise to helpful runtime errors.

**Static checking:** Check (the AST of) the program to ensure that all operations are applied in a type-meaningful way. If not, identify the error(s), and disallow the program from being run until corrected. This allows many program errors to be identified before execution.

## Approaches to type errors

**Laissez faire:** Even if an operation does not make sense for the data its being applied to, just go ahead and apply it to the (binary) machine representation of the data. In some cases this will do something harmful. In other cases it might even be useful.  
(Adopted, e.g., in C.)

**Dynamic checking:** At the point during execution at which a type mismatch (between operation and argument) is encountered, raise an error. This gives rise to helpful runtime errors.

**Static checking:** Check (the AST of) the program to ensure that all operations are applied in a type-meaningful way. If not, identify the error(s), and disallow the program from being run until corrected. This allows many program errors to be identified before execution.

## Approaches to type errors

**Laissez faire:** Even if an operation does not make sense for the data its being applied to, just go ahead and apply it to the (binary) machine representation of the data. In some cases this will do something harmful. In other cases it might even be useful.

(Adopted, e.g., in C.)

**Dynamic checking:** At the point during execution at which a type mismatch (between operation and argument) is encountered, raise an error. This gives rise to helpful runtime errors.

(Adopted, e.g., in Python.)

**Static checking:** Check (the AST of) the program to ensure that all operations are applied in a type-meaningful way. If not, identify the error(s), and disallow the program from being run until corrected. This allows many program errors to be identified before execution.

## Approaches to type errors

**Laissez faire:** Even if an operation does not make sense for the data its being applied to, just go ahead and apply it to the (binary) machine representation of the data. In some cases this will do something harmful. In other cases it might even be useful.

(Adopted, e.g., in C.)

**Dynamic checking:** At the point during execution at which a type mismatch (between operation and argument) is encountered, raise an error. This gives rise to helpful runtime errors.

(Adopted, e.g., in Python.)

**Static checking:** Check (the AST of) the program to ensure that all operations are applied in a type-meaningful way. If not, identify the error(s), and disallow the program from being run until corrected. This allows many program errors to be identified before execution. (Adopted, e.g., in Java and Haskell.)

In this lecture we look at static type-checking using a fragment of Haskell as the illustrative programming language.

We call the fragment of Haskell **Micro-Haskell** (**MH** for short).

MH is the basis of this year's Inf2A Assignment 1, which uses it to illustrate the full formal-language-processing pipeline.

For those who have never previously met Haskell or who could benefit from a Haskell refresher, we start with a gentle introduction to MH.

# Micro-Haskell: a crash course

In mathematics, we are used to defining functions via equations,  
e.g.  $f(x) = 3x + 7$ .

The idea in functional programming is that programs should look somewhat similar to mathematical definitions:

```
f x = x+x+x + 7 ;
```

This function expects an argument  $x$  of integer type (let's say), and returns a result of integer type. We therefore say the type of  $f$  is  $\text{Integer} \rightarrow \text{Integer}$  ("integer to integer").

By contrast, the definition

```
g x = x+x <= x+7 ;
```

returns a boolean result, so the type of  $g$  is  $\text{Integer} \rightarrow \text{Bool}$ .

## Multi-argument functions

What about a function of two arguments, say  $x :: \text{Integer}$  and  $y :: \text{Bool}$ ? E.g.

```
h x y = if y then x else -x ;
```

Think of  $h$  as a function that accepts arguments [one at a time](#). It accepts an integer and returns another function, which itself accepts a boolean and returns an integer.

So the type of  $h$  is  $\text{Integer} \rightarrow (\text{Bool} \rightarrow \text{Integer})$ . By convention, we treat  $\rightarrow$  as [right-associative](#), so we can write this just as  $\text{Integer} \rightarrow \text{Bool} \rightarrow \text{Integer}$ .

Note incidentally the use of 'if' to create [expressions](#) rather than commands. In Java, the above if-expression could be written as

$$(y ? x : -x)$$

# Typechecking in Micro-Haskell

In (Micro-)Haskell, the type of `h` is explicitly given as part of the function definition:

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else -x ;
```

The typechecker then checks that the expression on the RHS does indeed have type Integer, assuming `x` and `y` have the specified argument types Integer and Bool respectively.

Function definitions can also be **recursive**:

```
div :: Integer -> Integer -> Integer ;
div x y = if x < y then 0 else 1 + div (x + -y) y ;
```

Here the typechecker will check that the RHS has type Integer, assuming that `x` and `y` have type Integer and also that `div` itself has the stated type.

## Higher-order functions

The arguments of a function in MH can themselves be functions!

```
F :: (Integer -> Integer) -> Integer ;  
F g = g 0 + g 1 + g 2 + g 3;
```

The typechecker then checks that the expression on the RHS does indeed have type Integer, assuming x and y have the specified argument types Integer and Bool respectively.

For an example application of F, consider the following MH function.

```
inc :: Integer -> Integer ;  
inc x = x+1 ;
```

If we then type

```
F inc
```

into an evaluator (i.e., interpreter) for MH, the evaluator will compute that the result of the expression F inc is

## Higher-order functions

The arguments of a function in MH can themselves be functions!

```
F :: (Integer -> Integer) -> Integer ;  
F g = g 0 + g 1 + g 2 + g 3;
```

The typechecker then checks that the expression on the RHS does indeed have type Integer, assuming x and y have the specified argument types Integer and Bool respectively.

For an example application of F, consider the following MH function.

```
inc :: Integer -> Integer ;  
inc x = x+1 ;
```

If we then type

```
F inc
```

into an evaluator (i.e., interpreter) for MH, the evaluator will compute that the result of the expression F inc is 10.

In principle, the  $\rightarrow$  constructor can be iterated to produce very complex types, e.g.

$$(((\text{Integer} \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Integer}) \rightarrow \text{Integer}$$

Such monsters rarely arise in ordinary programs.

Nevertheless, MH (and full Haskell) has a precise way of checking whether the function definitions in the program correctly respect the types that have been assigned to them.

Before discussing this process, we summarize the types of MH.

## MH Types

The official grammar of MH types (in Assignment 1 handout) is

$$\begin{aligned} \textit{Type} &\rightarrow \textit{Type1} \textit{TypeOps} \\ \textit{TypeOps} &\rightarrow \epsilon \mid \rightarrow \textit{Type} \\ \textit{Type1} &\rightarrow \text{Integer} \mid \text{Bool} \mid (\textit{Type}) \end{aligned}$$

This is an LL(1) grammar for convenient parsing.

However, a parse tree for this grammar contains more detail than is required for understanding a type expression.

The following conceptually simpler grammar implements the **abstract syntax** of types

$$\textit{Type} \rightarrow \text{Integer} \mid \text{Bool} \mid \textit{Type} \rightarrow \textit{Type}$$

# Abstract Syntax Trees

The abstract syntax grammar is not appropriate for parsing:

- It is ambiguous
- It does not include all aspects of the concrete syntax. In particular, there are no brackets.

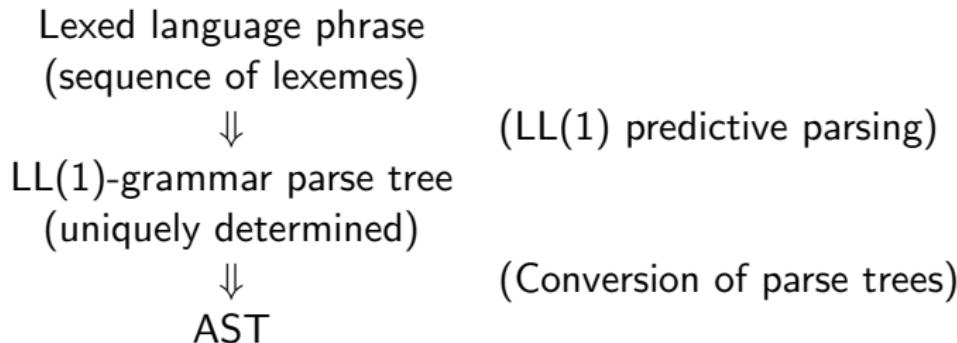
However **parse trees** for the abstract syntax grammar unambiguously correspond to types.

Instead of working with parse trees for the concrete LL(1) grammar, we convert such parse trees to parse trees for the abstract syntax grammar. Such parse trees are called **abstract syntax trees (AST)**.

## Concrete versus abstract syntax

The distinction between concrete and abstract syntax is not specific to types, but applies generally to formal and natural languages.

In the case of an LL(1)-predictively parsed formal languages, we have the following parsing pipeline:



# Type checking

Main ideas.

- ➊ Type checking is done **compositionally** by breaking down expressions into their subexpressions, type-checking the subexpressions, and ensuring that the top-level compound expression can then be given a type itself.
- ➋ Throughout the process, a **type environment** is maintained which records the types of all variables in the expression.

## Illustrative example

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 1+x ;
```

First the type environment  $\Gamma$  is set according to the type declaration.

$$\Gamma := h :: \text{Integer} \rightarrow \text{Bool} \rightarrow \text{Integer}$$

Next, the type environment is extended to assign types to the argument variables  $x$  and  $y$ .

$$\begin{aligned}\Gamma := & h :: \text{Integer} \rightarrow \text{Bool} \rightarrow \text{Integer}, \\ & x :: \text{Integer}, \\ & y :: \text{Bool}\end{aligned}$$

## Illustrative example (continued)

This is done in order to be consistent with the general rule

- In any expression  $e_1 e_2$  (a function application) we need  $e_1$  to have a function type  $t_1 \rightarrow t_2$  with  $e_2$  having the correct type  $t_1$  for its argument. The resulting type of  $e_1 e_2$  is then  $t_2$ .

Thus, in our example, we have types

$h\ x\ ::\ \text{Bool} \rightarrow \text{Integer}$

and

$h\ x\ y\ ::\ \text{Integer}$

## Illustrative example (continued)

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 1+x ;
```

We have

```
h x y  :: Integer
```

with the type environment

```
 $\Gamma := h :: \text{Integer} \rightarrow \text{Bool} \rightarrow \text{Integer},$ 
 $x :: \text{Integer},$ 
 $y :: \text{Bool}$ 
```

Our remaining task is to type-check (relative to  $\Gamma$ ) the **expression**:

```
if y then x else 1+x  :: Integer
```

## Illustrative example (continued)

General rule:

- In any expression `if e1 then e2 else e3` we need  $e_1$  to have type `Bool`, and  $e_2$  and  $e_3$  to have the same type  $t$ . The resulting type of `if e1 then e2 else e3` is then  $t$ .

In our example, we need to type-check

```
if y then x else 1+x :: Integer
```

we have  $y :: \text{Bool}$  and  $x :: \text{Integer}$  declared in  $\Gamma$ , so it remains only to type-check

```
1+x :: Integer
```

## Illustrative example (completed)

General rule:

- In any expression  $e_1 + e_2$  we need  $e_1$  and  $e_2$  to have type Integer. The resulting type of  $e_1 + e_2$  is then Integer.

In our example, we need to type-check

$$1+x :: \text{Integer}$$

we have  $x :: \text{Integer}$  declared in  $\Gamma$ , also the numeral 1 is (of course) given type Integer.

Thus indeed we have verified

$$1+x :: \text{Integer}$$

whence, putting everything together,

$$\text{if } y \text{ then } x \text{ else } 1+x :: \text{Integer}$$

as required.

## Static type checking — summary

The program is type-checked purely by looking at the AST of the program.

Thus type errors are picked up before the program is executed. Indeed, execution is disallowed for programs that do not type check.

Static type checking gives us a guarantee: **no type errors will occur during execution**.

This guarantee can be rigorously established as a mathematical theorem, using a mathematical model of program execution called **operational semantics**. Operational semantics lies at the heart of the Evaluator provided for Part D of Assignment 1. We shall meet operational semantics later in the course (Lecture 27).

# Fixing problems with grammars

## Informatics 2A: Lecture 13

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

22 October 2015

# LL(1) grammars: summary

Given a context-free grammar, the problem of **parsing** a string can be seen as that of constructing a **leftmost derivation**, e.g.

$$\begin{array}{lclclcl} \text{Exp} & \Rightarrow & \text{Exp} + \text{Exp} & \Rightarrow & \text{Num} + \text{Exp} & \Rightarrow & 1 + \text{Exp} \\ & \Rightarrow & 1 + \text{Num} & \Rightarrow & 1 + 2 & & \end{array}$$

At each stage, we expand the **leftmost nonterminal**. In general, it (seemingly) requires **magical powers** to know which rule to apply.

An **LL(1) grammar** is one in which the correct rule can always be determined from just the **nonterminal** to be expanded and the current **input symbol** (or end-of-input marker).

This leads to the idea of a **parse table**: a two-dimensional array (indexed by nonterminals and input symbols) in which the appropriate production can be looked up at each stage.

# Possible problems with grammars

LL(1) grammars allow for very **efficient parsing** (time linear in length of input string). Unfortunately, many “natural” grammars are not LL(1), for various reasons, e.g.

- ➊ They may be **ambiguous** (bad for computer languages)
- ➋ They may have rules with **shared prefixes**: e.g. how would we choose between the following productions?

Stmt → do Stmt while Cond

Stmt → do Stmt until Cond

- ➌ There may be **left-recursive rules**, where the LHS nonterminal appears at the start of the RHS: Exp → Exp + Exp

Sometimes such problems can be fixed: can replace our grammar by an equivalent LL(1) one. We'll look at ways of doing this.

# Problem 1: Ambiguity

We've seen many examples of **ambiguous** grammars. Some kinds of ambiguity are 'needless' and can be easily avoided. E.g. can replace

$$\text{List} \rightarrow \epsilon \mid \text{Item} \mid \text{List List}$$

by

$$\text{List} \rightarrow \epsilon \mid \text{Item List}$$

A similar trick works generally for any other kind of 'lists'.  
E.g. can replace

$$\text{List1} \rightarrow \text{Item} \mid \text{List1 ; List1}$$

by

$$\text{List1} \rightarrow \text{Item Rest} \quad \text{Rest} \rightarrow \epsilon \mid ; \text{Item Rest}$$

# Resolving ambiguity with added nonterminals

More serious example of ambiguity:

$$\begin{aligned} \text{Exp} \rightarrow & \text{ Num } | \text{ Var } | (\text{Exp}) | -\text{Exp} \\ & | \text{ Exp + Exp } | \text{ Exp - Exp } | \text{ Exp * Exp } \end{aligned}$$

We can disambiguate this by adding nonterminals to capture more subtle distinctions between different classes of expressions:

$$\begin{aligned} \text{Exp} \rightarrow & \text{ ExpA } | \text{ Exp + ExpA } | \text{ Exp - ExpA } \\ \text{ExpA} \rightarrow & \text{ ExpB } | \text{ ExpA * ExpB } \\ \text{ExpB} \rightarrow & \text{ ExpC } | -\text{ExpB} \\ \text{ExpC} \rightarrow & \text{ Num } | \text{ Var } | (\text{Exp}) \end{aligned}$$

Note that this builds in certain **design decisions** concerning what we want the rules of precedence to be.

N.B. our revised grammar is **unambiguous**, but not yet **LL(1)** ...

## Problem 2: Shared prefixes

Consider the two productions

$$\text{Stmt} \rightarrow \text{do Stmt while Cond}$$
$$\text{Stmt} \rightarrow \text{do Stmt until Cond}$$

On seeing the nonterminal Stmt and the terminal do, an LL(1) parser would have no way of choosing between these rules.

**Solution:** factor out the common part of these rules, so ‘delaying’ the decision until the relevant information becomes available:

$$\text{Stmt} \rightarrow \text{do Stmt Test}$$
$$\text{Test} \rightarrow \text{while Cond} \mid \text{until Cond}$$

This simple trick is known as **left factoring**.

## Problem 3: Left recursion

Suppose our grammar contains a rule like

$$\text{Exp} \rightarrow \text{Exp} + \text{ExpA}$$

**Problem:** whatever terminals Exp could begin with, Exp + ExpA could also begin with. So there's a danger our parser would apply this rule indefinitely:

$$\text{Exp} \Rightarrow \text{Exp} + \text{ExpA} \Rightarrow \text{Exp} + \text{ExpA} + \text{ExpA} \Rightarrow \dots$$

(In practice, we wouldn't even get this far: there'd be a clash in the parse table, e.g. at Num, Exp.)

So **left recursion** makes a grammar non-LL(1).

# Eliminating left recursion

Consider e.g. the rules

$$\text{Exp} \rightarrow \text{ExpA} \mid \text{Exp} + \text{ExpA} \mid \text{Exp} - \text{ExpA}$$

Taken together, these say that Exp can consist of ExpA followed by zero or more suffixes + ExpA or - ExpA.

So we just need to formalize this!

$$\text{Exp} \rightarrow \text{ExpA OpsA} \quad \text{OpsA} \rightarrow \epsilon \mid + \text{ExpA OpsA} \mid - \text{ExpA OpsA}$$

(Reminiscent of [Arden's rule](#).) Likewise:

$$\text{ExpA} \rightarrow \text{ExpB OpsB} \quad \text{OpsB} \rightarrow \epsilon \mid * \text{ExpB OpsB}$$

Together with the earlier rules for ExpB and ExpC, these give an **LL(1)** version of the grammar for arithmetic expressions on slide 5.

# The resulting LL(1) grammar

Exp  $\rightarrow$  ExpA OpsA  
OpsA  $\rightarrow$   $\epsilon$  | + ExpA OpsA | - ExpA OpsA  
ExpA  $\rightarrow$  ExpB OpsB  
OpsB  $\rightarrow$   $\epsilon$  | \* ExpB OpsB  
ExpB  $\rightarrow$  ExpC | - ExpB  
ExpC  $\rightarrow$  Num | Var | (Exp)

# Indirect left recursion

Left recursion can also arise in a more indirect way. E.g.

$$A \rightarrow a \mid Bc \qquad B \rightarrow b \mid Ad$$

By considering the combined effect of these rules, can see that they are equivalent to the following LL(1) grammar.

$$\begin{array}{ll} A \rightarrow aE \mid bcE & B \rightarrow bF \mid adF \\ E \rightarrow \epsilon \mid dcE & F \rightarrow \epsilon \mid cdF \end{array}$$

(Won't go into the systematic method here.)

# LL(1) grammars: summary

- Often (not always), a “natural” grammar for some language of interest can be massaged into an LL(1) grammar. This allows for very efficient parsing.
- Knowing a grammar is LL(1) also assures us that it is **unambiguous** — often non-trivial! By the same token, LL(1) grammars are poorly suited to **natural languages**.
- However, an LL(1) grammar may be less readable and intuitive than the original. It may also appear to mutilate the ‘natural’ structure of phrases. We must take care not to mutilate it so much that we can no longer ‘execute’ the phrase as intended.
- One can design realistic computer languages with LL(1) grammars. For less cumbersome syntax that ‘flows’ better, one might want to go a bit beyond LL(1) (e.g. to LR(1)), but the principles remain the same.

# Example of an LL(1) grammar

Here is the **repaired** programming language grammar from Lecture 8, as hinted at in Lecture 10. Combining it with our revised grammar for arithmetic expressions, we get an LL(1) grammar for a respectable programming language.

```
stmt    → if-stmt | while-stmt | begin-stmt | assg-stmt  
if-stmt → if bool-expr then stmt else stmt  
while-stmt → while bool-expr do stmt  
begin-stmt → begin stmt-list end  
stmt-list → stmt stmts  
stmts   →  $\epsilon$  | ; stmt stmts  
assg-stmt → VAR := arith-expr  
bool-expr → arith-expr compare-op arith-expr  
compare-op → < | > | <= | >= | == | != =
```

## Final topic: Chomsky Normal Form

Whilst on the subject of ‘transforming grammars into equivalent ones of some special kind’ ...

A context-free grammar  $\mathcal{G} = (N, \Sigma, P, S)$  is in **Chomsky normal form (CNF)** if all productions are of the form

$$A \rightarrow BC \quad \text{or} \quad A \rightarrow a \quad (A, B, C \in N, a \in \Sigma)$$

Theorem: Disregarding the empty string, every CFG  $\mathcal{G}$  is equivalent to a grammar  $\mathcal{G}'$  in Chomsky normal form. ( $\mathcal{L}(\mathcal{G}') = \mathcal{L}(\mathcal{G}) - \{\epsilon\}$ )

This is useful, because certain general parsing algorithms (e.g. the **CYK algorithm**, see Lecture 17) work best for grammars in CNF.

# Converting to Chomsky Normal Form

Consider for example the grammar

$$S \rightarrow TT \mid [S] \quad T \rightarrow \epsilon \mid (T)$$

**Step 1:** remove all  $\epsilon$ -productions, and for each rule  $X \rightarrow \alpha Y \beta$ , add a new rule  $X \rightarrow \alpha \beta$  whenever  $Y$  'can be empty'.

$$S \rightarrow TT \mid T \mid [S] \mid [] \quad T \rightarrow (T) \mid ()$$

**Step 2:** remove 'unit productions'  $X \rightarrow Y$ .

$$S \rightarrow TT \mid (T) \mid () \mid [S] \mid [] \quad T \rightarrow (T) \mid ()$$

Now all productions are of form  $X \rightarrow a$  or  $X \rightarrow x_1 \dots x_k$  ( $k \geq 2$ ).

# Converting to Chomsky Normal Form, ctd.

$$S \rightarrow TT \mid (T) \mid () \mid [S] \mid [] \quad T \rightarrow (T) \mid ()$$

**Step 3:** For each terminal  $a$ , add a nonterminal  $Z_a$  and a production  $Z_a \rightarrow a$ . In all rules  $X \rightarrow x_1 \dots x_k$  ( $k \geq 2$ ), replace each  $a$  by  $Z_a$ .

$$S \rightarrow TT \mid Z_{(} TZ_{)} \mid Z_{(} Z_{)} \mid Z_{[} SZ_{]} \mid Z_{[} Z_{]}$$

$$T \rightarrow Z_{(} TZ_{)} \mid Z_{(} Z_{)} \quad Z_{(} \rightarrow ( \quad Z_{)} \rightarrow ) \quad Z_{[} \rightarrow [ \quad Z_{]} \rightarrow ]$$

**Step 4:** For every production  $X \rightarrow Y_1 \dots Y_n$  with  $n \geq 3$ , add new symbols  $W_2, \dots, W_{n-1}$  and replace the production with  $X \rightarrow Y_1 W_2, \quad W_2 \rightarrow Y_2 W_3, \quad \dots, \quad W_{n-1} \rightarrow Y_{n-1} Y_n$ .

E.g.  $S \rightarrow Z_{(} TZ_{)} \mid Z_{[} SZ_{]}$  become

$$S \rightarrow Z_{(} W \quad W \rightarrow TZ_{)} \quad S \rightarrow Z_{[} V \quad V \rightarrow SZ_{]}$$

The resulting grammar is now in **Chomsky Normal Form**.

## Self-assessment question on context-free grammars

Consider the alphabet of ASCII characters. Let N be the lexical class of all non-alphabetic characters. Consider the following context-free grammar for a nonterminal P.

$$\begin{array}{l} P \rightarrow \epsilon \mid NP \mid PN \\ P \rightarrow a \mid aPa \mid aPA \mid APa \mid APA \mid A \\ P \rightarrow b \mid bPb \mid bPB \mid B Pb \mid B PB \mid B \\ \dots \quad \text{(23 similar lines for 'C' to 'Y')} \\ P \rightarrow z \mid zPz \mid zPZ \mid ZPz \mid ZPZ \mid Z \end{array}$$

Which of the following ASCII strings can be parsed as a P?

- ① never odd or even
- ② "Norma is as selfless as I am, Ron."
- ③ Live dirt up a side-track carted is a putrid evil.
- ④ I made reviled tubs repel; no, it is opposition, lepers, but delivered am I.

# More questions

Our grammar generates **palindromic strings**:

$$P \rightarrow \epsilon \mid N P \mid P N$$

$$P \rightarrow a \mid a P a \mid a P A \mid A P a \mid A P A \mid A$$

$$P \rightarrow b \mid b P b \mid b P B \mid B P b \mid B P B \mid B$$

... (23 similar lines for 'C' to 'Y')

$$P \rightarrow z \mid z P z \mid z P Z \mid Z P z \mid Z P Z \mid Z$$

Q. (self-assessment): Is this grammar LL(1)?

- 1 Yes.
- 2 No.
- 3 Don't know.

# More questions

Our grammar generates **palindromic strings**:

$$P \rightarrow \epsilon \mid N P \mid P N$$

$$P \rightarrow a \mid a P a \mid a P A \mid A P a \mid A P A \mid A$$

$$P \rightarrow b \mid b P b \mid b P B \mid B P b \mid B P B \mid B$$

... (23 similar lines for 'C' to 'Y')

$$P \rightarrow z \mid z P z \mid z P Z \mid Z P z \mid Z P Z \mid Z$$

Q. (self-assessment): Is this grammar LL(1)?

- 1 Yes.
- 2 No.
- 3 Don't know.

Q. (challenge) : Is it possible to provide an LL(1) grammar for the language of palindromes?

## Some light relief: Palindromic sentences

Our grammar recognises palindromic alphabetic strings, ignoring whitespace, punctuation, case distinctions, etc.

It is not too hard to construct such strings consisting entirely of English words. However, it's more satisfying to find examples that are coherent or interesting in some other way.

A famous example:

*A man, a plan, a canal — Panama!*

## Some light relief: Palindromic sentences

Our grammar recognises palindromic alphabetic strings, ignoring whitespace, punctuation, case distinctions, etc.

It is not too hard to construct such strings consisting entirely of English words. However, it's more satisfying to find examples that are coherent or interesting in some other way.

A famous example:

*A man, a plan, a canal — Panama!*

... which some smart aleck noticed could be tweaked to ...

*A dog, a plan, a canal — Pagoda!*

## Some light relief: Palindromic sentences

Our grammar recognises palindromic alphabetic strings, ignoring whitespace, punctuation, case distinctions, etc.

It is not too hard to construct such strings consisting entirely of English words. However, it's more satisfying to find examples that are coherent or interesting in some other way.

A famous example:

*A man, a plan, a canal — Panama!*

... which some smart aleck noticed could be tweaked to ...

*A dog, a plan, a canal — Pagoda!*

But probably there is nothing to equal ...

# Best English palindrome in the world?

(From Guy Steele, *Common Lisp Reference Manual*, 1983.)

*A man, a plan, a canoe, pasta, heros, rajahs, a  
coloratura, maps, snipe, percale, macaroni, a gag, a  
banana bag, a tan, a tag, a banana bag again (or a  
camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar,  
sore hats, a peon, a canal — Panama!*

# Reading

- [Making grammars LL\(1\)](#): former lecture notes available via the Course Schedule webpage.
- [Chomsky Normal Form](#): Kozen chapter 21, Jurafsky & Martin section 12.5.

# Morphology parsing

## Informatics 2A: Lecture 14

Shay Cohen

School of Informatics  
University of Edinburgh  
[scohen@inf.ed.ac.uk](mailto:scohen@inf.ed.ac.uk)

23 October 2015

- 1 Morphology parsing: the problem
- 2 Finite-state transducers
- 3 Finite-state transducers
- 4 FSTs for morphology parsing and generation

(This lecture is based on Jurafsky & Martin chapter 3, sections 1–7.)

# Morphology in other languages

Morphology is the study of the *structure of words*

English has relatively impoverished morphology.

Languages with rich morphology: Turkish, Arabic, Hungarian, Korean, and many more (actually, English is rather unique in having relatively simple morphology)

For example, Turkish is an *agglutinative* language, and words are constructed by concatenating *morphemes* together without changing them much:

[evlerinizden](#): “from your houses” (morphemes: ev-ler-iniz-den)

This lecture will mostly discuss how to build an English morphological analyser.

Stems and affixes (prefixes, suffixes, infixes and circumfixes) combine together

Four methods to combine them:

- Inflection (stem + grammar affix) - word does not change its grammatical class (*walk* → *walking*)
- Derivation (stem + grammar affix) - word changes its grammatical form (*computerize* → *computerization*)
- Compounding (stems together) - *doghouse*
- Cliticization - *I've*, *we're*, *he's*

Morphology can be concatenative or non-concatenative (e.g. templatic morphology as in Arabic)

# Morphological parsing: the problem

English has concatenative morphology. Words can be made up of a main **stem** (carrying the basic dictionary meaning) plus one or more **affixes** carrying grammatical information. E.g.:

Surface form:	cats	walking	smoohest
Lexical form:	cat+N+PL	walk+V+PresPart	smooth+Adj+Sup

**Morphological parsing** is the problem of extracting the lexical form from the surface form. (For speech processing, this includes identifying the word boundaries.)

We should take account of:

- Irregular forms (e.g. goose → geese)
- Systematic rules (e.g. 'e' inserted before suffix 's' after s,x,z,ch,sh: fox → foxes, watch → watches)

## Why bother?

- Any NLP tasks involving **grammatical parsing** will typically involve morphology parsing as a prerequisite.
- **Search engines:** e.g. a search for 'fox' should return documents containing 'foxes', and vice versa.
- Even a humble task like **spell checking** can benefit: e.g. is 'walking' a possible word form?

But why not just list all derived forms separately in our wordlist (e.g. walk, walks, walked, walking)?

- Might be OK for English, but not for a morphologically rich language — e.g. in Turkish, can pile up to 10 suffixes on a verb stem, leading to 40,000 possible forms for some verbs!
- Even for English, morphological parsing makes adding/learning new words easier.
- In speech processing, word breaks aren't known in advance.

# How expressive is morphology?

Morphemes are tacked together in a rather “regular” way.

This means that finite-state machines are a good way to model morphology. There is no need for “unbounded memory” to model it (there are no long range dependencies).

This is as opposed to syntax, the study of the order of words in a sentence, which we will learn about in another lecture

# Parsing and generation

**Parsing** here means going from the surface to the lexical form.

E.g. foxes → fox +N +PL.

**Generation** is the opposite process: fox +N +PL → foxes. It's helpful to consider these two processes together.

Either way, it's often useful to proceed via an intermediate form, corresponding to an analysis in terms of **morphemes** (= minimal meaningful units) before **orthographic rules** are applied.

Surface form:        foxes

Intermediate form: fox <sup>^</sup> s #

Lexical form:        fox +N +PL

(<sup>^</sup> means morpheme boundary, # means word boundary.)

N.B. The translation between surface and intermediate form is exactly the same if 'foxes' is a 3rd person singular verb!

## Finite-state transducers

We can consider  $\epsilon$ -NFAs (over an alphabet  $\Sigma$ ) in which transitions may also (optionally) produce *output* symbols (over a possibly different alphabet  $\Pi$ ).

E.g. consider the following machine with input alphabet  $\{a, b\}$  and output alphabet  $\{0, 1\}$ :

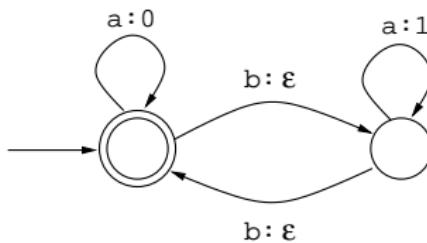
Such a thing is called a **finite state transducer**.

In effect, it specifies a (possibly multi-valued) translation from one regular language to another.

# Finite-state transducers

We can consider  $\epsilon$ -NFAs (over an alphabet  $\Sigma$ ) in which transitions may also (optionally) produce *output symbols* (over a possibly different alphabet  $\Pi$ ).

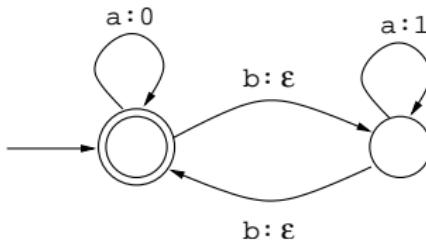
E.g. consider the following machine with input alphabet  $\{a, b\}$  and output alphabet  $\{0, 1\}$ :



Such a thing is called a **finite state transducer**.

In effect, it specifies a (possibly multi-valued) translation from one regular language to another.

## Quick exercise



What output will this produce, given the input *aabaaaabbab*?

- ① 001110
- ② 001111
- ③ 0011101
- ④ More than one output is possible.

Formally, a **finite state transducer**  $T$  with inputs from  $\Sigma$  and outputs from  $\Pi$  consists of:

- sets  $Q, S, F$  as in ordinary NFAs,
- a transition relation  $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Pi \cup \{\epsilon\}) \times Q$

From this, one can define a many-step transition relation

$\hat{\Delta} \subseteq Q \times \Sigma^* \times \Pi^* \times Q$ , where  $(q, x, y, q') \in \hat{\Delta}$  means “starting from state  $q$ , the input string  $x$  can be translated into the output string  $y$ , ending up in state  $q'$ .“ (Details omitted.)

Note that a finite state transducer can be run in either direction!

From  $T$  as above, we can obtain another transducer  $\bar{T}$  just by swapping the roles of inputs and outputs.

Formally, a **finite state transducer**  $T$  with inputs from  $\Sigma$  and outputs from  $\Pi$  consists of:

- sets  $Q$ ,  $S$ ,  $F$  as in ordinary NFAs,
- a transition relation  $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Pi \cup \{\epsilon\}) \times Q$

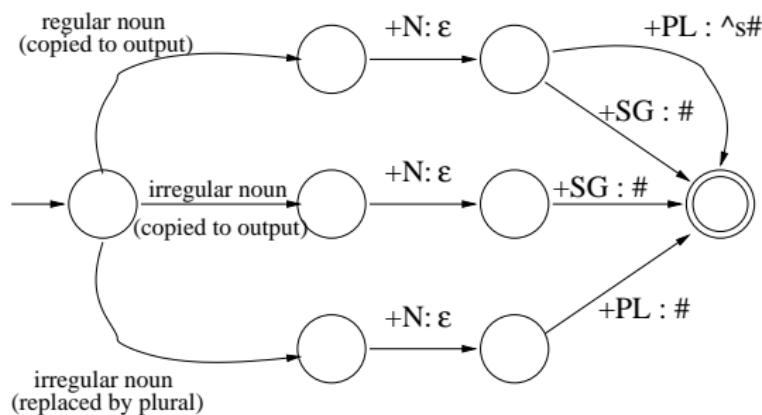
**Reminder:** Formally, an NFA with alphabet  $\Sigma$  consists of:

- A finite set  $Q$  of states.
- A transition relation  $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ ,
- A set  $S \subseteq Q$  of possible starting states,
- A set  $F \subseteq Q$  of accepting states.

## Stage 1: From lexical to intermediate form

Consider the problem of translating a lexical form like 'fox+N+PL' into an intermediate form like 'fox ^ s #', taking account of irregular forms like goose/geese.

We can do this with a transducer of the following schematic form:

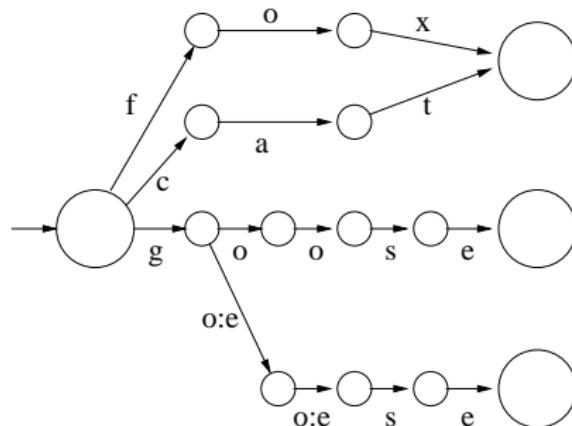


We treat each of  $+N$ ,  $+SG$ ,  $+PL$  as a single symbol.

The 'transition' labelled  $+PL : ^s\#$  abbreviates three transitions:  
 $+PL : ^$ ,  $\epsilon : s$ ,  $\epsilon : \#$ .

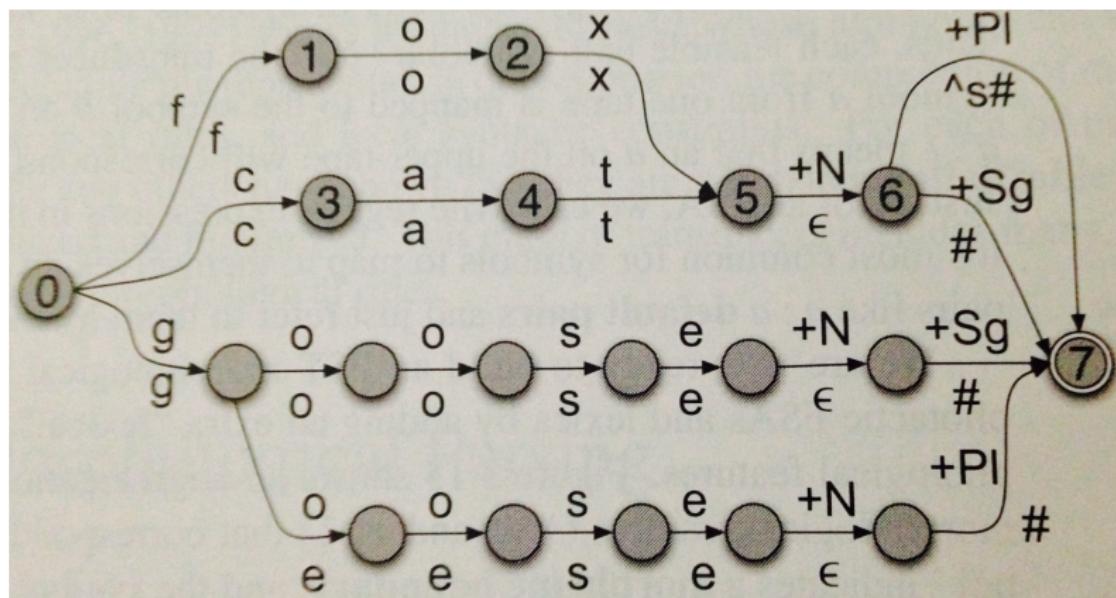
# The Stage 1 transducer fleshed out

The left hand part of the preceding diagram is an abbreviation for something like this (only a small sample shown):



Here, for simplicity, a single label  $u$  abbreviates  $u : u$ .

# Stage 1 in full



## Stage 2: From intermediate to surface form

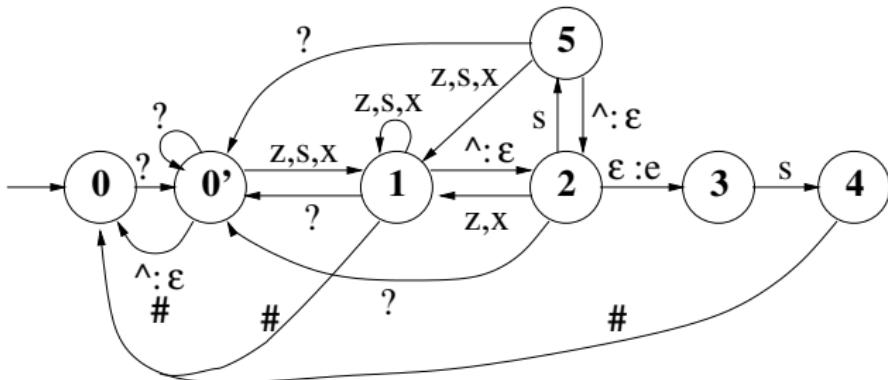
To convert a sequence of morphemes to surface form, we apply a number of **orthographic rules** such as the following.

- **E-insertion:** Insert e after s,z,x,ch,sh before a word-final morpheme -s. (fox → foxes)
- **E-deletion:** Delete e before a suffix beginning with e,i. (love → loving)
- **Consonant doubling:** Single consonants b,s,g,k,l,m,n,p,r,s,t,v are doubled before suffix -ed or -ing. (beg → begged)

We shall consider a simplified form of E-insertion, ignoring ch,sh.

(Note that this rule is oblivious to whether -s is a plural noun suffix or a 3rd person verb suffix.)

# A transducer for E-insertion (adapted from J+M)



Here ? may stand for any symbol except z,s,x, ^, #.  
(Treat # as a 'visible space character'.)

At a morpheme boundary following z,s,x, we arrive in State 2.  
If the ensuing input sequence is s#, our only option is to go via states 3 and 4. Note that there's no # -transition out of State 5.

State 5 allows e.g. 'ex^service^men#' to be translated to  
'exservicemen'.

## Putting it all together

FSTs can be **cascaded**: output from one can be input to another.

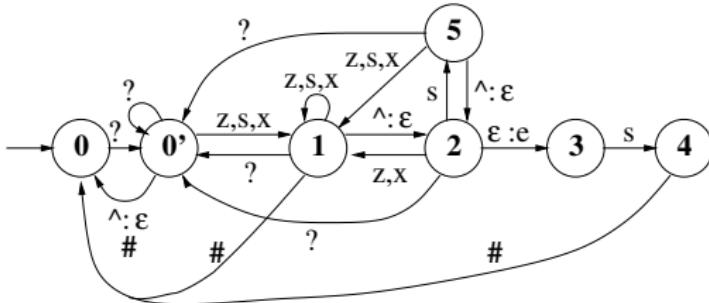
To go from lexical to surface form, use 'Stage 1' transducer followed by a bunch of orthographic rule transducers like the above. (Made more efficient by back-end compilation into one single transducer.)

The results of this **generation** process are typically **deterministic** (each lexical form gives a unique surface form), even though our transducers make use of non-determinism along the way.

Running the same cascade **backwards** lets us do **parsing** (surface to lexical form). Because of ambiguity, this process is frequently **non-deterministic**: e.g. 'foxes' might be analysed as fox+N+PL or fox+V+Pres+3SG.

Such ambiguities are not resolved by morphological parsing itself: left to a later processing stage.

## Quick exercise 2

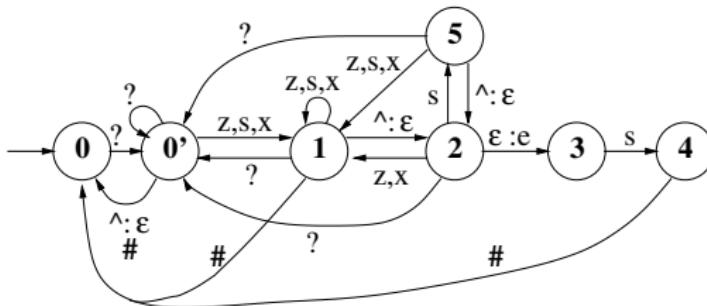


Apply this **backwards** to translate from surface to int. form.

Starting from state 0, how many **sequences of transitions** are compatible with the input string 'asses' ?

- 1
- 2
- 3
- 4
- More than 4

# Solution



On the input string 'asses', 10 transition sequences are possible!

- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$ , output ass $\hat{s}$ s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$ , output ass $\hat{e}$ s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{e} 0' \xrightarrow{s} 1$ , output asses
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$ , output as $\hat{s}$  $\hat{s}$ s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$ , output as $\hat{s}$  $\hat{e}$ s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{e} 0' \xrightarrow{s} 1$ , output as $\hat{e}$ ses
- Four of these can also be followed by  $1 \xrightarrow{\epsilon} 2$  (output  $\wedge$ ).

# The Porter Stemmer

Lexicon can be quite large with finite state transducers

Sometimes need to extract the stem in a very efficient fashion  
(such as in IR)

The Porter stemmer: a lexicon-free method for getting the stem of a given word

ATIONAL → ATE (e.g., relation → relate)

ING →  $\epsilon$  if stem contains a vowel (e.g. motoring → motor)

SSES → SS (e.g., grasses → grass)

Makes errors:

organization → organ

doing → doe

numerical → numerous

policy → police

A vibrant area of study

Mostly done by *learning from data*, just like many other NLP problems. Two main paradigms: unsupervised morphological parsing and supervised one.

NLP solvers are not perfect! They can make mistakes. Sometimes ambiguity can't even be resolved. BUT, for English, morphological analysis is highly accurate. With other languages, there is still a long way to go.

One of the basic tools that is used for many applications

- Speech recognition
- Machine translation
- Part-of-speech tagging
- ... and many more

### Part-of-speech tagging:

- What are parts of speech?
- What are they useful for?
- Zipf's law and the ambiguity of POS tagging
- One problem NLP solves really well (... for English)

# Parts-of-speech and the Lexicon in Natural Language

Informatics 2A: Lecture 15

Shay Cohen

School of Informatics  
University of Edinburgh

27 October 2015

We discussed morphological analysis (parsing, generation and recognition).

We described a finite-state transducer for analysing the morphological properties of nouns

This FST is a cascade of two FSTs: one for translating a string from an *lexical form* to an *intermediate form*, and one for translating a string from an intermediate form to a *surface form*

We described other ways of doing morphological analysis, such as using the Porter stemmer

- 1 Word classes and POS tags
- 2 Some specific word classes
- 3 Lexical ambiguity and word frequency

**Reading:** Jurafsky & Martin, Chapter 5.

Linguists have been classifying words for a long time ...

- Dionysius Thrax of Alexandria (c. 100 BC) wrote a grammatical sketch of Greek involving 8 parts-of-speech:

nouns	verbs	pronouns	prepositions
adverbs	conjunctions	participles	articles

- Thrax's list and minor variations on it dominated European language grammars and dictionaries for 2000 years.
- (Anyone sees an important POS missing?)

## Modern tagsets

In modern (English) NLP, larger (and more fine-grained) tagsets are preferred. E.g.

Penn Treebank	45 tags	<a href="http://bit.ly/1gwbird">http://bit.ly/1gwbird</a>
Brown corpus	87 tags	<a href="http://bit.ly/1jG9i2P">http://bit.ly/1jG9i2P</a>
C7 tagset	146 tags	<a href="http://bit.ly/1Mh36KX">http://bit.ly/1Mh36KX</a>

Trade-off between complexity and precision . . . and whatever tagset we use, there'll be some words that are hard to classify.

**Why do we need so many tags? We will see soon.**

## Distributional equivalence

Recall that for prog langs, a parser typically works entirely with tags produced by the lexer (e.g. IDENT, NUM). It won't care whether an identifier is x or y, or whether a numeral is 0 or 5.

**Consequence:** x and y have the same *distribution*: x can occur wherever y can, and vice versa.

The idea of POS tags is much the same: group the words of a language into classes of words with the same (or similar) distributions. E.g. the words

crocodile      pencil      mistake

are very different as regards meaning, but grammatically can occur in the same contexts. So let's classify them all as **nouns**.  
(More specifically, as *singular, countable, common nouns*.)

# Criteria for classifying words

When should words be put into the same class?

Three different criteria might be considered . . .

- **Distributional** criteria: Where can the words occur?
- **Morphological** criteria: What form does the word have? (E.g. -tion, -ize). What affixes can it take? (E.g. -s, -ing, -est).
- **Notional** (or semantic) criteria: What sort of concept does the word refer to? (E.g. nouns often refer to 'people, places or things'). More problematic: less useful for us.

We'll look at various parts-of-speech in terms of these criteria.

# Open and closed classes in natural language

There's a broad distinction between **open** and **closed** word classes:

- **Open classes** are typically large, have fluid membership, and are often stable under translation.
- Four major open classes are widely found in languages worldwide: *nouns*, *verbs*, *adjectives*, *adverbs*.
  - Virtually all languages have at least the first two.
  - All Indo-European languages (e.g. English) have all four.
- **Closed classes** are typically small, have relatively fixed membership, and the repertoire of classes varies widely between languages. E.g. *prepositions* (English, German), *post-positions* (Hungarian, Urdu, Korean), *particles* (Japanese), *classifiers* (Chinese), etc.
- Closed-class words (e.g. **of**, **which**, **could**) often play a structural role in the grammar as **function words**.

# Nouns

**Notionally**, nouns generally refer to living things (*mouse*), places (*Scotland*), non-living things (*harpoon*), or concepts (*marriage*).

**Formally**, *-ness*, *-tion*, *-ity*, and *-ance* tend to indicate nouns. (*happiness*, *exertion*, *levity*, *significance*).

**Distributionally**, we can examine the contexts where a noun appears and other words that appear in the same contexts. For example, nouns can appear with possession: "his car", "her idea".

**Notionally**, verbs refer to actions (*observe, think, give*).

**Formally**, words that end in *-ate* or *-ize* tend to be verbs, and ones that end in *-ing* are often the present participle of a verb (*automate, calibrate, equalize, modernize; rising, washing, grooming*).

**Distributionally**, we can examine the contexts where a verb appears and at other words that appear in the same contexts, which may include their arguments.

Different types of verbs have different distributional properties. For example, base form verbs can appear as infinitives: "to jump", "to learn".

# Example of noun and verb classes

## Nouns:

- Proper nouns: names such as Regina, IBM, Edinburgh
- Pronouns: he, she, it, they, we
- Common nouns
  - Count nouns: e.g. goat
  - Mass nouns: e.g. snow (? snows)

**Verbs** can be in base form, past tense, gerund... Also, consider auxiliary verbs.

## Why do we need so many tags?

What is the part of speech tag for “walking”? Use **linguistic tests**.

# Why do we need so many tags?

What is the part of speech tag for “walking”? Use **linguistic tests**.

Verb tests:

## Example

Walking quickly is awkward

Quickly walking is awkward

# Why do we need so many tags?

What is the part of speech tag for “walking”? Use **linguistic tests**.

Verb tests:

## Example

Walking quickly is awkward

Quickly walking is awkward

Noun tests:

## Example

Walking is awkward

Her walking is awkward

Fast walking is awkward

# Why do we need so many tags?

What is the part of speech tag for “walking”? Use **linguistic tests**.

Verb tests:

## Example

Walking quickly is awkward

Quickly walking is awkward

Noun tests:

## Example

Walking is awkward

Her walking is awkward

Fast walking is awkward

“Walking” has both properties of both noun and verb. A separate tag for gerunds?

**Notionally**, adjectives convey properties of or opinions about things that are nouns (*small, wee, sensible, excellent*).

**Formally**, words that end in *-al, -ble, and -ous* tend to be adjectives (*formal, gradual, sensible, salubrious, parlous*)

**Distributionally**, adjectives usually appear before a noun or after a form of *be*.

**Notionally**, adverbs convey properties of or opinions about actions or events (*quickly, often, possibly, unfortunately*) or adjectives (*really*).

**Formally**, words that end in *-ly* tend to be adverbs.

**Distributionally**, adverbs can appear next to a verb, or an adjective, or at the start of a sentence.

## Other classes (closed)

- **prepositions:** on, under, over, near, by, at, from, to, with
- **determiners:** a, an, the
- **conjunctions:** and, but, or, as, if, when
- **particles:** up, down, on, off, in, out, at, by
- **numerals:** one, two, three, first, second, third

# Importance of formal and distributional criteria

Often in reading, we come across **unknown words**. (Especially in computing literature!)

bootloader, distros, whitelist, diskdrak, borked  
(<http://www.linux.com/feature/150441>)  
revved, femtosecond, dogfooding  
(<http://hardware.slashdot.org/>)

Even if we don't know its meaning, formal and distributional criteria help people (and machines) recognize which (open) class an unknown word belongs to.

I really wish mandriva would redesign the diskdrak UI. The orphan bit is borked.

## Example of POS inference

Those **zorls** you **splarded** were **malgy**.

What is the part of speech of the word **malgy**?

- ① adverb
- ② noun
- ③ verb
- ④ adjective

## Example of POS inference

The highly-valued share plummeted over the course of the busy week .

Can you decide on the tags of each word?

## Example of POS inference

The highly-valued share plummeted over the course of the busy week .

Can you decide on the tags of each word?

The/ highly-valued/ share/ plummeted/ over/ the/  
course/ of/ the/ busy/ week/ .

## Example of POS inference

The highly-valued share plummeted over the course of the busy week .

Can you decide on the tags of each word?

The/DT highly-valued/JJ share/NN plummeted/VBD over/IN  
the/DT course/NN of/IN the/DT busy/JJ week/NN ./.

# The tagging problem

Given an input text, we want to tag each word correctly:

The/DT grand/JJ jury/NN commented/VBD on/IN a/DT  
number/NN of/IN other/JJ topics/NNS ./.

There/EX was/VBD still/JJ lemonade/NN in/IN the/DT  
bottle/NN ./.

(Many Brown/Penn tags are quite counterintuitive!)

- In the above, **number** and **bottle** are nouns not verbs — but how does our tagger tell?
- In the second example, **still** could be an adjective or an adverb — which seems more likely?

These issues lead us to consider **word frequencies** (among other things).

# Types of Lexical Ambiguity

Part of Speech (PoS) Ambiguity: e.g., *still*:

- ① *adverb*: at present, as yet
- ② *noun*: (1) silence; (2) individual frame from a film; (3) vessel for distilling alcohol
- ③ *adjective*: motionless, quiet
- ④ *transitive verb*: to calm

Sense Ambiguity: e.g., *intelligence*:

- ① Power of understanding
- ② Obtaining or dispersing secret information; also the persons engaged in obtaining or dispersing secret information

# Word Frequencies in Different Languages

Ambiguity by part-of-speech tags:

Language	Type-ambiguous	Token-ambiguous
English	13.2%	56.2%
Greek	<1%	19.14%
Japanese	7.6%	50.2%
Czech	<1%	14.5%
Turkish	2.5%	35.2%

Taken from real data for treebanks annotated with their POS tags

# Word Frequency – Properties of Words in Use

Take any corpus of English like the **Brown Corpus** or **Tom Sawyer** and sort its words by how often they occur.

word	Freq. ( $f$ )	Rank ( $r$ )	$f \cdot r$
the	3332	1	3332
and	2972	2	5944
a	1775	3	5235
he	877	10	8770
but	410	20	8400
be	294	30	8820
there	222	40	8880
one	172	50	8600
about	158	60	9480
more	138	70	9660
never	124	80	9920
Oh	116	90	10440

## Word Frequency – Properties of Words in Use

Take any corpus of English like the **Brown Corpus** or **Tom Sawyer** and sort its words by how often they occur.

word	Freq. ( $f$ )	Rank ( $r$ )	$f \cdot r$
two	104	100	10400
turned	51	200	10200
you'll	30	300	9000
name	21	400	8400
comes	16	500	8000
group	13	600	7800
lead	11	700	7700
friends	10	800	8000
begin	9	900	8100
family	8	1000	8000
brushed	4	2000	8000
sins	2	3000	6000

# Zipf's law

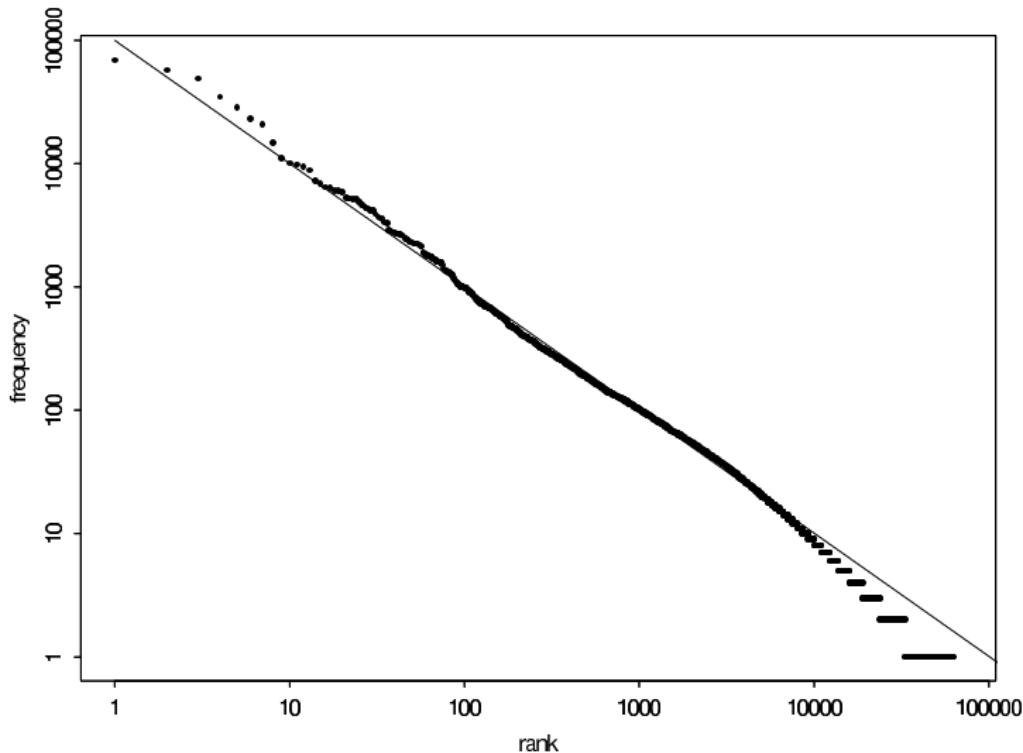
Given some corpus of natural language utterances, the **frequency** of any word is inversely proportional to its **rank in** the frequency table (observation made by Harvard linguist George Kingsley Zipf).

Zipf's law states that:  $f \propto \frac{1}{r}$

There is a constant  $k$  such that:  $f \cdot r = k$ .



# Zipf's law for the Brown corpus



According to Zipf's law:

- There is a very small number of very common words.
- There is a small-medium number of middle frequency words.
- There is a very large number of words that are infrequent.

(It's not fully understood why Zipf's law works so well for word frequencies.)

In fact, many other kinds of data conform closely to a **Zipfian distribution**:

- Populations of cities.
- Sizes of earthquakes.
- Amazon sales rankings.

Old POS taggers used to work in two stages, based on hand-written rules: the first stage identifies a set of possible POS for each word in the sentence (based on a lexicon), and the second uses a set of hand-crafted rules in order to select a POS from each of the lists for each word.

Example:

- ① If a word belongs to the set of determiners, tag is at DT.
- ② Tag a word next to a determiner as an adjective (JJ), if it ends with -ed.
- ③ If a word appears after is or are and it ends with ing, tag it as a verb VBG.

## Why do we need POS tags?

- They are often an essential ingredient in natural language applications
- Usually appear at the “bottom” of the pipeline
- For example: most of the syntactic variability (we will learn about that later) is determined by the sequence of POS tags in a sentence. POS tags are easier to predict than the full syntax, and therefore, by predicting the POS tags, we pave the way for identification of full phrases: noun phrases, verb phrases, etc.

## Extreme POS ambiguity...

**Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo**

## Extreme POS ambiguity...

**Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo**

Bison from Buffalo, which bison from Buffalo bully, themselves  
bully bison from Buffalo.

**Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo**

Bison from Buffalo, which bison from Buffalo bully, themselves  
bully bison from Buffalo.

**If police police police police, who police police police? Police  
police police police police police**

Look it up!

# Part-of-Speech Tagging

## Informatics 2A: Lecture 16

Shay Cohen

School of Informatics  
University of Edinburgh

29 October 2015

We discussed the POS tag lexicon

When do words belong to the same class? Three criteria

What tagset should we use?

What are the sources of ambiguity for POS tagging?

## 1 Automatic POS tagging: the problem

## 2 Methods for tagging

- Unigram tagging
- Bigram tagging
- Tagging using Hidden Markov Models: Viterbi algorithm
- Rule-based Tagging

**Reading:** Jurafsky & Martin, chapters (5 and) 6.

# Benefits of Part of Speech Tagging

- Essential preliminary to (anything that involves) parsing.
- Can help with **speech synthesis**. For example, try saying the sentences below out loud.
- Can help with **determining authorship**: are two given documents written by the same person? **Forensic linguistics**.

- ① *Have you read 'The Wind in the Willows'? (noun)*
- ② *The clock has stopped. Please wind it up. (verb)*
- ③ *The students tried to protest. (verb)*
- ④ *The students' protest was successful. (noun)*

## Corpus annotation

A **corpus** (plural **corpora**) is a computer-readable collection of NL text (or speech) used as a source of information about the language: e.g. what words/constructions can occur in practice, and with what frequencies.

The usefulness of a corpus can be enhanced by *annotating* each word with a POS tag, e.g.

```
Our/PRP\$ enemies/NNS are/VBP innovative/JJ and/CC  
resourceful/JJ ,/, and/CC so/RB are/VB we/PRP ./.  
They/PRP never/RB stop/VB thinking/VBG about/IN new/JJ  
ways/NNS to/TO harm/VB our/PRP\$ country/NN and/CC  
our/PRP\$ people/NN, and/CC neither/DT do/VB we/PRP ./.
```

Typically done by an automatic tagger, then hand-corrected by a native speaker, in accordance with specified **tagging guidelines**.

# POS tagging: difficult cases

Even for humans, tagging sometimes poses difficult decisions.  
Various tests can be applied, but they don't always yield clear answers.

E.g. Words in **-ing**: adjectives (JJ), or verbs in gerund form (VBG)?

<b>a boring/JJ lecture</b>	a very boring lecture ? a lecture that bores
<b>the falling/VBG leaves</b>	*the very falling leaves the leaves that fall
<b>a revolving/VBG? door</b>	*a very revolving door a door that revolves *the door seems revolving
<b>sparkling/JJ? lemonade</b>	? very sparkling lemonade lemonade that sparkles the lemonade seems sparkling

In view of such problems, we can't expect 100% accuracy from an automatic tagger.

- Need to distinguish **word tokens** (particular occurrences in a text) from **word types** (distinct vocabulary items).
- We'll count different inflected or derived forms (e.g. break, breaks, breaking) as distinct word types.
- A single word type (e.g. **still**) may appear with several POS.
- But most words have a clear **most frequent** POS.

**Question:** How many tokens and types in the following? Ignore case and punctuation.

*Esau sawed wood. Esau Wood would saw wood. Oh, the  
wood Wood would saw!*

- ① 14 tokens, 6 types
- ② 14 tokens, 7 types
- ③ 14 tokens, 8 types
- ④ None of the above.

# Extent of POS Ambiguity

The Brown corpus (1,000,000 word tokens) has 39,440 different word types.

- 35340 have only 1 POS tag anywhere in corpus (89.6%)
- 4100 (10.4%) have 2 to 7 POS tags

So why does just 10.4% POS-tag ambiguity by **word type** lead to difficulty?

This is thanks to *Zipfian distribution*: many high-frequency words have more than one POS tag.

In fact, more than 40% of the **word tokens** are ambiguous.

He wants **to/TO** go.

He went **to/IN** the store.

He wants **that/DT** hat.

It is obvious **that/CS** he wants a hat.

He wants a hat **that/WPS** fits.

# Word Frequencies in Different Languages

Ambiguity by part-of-speech tags:

Language	Type-ambiguous	Token-ambiguous
English	13.2%	56.2%
Greek	<1%	19.14%
Japanese	7.6%	50.2%
Czech	<1%	14.5%
Turkish	2.5%	35.2%

# Some tagging strategies

We'll look at several methods or strategies for automatic tagging.

- One simple strategy: just assign to each word its *most common tag*. (So **still** will *always* get tagged as an adverb — never as a noun, verb or adjective.) Call this *unigram tagging*, since we only consider one token at a time.
- Surprisingly, even this crude approach typically gives around 90% accuracy. (State-of-the-art is 96–98%).
- Can we do better? We'll look briefly at **bigram tagging**, then at **Hidden Markov Model tagging**.

# Bigram tagging

We can do much better by looking at *pairs of adjacent tokens*.

For each word (e.g. **still**), tabulate the frequencies of each possible POS *given the POS of the preceding word*.

Example (with made-up numbers):

still	DT	MD	JJ	...
NN	8	0	6	
JJ	23	0	14	
VB	1	12	2	
RB	6	45	3	

Given a new text, tag the words from left to right, assigning each word the most likely tag given the preceding one.

Could also consider **trigram** (or more generally **n-gram**) tagging, etc. But the frequency matrices would quickly get very large, and also (for realistic corpora) too 'sparse' to be really useful.

## Example

and a member of both countries , a serious the services of the Dole of . " Ross declined to buy beer at the winner of his wife , I can live with her hand who sleeps below 50 @-@ brick appealed to make his last week the size , Radovan Karadzic said . " The Dow Jones set aside from the economy that Samuel Adams was half @-@ filled with it , " but if that Yeltsin . " but analysts and goes digital Popcorn , you don 't . " this far rarer cases it is educable .

## Example

change his own home ; others ( such disagreements have characterized Diller 's team quickly launched deliberately raunchier , more recently , " said Michael Pasano , a government and ruling party " presidential power , and Estonia , which published photographs by him in running his own club

### Example

not to let nature take its course . " we've got one time to do it in three weeks and was criticized by Lebanon and Syria to use the killing of thousands of years of involvement in the plots .

# Problems with bigram tagging

- One incorrect tagging choice might have unintended effects:

	The	still	smoking	remains	of	the	campfire
<i>Intended:</i>	DT	RB	VBG	NNS	IN	DT	NN
<i>Bigram:</i>	DT	JJ	NN	VBZ	...		

- No lookahead: choosing the ‘most probable’ tag at one stage might lead to highly improbable choice later.

	The	still	was	smashed
<i>Intended:</i>	DT	NN	VBD	VBN
<i>Bigram:</i>	DT	JJ	VBD?	

We’d prefer to find the *overall most likely* tagging sequence given the bigram frequencies. This is what the **Hidden Markov Model (HMM)** approach achieves.

- The idea is to model the agent that might have generated the sentence by a semi-random process that outputs a sequence of words.
- Think of the output as **visible** to us, but the internal states of the process (which contain POS information) as **hidden**.
- For some outputs, there might be several possible ways of generating them i.e. several sequences of internal states. Our aim is to compute the sequence of hidden states with the **highest probability**.
- Specifically, our processes will be '**NFAs with probabilities**'. Simple, though not a very flattering model of human language users!

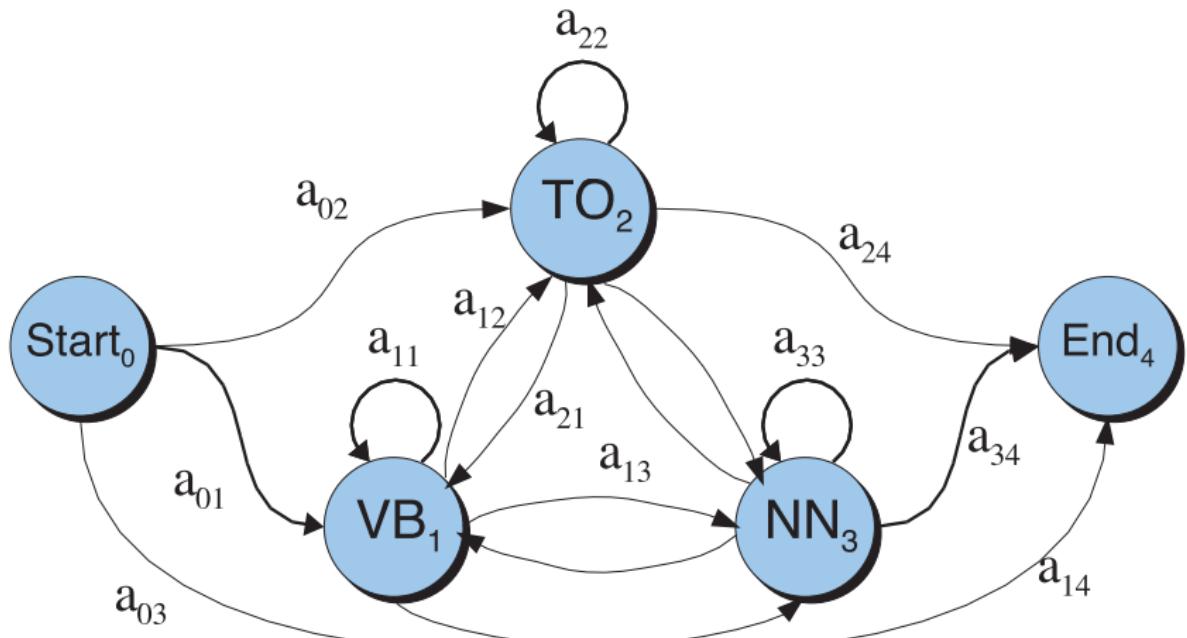
# Definition of Hidden Markov Models

For our purposes, a **Hidden Markov Model (HMM)** consists of:

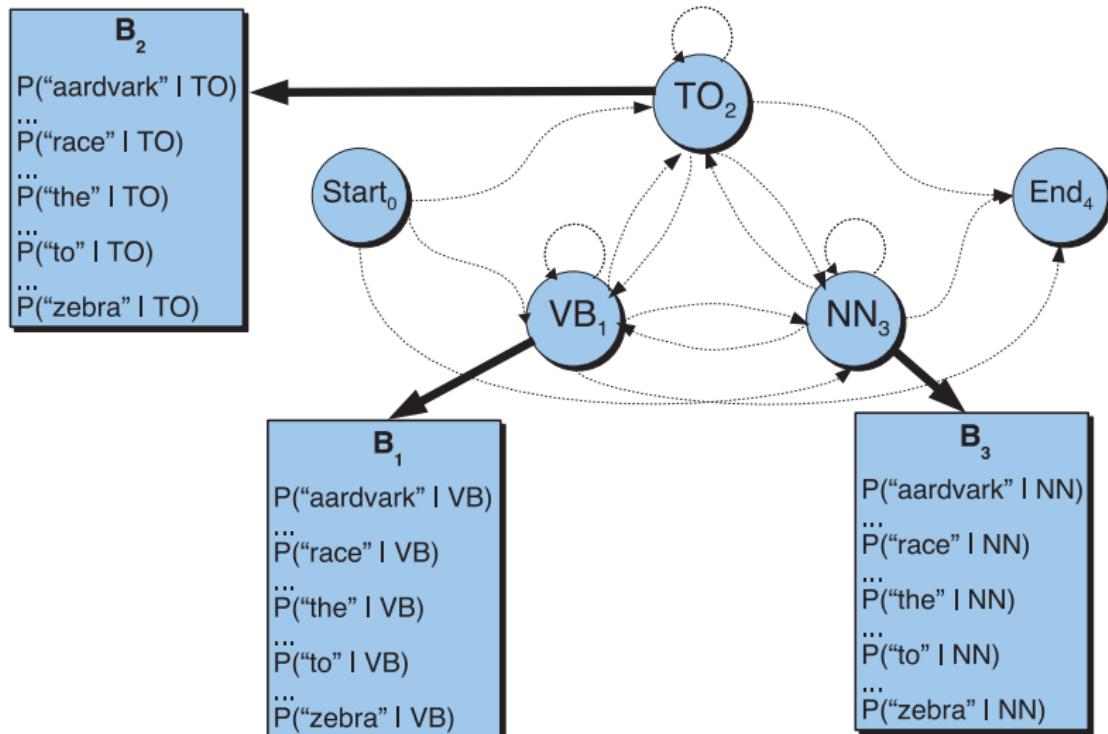
- A set  $Q = \{q_0, q_1, \dots, q_T\}$  of **states**, with  $q_0$  the start state.  
(Our non-start states will correspond to *parts-of-speech*).
- A **transition probability** matrix  
 $A = (a_{ij} \mid 0 \leq i \leq T, 1 \leq j \leq T)$ , where  $a_{ij}$  is the probability  
of jumping from  $q_i$  to  $q_j$ . For each  $i$ , we require  $\sum_{j=1}^T a_{ij} = 1$ .
- For each non-start state  $q_i$  and word type  $w$ , an **emission probability**  $b_i(w)$  of outputting  $w$  upon entry into  $q_i$ . (Ideally,  
for each  $i$ , we'd have  $\sum_w b_i(w) = 1$ .)

We also suppose we're given an **observed sequence**  $w_1, w_2 \dots, w_n$   
of word tokens generated by the HMM.

# Transition Probabilities



# Emission Probabilities



## Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a very rich history .

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh

NNP

$$p(\text{NNP}|\langle s \rangle) \times p(\text{Edinburgh}|\text{NNP})$$

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has  
NNP VBZ

$$p(\text{NNP}|\langle s \rangle) \times p(\text{Edinburgh}|\text{NNP})$$

$$p(\text{VBZ}|\text{NNP}) \times p(\text{has}|\text{VBZ})$$

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a  
NNP VBZ DT

$$p(NNP| \langle s \rangle) \times p(Edinburgh| NNP)$$

$$p(VBZ| NNP) \times p(has| VBZ)$$

$$p(DT| VBZ) \times p(a| DT)$$

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a very  
NNP VBZ DT RB

$$p(NNP|\langle s \rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a very rich  
NNP VBZ DT RB JJ

$$p(NNP|\langle s \rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

$$p(JJ|RB) \times p(rich|JJ)$$

# Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a very rich history  
NNP VBZ DT RB JJ NN

$$p(NNP|\langle s \rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

$$p(JJ|RB) \times p(rich|JJ)$$

$$p(NN|JJ) \times p(history|NN)$$

# Transition and Emission Probabilities

	<b>VB</b>	<b>TO</b>	<b>NN</b>	<b>PRP</b>
<b>&lt;s&gt;</b>	.019	.0043	.041	.67
<b>VB</b>	.0038	.035	.047	.0070
<b>TO</b>	.83	0	.00047	0
<b>NN</b>	.0040	.016	.087	.0045
<b>PRP</b>	.23	.00079	.001	.00014

	<b>I</b>	<b>want</b>	<b>to</b>	<b>race</b>
<b>VB</b>	0	.0093	0	.00012
<b>TO</b>	0	0	.99	0
<b>NN</b>	0	.000054	0	.00057
<b>PRP</b>	.37	0	0	0

# How Do we Search for Best Tag Sequence?

We have defined an HMM, but how do we use it? We are given a **word sequence** and must find their corresponding **tag sequence**.

- It's easy to compute the probability of generating a word sequence  $w_1 \dots w_n$  via a specific tag sequence  $t_1 \dots t_n$ : let  $t_0$  denote the start state, and compute

$$\prod_{i=1}^T P(t_i|t_{i-1}).P(w_i|t_i) \quad (1)$$

using the transition and emission probabilities.

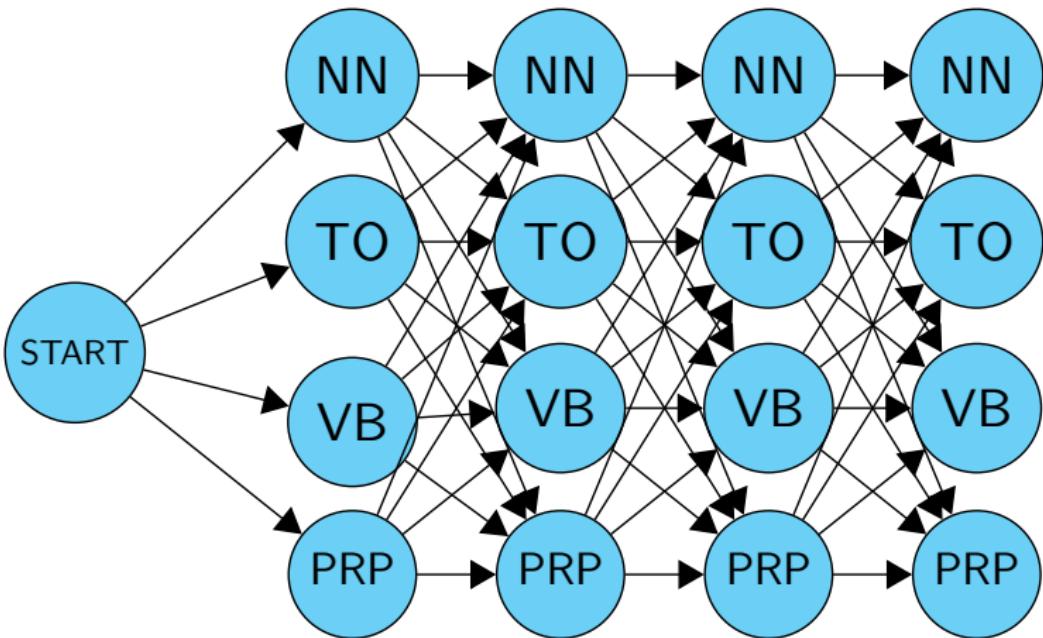
- But how do we find the most likely tag sequence?**
- We can do this efficiently using **dynamic programming** and the **Viterbi algorithm**.

## Question

Given  $n$  word tokens and a tagset with  $T$  choices per token, how many tag sequences do we have to evaluate?

- ①  $|T|$  tag sequences
- ②  $n$  tag sequences
- ③  $|T| \times n$  tag sequences
- ④  $|T|^n$  tag sequences

# The HMM trellis



I      want      to      race

# The Viterbi Algorithm

Keep a chart of the form  $\text{Table}(\text{POS}, i)$  where  $\text{POS}$  ranges over the POS tags and  $i$  ranges over the indices in the sentence.

For all  $T$  and  $i$ :

$$\text{Table}(T, i + 1) \leftarrow \max_{T'} \text{Table}(T', i) \times p(T | T') \times p(w_{i+1} | T)$$

and

$$\text{Table}(T, 0) \leftarrow p(T | \langle s \rangle)$$

$\text{Table}(., n)$  will contain the **probability** of the most likely sequence.  
To get the actual sequence, we need backpointers.

# The Viterbi algorithm

Let's now tag the newspaper headline:

deal talks fail

Note that each token here could be a noun (N) or a verb (V).

We'll use a toy HMM given as follows:

	to N	to V		deal	fail	talks		
from start	.8	.2	N	.2	.05	.2		
from N	.4	.6	V	.3	.3	.3		
from V	.8	.2	Emissions					
Transitions		Emissions						

# The Viterbi matrix

	deal	talks	fail
N			
V			
	to N	to V	
from start	.8	.2	
from N	.4	.6	
from V	.8	.2	
		deal	fail
N		.2	.05
V		.3	.3
		talks	

Transitions    Emissions

$$\text{Table}(T, i+1) \leftarrow \max_{T'} \text{Table}(T', i) \times p(T|T') \times p(w_{i+1}|T)$$

# The Viterbi matrix

	deal	talks	fail
N	$.8 \times .2 = .16$	$\leftarrow .16 \times .4 \times .2 = .0128$ (since $.16 \times .4 > .06 \times .8$ )	$\swarrow .0288 \times .8 \times .05 = .001152$ (since $.0128 \times .4 < .0288 \times .8$ )
V	$.2 \times .3 = .06$	$\nwarrow .16 \times .6 \times .3 = .0288$ (since $.16 \times .6 > .06 \times .2$ )	$\nwarrow .0128 \times .6 \times .3 = .002304$ (since $.0128 \times .6 > .0288 \times .2$ )

Looking at the highest probability entry in the final column and chasing the backpointers, we see that the tagging **N N V** wins.

## The Viterbi Algorithm: second example

$q_4$	NN	0					
$q_3$	TO	0					
$q_2$	VB	0					
$q_1$	PRP	0					
$q_o$	start	<b>1.0</b>					
	$\langle s \rangle$		I	want		to	race
			$w_1$	$w_2$	$w_3$	$w_4$	

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

# The Viterbi Algorithm

$q_4$	NN	0				
$q_3$	TO	0				
$q_2$	VB	0				
$q_1$	PRP	0				
$q_o$	start	<b>1.0</b>				
	<s>	I	want	to	race	
		$w_1$	$w_2$	$w_3$	$w_4$	

- ① Create probability matrix, with one column for each observation (i.e., word token), and one row for each non-start state (i.e., POS tag).
- ② We proceed by filling cells, column by column.
- ③ The entry in column  $i$ , row  $j$  will be the **probability of the most probable route to state  $q_j$  that emits  $w_1 \dots w_i$** .

# The Viterbi Algorithm

$q_4$	NN	0	$1.0 \times .041 \times 0$			
$q_3$	TO	0	$1.0 \times .0043 \times 0$			
$q_2$	VB	0	$1.0 \times .19 \times 0$			
$q_1$	PRP	0	$1.0 \times .67 \times .37$			
$q_o$	start	<b>1.0</b>				
		<s>	I	want	to	race
			$w_1$	$w_2$	$w_3$	$w_4$

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

- $v_{i-1}(k)$  is **previous Viterbi path probability**,  $a_{kj}$  is **transition probability**, and  $b_j(w_i)$  is **emission probability**.
- There's also an (implicit) **backpointer** from cell  $(i, j)$  to the relevant  $(i - 1, k)$ , where  $k$  maximizes  $v_{i-1}(k) a_{kj}$ .

# The Viterbi Algorithm

$q_4$	NN	0	0	$.025 \times .0012 \times 0.000054$		
$q_3$	TO	0	0	$.025 \times .00079 \times 0$		
$q_2$	VB	0	0	$.025 \times .23 \times .0093$		
$q_1$	PRP	0	<b>.025</b>	$.025 \times .00014 \times 0$		
$q_0$	start	<b>1.0</b>				
		<s>	I	want	to	race
			$w_1$	$w_2$	$w_3$	$w_4$

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

- $v_{i-1}(k)$  is **previous Viterbi path probability**,  $a_{kj}$  is **transition probability**, and  $b_j(w_i)$  is **emission probability**.
- There's also an (implicit) **backpointer** from cell  $(i, j)$  to the relevant  $(i - 1, k)$ , where  $k$  maximizes  $v_{i-1}(k) a_{kj}$ .

# The Viterbi Algorithm

$q_4$	NN	0	0	.000000002	.000053 × .047 × 0	
$q_3$	TO	0	0	0	.000053 × .035 × .99	
$q_2$	VB	0	0	<b>.00053</b>	.000053 × .0038 × 0	
$q_1$	PRP	0	<b>.025</b>	0	.000053 × .0070 × 0	
$q_0$	start	<b>1.0</b>				
	<s>		I	want	to	race
			$w_1$	$w_2$	$w_3$	$w_4$

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

- $v_{i-1}(k)$  is **previous Viterbi path probability**,  $a_{kj}$  is **transition probability**, and  $b_j(w_i)$  is **emission probability**.
- There's also an (implicit) **backpointer** from cell  $(i, j)$  to the relevant  $(i - 1, k)$ , where  $k$  maximizes  $v_{i-1}(k) a_{kj}$ .

# The Viterbi Algorithm

$q_4$	NN	0	0	.0000000020	.0000018 × .00047 × .00057
$q_3$	TO	0	0	.0000018	.0000018 × 0 × 0
$q_2$	VB	0	0	.00053	.0000018 × .83 × .00012
$q_1$	PRP	0	.025	0	.0000018 × 0 × 0
$q_0$	start	1.0			
	< s >	I	want	to	race
		$w_1$	$w_2$	$w_3$	$w_4$

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

- $v_{i-1}(k)$  is **previous Viterbi path probability**,  $a_{kj}$  is **transition probability**, and  $b_j(w_i)$  is **emission probability**.
- There's also an (implicit) **backpointer** from cell  $(i, j)$  to the relevant  $(i - 1, k)$ , where  $k$  maximizes  $v_{i-1}(k) a_{kj}$ .

# The Viterbi Algorithm

$q_4$	NN	0	0	.000000002	0	4.8222e-13
$q_3$	TO	0	0	0	<b>.0000018</b>	0
$q_2$	VB	0	0	<b>.00053</b>	0	<b>1.7928e-10</b>
$q_1$	PRP	0	<b>.025</b>	0	0	0
$q_0$	start	<b>1.0</b>				
	<s>	I		want	to	race
		$w_1$		$w_2$	$w_3$	$w_4$

- For each state  $q_j$  at time  $i$ , compute

$$v_i(j) = \max_{k=1}^n v_{i-1}(k) a_{kj} b_j(w_i)$$

- $v_{i-1}(k)$  is **previous Viterbi path probability**,  $a_{kj}$  is **transition probability**, and  $b_j(w_i)$  is **emission probability**.
- There's also an (implicit) **backpointer** from cell  $(i, j)$  to the relevant  $(i - 1, k)$ , where  $k$  maximizes  $v_{i-1}(k) a_{kj}$ .

# Connection between HMMs and finite state machines

Hidden Markov models are finite state machines with probabilities added to them.

If we think of finite state automaton as generating a string when randomly going through states (instead of scanning a string), then hidden Markov models are such FSMs where there is a specific probability for generating each symbol at each state, and a specific probability for transitioning from one state to another.

As such, the Viterbi algorithm can be used to find the most likely sequence of *states* in a probabilistic FSM, given a specific input string.

**Question: where do the probabilities come from?**

<http://nlp.stanford.edu:8080/parser/>

- Relies both on “distributional” and “morphological” criteria
- Uses a model similar to hidden Markov models

# Rule-based Tagging

## Basic idea:

- ① Assign each token all its possible tags.
- ② Apply rules that eliminate all tags for a token that are inconsistent with its context.

### Example

the	DT (determiner)	the	DT (determiner)
can	MD (modal)	can	MD (modal) <span style="color:red">X</span>
	NN (sg noun)		NN (sg noun) <span style="color:green">✓</span>
	VB (base verb)		VB (base verb) <span style="color:red">X</span>

Assign any unknown word tokens a tag that is consistent with its context (eg, the **most frequent** tag).

# Rule-based tagging

Rule-based tagging often used a large set of hand-crafted context-sensitive rules.

**Example (schematic):**

```
if (-1 DT) /* if previous word is a determiner */  
elim MD, VB /* eliminate modals and base verbs */
```

**Problem:** Cannot eliminate all POS ambiguity.

# Phrase Structure and Parsing as Search

## Informatics 2A: Lecture 17

Shay Cohen

School of Informatics  
University of Edinburgh

30 October 2015

## 1 Phrase Structure

- Heads and Phrases
- Desirable Properties of a Grammar
- A Fragment of English

## 2 Grammars and Parsing

- Recursion
- Structural Ambiguity
- Recursive Descent Parsing
- Shift-Reduce Parsing

Part-of-speech tagging and its applications

The use of hidden Markov models for POS tagging

The Viterbi algorithm

Noun (N): Noun Phrase (NP)

Adjective (A): Adjective Phrase (AP)

Verb (V): Verb Phrase (VP)

Preposition (P): Prepositional Phrase (PP)

- So far we have looked at terminals (words or POS tags).
- Today, we'll look at non-terminals, which correspond to **phrases**.
- The **class** that a word belongs to is closely linked to the name of the **phrase** it customarily appears in.
- In a X-phrase (eg **NP**), the key occurrence of X (eg **N**) is called the **head**.
- In English, the head tends to appear in the middle of a phrase.

English NPs are commonly of the form:

(Det) Adj\* **Noun** (PP | RelClause)\*

**NP**: *the angry duck that tried to bite me*, **head**: *duck*.

VPs are commonly of the form:

(Aux) Adv\* **Verb** Arg\* Adjunct\*

Arg → NP | PP

Adjunct → PP | AdvP | ...

**VP**: *usually eats artichokes for dinner*, **head**: *eat*.

In Japanese, Korean, Hindi, Urdu, and other **head-final** languages, the head is at the end of its associated phrase.

In Irish, Welsh, Scots Gaelic and other **head-initial** languages, the head is at the beginning of its associated phrase.

# Desirable Properties of a Grammar

Chomsky specified two properties that make a grammar “interesting and satisfying”:

- It should be a **finite** specification of the strings of the language, rather than a list of its sentences.
- It should be **revealing**, in allowing strings to be associated with meaning (semantics) in a systematic way.

We can add another desirable property:

- It should capture **structural** and **distributional** properties of the language. (E.g. where heads of phrases are located; how a sentence transforms into a question; which phrases can float around the sentence.)

# Desirable Properties of a Grammar

- Context-free grammars (CFGs) provide a pretty good approximation.
- Some features of NLs are more easily captured using mildly context-sensitive grammars, as well see later in the course.
- There are also more modern grammar formalisms that better capture structural and distributional properties of human languages. (E.g. combinatory categorial grammar.)
- But LL(1) grammars and the like definitely aren't enough for NLs. Even if we could make a NL grammar LL(1), we wouldn't want to: this would artificially suppress ambiguities, and would often mutilate the 'natural' structure of sentences.

# A Tiny Fragment of English

Let's say we want to capture in a grammar the structural and distributional properties that give rise to sentences like:

A duck walked in the park.	NP,V,PP
The man walked with a duck.	NP,V,PP
You made a duck.	Pro,V,NP
You made her duck.	? Pro,V,NP
A man with a telescope saw you.	NP,PP,V,Pro
A man saw you with a telescope.	NP,V,Pro,PP
You saw a man with a telescope.	Pro,V,NP,PP

We want to write **grammatical rules** that generate these phrase structures, and **lexical rules** that generate the words appearing in them.

# Grammar for the Tiny Fragment of English

Grammar G1 generates the sentences on the previous slide:

## Grammatical rules

$S \rightarrow NP\ VP$   
 $NP \rightarrow Det\ N$   
 $NP \rightarrow Det\ N\ PP$   
 $NP \rightarrow Pro$   
 $VP \rightarrow V\ NP\ PP$   
 $VP \rightarrow V\ NP$   
 $VP \rightarrow V$   
 $PP \rightarrow Prep\ NP$

## Lexical rules

$Det \rightarrow a \mid the \mid her$  (determiners)  
 $N \rightarrow man \mid park \mid duck \mid telescope$  (nouns)  
 $Pro \rightarrow you$  (pronoun)  
 $V \rightarrow saw \mid walked \mid made$  (verbs)  
 $Prep \rightarrow in \mid with \mid for$  (prepositions)

Does G1 produce a finite or an infinite number of sentences?

# Recursion

Recursion in a grammar makes it possible to generate an infinite number of sentences.

In direct recursion, a non-terminal on the LHS of a rule also appears on its RHS. The following rules add direct recursion to G1:

$VP \rightarrow VP \text{ Conj } VP$

$\text{Conj} \rightarrow \text{and} \mid \text{or}$

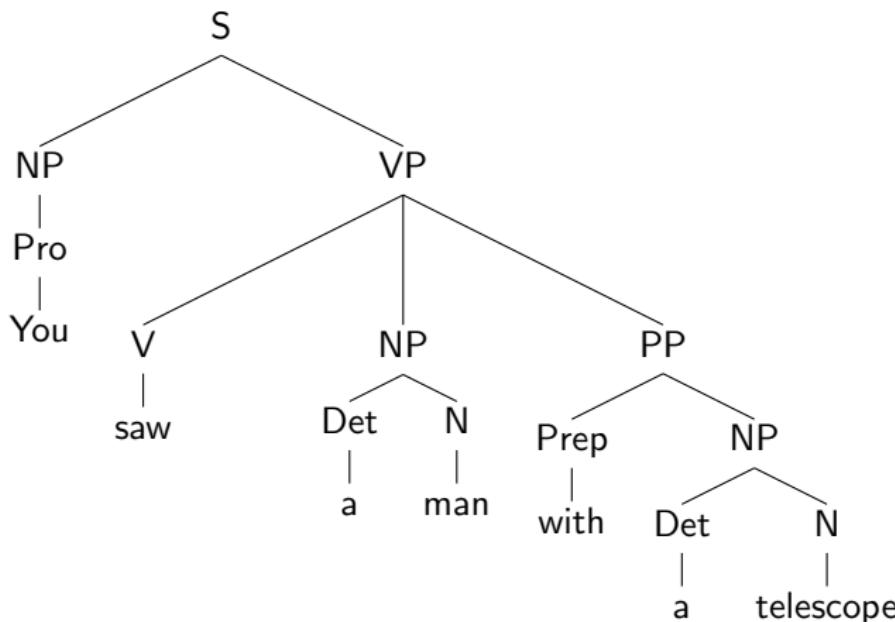
In indirect recursion, some non-terminal can be expanded (via several steps) to a sequence of symbols containing that non-terminal:

$NP \rightarrow \text{Det } N \text{ PP}$

$PP \rightarrow \text{Prep } NP$

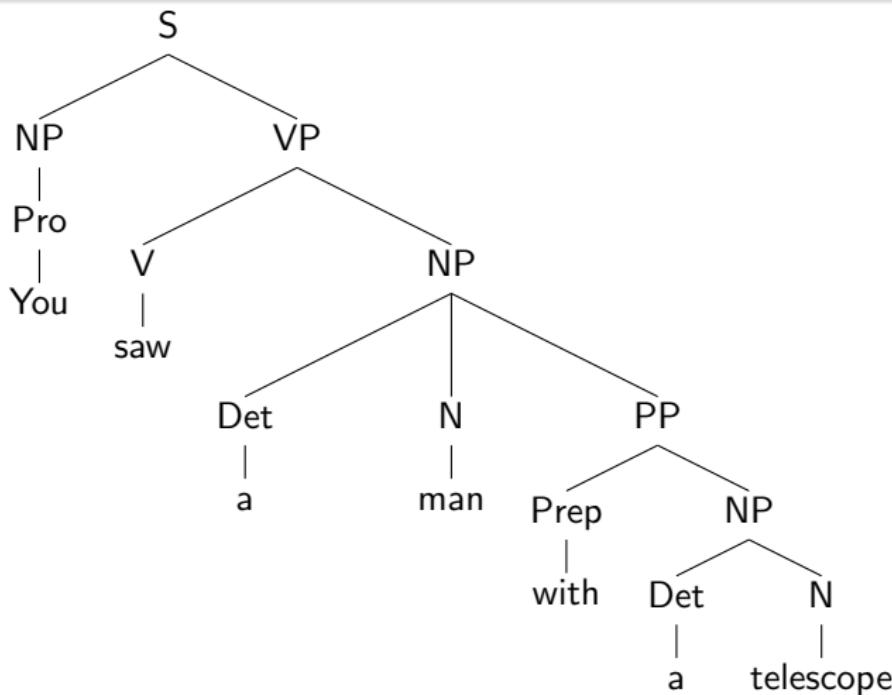
# Structural Ambiguity

You saw a man with a telescope.



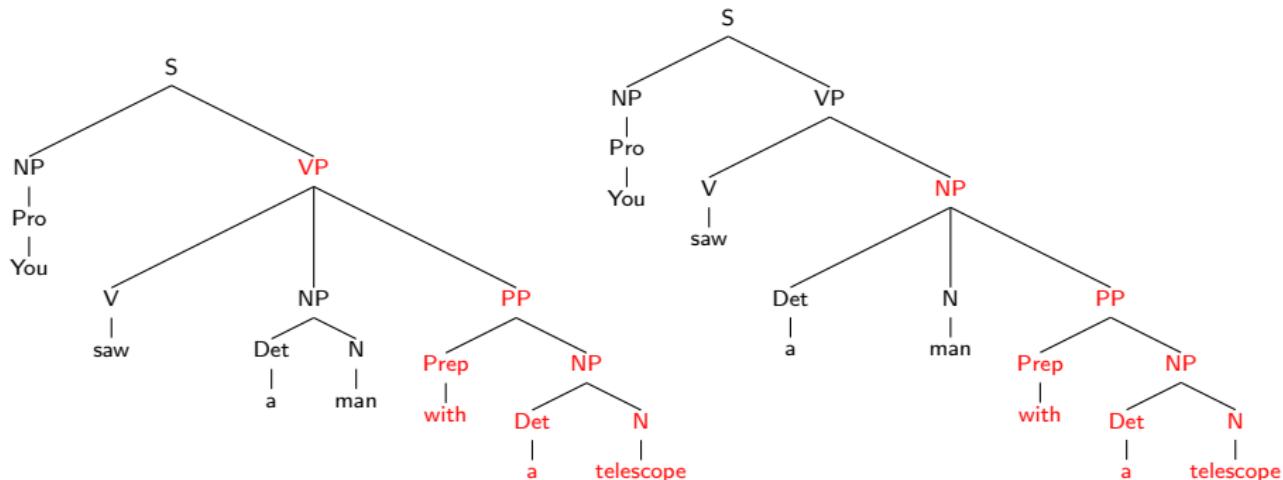
# Structural Ambiguity

You saw a man with a telescope.



# Structural Ambiguity

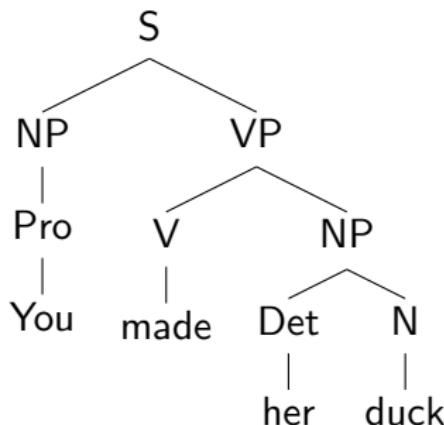
You saw a man with a telescope.



This illustrates **attachment ambiguity**: the PP can be a part of the VP or of the NP. Note that there's no **POS ambiguity** here.

# Structural Ambiguity

Grammar G1 only gives us one analysis of *you made her duck*.



There is another, ditransitive (i.e., two-object) analysis of this sentence – one that underlies the pair:

What did you make for her?

You made her duck.

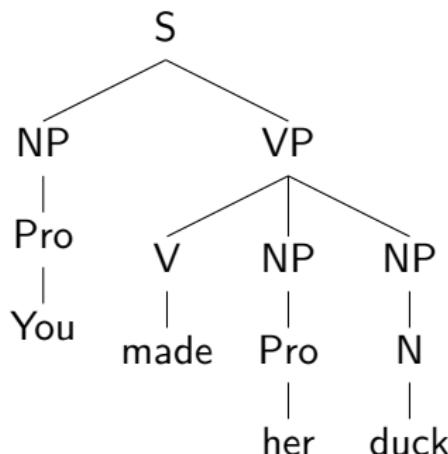
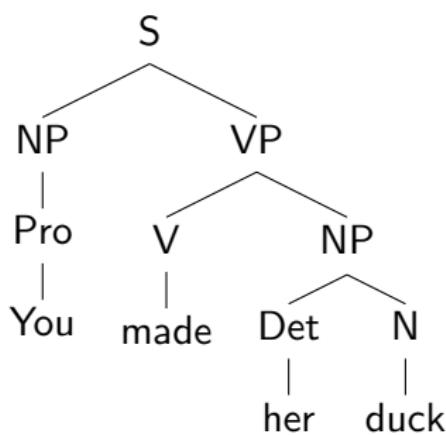
## Structural Ambiguity

For this alternative, G1 also needs rules like:

NP → N

$\text{VP} \rightarrow \text{V NP NP}$

Pro → her



In this case, the **structural ambiguity** is rooted in **POS ambiguity**.

# Structural Ambiguity

There is a third analysis as well, one that underlies the pair:

What did you make her do?

You made her duck. (move head or body quickly downwards)

Here, the **small clause** (*her duck*) is the direct object of a verb.

Similar **small clauses** are possible with verbs like *see*, *hear* and *notice*, but not *ask*, *want*, *persuade*, etc.

G1 needs a rule that requires accusative case-marking on the subject of a small clause and no tense on its verb.:

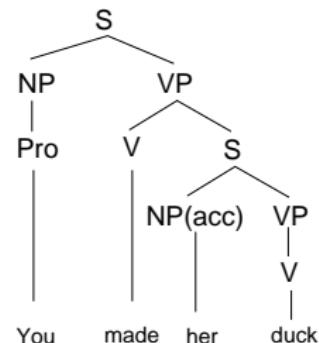
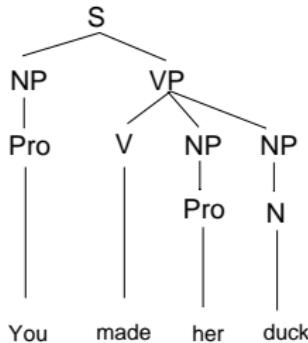
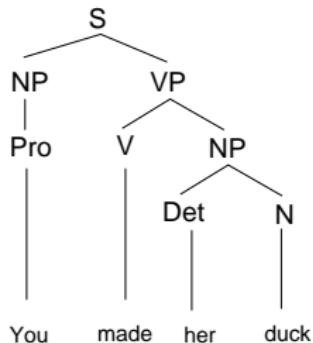
$\text{VP} \rightarrow \text{V S1}$

$\text{S1} \rightarrow \text{NP}(\text{acc}) \text{ VP}(\text{untensed})$

$\text{NP}(\text{acc}) \rightarrow \text{her} \mid \text{him} \mid \text{them}$

# Structural Ambiguity

Now we have three analyses for *you made her duck*:



How can we compute these analyses automatically?

A **parser** is an algorithm that computes a structure for an input string given a grammar. All parsers have two fundamental properties:

- **Directionality**: the sequence in which the structures are constructed (e.g., top-down or bottom-up).
- **Search strategy**: the order in which the search space of possible analyses is explored (e.g., depth-first, breadth-first).

For instance, LL(1) parsing is **top-down** and **depth-first**.

## Coming up: A zoo of parsing algorithms

As we've noted, LL(1) isn't good enough for NL. We'll be looking at other parsing algorithms that work for more general CFGs.

- **Recursive descent** parsers (top-down). Simple and very general, but inefficient. Other problems
- **Shift-reduce** parsers (bottom-up).
- The **Cocke-Younger-Kasami** algorithm (bottom up). Works for any CFG with reasonable efficiency.
- The **Earley** algorithm (top down). Chart parsing enhanced with prediction.

A **recursive descent** parser treats a grammar as a specification of how to break down a top-level goal into subgoals. Therefore:

- Parser searches through the trees licensed by the grammar to find the one that has the required sentence along its yield.
- **Directionality** = **top-down**: It starts from the start symbol of the grammar, and works its way down to the terminals.
- **Search strategy** = **depth-first**: It expands a given terminal as far as possible before proceeding to the next one.

# Algorithm Sketch: Recursive Descent Parsing

- ① The top-level goal is to derive the start symbol ( $S$ ).
- ② Choose a **grammatical rule** with  $S$  as its LHS (e.g.,  $S \rightarrow NP\ VP$ ), and replace  $S$  with the RHS of the rule (the subgoals; e.g.,  $NP$  and  $VP$ ).
- ③ Choose a rule with the leftmost subgoal as its LHS (e.g.,  $NP \rightarrow Det\ N$ ). Replace the subgoal with the RHS of the rule.
- ④ Whenever you reach a **lexical rule** (e.g.,  $Det \rightarrow the$ ), match its RHS against the current position in the input string.
  - If it matches, move on to next position in the input.
  - If it doesn't, try next lexical rule with the same LHS.
  - If no rules with same LHS, backtrack to most recent choice of grammatical rule and choose another rule with the same LHS.
  - If no more grammatical rules, back up to the previous subgoal.
- ⑤ Iterate until the whole input string is consumed, or you fail to match one of the positions in the input. Backtrack on failure.

# Recursive Descent Parsing

s

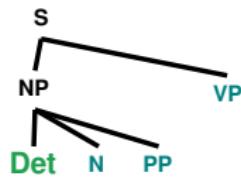
the dog saw a man in the park

# Recursive Descent Parsing



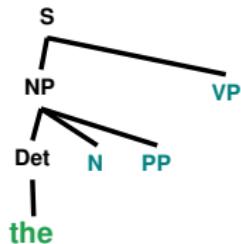
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



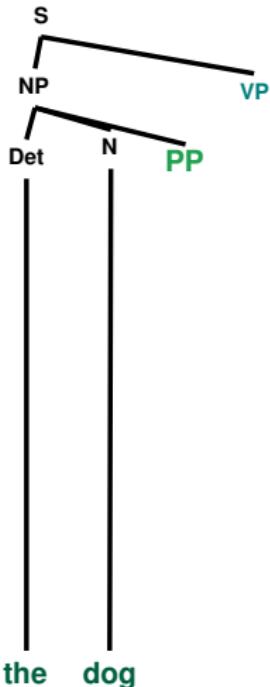
the dog saw a man in the park

# Recursive Descent Parsing



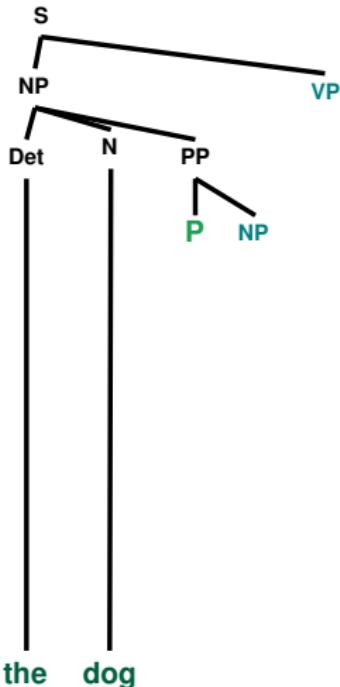
the dog saw a man in the park

# Recursive Descent Parsing



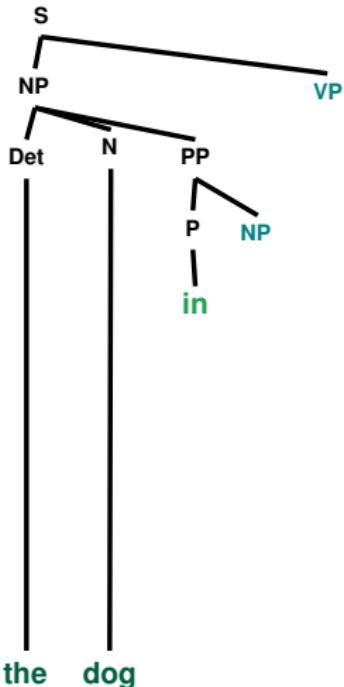
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



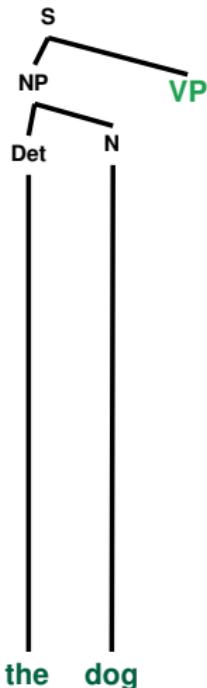
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



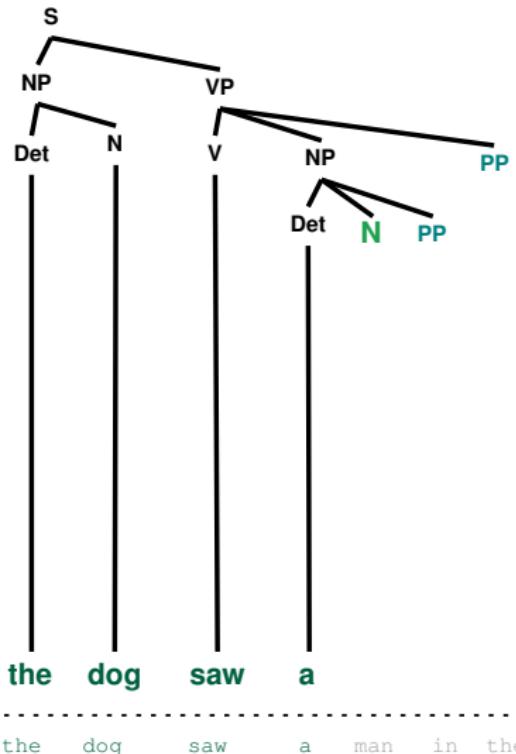
the dog saw a man in the park

# Recursive Descent Parsing

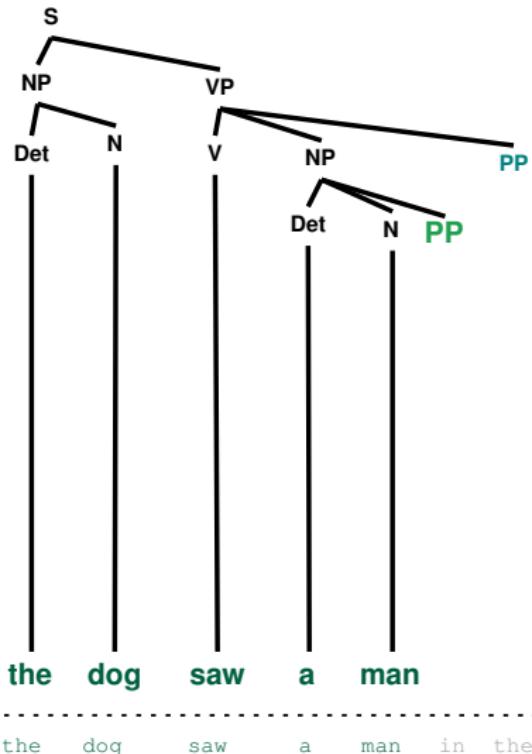


the dog saw a man in the park

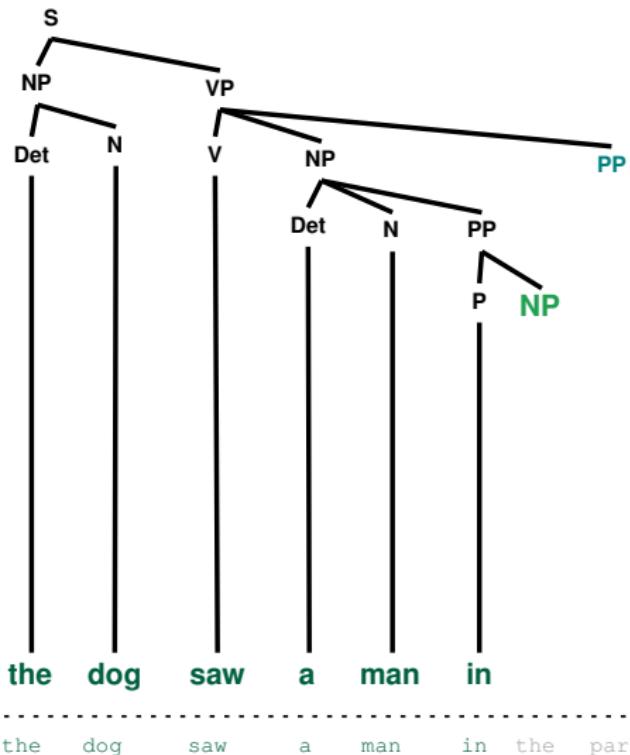
# Recursive Descent Parsing



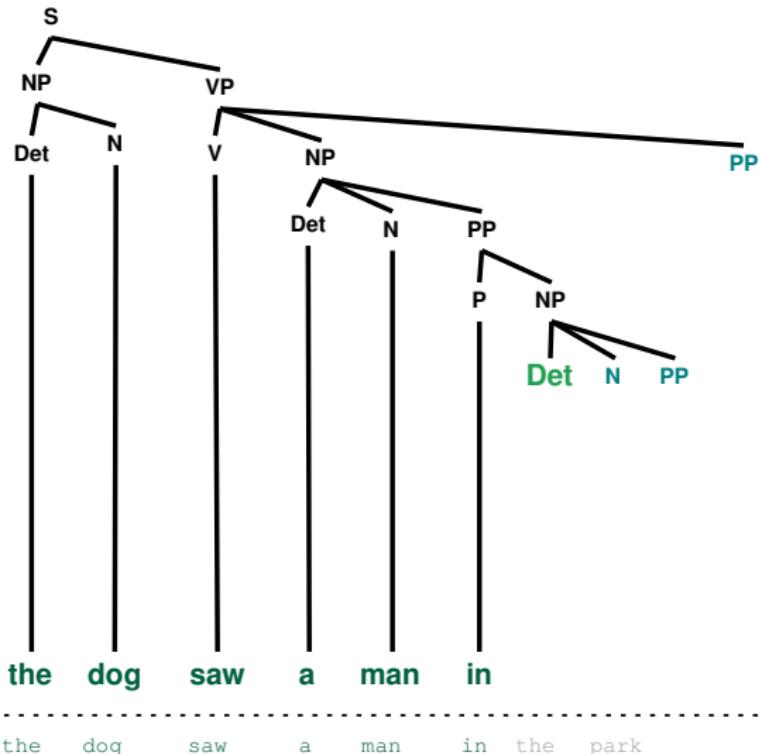
# Recursive Descent Parsing



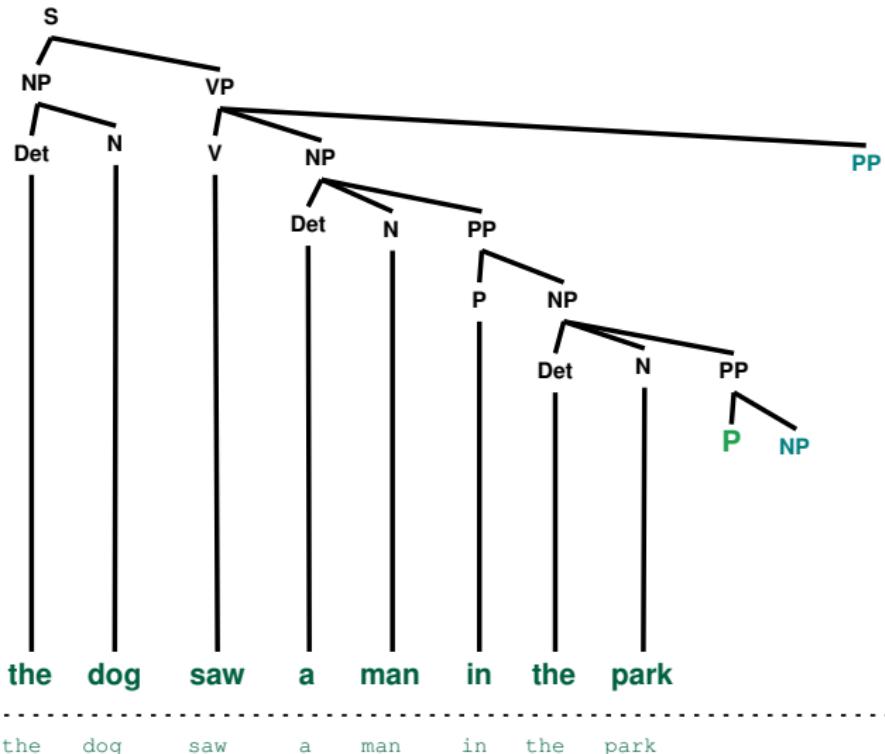
# Recursive Descent Parsing



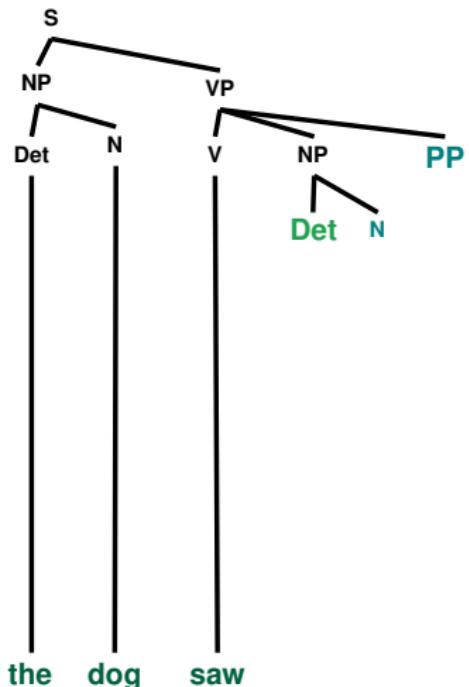
# Recursive Descent Parsing



# Recursive Descent Parsing

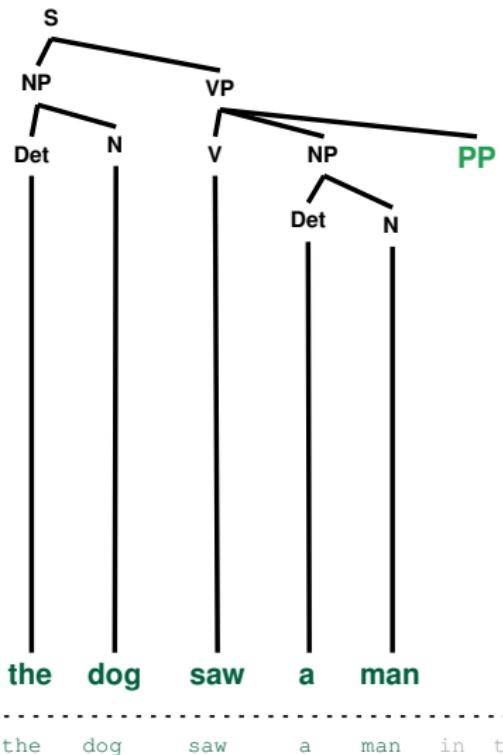


# Recursive Descent Parsing



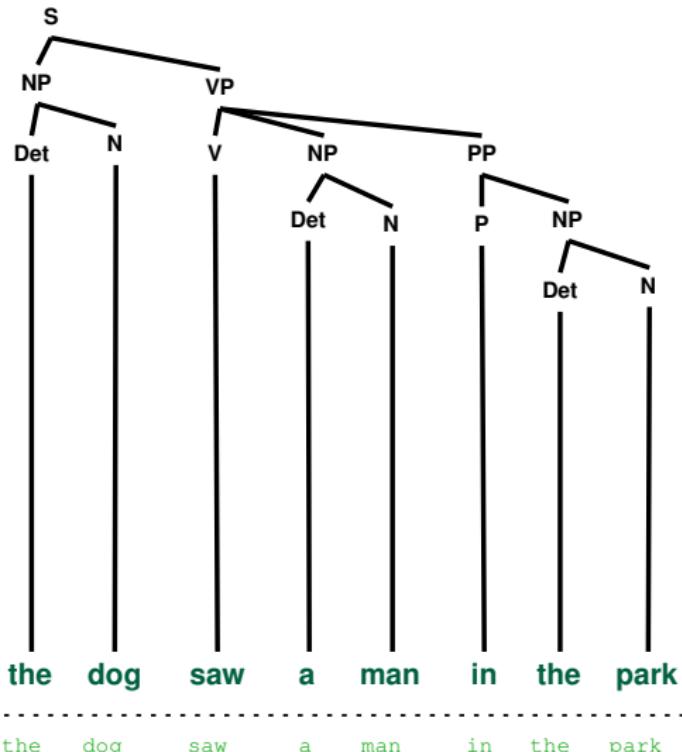
the    dog    saw    a    man    in    the    park

# Recursive Descent Parsing



the    dog    saw    a    man    in    the    park

# Recursive Descent Parsing



A **Shift-Reduce** parser tries to find sequences of words and phrases that correspond to the **righthand** side of a grammar production and replace them with the lefthand side:

- **Directionality** = **bottom-up**: starts with the words of the input and tries to build trees from the words up.
- **Search strategy** = **breadth-first**: starts with the words, then applies rules with matching right hand sides, and so on until the whole sentence is reduced to an S.

## Algorithm Sketch: Shift-Reduce Parsing

Until the words in the sentences are substituted with S:

- Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (**shift**)
- Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (**reduce**)

A shift-reduce parser implemented using a stack:

- ① start with an empty stack
- ② a **shift** action pushes the current input symbol onto the stack
- ③ a **reduce** action replaces  $n$  items with a single item

# Shift-Reduce Parsing

Stack	Remaining
	my dog saw a man in the park

# Shift-Reduce Parsing

Stack	Remaining
Det my	dog saw a man in the park

# Shift-Reduce Parsing

Stack	Remaining
Det      N my      dog	saw a man in the park

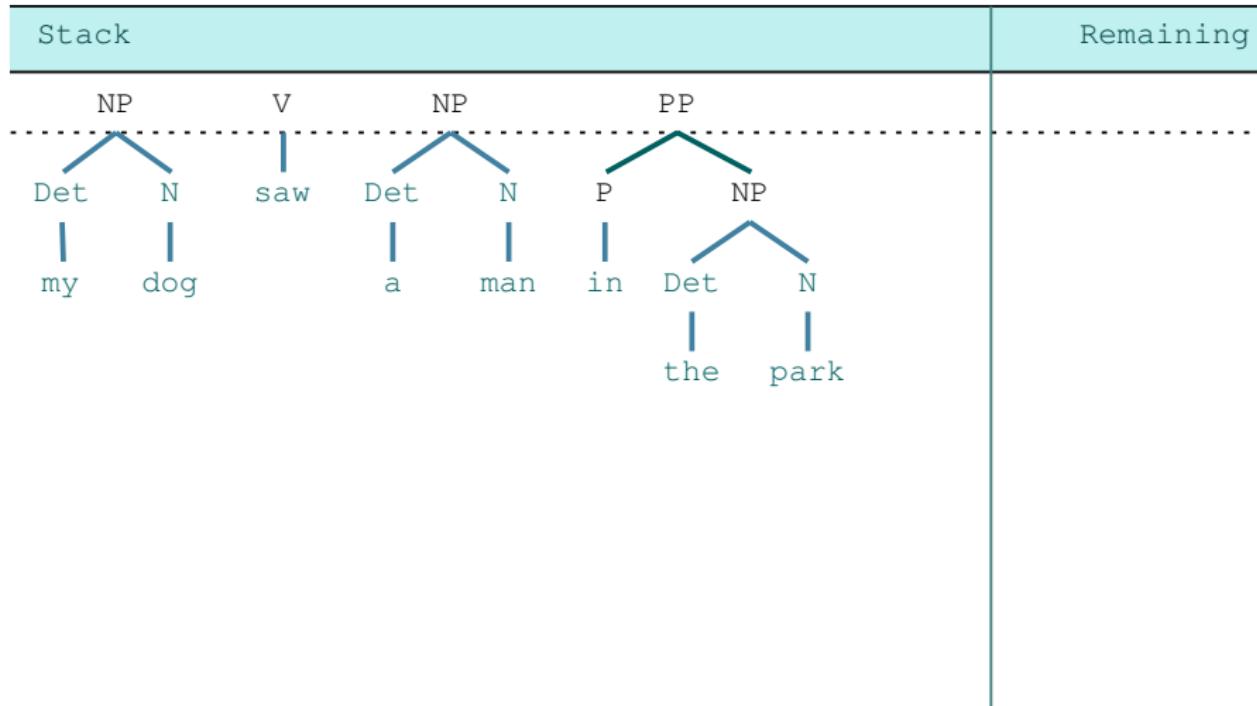
# Shift-Reduce Parsing

Stack	Remaining
<p>NP</p> <p>Det      N</p> <p>my      dog</p> 	saw a man in the park

# Shift-Reduce Parsing

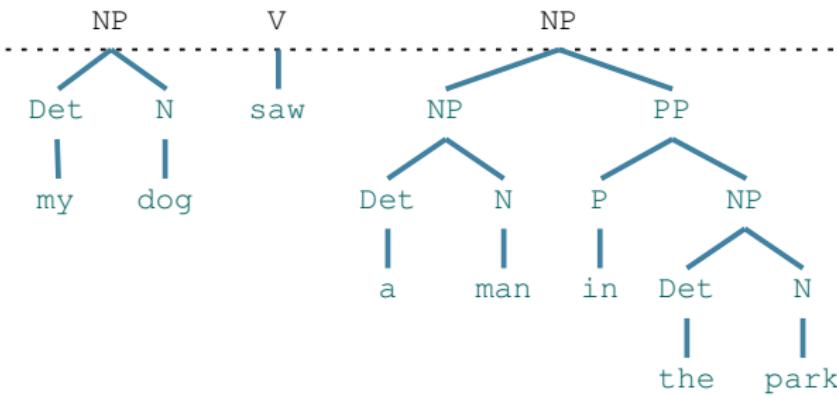
Stack	Remaining
<p>NP                    V                    NP</p> <p>Det                N                saw                Det                N</p> <p>my                dog                                     a                man</p> <pre>graph TD; NP1[NP] --&gt; Det1[Det]; NP1 --&gt; N1[N]; Det1 --&gt; my1[my]; N1 --&gt; dog1[dog]; V1[V] --&gt; saw1[saw]; NP2[NP] --&gt; Det2[Det]; NP2 --&gt; N2[N]; Det2 --&gt; a1[a]; N2 --&gt; man1[man];</pre>	in the park

# Shift-Reduce Parsing

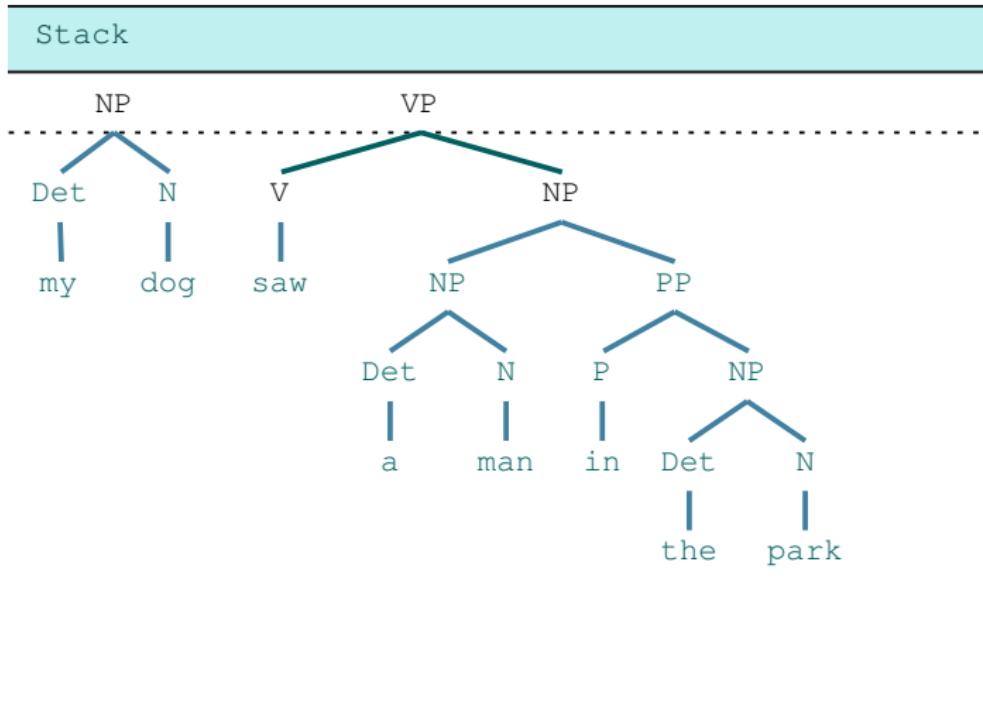


# Shift-Reduce Parsing

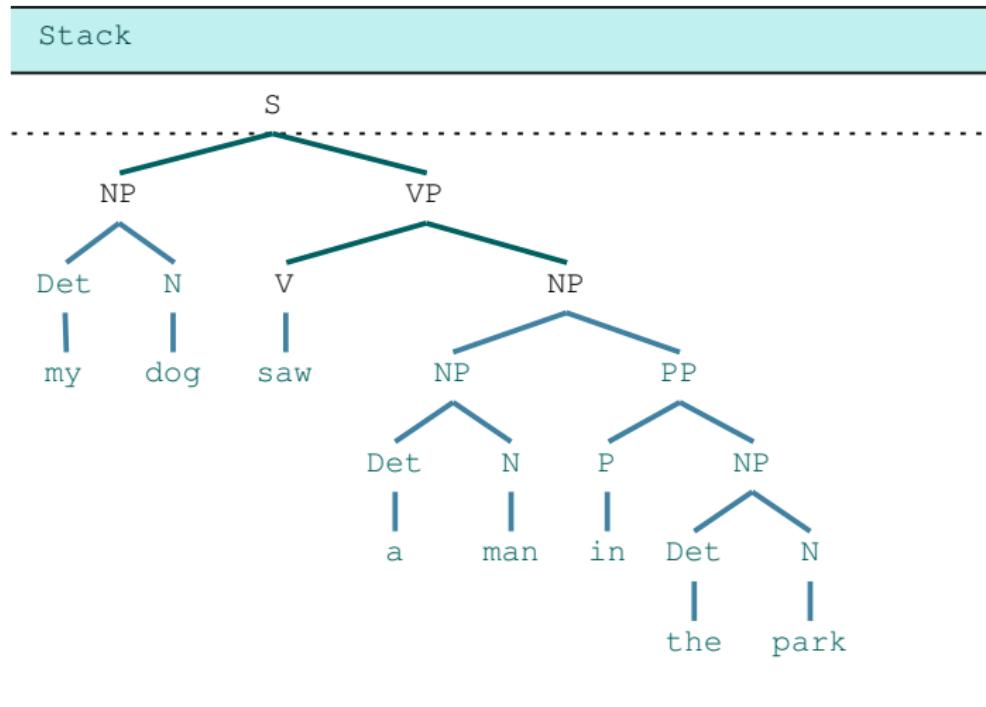
Stack



# Shift-Reduce Parsing



# Shift-Reduce Parsing



## Shift-reduce parsers and pushdown automata

Shift-reduce parsing is equivalent to a pushdown automaton constructed from the CFG (with one state  $q_0$ ):

- start with empty stack
- shift: a transition in the PDA from  $q_0$  (to  $q_0$ ) putting a terminal symbol on the stack
- reduce: whenever the righthand side of a rule appears on top of the stack, pop the RHS and push the lefthand side (still staying in  $q_0$ ). Don't consume anything from the input.
- accept the string if the start symbol is in the stack and the end of string has been reached

If there is some derivation for a given sentence under the CFG, there will be a sequence of actions for which this NPDA accepts the string

**If there is some derivation for a given sentence under the CFG, there will be a sequence of actions for which this NPDA accepts the string**

But how do we find this derivation?

One way to do this is using so-called generalised LR parsing, which explores all possible paths of the above NPDA

Modern parsers do it differently, because GLR can be exponential in the worst-case

Shift-reduce parsers are highly efficient, they are linear in the length of the string, if they explore only one path

How to do that? **Learn from data** what actions to take at each point, and try to make the optimal decisions so that the correct parser will be found

This keeps the parser linear in the length of the string, but one small error can propagate through the whole parse, and lead to the wrong parse tree

# Try it out Yourselves!

## Recursive Descent Parser

```
>>> from nltk.app import rdparser  
>>> rdparser()
```

## Shift-Reduce Parser

```
>>> from nltk.app import srparser  
>>> srparser()
```

- We use CFGs to represent NL grammars
- Grammars need recursion to produce infinite sentences
- Most NL grammars have structural ambiguity
- A parser computes structure for an input automatically
- Recursive descent and shift-reduce parsing
- We'll examine more parsers in Lectures 17–22

**Reading:** J&M (2nd edition) Chapter 12 (intro – section 12.3), Chapter 13 (intro – section 13.3)

**Next lecture:** The CYK algorithm

# Chart Parsing: the CYK Algorithm

Informatics 2A: Lecture 18

Shay Cohen

3 November 2015



Deterministic parsing (e.g., LL(1)) aims to address a limited amount of local ambiguity – the problem of not being able to decide uniquely which grammar rule to use next in a left-to-right analysis of the input string.

By re-structuring the grammar, the parser can make a unique decision, based on a limited amount of look-ahead.

Recursive Descent parsing also demands grammar restructuring, in order to eliminate left-recursive rules that can get it into a hopeless loop.

## Left Recursion

But grammars for natural human languages should be **revealing**, re-structuring the grammar may destroy this. (Indirectly) left-recursive rules are needed in English.

$NP \rightarrow DET\ N$

$NP \rightarrow NPR$

$DET \rightarrow NP\ 's$

These rules generate NPs with possessive modifiers such as:

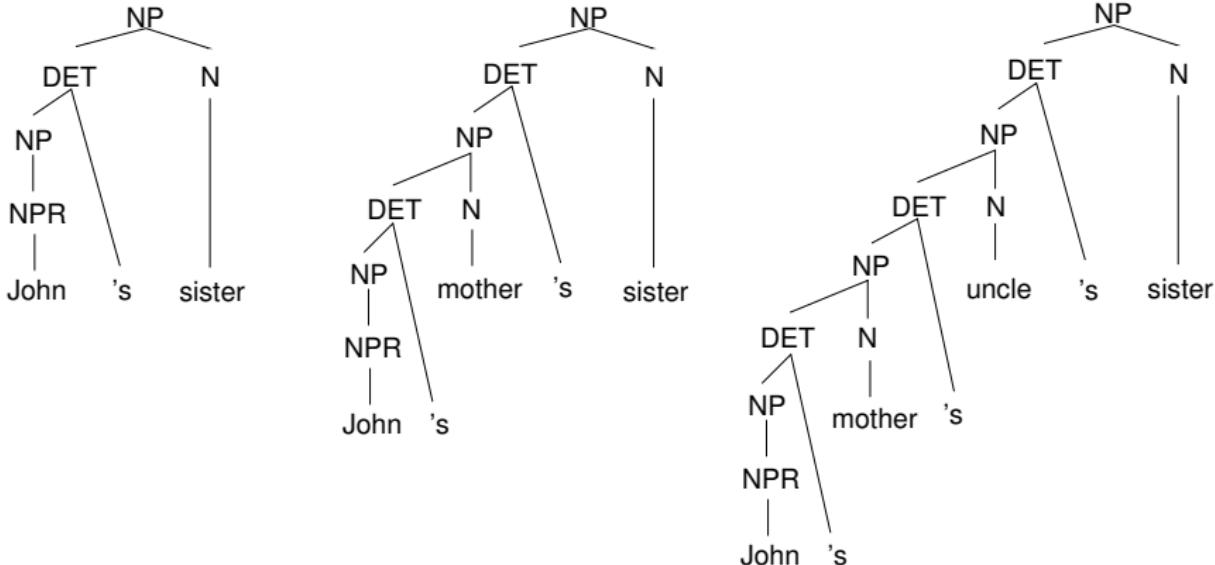
John's sister

John's mother's sister

John's mother's uncle's sister

John's mother's uncle's sister's niece

# Left Recursion



We don't want to re-structure our grammar rules just to be able to use a particular approach to parsing. Need an alternative.

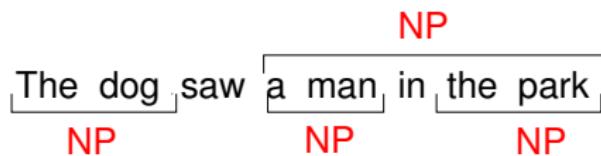
# Problems with Parsing as Search

- ① A **recursive descent parser** (top-down) will do badly if there are many different rules for the same LHS. Hopeless for rewriting parts of speech (preterminals) with words (terminals).
- ② A **shift-reduce parser** (bottom-up) does a lot of useless work: many phrase structures will be locally possible, but globally impossible. Also inefficient when there is much lexical ambiguity.
- ③ Both strategies do repeated work by **re-analyzing** the same substring many times.

We will see how **chart parsing** solves the re-parsing problem, and also copes well with ambiguity.

# Dynamic Programming

With a CFG, a parser should be able to avoid re-analyzing sub-strings because the analysis of any sub-string is **independent** of the rest of the parse.



The parser's exploration of its search space can exploit this independence if the parser uses **dynamic programming**.

Dynamic programming is the basis for all **chart parsing** algorithms.

# Parsing as Dynamic Programming

- Given a problem, systematically fill a table of solutions to sub-problems: this is called **memoization**.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them.
- For parsing, the sub-problems are analyses of sub-strings and correspond to **constituents** that have been found.
- Sub-trees are stored in a **chart** (aka **well-formed substring table**), which is a record of all the substructures that have ever been built during the parse.

Solves **re-parsing problem**: sub-trees are looked up, not re-parsed!  
Solves **ambiguity problem**: chart implicitly stores all parses!

# Depicting a Chart

A **chart** can be depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span (ie, starting **right before** the first word, ending **right after** the final one);
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index.
- It can contain information about the **type** of constituent (or constituents) that span(s) the substring, pointers to its sub-constituents, and/or **predictions** about what constituents might follow the substring.

# CYK Algorithm

CYK (Cocke, Younger, Kasami) is an algorithm for recognizing and recording constituents in the chart.

- Assumes that the grammar is in Chomsky Normal Form: rules all have form  $A \rightarrow BC$  or  $A \rightarrow w$ .
- Conversion to CNF can be done automatically.

NP	$\rightarrow$	Det Nom	NP	$\rightarrow$	Det Nom
Nom	$\rightarrow$	N   OptAP Nom	Nom	$\rightarrow$	book   orange   AP Nom
OptAP	$\rightarrow$	$\epsilon$   OptAdv A	AP	$\rightarrow$	heavy   orange   Adv A
A	$\rightarrow$	heavy   orange	A	$\rightarrow$	heavy   orange
Det	$\rightarrow$	a	Det	$\rightarrow$	a
OptAdv	$\rightarrow$	$\epsilon$   very	Adv	$\rightarrow$	very
N	$\rightarrow$	book   orange			

# CYK: an example

Let's look at a simple example before we explain the general case.

## Grammar Rules in CNF

NP	$\rightarrow$	Det Nom
Nom	$\rightarrow$	<i>book</i>   <i>orange</i>   AP Nom
AP	$\rightarrow$	<i>heavy</i>   <i>orange</i>   Adv A
A	$\rightarrow$	<i>heavy</i>   <i>orange</i>
Det	$\rightarrow$	<i>a</i>
Adv	$\rightarrow$	<i>very</i>

(N.B. Converting to CNF sometimes breeds duplication!)

Now let's parse: *a very heavy orange book*

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a					
1 very					
2 heavy					
3 orange					
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				
1 very					
2 heavy					
3 orange					
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				
1 very		Adv			
2 heavy					
3 orange					
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 a	2 very	3 heavy	4 orange	5 book
0 a	Det				
1 very		Adv			
2 heavy			A,AP		
3 orange					
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 a	2 very	3 heavy	4 orange	5 book
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP		
3 orange					
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP		
3 orange				Nom,A,AP	
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					Nom

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	Nom
4 book					Nom

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

	1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	Nom
3 orange				Nom,A,AP	Nom
4 book					Nom

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

		1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				NP	
1 very		Adv	AP	Nom	Nom	
2 heavy			A,AP	Nom	Nom	
3 orange				Nom,A,AP	Nom	
4 book						Nom

# Filling out the CYK chart

0 a 1 very 2 heavy 3 orange 4 book 5

		1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0 a	Det				NP	NP
1 very		Adv	AP	Nom	Nom	
2 heavy			A,AP	Nom	Nom	
3 orange				Nom,A,AP	Nom	
4 book						Nom

# CYK: The general algorithm

```
function CKY-Parse(words, grammar) returns table for  
j  $\leftarrow$  from 1 to LENGTH(words) do  
  table[j - 1, j]  $\leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$   
  for i  $\leftarrow$  from j - 2 downto 0 do  
    for k  $\leftarrow$  i + 1 to j - 1 do  
      table[i, j]  $\leftarrow$  table[i, j]  $\cup$   
       $\{A \mid A \rightarrow BC \in grammar,$   
       $B \in table[i, k]$   
       $C \in table[k, j]\}$ 
```

# CYK: The general algorithm

**function** CKY-Parse(*words*, *grammar*) **returns** *table* **for**

*j*  $\leftarrow$  **from** 1 **to** LENGTH(*words*) **do**

loop over the columns

*table*[*j* - 1, *j*]  $\leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$

fill bottom cell

**for** *i*  $\leftarrow$  **from** *j* - 2 **downto** 0 **do**

fill row *i* in column *j*

**for** *k*  $\leftarrow$  *i* + 1 **to** *j* - 1 **do**

loop over split locations

*table*[*i*, *j*]  $\leftarrow$  *table*[*i*, *j*]  $\cup$

between *i* and *j*

$\{A \mid A \rightarrow BC \in grammar,$

$B \in table[i, k]$

$C \in table[k, j]\}$

Check the grammar  
for rules that  
link the constituent  
in  $[i, k]$  with those  
in  $[k, j]$ . For each  
rule found store  
LHS in cell  $[i, j]$ .

## A succinct representation of CKY

We have a Boolean table called Chart, such that  $\text{Chart}[A, i, j]$  is true if there is a sub-phrase according the grammar that dominates words  $i$  through words  $j$

Build this chart recursively, similarly to the Viterbi algorithm:

For  $j > i + 1$ :

$$\text{Chart}[A, i, j] = \bigvee_{k=i+1}^{j-1} \bigvee_{A \rightarrow B C} \text{Chart}[B, i, k] \wedge \text{Chart}[C, k, j]$$

Seed the chart, for  $i + 1 = j$ :

$\text{Chart}[A, i, i + 1]$  = True if there exists a rule  $A \rightarrow w_{i+1}$  where  $w_{i+1}$  is the  $(i + 1)$ th word in the string

- So far, we just have a chart **recognizer**, a way of determining whether a string belongs to the given language.
- Changing this to a **parser** requires recording which existing constituents were combined to make each new constituent.
- This requires another field to record the one or more ways in which a constituent spanning  $(i,j)$  can be made from constituents spanning  $(i,k)$  and  $(k,j)$ . (More clearly displayed in **graph** representation, see next lecture.)
- In any case, for a fixed grammar, the CYK algorithm runs in time  $O(n^3)$  on an input string of  $n$  tokens.
- The algorithm identifies **all possible parses**.

# CYK-style parse charts

Even without converting a grammar to CNF, we can draw CYK-style parse charts:

		1 <i>a</i>	2 <i>very</i>	3 <i>heavy</i>	4 <i>orange</i>	5 <i>book</i>
0	<i>a</i>	Det			NP	NP
1	<i>very</i>		OptAdv	OptAP	Nom	Nom
2	<i>heavy</i>			A,OptAP	Nom	Nom
3	<i>orange</i>				N,Nom,A,AP	Nom
4	<i>book</i>					N,Nom

(We haven't attempted to show  $\epsilon$ -phrases here. Could in principle use cells below the main diagonal for this . . . )

However, CYK-style parsing will have run-time worse than  $O(n^3)$  if e.g. the grammar has rules  $A \rightarrow BCD$ .

## Second example

### Grammar Rules in CNF

$S \rightarrow NP\ VP$	$Nominal \rightarrow book flight money$
$S \rightarrow X1\ VP$	$Nominal \rightarrow Nominal\ noun$
$X1 \rightarrow Aux\ VP$	$Nominal \rightarrow Nominal\ PP$
$S \rightarrow book include prefer$	$VP \rightarrow book include prefer$
$S \rightarrow Verb\ NP$	$VPVerb \rightarrow NP$
$S \rightarrow X2$	$VP \rightarrow X2\ PP$
$S \rightarrow Verb\ PP$	$X2 \rightarrow Verb\ NP$
$S \rightarrow VP\ PP$	$VP \rightarrow Verb\ NP$
$NP \rightarrow TWA Houston$	$VP \rightarrow VP\ PP$
$NP \rightarrow Det\ Nominal$	$PP \rightarrow Preposition\ NP$
$Verb \rightarrow book include prefer$	$Noun \rightarrow book flight money$

Let's parse *Book the flight through Houston!*

## Second example

### Grammar Rules in CNF

$S \rightarrow NP\ VP$	$Nominal \rightarrow book flight money$
$S \rightarrow X1\ VP$	$Nominal \rightarrow Nominal\ noun$
$X1 \rightarrow Aux\ VP$	$Nominal \rightarrow Nominal\ PP$
$S \rightarrow book include prefer$	$VP \rightarrow book include prefer$
$S \rightarrow Verb\ NP$	$VPVerb \rightarrow NP$
$S \rightarrow X2$	$VP \rightarrow X2\ PP$
$S \rightarrow Verb\ PP$	$X2 \rightarrow Verb\ NP$
$S \rightarrow VP\ PP$	$VP \rightarrow Verb\ NP$
$NP \rightarrow TWA Houston$	$VP \rightarrow VP\ PP$
$NP \rightarrow Det\ Nominal$	$PP \rightarrow Preposition\ NP$
$Verb \rightarrow book include prefer$	$Noun \rightarrow book flight money$

Let's parse *Book the flight through Houston!*

## Second example

Book            the            flight            through            Houston

S, VP, Verb, Nominal, Noun [0, 1]				

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]				
	Det [1, 2]			

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]				
	Det [1, 2]			
		Nominal, Noun [2, 3]		

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]				
	Det [1, 2]			
		Nominal, Noun [2, 3]		
			Prep [3, 4]	

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]				
	Det [1, 2]			
		Nominal, Noun [2, 3]		
			Prep [3, 4]	
				NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]			
	Det [1, 2]			
		Nominal, Noun [2, 3]		
		Prep [3, 4]		
			NP, Proper- Noun [4, 5]	

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]			
	Det [1, 2]	NP [1, 3]		
		Nominal, Noun [2, 3]		
		Prep [3, 4]		
			NP, Proper- Noun [4, 5]	

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]		
	Det [1, 2]	NP [1, 3]		
		Nominal, Noun [2, 3]		
		Prep [3, 4]		NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]		
	Det [1, 2]	NP [1, 3]		
		Nominal, Noun [2, 3]	[2, 4]	
		Prep [3, 4]		NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]		
	Det [1, 2]	NP [1, 3]	[1, 4]	
		Nominal, Noun [2, 3]	[2, 4]	
		Prep [3, 4]		NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	
	Det [1, 2]	NP [1, 3]	[1, 4]	
		Nominal, Noun [2, 3]	[2, 4]	
		Prep [3, 4]		NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	
	Det [1, 2]	NP [1, 3]	[1, 4]	
		Nominal, Noun [2, 3]	[2, 4]	
		Prep [3, 4]	PP [3, 5]	
			NP, Proper- Noun [4, 5]	

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	
	Det [1, 2]	NP [1, 3]	[1, 4]	
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep [3, 4]	PP [3, 5]
				NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep [3, 4]	PP [3, 5]
				NP, Proper- Noun [4, 5]

## Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	S <sub>1</sub> , VP, X2, S <sub>2</sub> , VP, S <sub>3</sub> [0, 5]
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep [3, 4]	PP [3, 5]
				NP, Proper- Noun [4, 5]

# Visualizing the Chart

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]		S, VP, X2 [0, 3]		S <sub>1</sub> , VP, X2, S <sub>2</sub> , VP, S <sub>3</sub> [0, 5]
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep ← PP [3, 4] ↓ [3, 5]	NP, Proper- Noun [4, 5]

# Visualizing the Chart

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]		S, VP, X2 [0, 3]		$S_1$ , VP, X2, $S_2$ , VP, $S_3$ [0, 5]
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	←	Nominal [2, 5]
		[2, 4]		
			Prep [3, 4]	PP [3, 5]
				NP, Proper- Noun [4, 5]

# A Tribute to CKY (part 1/3)

*You, my CKY algorithm,  
dictate every parser's rhythm,  
if Cocke, Younger and Kasami hadn't bothered,  
all of our parsing dreams would have been shattered.*

*You are so simple, yet so powerful,  
and with the proper semiring and time,  
you will be truthful,  
to return the best parse - anything less would be a crime.*

*With dynamic programming or memoization,  
you are one of a kind,  
I really don't need to mention,  
if it werent for you, all syntax trees would be behind.*

## A Tribute to CKY (part 2/3)

*Failed attempts have been made to show there are better,  
for example, by using matrix multiplication,  
all of these impractical algorithms didn't matter  
you came out stronger, insisting on just using summation.*

*All parsing algorithms to you hail,  
at least those with backbones which are context-free,  
you will never become stale,  
as long as we need to have a syntax tree.*

*It doesn't matter that the C is always in front,  
or that the K and Y can swap,  
you are still on the same hunt,  
maximizing and summing, nonstop.*

# A Tribute to CKY (part 3/3)

*Every Informatics student knows you intimately,  
they have seen your variants dozens of times,  
you have earned that respect legitimately,  
and you will follow them through their primes.*

*CKY, going backward and forward,  
inside and out,  
it is so straightforward -  
You are the best, there is no doubt.*

- Parsing as search is inefficient (typically exponential time).
- Alternative: use dynamic programming and memoize sub-analysis in a chart to avoid duplicate work.
- The chart can be visualized as a matrix.
- The CYK algorithm builds a chart in  $O(n^3)$  time. The basic version gives just a recognizer, but it can be made into a parser if more info is recorded in the chart.

**Reading:** J&M (2nd ed), Chapter. 13, Sections 13.3–13.4  
NLTK Book, Chapter. 8 (*Analyzing Sentence Structure*), Section 8.4

**Next lecture:** the Earley parser or dynamic programming for top-down parsing

# Earley Parsing

Informatics 2A: Lecture 19

Shay Cohen

5 November 2015

## 1 The CYK chart as a graph

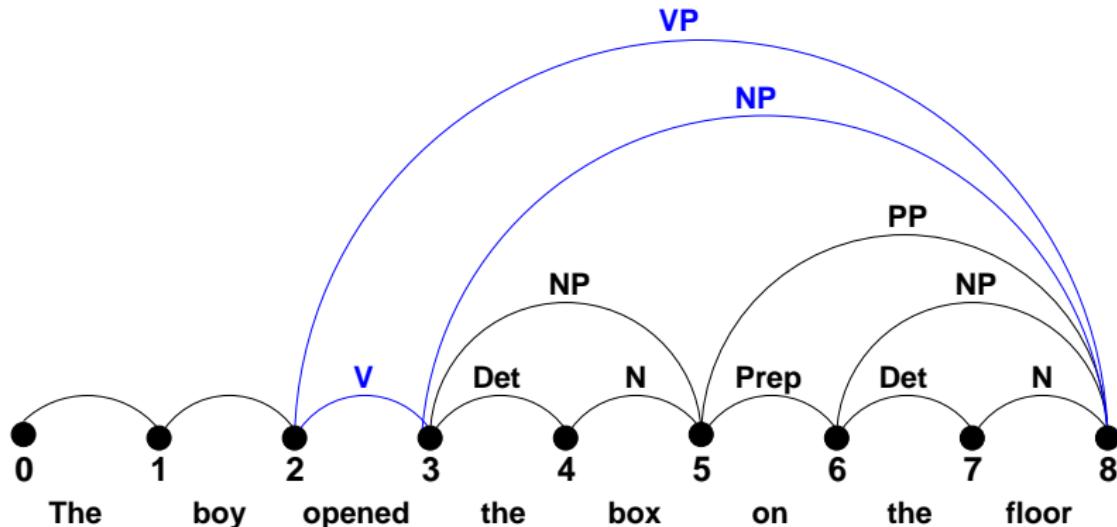
- What's wrong with CYK
- Adding Prediction to the Chart

## 2 The Earley Parsing Algorithm

- The PREDICTOR Operator
- The SCANNER Operator
- The COMPLETER Operator
- Earley parsing: example
- Comparing Earley and CYK

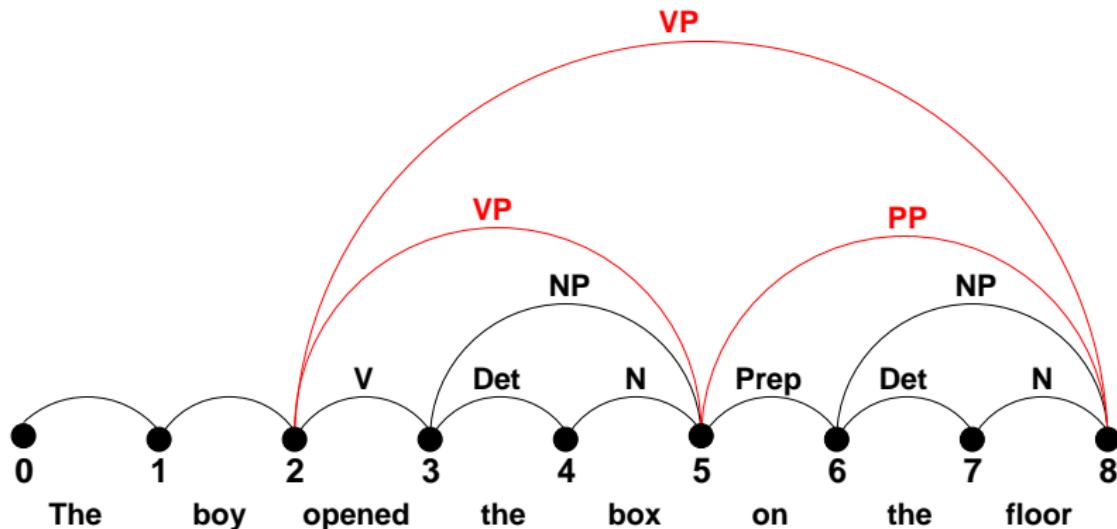
# Graph representation

The CYK chart can also be represented as a **graph**. E.g. for a certain grammar containing rules  $VP \rightarrow V \ NP$  and  $VP \rightarrow VP \ PP$ :



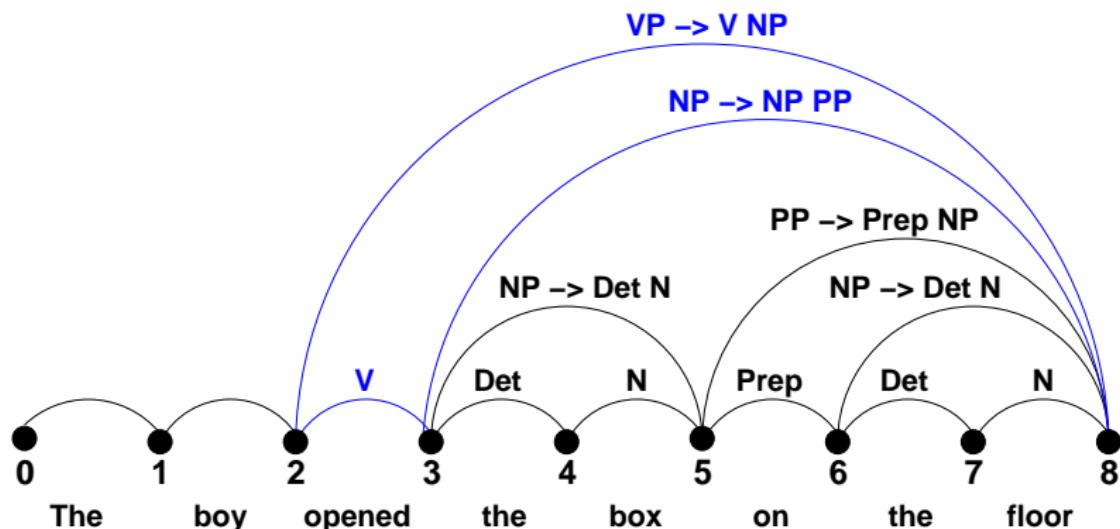
# Graph representation

An alternative analysis. Note we don't know which production the VP arc [2, 8] represents:  $VP \rightarrow V\ NP$  or  $VP \rightarrow VP\ PP$ .



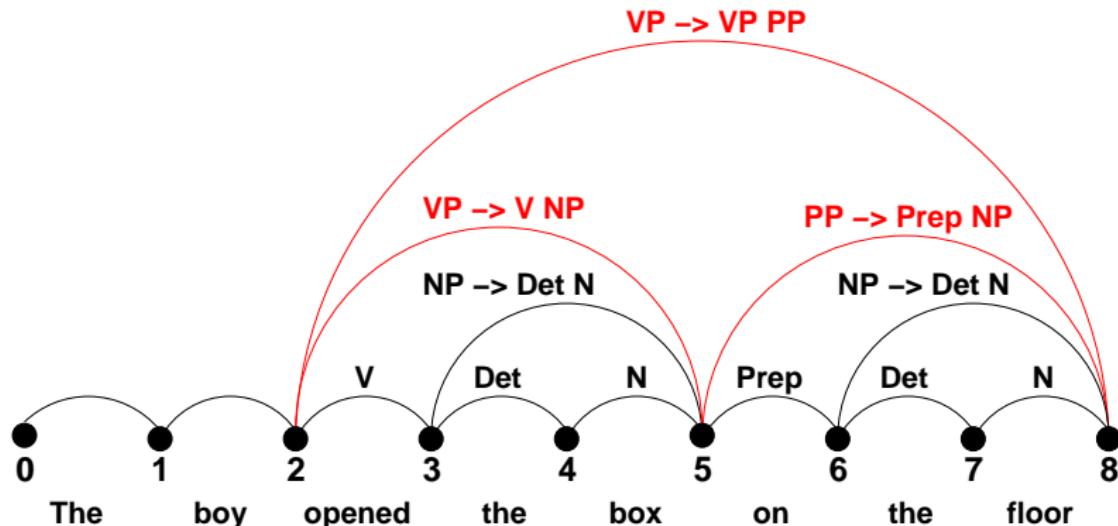
# CYK Chart entries

If the entire **production** were recorded, rather than just its LHS (ie, the constituent that it analyses), then we'd (usually) know.

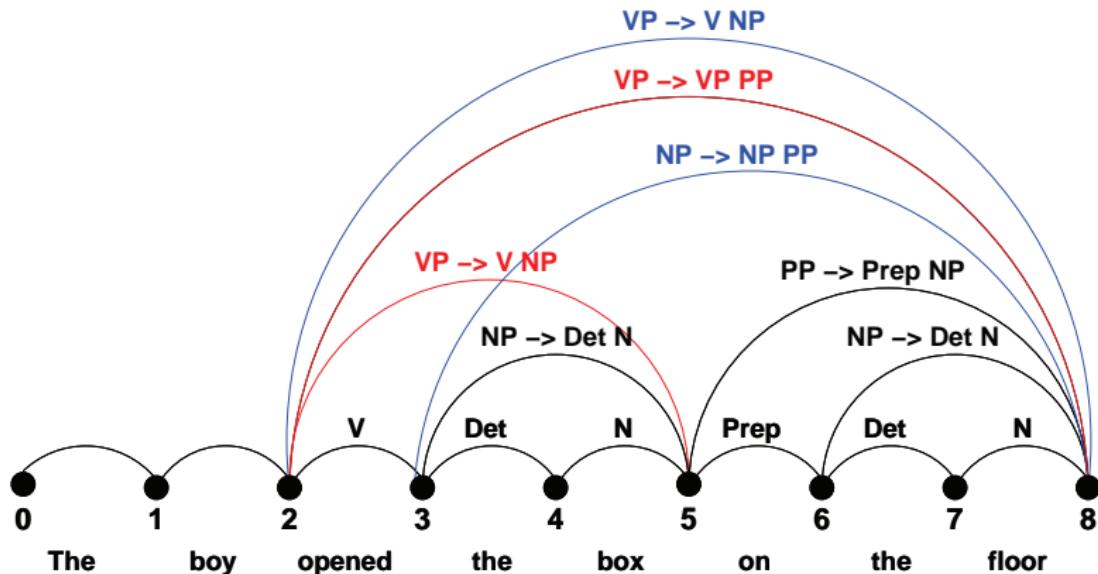


# CYK Chart entries

If the entire **production** were recorded, rather than just its LHS (ie, the constituent that it analyses), then we'd (usually) know.



## Chart entries: Both analyses



The CYK algorithm avoids redundant work by storing in a chart all the constituents it finds.

But it populates the table with **phantom constituents**, that don't form part of any complete parse. Can be a significant problem in long sentences.

The idea of the *Earley algorithm* is to avoid this, by only building constituents that are compatible with the input read so far.

**Key idea:** as well as **completed productions** (ones whose entire RHS have been recognized), we also record **incomplete productions** (ones for which there may so far be only partial evidence).

- **Incomplete productions** (aka **incomplete constituents**) are effectively **predictions** about what might come next and what will be learned from finding it.
- **Incomplete constituents** can be represented using an extended form of production rule called a **dotted rule**, e.g.  
 $VP \rightarrow V \bullet NP$ .
- The **dot** indicates how much of the RHS has already been found. The rest is a prediction of what is to come.

- Allows arbitrary CFGs
- Top-down control
- Fills a table in a single sweep over the input
- Table entries represent:
  - Completed constituents and their locations
  - In-progress constituents
  - Predicted constituents

## States

The table entries are called states and are represented with dotted-rules.

 $S \rightarrow \bullet VP [0,0]$ 

A *VP* is predicted at the start of the sentence

 $NP \rightarrow Det \bullet Nominal [1,2]$ 

An *NP* is in progress; seen *Det*, *Nominal* is expected

 $VP \rightarrow V NP \bullet [0,3]$ 

A *VP* has been found starting at 0 and ending at 3

Once chart is populated there should be an *S* the final column that spans from 0 to *N* and is complete:  $S \rightarrow \alpha \bullet [0, N]$ . If that's the case you're done.

# Sketch of Earley Algorithm

- ① **Predict** all the states you can upfront, working top-down from  $S$
- ② For each word in the input:
  - ① **Scan in** the word.
  - ② **Complete** or extend existing states based on matches.
  - ③ Add new **predictions**.
- ③ When out of words, look at the chart to see if you have a winner.

The algorithm uses three basic operations to process states in the chart: PREDICTOR and COMPLETER add states to the chart entry being processed; SCANNER adds a state to the next chart entry.

- Creates new states representing top-down expectations
- Applied to any state that has a non-terminal (other than a part-of-speech category) immediately to right of dot
- Application results in creation of one new state for each alternative expansion of that non-terminal
- New states placed into same chart entry as generating state

$S \rightarrow \bullet VP, [0,0]$		
$VP$	$\rightarrow \bullet$	$Verb, [0,0]$
$VP$	$\rightarrow \bullet$	$Verb\ NP, [0,0]$
$VP$	$\rightarrow \bullet$	$Verb\ NP\ PP, [0,0]$
$VP$	$\rightarrow \bullet$	$Verb\ PP, [0,0]$
$VP$	$\rightarrow \bullet$	$VP\ PP, [0,0]$

- Applies to states with a part-of-speech category to right of dot
- Incorporates into chart a state corresponding to prediction of a word with particular part-of-speech
- Creates new state from input state with dot advanced over predicted input category
- Unlike CYK, only parts-of-speech of a word that are predicted by some existing state will enter the chart (top-down input)

$VP \rightarrow \bullet \ Verb \ NP, [0,0]$

$VP \rightarrow book \bullet NP, [0,1]$

## COMPLETER

- Applied to state when its dot has reached right end of the rule
- This means that parser has successfully discovered a particular grammatical category over some span of the input
- COMPLETER finds and advances all previously created states that were looking for this category at this position in input
- Creates states copying the older state, advancing dot over expected category, and installing new state in chart

$NP \rightarrow Det\ Nominal \bullet, [1,3]$

finds state

$VP \rightarrow Verb \bullet NP, [0,1]$

finds state

$VP \rightarrow Verb \bullet NP\ PP, [0,1]$

- Applied to state when its dot has reached right end of the rule
- This means that parser has successfully discovered a particular grammatical category over some span of the input
- COMPLETER finds and advances all previously created states that were looking for this category at this position in input
- Creates states copying the older state, advancing dot over expected category, and installing new state in chart

$$NP \rightarrow Det\ Nominal \bullet, [1,3]$$

finds state	VP	$\rightarrow$	Verb $\bullet$ $NP, [0,1]$
finds state	VP	$\rightarrow$	Verb $\bullet$ $NP\ PP, [0,1]$
adds complete state	VP	$\rightarrow$	Verb $NP \bullet, [0,3]$
adds incomplete state	VP	$\rightarrow$	Verb $NP \bullet\ PP, [0,3]$

# Earley parsing: example

We will use the grammar to parse the sentence “*Book that flight*”.

## Grammar Rules

$S \rightarrow NP\ VP$	$VP \rightarrow Verb$
$S \rightarrow Aux\ NP\ VP$	$VP \rightarrow Verb\ NP$
$S \rightarrow VP$	$VP \rightarrow Verb\ NP\ PP$
$NP \rightarrow Pronoun$	$VP \rightarrow Verb\ PP$
$NP \rightarrow Proper-Noun$	$VP \rightarrow VP\ PP$
$NP \rightarrow Det\ Nominal$	$PP \rightarrow Preposition\ NP$
$Nominal \rightarrow Noun$	$Verb \rightarrow book include prefer$
$Nominal \rightarrow Nominal\ Noun$	$Noun \rightarrow book flight meal$
$Nominal \rightarrow Nominal\ PP$	$Det \rightarrow that this these$

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[0]

state	rule	start/end	reason
S1	$S \rightarrow \bullet NP VP$	[0,0]	Predictor
S2	$S \rightarrow \bullet Aux NP VP$	[0,0]	Predictor
S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
S5	$NP \rightarrow \bullet Proper-Noun$	[0,0]	Predictor
S6	$NP \rightarrow \bullet Det Nominal$	[0,0]	Predictor
S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
S8	$VP \rightarrow \bullet Verb NP$	[0,0]	Predictor
S9	$VP \rightarrow \bullet Verb NP PP$	[0,0]	Predictor
S10	$VP \rightarrow \bullet Verb PP$	[0,0]	Predictor
S11	$VP \rightarrow \bullet VP PP$	[0,0]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$\text{Verb} \rightarrow \text{book } \bullet$	[0,1]	Scanner
S13	$\text{VP} \rightarrow \text{Verb } \bullet$	[0,1]	Completer
S14	$\text{VP} \rightarrow \text{Verb } \bullet \text{ NP}$	[0,1]	Completer
S15	$\text{VP} \rightarrow \text{Verb } \bullet \text{ NP PP}$	[0,1]	Completer
S16	$\text{VP} \rightarrow \text{Verb } \bullet \text{ PP}$	[0,1]	Completer
S17	$S \rightarrow \text{VP } \bullet$	[0,1]	Completer
S18	$\text{VP} \rightarrow \text{VP } \bullet \text{ PP}$	[1,1]	Completer
S19	$\text{NP} \rightarrow \bullet \text{ Pronoun}$	[1,1]	Predictor
S20	$\text{NP} \rightarrow \bullet \text{ Proper-Noun}$	[1,1]	Predictor
S21	$\text{NP} \rightarrow \bullet \text{ Det Nominal}$	[1,1]	Predictor
S22	$\text{PP} \rightarrow \bullet \text{ Prep NP}$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
<b>S14</b>	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
<b>S19</b>	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
<b>S20</b>	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
<b>S21</b>	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep NP$	[1,1]	Predictor

## Earley parsing: example[1]

state	rule	start/end	reason
S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
S15	$VP \rightarrow Verb \bullet NP PP$	[0,1]	Completer
S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
S17	$S \rightarrow VP \bullet$	[0,1]	Completer
S18	$VP \rightarrow VP \bullet PP$	[1,1]	Completer
S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
S20	$NP \rightarrow \bullet Proper-Noun$	[1,1]	Predictor
S21	$NP \rightarrow \bullet Det\ Nominal$	[1,1]	Predictor
S22	$PP \rightarrow \bullet Prep\ NP$	[1,1]	Predictor

## Earley parsing: example[2]

state	rule	start/end	reason
S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
S24	$NP \rightarrow Det \bullet Nominal$	[1,2]	Completer
S25	$Nominal \rightarrow \bullet Noun$	[2,2]	Predictor
S26	$Nominal \rightarrow \bullet Nominal Noun$	[2,2]	Predictor
S27	$Nominal \rightarrow \bullet Nominal PP$	[2,2]	Predictor

## Earley parsing: example[2]

state	rule	start/end	reason
S23	$\text{Det} \rightarrow \text{that} \bullet$	[1,2]	Scanner
S24	$\text{NP} \rightarrow \text{Det } \bullet \text{ Nominal}$	[1,2]	Completer
S25	$\text{Nominal} \rightarrow \bullet \text{ Noun}$	[2,2]	Predictor
S26	$\text{Nominal} \rightarrow \bullet \text{ Nominal Noun}$	[2,2]	Predictor
S27	$\text{Nominal} \rightarrow \bullet \text{ Nominal PP}$	[2,2]	Predictor

## Earley parsing: example[2]

state	rule	start/end	reason
S23	$\text{Det} \rightarrow \text{that } \bullet$	[1,2]	Scanner
S24	$\text{NP} \rightarrow \text{Det } \bullet \text{ Nominal}$	[1,2]	Completer
S25	$\text{Nominal} \rightarrow \bullet \text{ Noun}$	[2,2]	Predictor
S26	$\text{Nominal} \rightarrow \bullet \text{ Nominal Noun}$	[2,2]	Predictor
S27	$\text{Nominal} \rightarrow \bullet \text{ Nominal PP}$	[2,2]	Predictor

## Earley parsing: example[2]

state	rule	start/end	reason
S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
S24	$NP \rightarrow Det \bullet Nominal$	[1,2]	Completer
S25	$Nominal \rightarrow \bullet Noun$	[2,2]	Predictor
S26	$Nominal \rightarrow \bullet Nominal Noun$	[2,2]	Predictor
S27	$Nominal \rightarrow \bullet Nominal PP$	[2,2]	Predictor

## Earley parsing: example[3]

state	rule	start/end	reason
S28	$Noun \rightarrow \bullet flight$	[2,3]	Scanner
S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
S31	$Nominal \rightarrow Nominal \bullet\ Noun$	[2,3]	Completer
S32	$Nominal \rightarrow Nominal \bullet\ PP$	[2,3]	Completer
S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
S34	$VP \rightarrow Verb\ NP \bullet\ PP$	[0,3]	Completer
S35	$PP \rightarrow Prep \bullet\ NP$	[3,3]	Predictor
S36	$S \rightarrow VP \bullet$	[0,3]	Completer
S37	$VP \rightarrow VP \bullet\ PP$	[0,3]	Completer

## Earley parsing: example[3]

state	rule	start/end	reason
S28	$\text{Noun} \rightarrow \bullet \text{ flight}$	[2,3]	Scanner
S29	$\text{Nominal} \rightarrow \text{Noun} \bullet$	[2,3]	Completer
S30	$\text{NP} \rightarrow \text{Det Nominal} \bullet$	[1,3]	Completer
S31	$\text{Nominal} \rightarrow \text{Nominal} \bullet \text{ Noun}$	[2,3]	Completer
S32	$\text{Nominal} \rightarrow \text{Nominal} \bullet \text{ PP}$	[2,3]	Completer
S33	$\text{VP} \rightarrow \text{Verb NP} \bullet$	[0,3]	Completer
S34	$\text{VP} \rightarrow \text{Verb NP} \bullet \text{ PP}$	[0,3]	Completer
S35	$\text{PP} \rightarrow \text{Prep} \bullet \text{ NP}$	[3,3]	Predictor
S36	$\text{S} \rightarrow \text{VP} \bullet$	[0,3]	Completer
S37	$\text{VP} \rightarrow \text{VP} \bullet \text{ PP}$	[0,3]	Completer

## Earley parsing: example[3]

state	rule	start/end	reason
S28	$Noun \rightarrow \bullet flight$	[2,3]	Scanner
S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
S31	$Nominal \rightarrow Nominal \bullet\ Noun$	[2,3]	Completer
S32	$Nominal \rightarrow Nominal \bullet\ PP$	[2,3]	Completer
S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
S34	$VP \rightarrow Verb\ NP \bullet\ PP$	[0,3]	Completer
S35	$PP \rightarrow Prep \bullet\ NP$	[3,3]	Predictor
S36	$S \rightarrow VP \bullet$	[0,3]	Completer
S37	$VP \rightarrow VP \bullet\ PP$	[0,3]	Completer

## Earley parsing: example[3]

state	rule	start/end	reason
S28	$Noun \rightarrow \bullet flight$	[2,3]	Scanner
S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
S31	$Nominal \rightarrow Nominal \bullet\ Noun$	[2,3]	Completer
S32	$Nominal \rightarrow Nominal \bullet\ PP$	[2,3]	Completer
S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
S34	$VP \rightarrow Verb\ NP \bullet\ PP$	[0,3]	Completer
S35	$PP \rightarrow Prep \bullet\ NP$	[3,3]	Predictor
S36	$S \rightarrow VP \bullet$	[0,3]	Completer
S37	$VP \rightarrow VP \bullet\ PP$	[0,3]	Completer

# The Earley Algorithm

---

```
function EARLEY-PARSE(words, grammar) returns chart
    ENQUEUE(( $\gamma \rightarrow \bullet S, [0, 0]$ ), chart[0])
    for i  $\leftarrow$  from 0 to LENGTH(words) do
        for each state in chart[i] do
            if INCOMPLETE?(state) and
                NEXT-CAT(state) is not a part of speech then
                    PREDICTOR(state)
                elseif INCOMPLETE?(state) and
                    NEXT-CAT(state) is a part of speech then
                    SCANNER(state)
                else
                    COMPLETER(state)
            end
        end
    return(chart)
```

---

# The Earley Algorithm

```
procedure PREDICTOR( $(A \rightarrow \alpha \bullet B \beta, [i, j])$ )
    for each  $(B \rightarrow \gamma)$  in GRAMMAR-RULES-FOR( $B, grammar$ ) do
        ENQUEUE( $(B \rightarrow \bullet \gamma, [j, j]), chart[j]$ )
    end

procedure SCANNER( $(A \rightarrow \alpha \bullet B \beta, [i, j])$ )
    if  $B \subset \text{PARTS-OF-SPEECH}(word[j])$  then
        ENQUEUE( $(B \rightarrow word[j], [j, j+1]), chart[j+1]$ )
    end

procedure COMPLETER( $(B \rightarrow \gamma \bullet, [j, k])$ )
    for each  $(A \rightarrow \alpha \bullet B \beta, [i, j])$  in  $chart[j]$  do
        ENQUEUE( $(A \rightarrow \alpha B \bullet \beta, [i, k]), chart[k]$ )
    end
```

# Parsing the Input

As with CYK we have formulated a **recognizer**. We can change it to a **parser** by adding backpointers so that each state knows where it came from.

Chart[1]	S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
Chart[2]	S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
Chart[3]	S28	$Noun \rightarrow flight \bullet$	[2,3]	Scanner
	S29	$Nominal \rightarrow Noun \bullet$	[2,3]	(S28)
	S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	(S23, S29)
	S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	(S12, S30)
	S36	$S \rightarrow VP \bullet$	[0,3]	(S33)

## Comparing Earley and CYK

- For such a simple example, there seems to be a lot of useless stuff in the chart.
- We are predicting phrases that aren't there at all!
- That's the flipside to the CYK problem.

- For such a simple example, there seems to be a lot of useless stuff in the chart.
- **We are predicting phrases that aren't there at all!**
- That's the flipside to the CYK problem.

**Did we solve ambiguity?**

- For such a simple example, there seems to be a lot of useless stuff in the chart.
- We are predicting phrases that aren't there at all!
- That's the flipside to the CYK problem.

**Did we solve ambiguity?** Both CYK and Earley may result in multiple  $S$  structures for the  $[0, N]$  table entry. Of course, neither can tell us which one is 'right'.

# The Asymptotic Complexity of Earley and CKY

- Both algorithms are cubic in  $n$  (length of string)
- CKY needs to construct  $O(n^2)$  elements in the chart (in the worst-case), and processing each element to create it is  $O(n)$ , so it is  $O(n^3)$  in total
- Earley also needs to construct  $O(n^2)$  elements, and the COMPLETER operation takes  $O(n)$  time. It could potentially run on  $O(n^2)$  elements, so the complexity is again  $O(n^3)$

## More about Asymptotic Complexity of Earley

- The COMPLETER operation really takes  $O(i^2)$  at iteration  $i$
- For unambiguous grammars, Earley shows that the COMPLETER operation can take at most  $O(i)$  time
- This means that the complexity for unambiguous grammars is  $O(n^2)$
- There are also some specialised grammars for which the Earley algorithm takes  $O(n)$  time

## Connection between the Earley Algorithm and CKY

What happens if we run the Earley algorithm on a grammar in Chomsky normal form?

- This is essentially CKY with top-down filtering
- It will only create (completed) elements in the chart, if there is a left-most derivation that leads to that constituent

- The Earley algorithm uses dynamic programming to implement a **top-down** search strategy.
- Single left to right pass that fills chart with entries.
- Dotted rule represents progress in recognizing RHS of rule.
- Algorithm always moves forward, never backtracks to previous chart entry, once it has moved on.
- States are processed using PREDICTOR, COMPLETER, SCANNER operations.

**Reading:** Same as for Lecture 17

**Next lecture:** Resolving ambiguity using statistical parsing.

# Probabilistic Context-Free Grammars

## Informatics 2A: Lecture 20

Shay Cohen

6 November, 2015

## 1 Motivation

## 2 Probabilistic Context-Free Grammars

- Definition
- Conditional Probabilities
- Applications
- Probabilistic CYK

Three things motivate the use of probabilities in grammars and parsing:

- ① Syntactic disambiguation – main motivation
- ② Coverage – issues in developing a grammar for a language
- ③ Representativeness – adapting a parser to new domains, texts.

## Motivation 1: Ambiguity

- Amount of ambiguity increases with sentence length.
- Real sentences are fairly long (avg. sentence length in the *Wall Street Journal* is 25 words).
- The amount of (unexpected!) ambiguity increases rapidly with sentence length. This poses a problem, even for chart parsers, if they have to keep track of all possible analyses.
- It would reduce the amount of work required if we could **ignore** improbable analyses.

A second provision passed by the Senate and House would eliminate a rule allowing companies that post losses resulting from LBO debt to receive refunds of taxes paid over the previous three years. [wsj\_1822] (33 words)

## Motivation 2: Coverage

- It is actually very difficult to write a grammar that covers all the constructions used in ordinary text or speech.
- Typically hundreds of rules are required in order to capture both all the different linguistic patterns and all the different possible analyses of the same pattern. (How many grammar rules did we have to add to cover three different analyses of *You made her duck?*)
- Ideally, one wants to **induce** (learn) a grammar from a corpus.
- Grammar induction requires probabilities.

## Motivation 3: Representativeness

The likelihood of a particular construction can vary, depending on:

- **register** (formal vs. informal): eg, *greenish*, *alot*, subject-drop (*Want a beer?*) are all more probable in informal than formal register;
- **genre** (newspapers, essays, mystery stories, jokes, ads, etc.): Clear from the difference in PoS-taggers trained on different genres in the Brown Corpus.
- **domain** (biology, patent law, football, etc.).

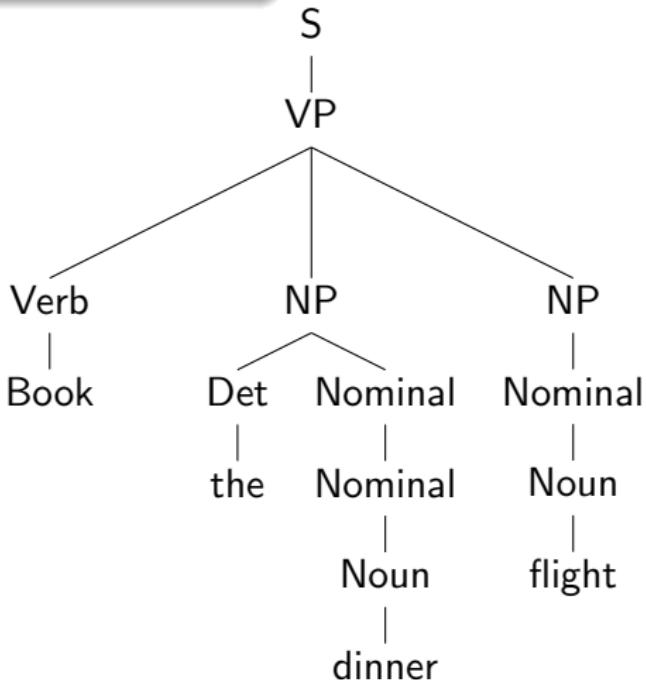
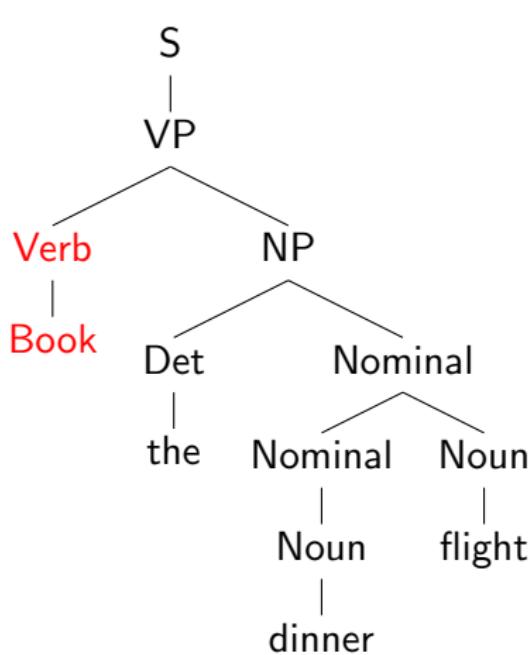
Probabilistic grammars and parsers can reflect these kinds of distributions.

## Example Parses for an Ambiguous Sentence

Book the dinner flight.

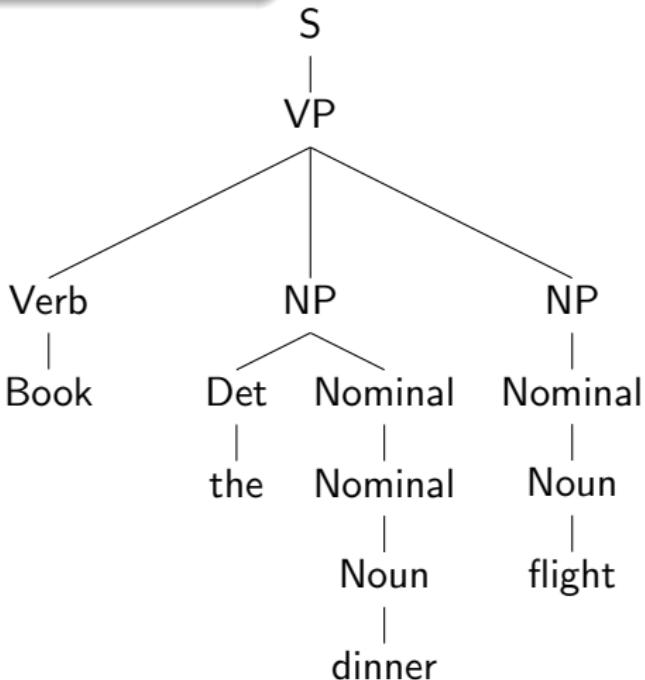
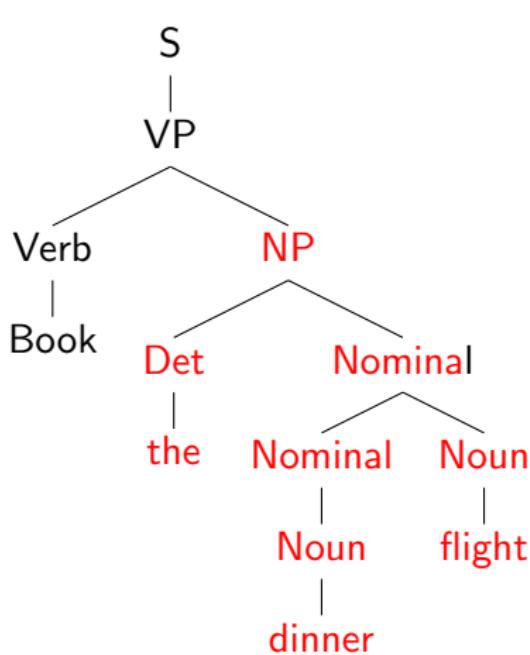
# Example Parses for an Ambiguous Sentence

Book the dinner flight.



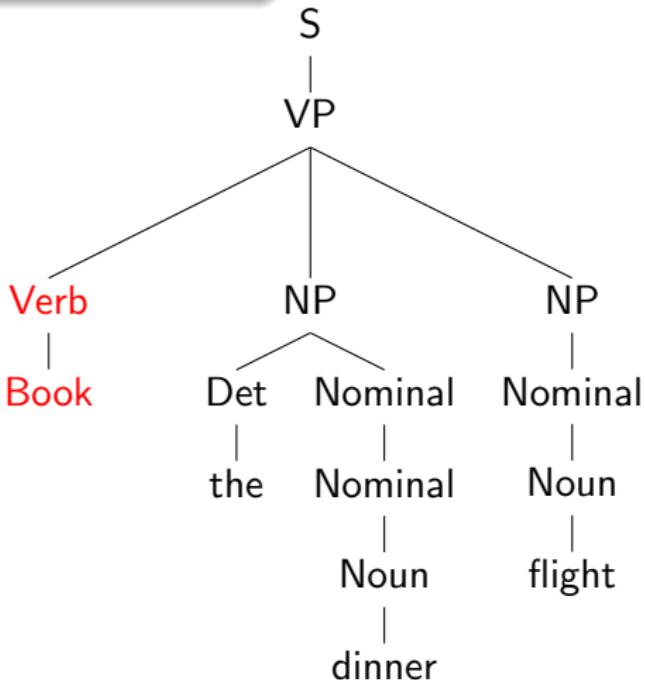
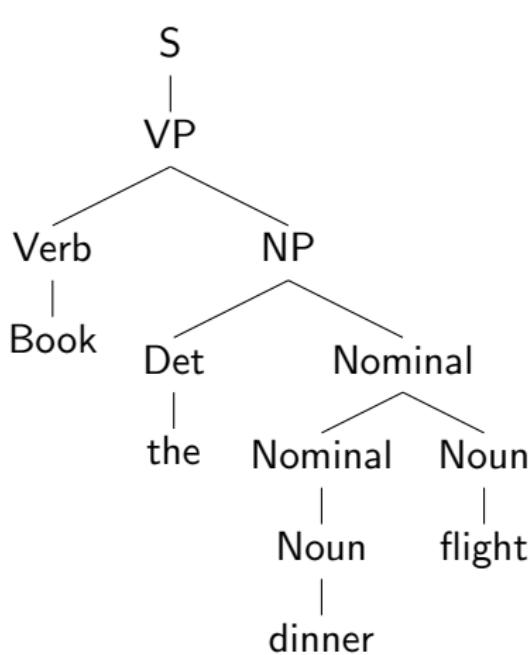
# Example Parses for an Ambiguous Sentence

Book the dinner flight.



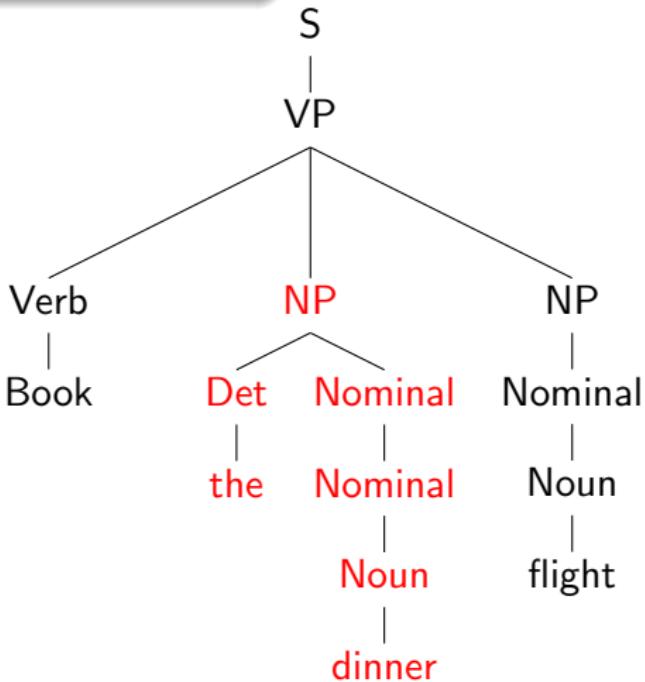
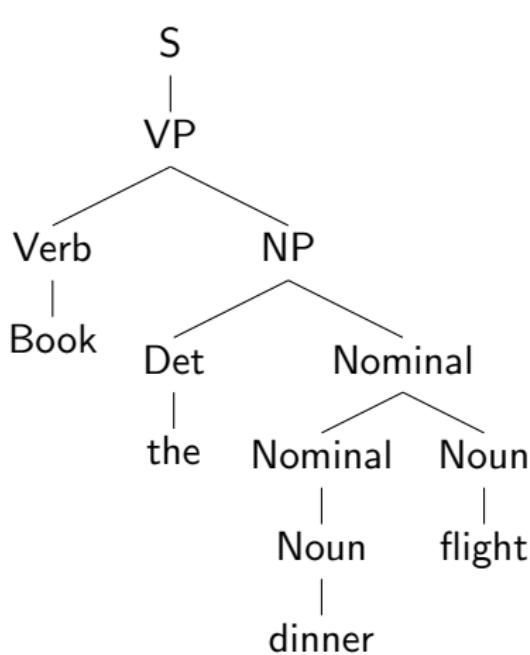
# Example Parses for an Ambiguous Sentence

Book the dinner flight.



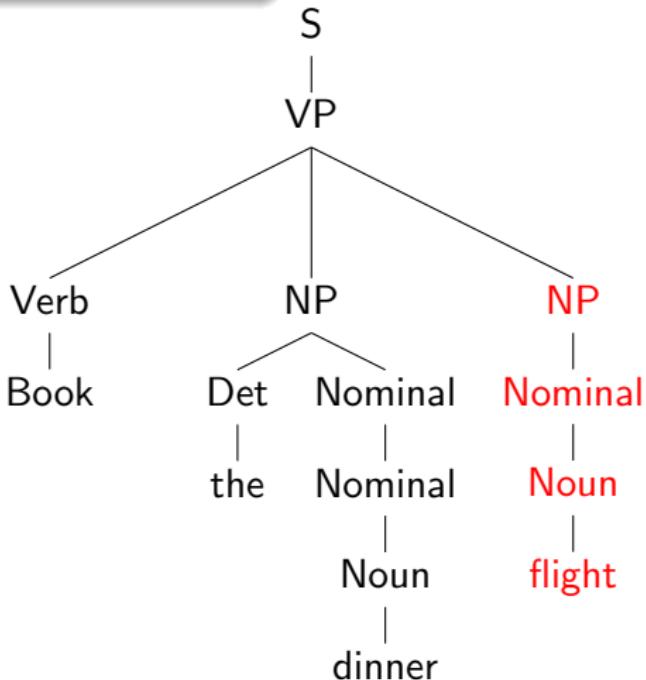
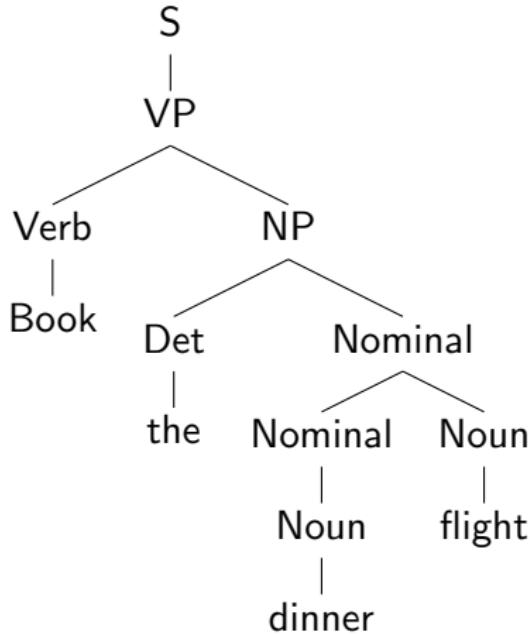
# Example Parses for an Ambiguous Sentence

Book the dinner flight.



# Example Parses for an Ambiguous Sentence

Book the dinner flight.



A PCFG  $\langle N, \Sigma, R, S \rangle$  is defined as follows:

- $N$  is the set of non-terminal symbols
- $\Sigma$  is the terminals (disjoint from  $N$ )
- $R$  is a set of rules of the form  $A \rightarrow \beta[p]$   
where  $A \in N$  and  $\beta \in (\sigma \cup N)^*$ ,  
and  $p$  is a number between 0 and 1
- $S$  a start symbol,  $S \in N$

A PCFG is a CFG in which each rule is associated with a probability.

What does the  $p$  associated with each rule express?

It expresses the probability that the LHS non-terminal will be expanded as the RHS sequence.

- $P(A \rightarrow \beta | A)$
- $\sum_{\beta} P(A \rightarrow \beta | A) = 1$
- The sum of the probabilities associated with all of the rules expanding the non-terminal  $A$  is required to be 1.

$$A \rightarrow \beta [p] \quad \text{or} \quad P(A \rightarrow \beta | A) = p \quad \text{or} \quad P(A \rightarrow \beta) = p$$

# Example Grammar

$S \rightarrow NP\ VP$	[.80]	$Det \rightarrow the$	[.10]
$S \rightarrow Aux\ NP\ VP$	[.15]	$Det \rightarrow a$	[.90]
$S \rightarrow VP$	[.05]	$Noun \rightarrow book$	[.10]
$NP \rightarrow Pronoun$	[.35]	$Noun \rightarrow flight$	[.30]
$NP \rightarrow Proper-Noun$	[.30]	$Noun \rightarrow dinner$	[.60]
$NP \rightarrow Det\ Nominal$	[.20]	$Proper-Noun \rightarrow Houston$	[.60]
$NP \rightarrow Nominal$	[.15]	$Proper-Noun \rightarrow NWA$	[.40]
$Nominal \rightarrow Noun$	[.75]	$Aux \rightarrow does$	[.60]
$Nominal \rightarrow Nominal\ Noun$	[.05]	$Aux \rightarrow can$	[.40]
$VP \rightarrow Verb$	[.35]	$Verb \rightarrow book$	[.30]
$VP \rightarrow Verb\ NP$	[.20]	$Verb \rightarrow include$	[.30]
$VP \rightarrow Verb\ NP\ PP$	[.10]	$Verb \rightarrow prefer$	[.20]
$VP \rightarrow Verb\ PP$	[.15]	$Verb \rightarrow sleep$	[.20]

## PCFGs as a random process

Start with the root node, and at each step, probabilistically expand the nodes until you hit a terminal symbol:

Question: Does this process always have to terminate?

Question: Does this process always have to terminate?

Consider the grammar, for some  $\epsilon > 0$ :

## Example

$S \rightarrow S S$  with probability  $0.5 + \epsilon$

$S \rightarrow a$  with probability  $0.5 - \epsilon$

Question: Does this process always have to terminate?

Consider the grammar, for some  $\epsilon > 0$ :

## Example

$S \rightarrow S S$  with probability  $0.5 + \epsilon$

$S \rightarrow a$  with probability  $0.5 - \epsilon$

Can potentially not terminate.

We get a “monster tree” with infinite number of nodes.

When we read a grammar off a treebank, that kind of grammar is highly unlikely to arise.

We have a “Markovian” process here (limited memory of history)

Everything above a given node in the tree is conditionally independent of everything below that node if we know what is the nonterminal in that node

Another way to think about it: once we get to a new nonterminal and continue from there, we forget the whole derivation up to that point, and focus on that nonterminal as if it is a new root node

Too strong independence assumptions for natural language data.

# PCFGs and disambiguation

- A PCFG assigns a probability to every parse tree or derivation associated with a sentence.
- This probability is the product of the rules applied in building the parse tree.

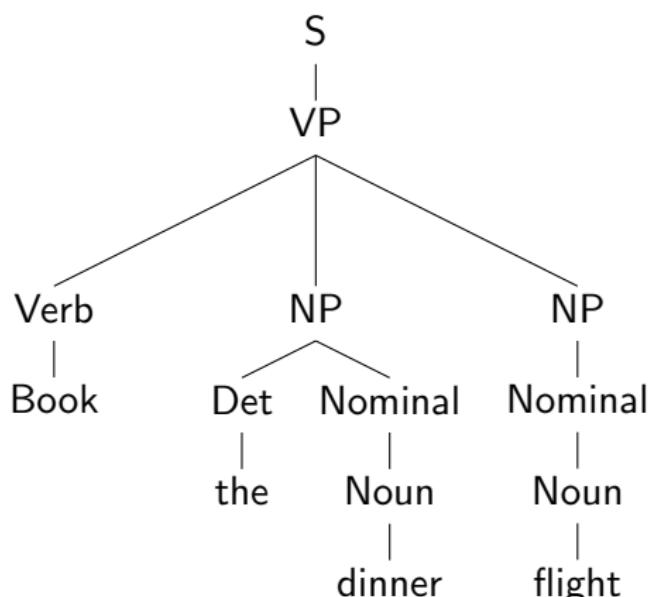
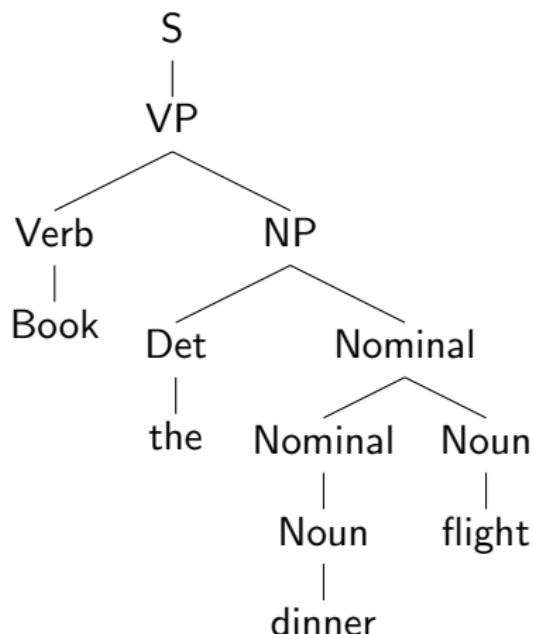
$$P(T, S) = \prod_{i=1}^n P(A_i \rightarrow \beta_i) \quad n \text{ is number of rules in } T$$

$$P(T, S) = P(T)P(S|T) = P(S)P(T|S) \text{ by definition}$$

But  $P(S|T) = 1$  because  $S$  is determined by  $T$

$$\text{So } P(T, S) = P(T)$$

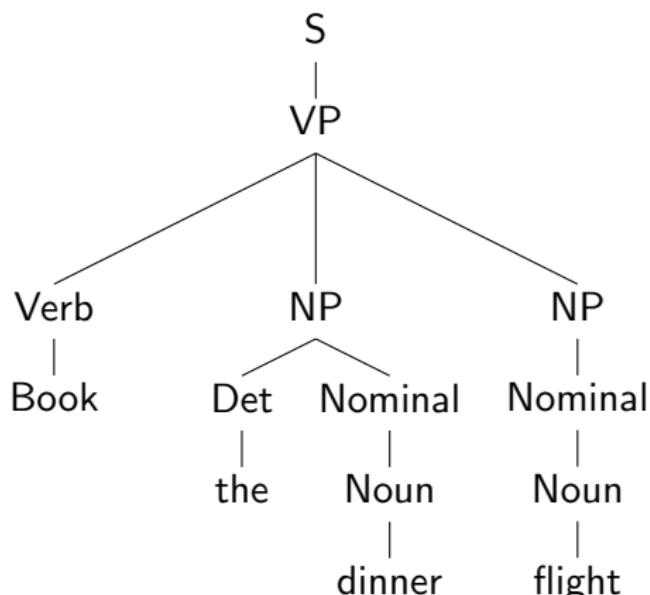
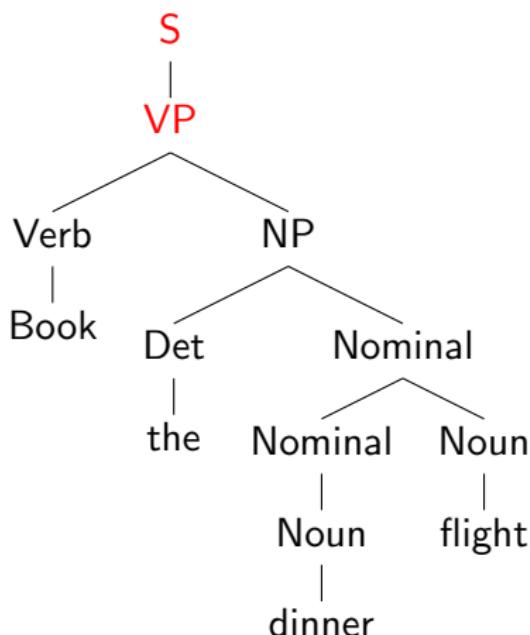
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

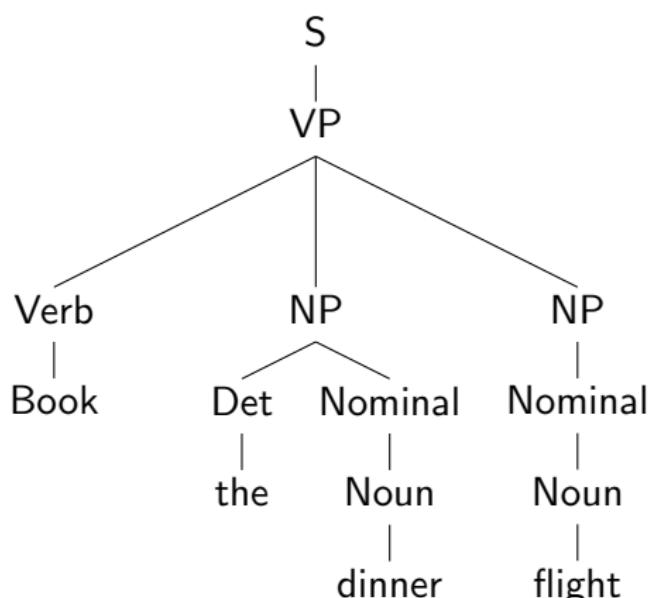
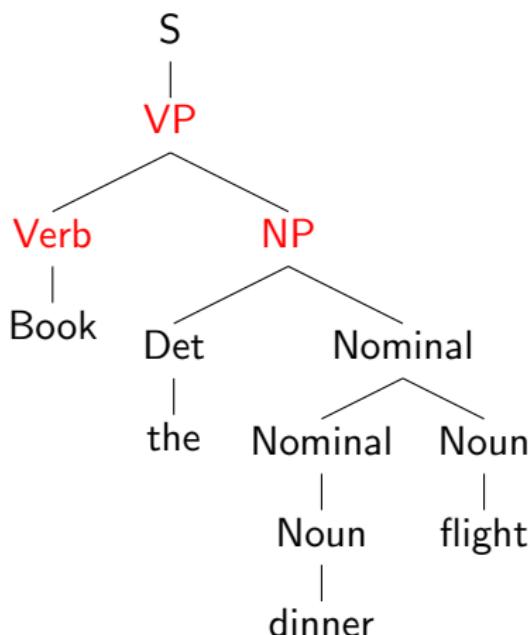
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

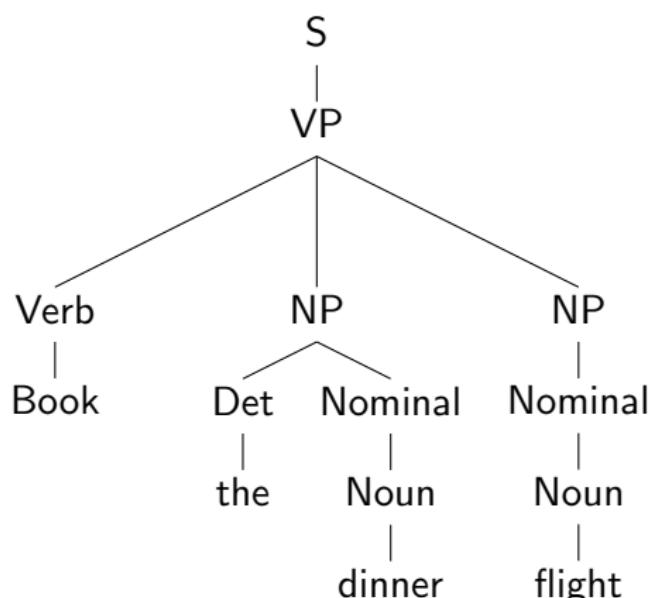
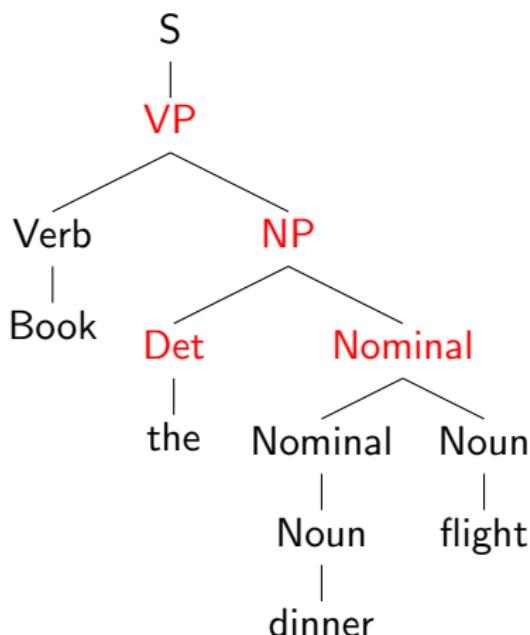
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

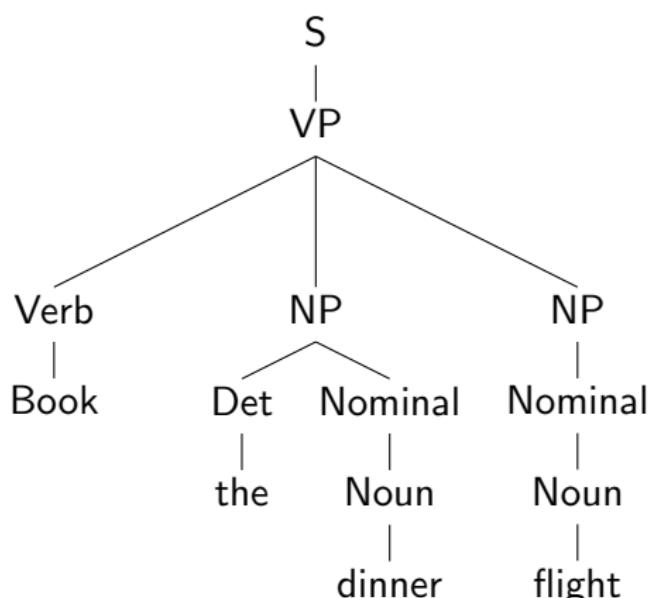
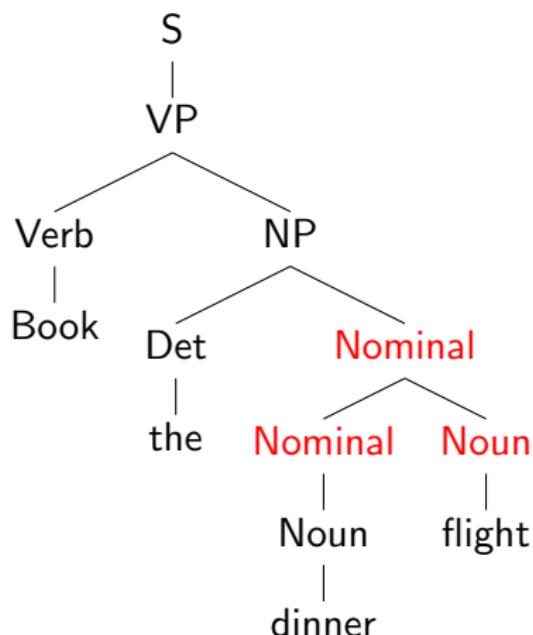
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

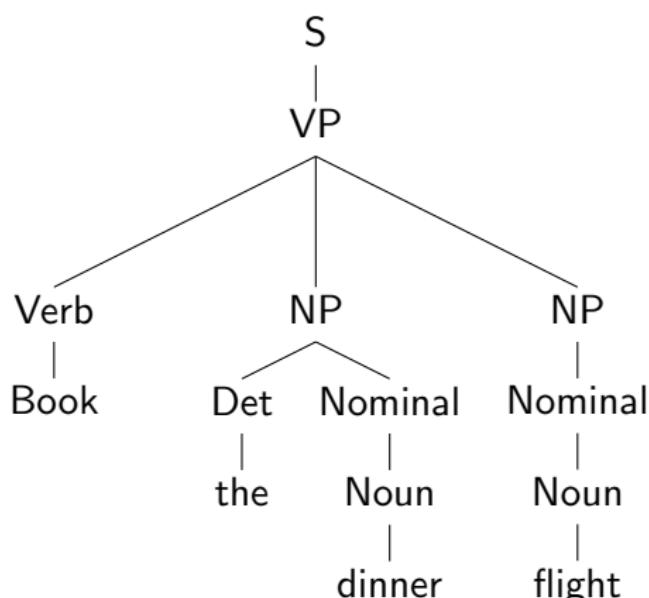
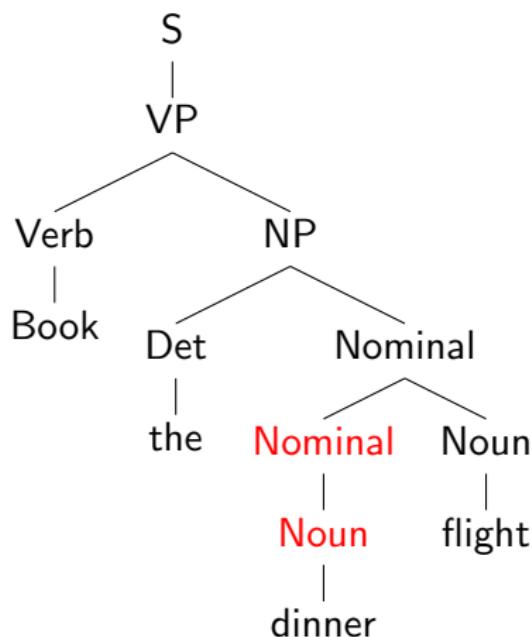
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * \textcolor{red}{.20} * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

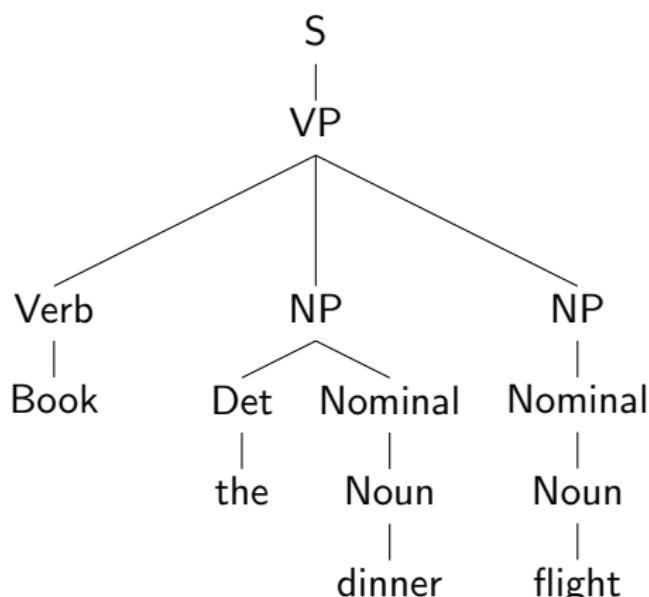
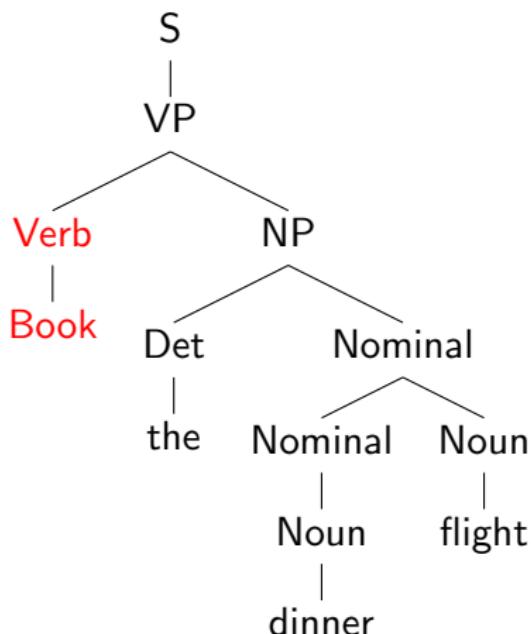
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

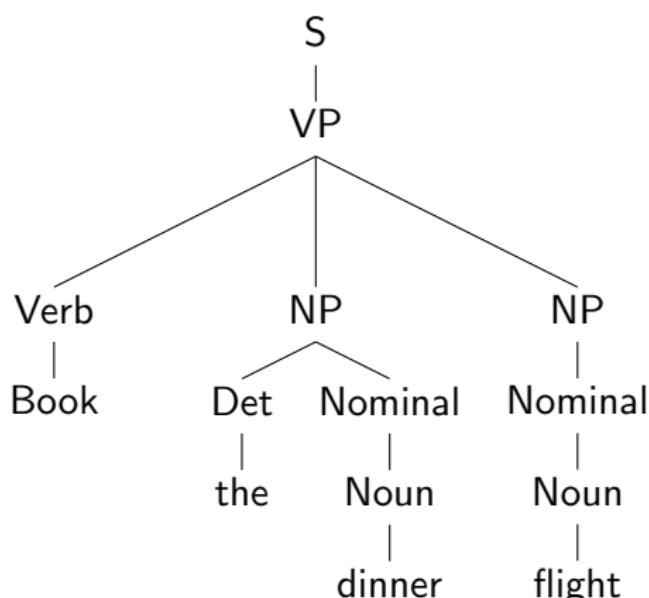
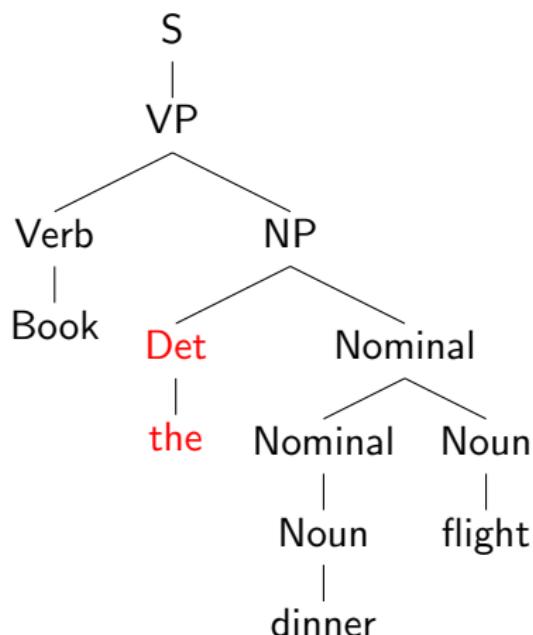
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

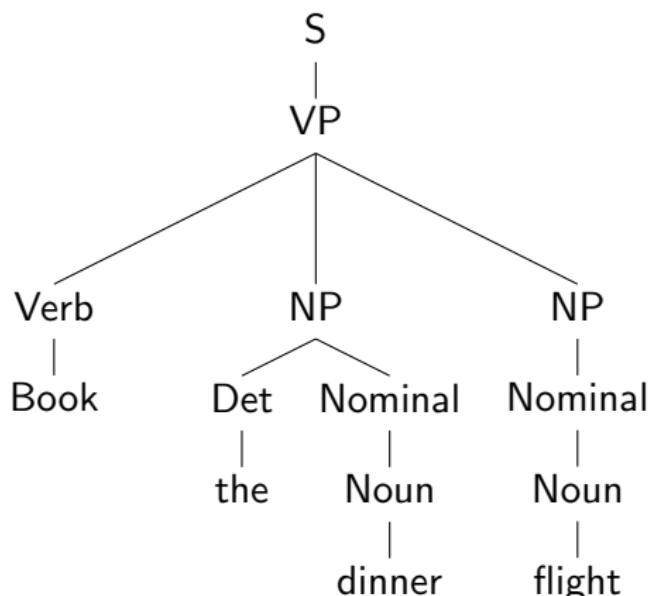
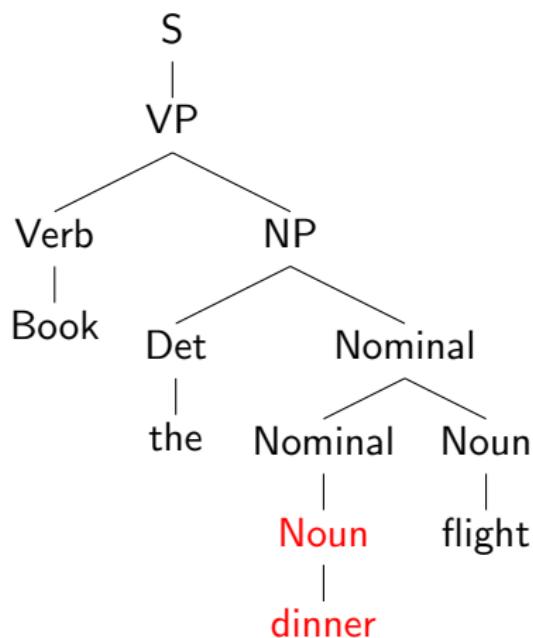
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * \textcolor{red}{.60} * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

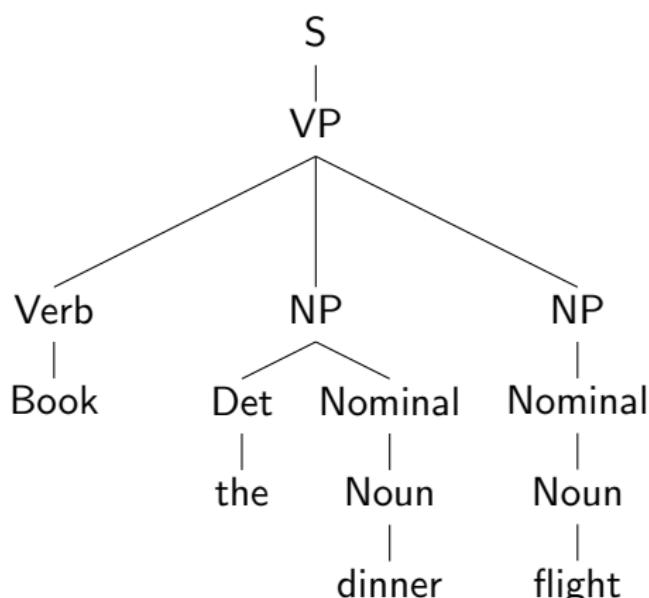
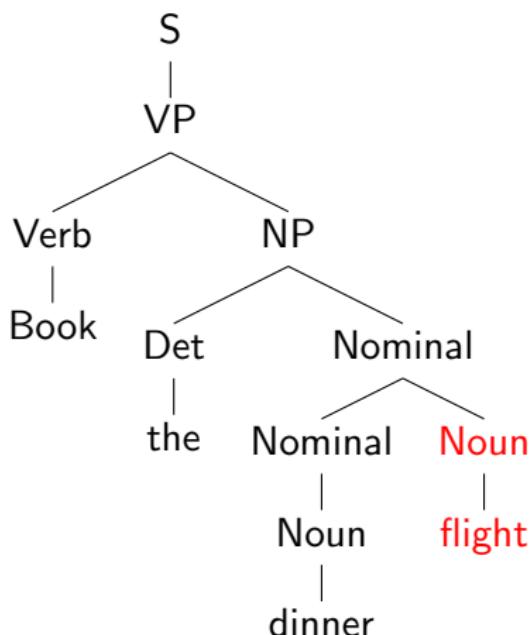
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * \textcolor{red}{.10} * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

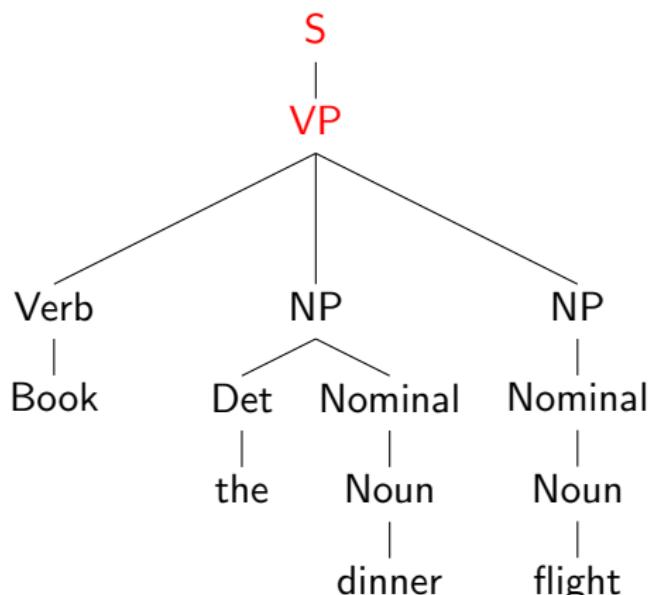
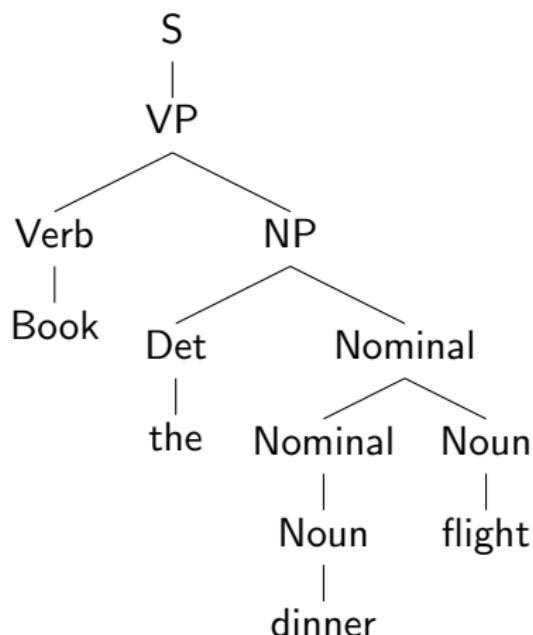
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = 2.2 \times 10^{-6}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = 6.1 \times 10^{-7}$$

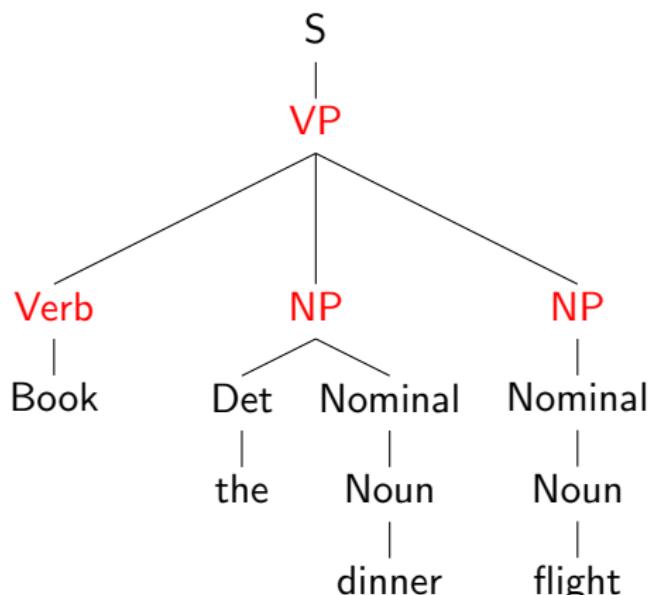
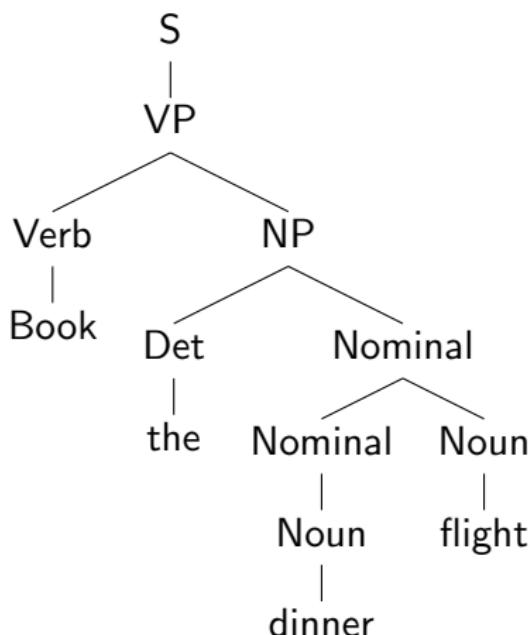
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = \mathbf{.05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = 6.1 \times 10^{-7}}$$

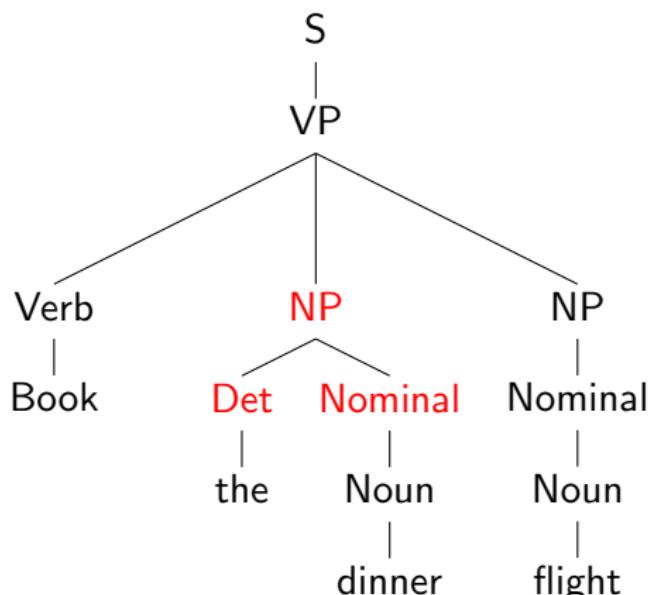
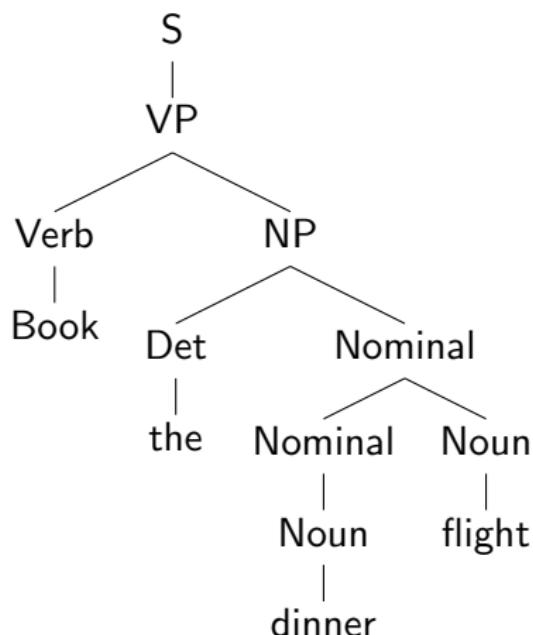
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * \mathbf{.10} * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

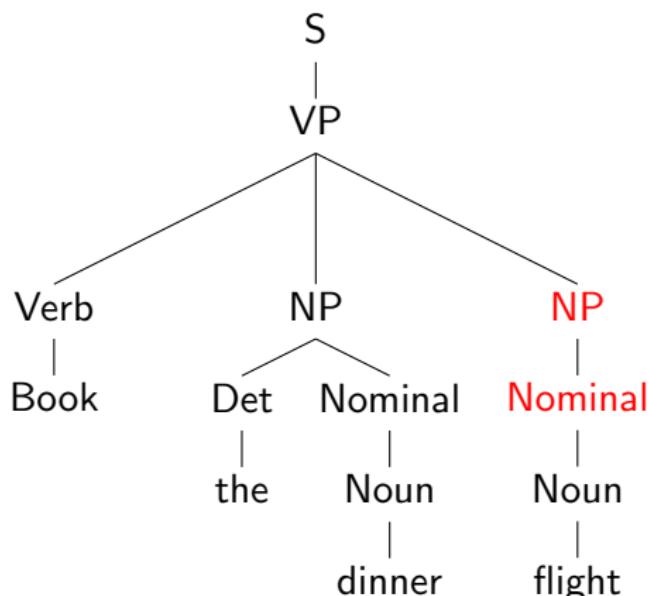
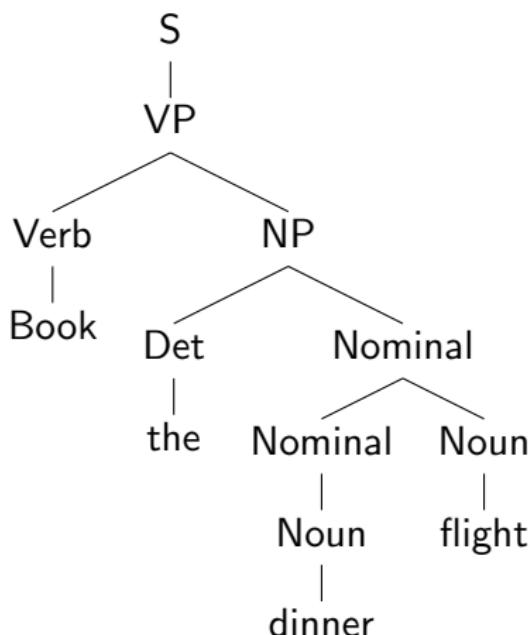
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * \mathbf{.20} * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

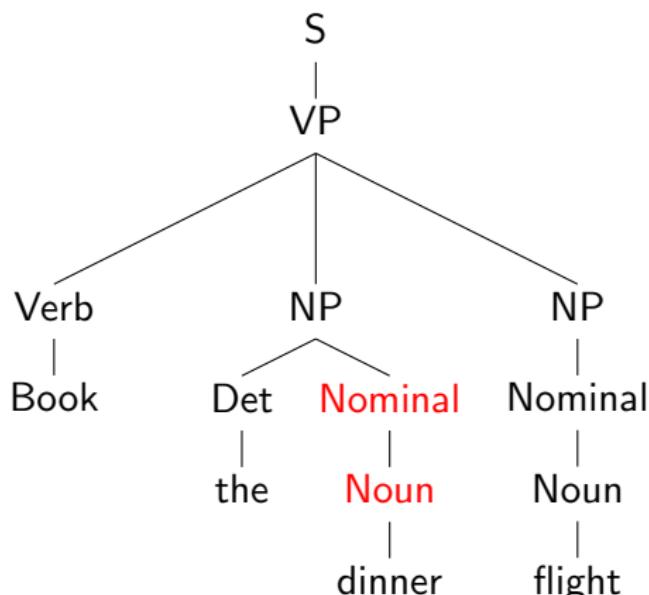
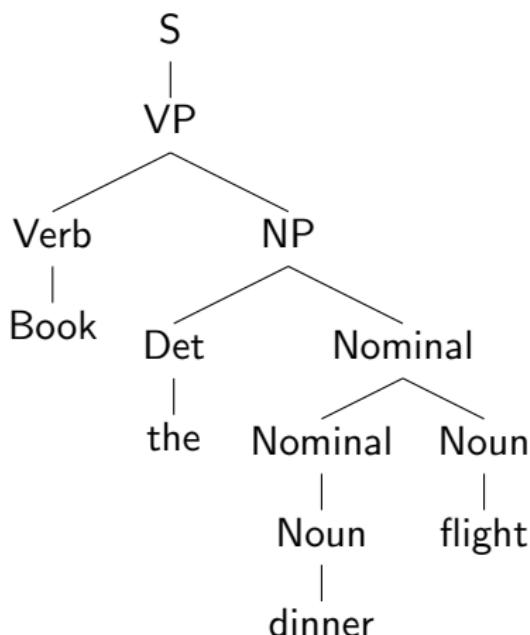
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * \mathbf{.15} * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

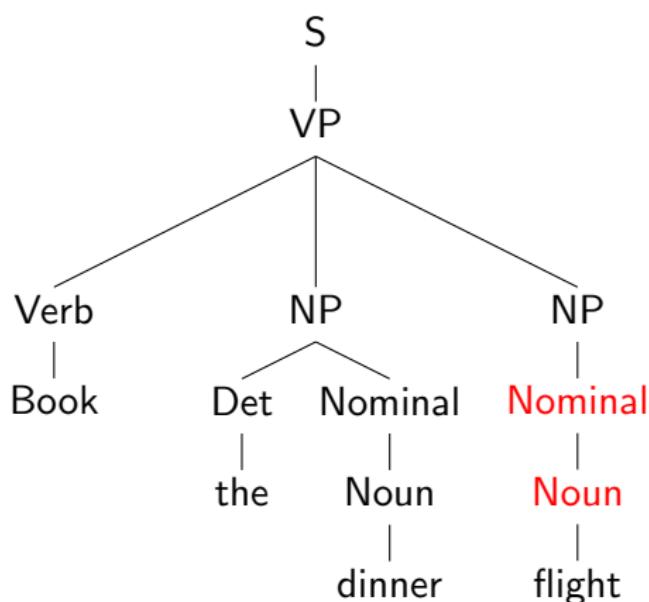
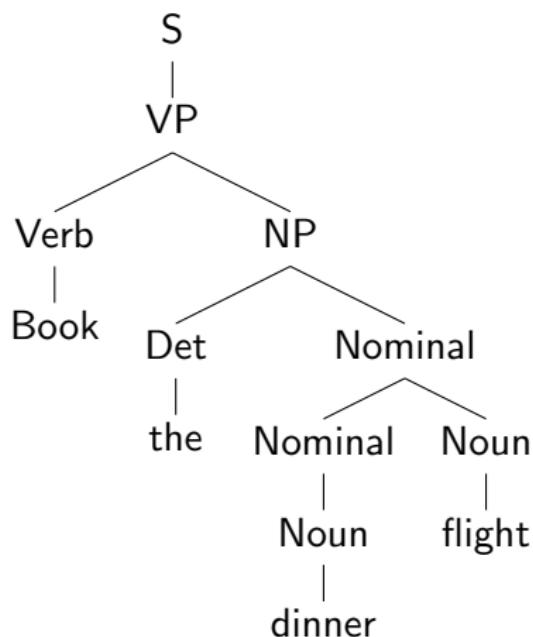
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * \mathbf{.75} * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

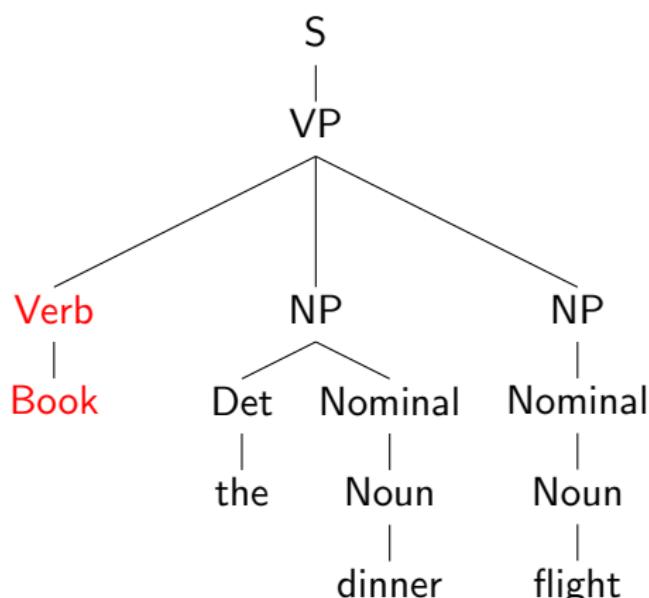
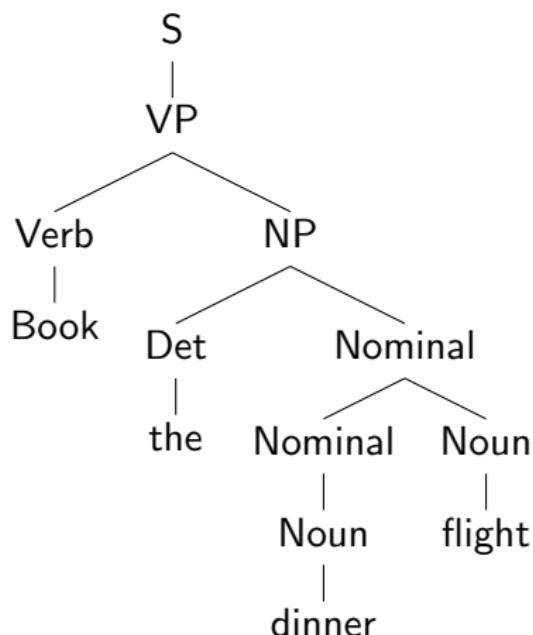
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

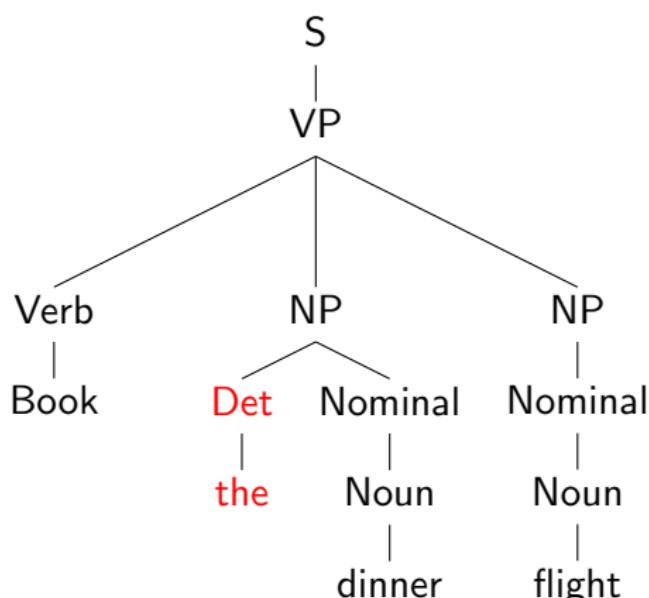
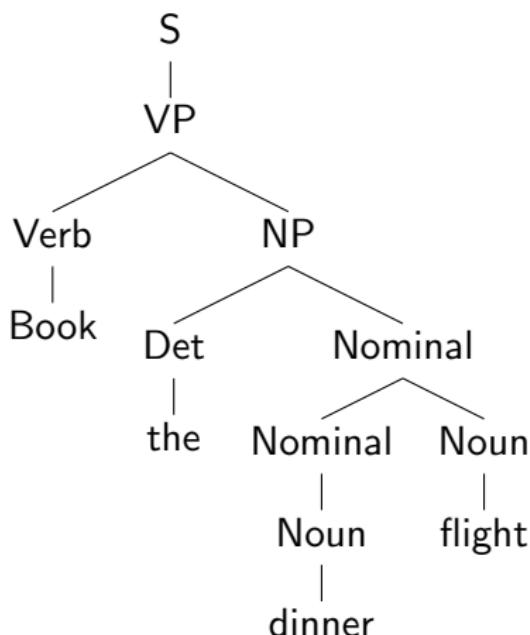
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * \mathbf{.30} * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

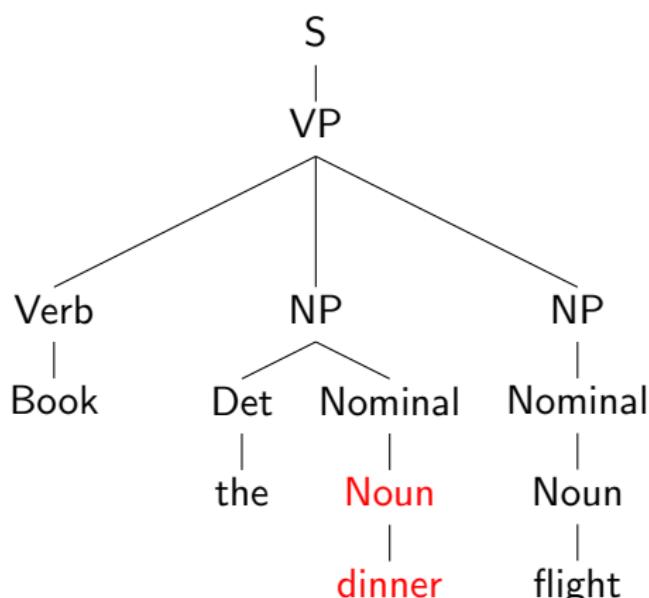
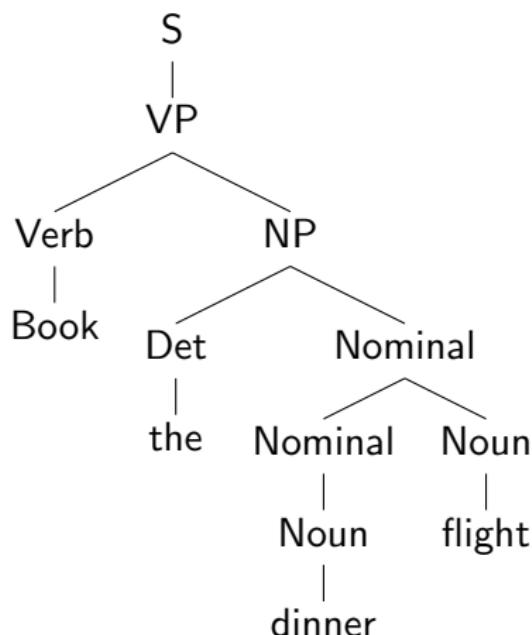
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * \mathbf{.60} * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

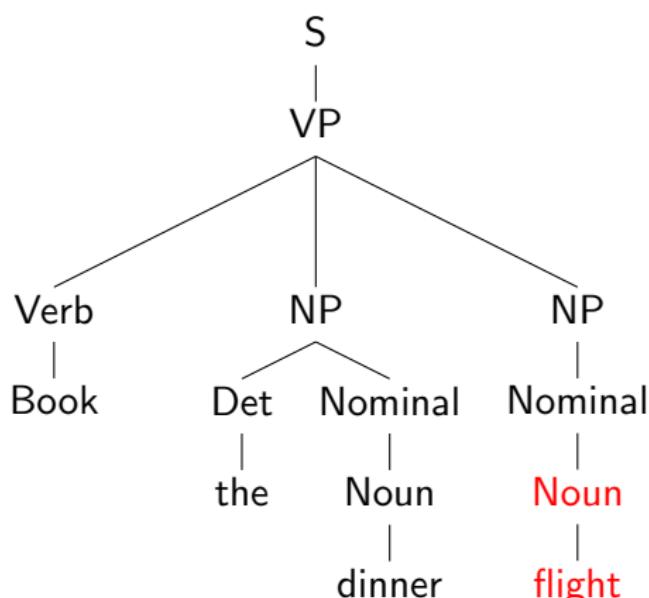
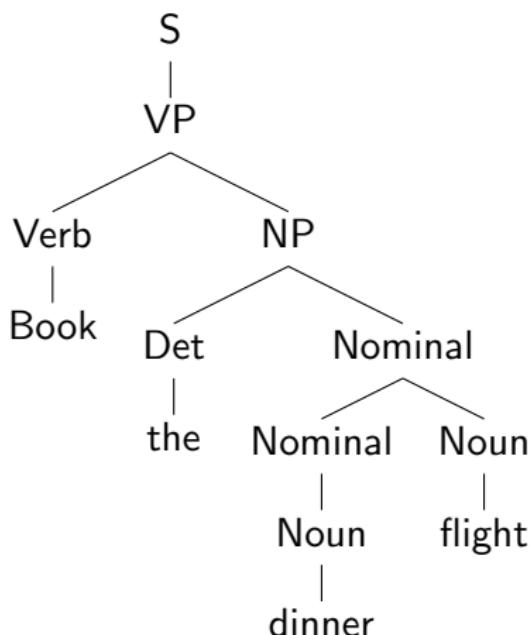
# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * \mathbf{.10} * .40 = \mathbf{6.1 \times 10^{-7}}$$

# Application 1: Disambiguation



$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$

$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * \mathbf{.40} = \mathbf{6.1 \times 10^{-7}}$$

## Application 2: Language Modelling

As well as assigning probabilities to parse trees, a PCFG assigns a probability to every sentence generated by the grammar. This is useful for **language modelling**.

The probability of a sentence is the sum of the probabilities of each parse tree associated with the sentence:

$$P(S) = \sum_{T \text{ s.t. } \text{yield}(T)=S} P(T, S)$$

$$P(S) = \sum_{s \text{.t. } \text{yield}(T)=S} P(T)$$

**When is it useful to know the probability of a sentence?**

When ranking the output of speech recognition, machine translation, and error correction systems.

Many probabilistic parsers use a probabilistic version of the CYK bottom-up chart parsing algorithm.

Sentence  $S$  of length  $n$  and CFG grammar with  $V$  non-terminals

## Ordinary CYK

$2 \cdot d(n + 1) * (n + 1)$  array where a value in cell  $(i, j)$  is list of non-terminals spanning position  $i$  through  $j$  in  $S$ .

## Probabilistic CYK

$3 \cdot d(n + 1) * (n + 1) * V$  array where a value in cell  $(i, j, K)$  is probability of non-terminal  $K$  spanning position  $i$  through  $j$  in  $S$

As with regular CYK, probabilistic CYK assumes that the grammar is in Chomsky-normal form (rules  $A \rightarrow B \ C$  or  $A \rightarrow w$ ).

# Oneliner for probabilistic CYK

$$\text{Chart}[A, i, j] = \max_{\substack{i < k < j \\ A \rightarrow B C \in G}} \text{Chart}[B, i, k] \times \text{Chart}[C, k, j] \times p(A \rightarrow B C)$$

Question: what order over  $(i, j)$  do we use to construct the table?

# Oneliner for probabilistic CYK

$$\text{Chart}[A, i, j] = \max_{\substack{i < k < j \\ A \rightarrow B C \in G}} \text{Chart}[B, i, k] \times \text{Chart}[C, k, j] \times p(A \rightarrow B C)$$

Question: what order over  $(i, j)$  do we use to construct the table?  
Remember from class about non-probabilistic CYK:

$$\text{Chart}[A, i, j] = \bigvee_{k=i+1}^{j-1} \bigvee_{A \rightarrow B C} \text{Chart}[B, i, k] \wedge \text{Chart}[C, k, j]$$

More advanced view of this: probabilistic CYK is the same as CYK, only with a different *semiring*

```
function Probabilistic-CYK(words, grammar) returns most
probable parse and its probability
    for j ← from 1 to LENGTH(words) do
        for all {A|A → words[j] ∈ grammar}
            table[j − 1, j, A] ←  $P(A \rightarrow words[j])$ 
        for i ← from j − 2 downto 0 do
            for all {A|A → BC ∈ grammar,
                and table[i, k, B] > 0 and table[k, j, C] > 0}
                if (table[i, j, A] <  $P(A \rightarrow BC) \times table[i, k, B] \times table[k, j, C]$ ) then
                    table[i, j, A] ←  $P(A \rightarrow BC) \times table[i, k, B] \times table[k, j, C]$ 
                    back[i, j, A] ← {k, B, C}
            return
            BUILD-TREE(back[1, LENGTH(words), S]), table[1, LENGTH(words), S]
```

## Visualizing the Chart

The	flight	includes	a	meal
Det: .40				
[0, 1]				

$S \rightarrow NP\ VP$  .80      *Det*  $\rightarrow$  *the*.40

*NP* → *Det N*.30    *Det* → *a* .40

$VP \rightarrow V\ NP$ .20     $N \rightarrow meal$ .01

$V \Rightarrow includes .05$      $N \Rightarrow flight .02$

## Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]				
	N: .02 [1, 2]			

$S \rightarrow NP\ VP$	.80	$Det \rightarrow the$	.40
$NP \rightarrow Det\ N$	.30	$Det \rightarrow a$	.40
$VP \rightarrow V\ NP$	.20	$N \rightarrow meal$	.01
$V \rightarrow includes$	.05	$N \rightarrow flight$	.02

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]				
	N: .02 [1, 2]			
		V: .05 [2, 3]		

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]				
	N: .02 [1, 2]			
		V: .05 [2, 3]		
			Det: .40 [3, 4]	

$S \rightarrow NP\ VP\ .80$    *Det → the.40*  
 $NP \rightarrow Det\ N\ .30$    *Det → a .40*  
 $VP \rightarrow V\ NP\ .20$    *N → meal .01*  
 $V \rightarrow includes.05$    *N → flight.02*

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]				
	N: .02 [1, 2]			
		V: .05 [2, 3]		
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40    × .02 = .0024 [0, 2]			
	N: .02 [1, 2]			
		V: .05 [2, 3]		
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40    × .02 = .0024 [0, 2]			
	N: .02 [1, 2]	[1, 3]	V: .05 [2, 3]	
				Det: .40 [3, 4]
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40      × .02 = .0024 [0, 2]	[0, 3]		
	N: .02 [1, 2]	[1, 3]		
		V: .05 [2, 3]		
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40 × .02 = .0024 [0, 2]	[0, 3]		
	N: .02 [1, 2]	[1, 3]		
		V: .05 [2, 3]	[2, 4]	
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40      × .02 = .0024 [0, 2]	[0, 3]		
	N: .02 [1, 2]	[1, 3]	[1, 4]	
		V: .05 [2, 3]	[2, 4]	
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40      × .02 = .0024 [0, 2]	[0, 3]	[0, 4]	
	N: .02 [1, 2]	[1, 3]	[1, 4]	
		V: .05 [2, 3]	[2, 4]	
			Det: .40 [3, 4]	
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40 × .02 = .0024 [0, 2]	[0, 3]	[0, 4]	
	N: .02 [1, 2]	[1, 3]	[1, 4]	
		V: .05 [2, 3]	[2, 4]	
			Det: .40 [3, 4]	NP: .30 × .40 × .01 = 0.0012 [3, 5]
				N: .01 [4, 5]

$S \rightarrow NP\ VP\ .80$     $Det \rightarrow the\ .40$   
 $NP \rightarrow Det\ N\ .30$     $Det \rightarrow a\ .40$   
 $VP \rightarrow V\ NP\ .20$     $N \rightarrow meal\ .01$   
 $V \rightarrow includes\ .05$     $N \rightarrow flight\ .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40 × .02 = .0024 [0, 2]	[0, 3]	[0, 4]	
	N: .02 [1, 2]	[1, 3]	[1, 4]	
		V: .05 [2, 3]	[2, 4]	VP: .20 × .05 × 0.0012 = 0.000012 [2, 5]
			Det: .40 [3, 4]	NP: .30 × .40 × .01 = 0.0012 [3, 5]
				N: .01 [4, 5]

$S \rightarrow NP VP .80$     $Det \rightarrow the .40$   
 $NP \rightarrow Det N .30$     $Det \rightarrow a .40$   
 $VP \rightarrow V NP .20$     $N \rightarrow meal .01$   
 $V \rightarrow includes .05$     $N \rightarrow flight .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40 × .02 = .0024 [0, 2]	[0, 3]	[0, 4]	
	N: .02 [1, 2]	[1, 3]	[1, 4]	[1, 5]
		V: .05 [2, 3]	[2, 4]	VP: .20 × .05 × 0.0012 = 0.000012 [2, 5]
			Det: .40 [3, 4]	NP: .30 × .40 × .01 = 0.0012 [3, 5]
				N: .01 [4, 5]

$S \rightarrow NP VP .80$     $Det \rightarrow the .40$   
 $NP \rightarrow Det N .30$     $Det \rightarrow a .40$   
 $VP \rightarrow V NP .20$     $N \rightarrow meal .01$   
 $V \rightarrow includes .05$     $N \rightarrow flight .02$

# Visualizing the Chart

The	flight	includes	a	meal
Det: .40 [0, 1]	NP: .30 × .40 .02 = .0024 [0, 2]			S: .80 × .0024 × .000012 = .000000023 [0, 5]
	N: .02 [1, 2]	[0, 3]	[0, 4]	[1, 5]
		V: .05 [1, 3]	[1, 4]	VP: .20 × .05 × 0.0012 = 0.000012 [1, 5]
			[2, 3]	[2, 4] [2, 5]
			Det: .40 [3, 4]	NP: .30 × .40 × .01 = 0.0012 [3, 5]
				N: .01 [4, 5]

$S \rightarrow NP VP .80$     $Det \rightarrow the .40$   
 $NP \rightarrow Det N .30$     $Det \rightarrow a .40$   
 $VP \rightarrow V NP .20$     $N \rightarrow meal .01$   
 $V \rightarrow includes .05$     $N \rightarrow flight .02$

## Probabilistic CYK: more tricky example

S → NP VP (1.0)  
NP → N (0.6) | A NP (0.2) | NP N (0.2)  
VP → V (0.8) | V Adv (0.2)  
N → orange (0.3) | tree (0.5) | blossoms (0.2)  
A → orange (1.0)  
V → blossoms (1.0)  
Adv → early (1.0)

(Not quite in CNF, but never mind.) We'll parse:

orange tree blossoms early

# The probabilistic CYK-style chart

	orange	tree	blossoms	early
orange	N (0.3) A (1.0) NP (0.18)	NP (0.06)	S (0.048) NP (0.0024)	S (0.012)
tree		N (0.5) NP (0.3)	NP (0.012)	S (0.06)
blossoms			N (0.2) V (1.0) NP (0.12) VP (0.8)	VP (0.2)
early				Adv(1.0)

## The probabilistic CYK-style chart: some comments

- The phrase **orange tree** gets 0.06 for its best analysis as an *NP*, since

$$0.06 = 0.2 * 1.0 * 0.3 \quad (\text{for } NP \rightarrow A \ NP)$$

$$\text{beats } 0.018 = 0.18 * 0.5 * 0.2 \quad (\text{for } NP \rightarrow NP \ N).$$

Only the higher probability is recorded in the chart.

- For **orange tree blossoms**, there are now two analyses as *NP*, each with probability 0.0024.
- There is also an analysis of **orange tree blossoms** as *S*. This doesn't compete with its analysis as *NP*, so both are recorded.

# Question

$S \rightarrow NP\ VP$	$Det \rightarrow the$
$NP \rightarrow Det\ N$	$Det \rightarrow a$
$VP \rightarrow V\ NP$	$N \rightarrow meal$
$V \rightarrow includes$	$N \rightarrow flight$

- ① Someone tells you that for each non-terminal  $X$ , the rules with LHS  $X$  are ‘equally likely’. What is the probability of **the flight includes a meal**?
- (1) 1    (2) 1/4    (3) 1/16    (d) 1/256

# Reduction of HMM to PCFG

Consider the following toy HMM:

	to N	to V		deal	fail	talks
from start	.8	.2	N	.2	.05	.2
from N	.4	.6	V	.3	.3	.3
from V	.8	.2				

It can be converted to a PCFG:

$$S \rightarrow \text{deal } N \quad p(N|\langle s \rangle) \times p(\text{deal} | N) = 0.8 \times 0.2 = 0.16$$

$$S \rightarrow \text{fail } N \quad p(N|\langle s \rangle) \times p(\text{fail} | N) = 0.8 \times 0.05 = 0.04$$

$$S \rightarrow \text{talks } N \quad p(N|\langle s \rangle) \times p(\text{talks} | N)$$

$$S \rightarrow \text{deal } V \quad p(N|\langle s \rangle) \times p(\text{deal} | V)$$

...

$$N \rightarrow \text{deal } N \quad p(N|N) \times p(\text{deal} | N) = 0.4 \times 0.2 = 0.08$$

$$N \rightarrow \text{deal } V \quad p(V|N) \times p(\text{deal} | V)$$

...

$$N \rightarrow \epsilon \quad 1.0 \text{ (just a weight)}$$

$$V \rightarrow \text{deal } N \quad p(N|V) \times p(\text{deal} | N)$$

$$V \rightarrow \text{deal } V \quad p(V|V) \times p(\text{deal} | V)$$

...

$$V \rightarrow \epsilon \quad 1.0 \text{ (just a weight)}$$

- A PCFG is a CFG with each rule annotated with a probability;
- the sum of the probabilities of all rules that expand the same non-terminal must be 1;
- probability of a parse tree is the product of the probabilities of all the rules used in this parse;
- probability of sentence is sum of probabilities of all its parses;
- applications for PCFGs: disambiguation, language modeling;
- Probabilistic CYK algorithm.

**Next lecture:** But where do the rule probabilities come from?

# Parameter Estimation and Lexicalization for PCFGs

Informatics 2A: Lecture 21

Shay Cohen

10 November 2015

## 1 Standard PCFGs

- Parameter Estimation
- Problem 1: Assuming Independence
- Problem 2: Ignoring Lexical Information

## 2 Lexicalized PCFGs

- Lexicalization
- Head Lexicalization

Reading:

*J&M 2<sup>nd</sup> edition, ch. 14.2–14.6,*

*NLTK Book, Chapter 8, final section on Weighted Grammar.*

## Question

$S \rightarrow NP\ VP$	(1.0)	$NPR \rightarrow John$	(0.5)
$NP \rightarrow DET\ N$	(0.7)	$NPR \rightarrow Mary$	(0.5)
$NP \rightarrow NPR$	(0.3)	$V \rightarrow saw$	(0.4)
$VP \rightarrow V\ PP$	(0.7)	$V \rightarrow loves$	(0.6)
$VP \rightarrow V\ NP$	(0.3)	$DET \rightarrow a$	(1.0)
$PP \rightarrow Prep\ NP$	(1.0)	$N \rightarrow cat$	(0.6)
		$N \rightarrow saw$	(0.4)

What is the probability of the sentence *John saw a saw*?

- ① 0.02
- ② 0.00016
- ③ 0.00504
- ④ 0.0002

# Parameter Estimation

In a PCFG every rule is associated with a probability.  
But where do these rule probabilities come from?

Use a large **parsed corpus** such as the Penn Treebank.

( (S	
(NP-SBJ (DT That) (JJ cold)	
(, ,)	
(JJ empty) (NN sky) )	$S \rightarrow NP-SBJ VP$
(VP (VBD was)	$VP \rightarrow VBD ADJP-PRD$
(ADJP-PRD (JJ full)	$PP \rightarrow IN NP$
(PP (IN of)	$NP \rightarrow NN CC NN$
(NP (NN fire)	
(CC and)	etc.
(NN light) ))))	
(. .) ))	

In a PCFG every rule is associated with a probability.  
But where do these rule probabilities come from?

Use a large **parsed corpus** such as the Penn Treebank.

- Obtain **grammar rules** by reading them off the trees.
- Calculate number of times  $\text{LHS} \rightarrow \text{RHS}$  occurs over number of times  $\text{LHS}$  occurs.

$$P(\alpha \rightarrow \beta | \alpha) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\sum_{\gamma} \text{Count}(\alpha \rightarrow \gamma)} = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]]

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP\ VP$	S	2/4
$r2$	$S \rightarrow NP\ VP\ AP$	S	2/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]]

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP\ VP$	$S$	$2/4$
$r2$	$S \rightarrow NP\ VP\ AP$	$S$	$2/4$

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]'

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP\ VP$	S	2/4
$r2$	$S \rightarrow NP\ VP\ AP$	S	2/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]]

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP VP$	S	2/4
$r2$	$S \rightarrow NP VP AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]'

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP VP$	S	2/4
$r2$	$S \rightarrow NP VP AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4
$r4$	$NP \rightarrow bananas$	NP	1/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]'

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP\ VP$	S	2/4
$r2$	$S \rightarrow NP\ VP\ AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4
$r4$	$NP \rightarrow bananas$	NP	1/4
$r5$	$VP \rightarrow grows$	VP	3/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]]

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP VP$	S	2/4
$r2$	$S \rightarrow NP VP AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4
$r4$	$NP \rightarrow bananas$	NP	1/4
$r5$	$VP \rightarrow grows$	VP	3/4
$r6$	$VP \rightarrow grow$	VP	1/4

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]'

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP VP$	S	2/4
$r2$	$S \rightarrow NP VP AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4
$r4$	$NP \rightarrow bananas$	NP	1/4
$r5$	$VP \rightarrow grows$	VP	3/4
$r6$	$VP \rightarrow grow$	VP	1/4
$r7$	$AP \rightarrow fast$	AP	1/2

# Parameter Estimation

Corpus of parsed sentences:

'S1: [S [NP grass] [VP grows]]'  
'S2: [S [NP grass] [VP grows] [AP slowly]]]  
'S3: [S [NP grass] [VP grows] [AP fast]]]  
'S4: [S [NP bananas] [VP grow]]'

Compute PCFG probabilities:

$r$	Rule	$\alpha$	$P(r \alpha)$
$r1$	$S \rightarrow NP VP$	S	2/4
$r2$	$S \rightarrow NP VP AP$	S	2/4
$r3$	$NP \rightarrow grass$	NP	3/4
$r4$	$NP \rightarrow bananas$	NP	1/4
$r5$	$VP \rightarrow grows$	VP	3/4
$r6$	$VP \rightarrow grow$	VP	1/4
$r7$	$AP \rightarrow fast$	AP	1/2
$r8$	$AP \rightarrow slowly$	AP	1/2

# Parameter Estimation

With these parameters (rule probabilities), we can now compute the probabilities of the four sentences S1–S4:

$$\begin{aligned} P(S1) &= P(r1|S)P(r3|NP)P(r5|VP) \\ &= 2/4 \cdot 3/4 \cdot 3/4 = 0.28125 \end{aligned}$$

$$\begin{aligned} P(S2) &= P(r2|S)P(r3|NP)P(r5|VP)P(r7|AP) \\ &= 2/4 \cdot 3/4 \cdot 3/4 \cdot 1/2 = 0.140625 \end{aligned}$$

$$\begin{aligned} P(S3) &= P(r2|S)P(r3|NP)P(r5|VP)P(r7|AP) \\ &= 2/4 \cdot 3/4 \cdot 3/4 \cdot 1/2 = 0.140625 \end{aligned}$$

$$\begin{aligned} P(S4) &= P(r1|S)P(r4|NP)P(r6|VP) \\ &= 2/4 \cdot 1/4 \cdot 1/4 = 0.03125 \end{aligned}$$

One criterion for finding rule weights of a PCFG (or parameters in general) is the *maximum likelihood* criterion.

It means we want to find rule weights which make the treebank we observe most likely if we multiply in all probabilities together (we assume the trees are independent)

Counting and normalising satisfies this criterion

What if we don't have a treebank, but we do have an unparsed corpus and (non-probabilistic) parser?

- ① Take a CFG and set all rules to have equal probability.
- ② Parse the (flat) corpus with the CFG.
- ③ Adjust the probabilities.
- ④ Repeat steps two and three until probabilities converge.

This is the **inside-outside algorithm** (Baker, 1979), a type of Expectation Maximisation algorithm. It can also be used to induce a grammar, but only with limited success.

While standard PCFGs are already useful for some purposes, they can produce poor result when used for disambiguation.

Why is that?

- ① They assume the rule choices are independent of one another.
- ② They ignore lexical information until the very end of the analysis, when word classes are rewritten to word tokens.

How can this lead to bad choices among possible parses?

## Problem 1: Assuming Independence

By definition, a CFG assumes that the expansion of non-terminals is completely **independent**. It doesn't matter:

- where a non-terminal is in the analysis;
- what else is (or isn't) in the analysis.

The same assumption holds for standard PCFGs: The probability of a rule is the same, no matter

- where it is applied in the analysis;
- what else is (or isn't) in the analysis.

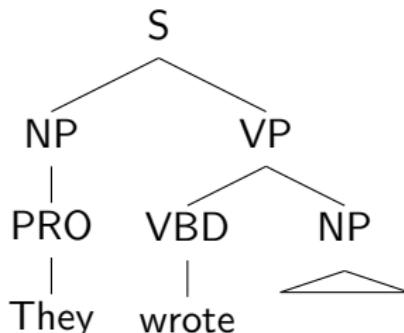
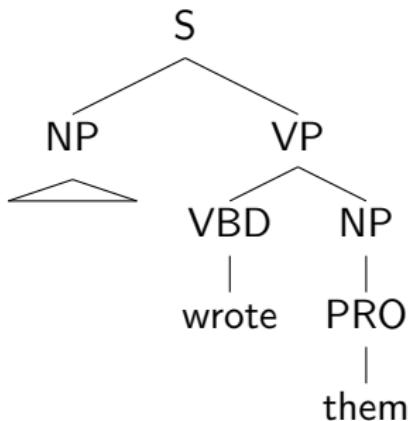
**But this assumption is too simple!**

# Problem 1: Assuming Independence

$$\begin{aligned} S &\rightarrow NP \ VP \\ VP &\rightarrow VBD \ NP \end{aligned}$$

$$\begin{aligned} NP &\rightarrow PRO \\ NP &\rightarrow DT \ NOM \end{aligned}$$

The above rules assign the same probability to both these trees, because they use the same re-write rules, and probability calculations do not depend on where rules are used.



## Problem 1: Assuming independence

But in speech corpora, 91% of 31021 subject NPs are pronouns:

- (1)     a. **She**'s able to take her baby to work with her.
- b. My wife worked until **we** had a family.

while only 34% of 7489 object NPs are pronouns:

- (2)     a. Some laws absolutely prohibit **it**.
- b. It wasn't clear how NL and Mr. Simmons would respond if Georgia Gulf spurns **them** again.

So the probability of NP → PRO should depend on **where** in the analysis it applies (e.g., subject or object position).

## Another example of independence

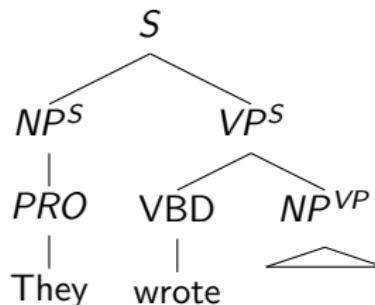
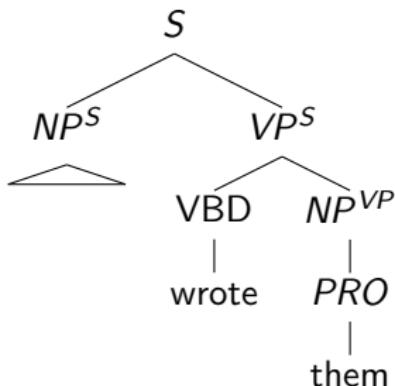
Question: which tree will get higher probability?

# Addressing the independence problem

One way of introducing greater sensitivity into PCFGs is via **parent annotation**: subdivide (all or some) non-terminal categories according to the non-terminal that appears as the node's immediate parent. E.g.  $NP$  subdivides into  $NP^S$ ,  $NP^{VP}$ , ...

$$\begin{aligned} S &\rightarrow NP^S \ VP^S \\ VP^S &\rightarrow VBD^{VP} \ NP^{VP} \end{aligned}$$

$$\begin{aligned} NP^S &\rightarrow PRO \\ NP^{VP} &\rightarrow PRO, \text{ etc.} \end{aligned}$$



## Addressing the independence problem

Node-splitting via **parent annotation** allows different probabilities to be assigned e.g. to the rules

$$NP^S \rightarrow PRO, \quad NP^{VP} \rightarrow PRO$$

However, too much node-splitting can mean not enough data to obtain realistic rule probabilities, unless we have an enormous training corpus.

There are even algorithms that try to identify the optimal amount of node-splitting for a given training set!

## Problem 2: Ignoring Lexical Information

$S \rightarrow NP\ VP$	$N \rightarrow sack\   bin\   \dots$
$NP \rightarrow NNS\   NN$	$NNS \rightarrow students$
$VP \rightarrow VBD\ NP\   VBD\ NP\ PP$	$V \rightarrow dumped\   spotted$
$PP \rightarrow P\ NP$	$DT \rightarrow a\   the$
$NP \rightarrow DT\ NN$	$P \rightarrow in$

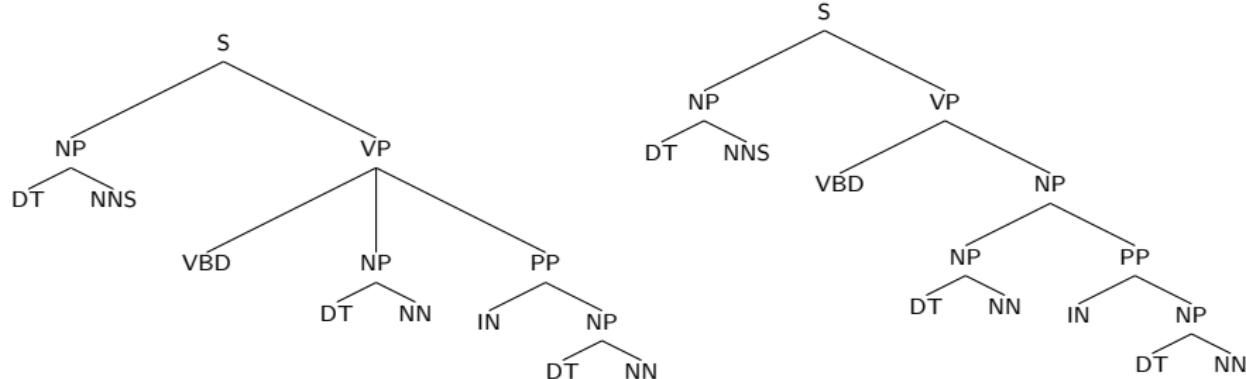
Consider the sentences:

- (3) a. The students dumped the sack in the bin.  
b. The students spotted the flaw in the plan.

Because rules for rewriting non-terminals ignore word tokens until the very end, let's consider these simply as strings of POS tags:

- (4) DT NNS VBD DT NN IN DT NN

## Problem 2: Ignoring Lexical Information



Which do we want for *The students dumped the sack in the bin*?

Which for *The students spotted the flaw in the plan*?

The most appropriate analysis depends in part on the **actual words** occurring. The word *dumped*, implying motion, is more likely to have an associated prepositional phrase than *spotted*.

A PCFG can be lexicalised by associating a word with every non-terminal in the grammar.

It is **head-lexicalised** if the word is the head of the constituent described by the non-terminal.

Each non-terminal has a **head** that determines syntactic properties of phrase (e.g., which other phrases it can combine with).

## Example

Noun Phrase (NP): Noun

Adjective Phrase (AP): Adjective

Verb Phrase (VP): Verb

Prepositional Phrase (PP): Preposition

# Lexicalization

We can lexicalize a PCFG by annotating each non-terminal with its **head word**, starting with the terminals – replacing

VP	$\rightarrow$	V NP PP	VP	$\rightarrow$	V NP
NP	$\rightarrow$	DT NN	NP	$\rightarrow$	NP PP
NP	$\rightarrow$	NNS	PP	$\rightarrow$	P NP

with rules such as

VP(dumped)	$\rightarrow$	V(dumped) NP(sack) PP(in)
VP(spotted)	$\rightarrow$	V(spotted) NP(flaw) PP(in)
VP(dumped)	$\rightarrow$	V(dumped) NP(sack)
VP(spotted)	$\rightarrow$	V(spotted) NP(flaw)
NP(flaw)	$\rightarrow$	DT(the) NN(flaw)
PP(in)	$\rightarrow$	P(in) NP(bin)
PP(in)	$\rightarrow$	P(in) NP(plan)

## Head Lexicalization

In principle, each of these rules can now have its own probability. But that would mean a ridiculous expansion in the set of grammar rules, with no parsed corpus large enough to estimate their probabilities accurately.

Instead we just lexicalize the **head** of phrase:

VP(dumped)	→ V(dumped) NP PP
VP(spotted)	→ V(spotted) NP PP
VP(dumped)	→ V(dumped) NP
VP(spotted)	→ V(spotted) NP
NP(flaw)	→ DT NN(flaw)
PP(in)	→ P(in) NP

Such grammars are called **lexicalized PCFGs** or, alternatively, **probabilistic lexicalized CFGs**.

# Head Lexicalization

For lexicalized PCFGs, rule probabilities can be reasonably estimated from a corpus.

In the simplest version, the lexicalized rules are supplemented by **head selection rules**, whose probabilities can also be estimated from a corpus:

VP	→	VP(dumped)
VP	→	VP(spotted)
NP	→	NP(sack)
NP	→	NP(flaw)
PP	→	PP(in)

## How many parameters in a lexicalized PCFG?

When all phrases are annotated with head words (say the grammar is in Chomsky normal form, and we have a vocabulary of size  $V$  and  $N$  nonterminals)?

When only the head phrase is annotated with a head word?

# Combining approaches

We can also combine the ideas of **head lexicalization** with **parent annotation**, leading to rules like

$$\begin{array}{ll} NP^{VP(dumped)} & \rightarrow NP(sack)^{VP(dumped)} \\ NP^{VP(spotted)} & \rightarrow NP(flaw)^{VP(spotted)} \\ PP^{VP(dumped)} & \rightarrow PP(in)^{VP(dumped)} \end{array}$$

The probabilities for such rules can be used to take account of commonly occurring **word combinations**, e.g. of verb-object or verb-preposition. These include **long-distance** correlations invisible to **N-gram** technology.

Grammars with these doubly-lexicalized rules are still feasible, given enough training data. This is roughly the idea behind the **Collins parser**.

## A highly simplified version of Collins model 1

Let a rule be  $\text{LHS} \rightarrow L_n L_{n-1} \dots L_1 H R_1 \dots R_{n-1} R_n$

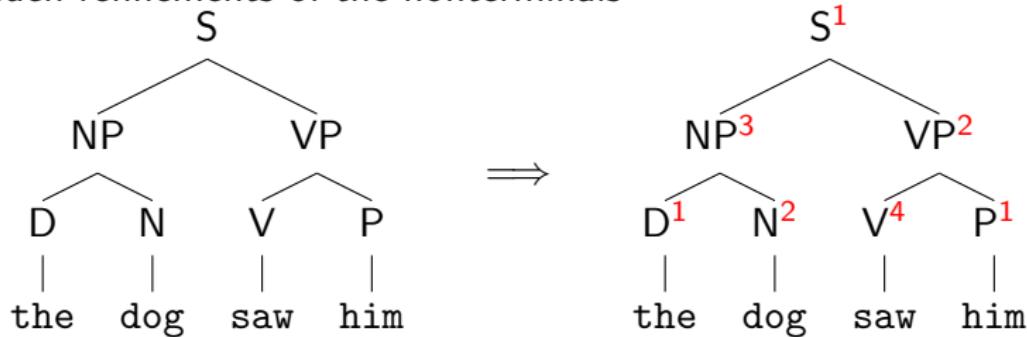
Idea: break it into smaller probabilities, “generating” the head and then left dependents and right dependents:

$$p(H|\text{LHS}) \times p_L(L_1|H, \text{LHS}) \dots p_L(L_n|H, \text{LHS}) \times p_L(\text{STOP} | L_n, \text{LHS}) \\ \times p_R(R_1|H, \text{LHS}) \dots p_R(R_n|H, \text{LHS}) \times p_R(\text{STOP} | R_n, \text{LHS})$$

Now can estimate each term separately, and the sparsity issue will be less severe

Lexicalization and node-splitting require careful thought about the linguistic meaning of the splitting

We can try to automatically induce these splittings by introducing *hidden* refinements of the nonterminals



Now our PCFG looks like:

$$S[1] \rightarrow NP[2] VP[3]$$

$$VP[4] \rightarrow VBD[1] NP[3]$$

$$NP[1] \rightarrow PRO[2]$$

$$NP[2] \rightarrow PRO[1], \text{ etc.}$$

**Question:** What kind of normalization do we expect now (say we have  $m$  hidden states)?

# How to Estimate Hidden State Grammars?

Similar to the inside-outside algorithm:

- Start with equal probabilities for all rules with hidden states
- Parse the treebank such that for each sentence the correct tree has to be returned, while maximising the probability of the states

$$H = \arg \max p(H, T, W)$$

( $W$  is the sentence,  $T$  is the treebank sentence, and  $H$  is a set of hidden states associated with each node in  $T$ )

- Re-estimate the hidden state PCFG rules as if you have node splitting on the nonterminals dictated by  $H$

Experiments on the Penn treebank show that approximately 30-60 states are needed for getting good coverage

- The rule probabilities of a PCFG can be estimated by counting how often the rules occur in a corpus.
- The usefulness of PCFGs is limited by the lack of lexical information and by strong independence assumptions.
- These limitations can be overcome by lexicalizing the grammars, i.e., by conditioning the rule probabilities on the head word of the rule.

## Demo: the Stanford parser:

<http://nlp.stanford.edu:8080/parser/>

# Agreement and Types in Natural Language

## Informatics 2A: Lecture 22

Shay Cohen

12 November 2015

- We've met the concept of **types** in programming languages, along with the idea of **typing constraints** on programs.
- Types also play a variety of roles in NL: e.g.
  - for disambiguation (via **selectional restrictions**),
  - for NL semantics (as in upcoming lecture).
- Furthermore, some phenomena that would be typically handled via types in a PL context (notably **agreement**) are often handled in other ways in NL.

We'll briefly survey this material in this lecture.

# Agreement phenomena

In PLs, typing rules enforce **type agreement** between different (often separated) constituents of a program:

```
int i=0; ...; if (i>2) ...
```

There are somewhat similar phenomena in NL: constituents of a sentence (often separated) may be constrained to agree on an attribute such as person, number, gender.

- You, I imagine, are unable to attend.
- The hills are looking lovely today, aren't they?
- He came very close to injuring himself.

# Agreement in various languages

These examples illustrate that in English:

- Verbs agree in **person** and **number** with their subjects;
- Tag questions agree in **person**, **number**, **tense** and **mode** with their main statement, and have the opposite **polarity**.
- Reflexive pronouns follow suit in **person**, **number** and **gender**.

French has much more by way of agreement phenomena:

- Adjectives agree with their head noun in gender and number.

Le petit chien,      La petite souris,      Les petites mouches

- Participles of *être* verbs agree with their subject:

Il est arrivé,      Elles sont arrivées

- Participles of other verbs agree with preceding direct objects:

Il a vu la femme,      Il l'a vue

How can we capture these kinds of constraints in a grammar?

# Agreement rules: why bother?

Modelling agreement is obviously important if we want to **generate** grammatically correct NL text.

But even for **understanding** input text, agreement can be useful for resolving ambiguity.

E.g. the following sentence is ambiguous . . .

The boy who eats flies ducks.

. . . whilst the following are less so:

The boys who eat fly ducks.

The boys who eat flies duck.

## Node-splitting via attributes

One solution is to refine our grammar by splitting certain non-terminals according to various *attributes*. Examples of attributes and their associated values are:

- **Person**: 1st, 2nd, 3rd
- **Number**: singular, plural
- **Gender**: masculine, feminine, neuter
- **Case**: nominative, accusative, dative, ...
- **Tense**: present, past, future, ...

In principle these are language-specific, though certain common patterns recur in many languages.

We can then split phrase categories as the language demands, e.g.

- Split NP on person, number, case (e.g. NP[3,sg,nom]),
- Split VP on person, number, tense (e.g. VP[3,sg,fut]).

# Parameterized CFG productions

We can often use such attributes to enforce agreement constraints.

This works because of the **head phrase structure** typical of NLs.

E.g. we may write parameterized rules such as:

$$\begin{array}{lcl} S & \rightarrow & NP[p,n,nom] VP[p,n] \\ NP[3,n,c] & \rightarrow & Det[n] Nom[n] \end{array}$$

Each of these really abbreviates a finite number of rules obtained by specializing the attribute variables. (Still a CFG!)

When specializing, each variable must take the same value everywhere, e.g.

$$\begin{array}{lcl} S & \rightarrow & NP[3,sg,nom] VP[3,sg] \\ S & \rightarrow & NP[1,pl,nom] VP[1,pl] \\ NP[3,pl,acc] & \rightarrow & Det[pl] Nom[pl] \end{array}$$

Parsing algorithms can be adapted to work with this machinery: don't have to 'build' all the specialized rules individually.

## Example: subject-verb agreement in English

S	→	NP[p,n,nom] VP[p,n]
NP[p,n,c]	→	Pro[p,n,c]
Pro[1,sg,nom]	→	<i>I</i> , etc.
Pro[1,sg,acc]	→	<i>me</i> , etc.
NP[3,n,c]	→	Det[n] Nom[n] RelOpt[n]
Nom[n]	→	N[n]   Adj Nom[n]
N[sg]	→	<i>person</i> , etc.
N[pl]	→	<i>people</i> , etc.
RelOpt[n]	→	$\epsilon$   <i>who</i> VP[3,n]
VP[p,n]	→	VV[p,n] NP[p',n',acc]
VV[p,n]	→	V[p,n]   BE[p,n] VG
V[3,sg]	→	<i>teaches</i> , etc.
BE[p,n]	→	<i>is</i> , etc.
VG	→	<i>teaching</i> , etc.

(Other rules omitted.)

# Some disadvantages of rule splitting for agreement

There is a huge proliferation of primitive grammatical categories

## Example

Non3sgVPto, NPmass, 3sgNP, Non3sgAux, ...

This leads to a large number of grammar rules and a loss of generality in the grammar

A fix: constraint-based representation scheme based on unification

# Feature Structures

Defined in terms of attribute-value matrices (AVMs):

subj	$\begin{bmatrix} \text{pers} & 3 \\ \text{num} & \text{sg} \\ \text{gend} & \text{masc} \\ \text{pred} & \text{pro} \end{bmatrix}$
pred	eat⟨SUBJ, OBJ⟩
obj	$\begin{bmatrix} \text{pers} & 3 \\ \text{num} & \text{pl} \\ \text{gend} & \text{fem} \\ \text{pred} & \text{pro} \end{bmatrix}$

Nested set of attributes

# How to use Feature Structures?

Each lexical rule is attached with a lexical AVM

## Example

Det → this [Det AGREEMENT [ NUMBER Sg ] ]

Det → these [Det AGREEMENT [ NUMBER Pl ] ]

Aux → do [Det AGREEMENT [ NUMBER Pl, PERSON 3rd ] ]

Aux → does [Det AGREEMENT [ NUMBER Sg, PERSON 3rd ] ]

# How to use Feature Structures?

Each lexical rule is attached with a lexical AVM

## Example

- Det → this [Det AGREEMENT [ NUMBER Sg ] ]
- Det → these [Det AGREEMENT [ NUMBER Pl ] ]
- Aux → do [Det AGREEMENT [ NUMBER Pl, PERSON 3rd ] ]
- Aux → does [Det AGREEMENT [ NUMBER Sg, PERSON 3rd ] ]

Grammar rules are specified with constraints and copying instructions

## Example

- VP → Verb NP [VP AGREEMENT] = [Verb AGREEMENT]
- NP → Det Nominal [NP HEAD] = [Nominal Head]  
[Det HEAD AGREEMENT]  
= [Nominal HEAD AGREEMENT]

Whenever phrases are conjoined, for example, two phrases in the CYK parser, we do *feature unification*.

This means we check whether we satisfy the constraints attached to the rule, and if so, when we create the new phrase, we also create a new feature structure for it

Unification is not a trivial algorithm because the attribute-values can be *shared* in an AVM, using pointers

## Example of Shared Attributes

CAT	VP										
AGREEMENT	1										
HEAD	<table><tr><td>AGREEMENT</td><td>1</td><td><table><tr><td>PERSON</td><td>3</td></tr><tr><td>NUMBER</td><td>SG</td></tr><tr><td>GENDER</td><td>MASC</td></tr></table></td></tr></table>	AGREEMENT	1	<table><tr><td>PERSON</td><td>3</td></tr><tr><td>NUMBER</td><td>SG</td></tr><tr><td>GENDER</td><td>MASC</td></tr></table>	PERSON	3	NUMBER	SG	GENDER	MASC	
AGREEMENT	1	<table><tr><td>PERSON</td><td>3</td></tr><tr><td>NUMBER</td><td>SG</td></tr><tr><td>GENDER</td><td>MASC</td></tr></table>	PERSON	3	NUMBER	SG	GENDER	MASC			
PERSON	3										
NUMBER	SG										
GENDER	MASC										

The 1 refers to the same sub-AVM

Types are also very useful if we wish to describe the **semantics** (i.e., meaning) of natural languages. For example, we can use types employed in **logic** to model the meanings of various phrase types.

## Basic Types

- ①  $e$  — the type of real-world *entities* such as Inf2a, Stuart, John.
- ②  $t$  — the type of *facts with truth value* like 'Inf2a is amusing'.

From these two basic types, we may construct more complex types via the **function type** constructor.

# From basic to complex formal types

Where PL people write  $\sigma \rightarrow \tau$ , NL people often write  $\langle \sigma, \tau \rangle$ . E.g.:

- $\langle e, t \rangle$ : **unary predicates** – functions from entities to facts.
- $\langle e, \langle e, t \rangle \rangle$ : **binary predicates** – functions from entities to unary predicates.
- $\langle \langle e, t \rangle, t \rangle$ : **type-raised entities** – functions from unary predicates to truth values.

- Inf2a, Stuart :  $e$
- enjoys :  $\langle e, \langle e, t \rangle \rangle$
- enjoys Inf2a, is amusing :  $\langle e, t \rangle$
- Inf2a is amusing, Stuart enjoys Inf2a :  $t$
- every student :  $\langle \langle e, t \rangle, t \rangle$

This simple system of types will be enough to be going on with (see Lecture 24). But for more precise semantic modelling, a much richer type system is desirable.

# Different types of entities in NL

We can distinguish those entities we can **count** and those we can't:

- A student kept **a chicken** in her room.
- A student kept **two chickens** in her room.
- I ate **rice** and drank **milk**.
- \*I ate **two rices** and drank **two milks**.

- **individuals** (things we can count): one student, two students, one chicken, many chickens, one room, many rooms
- **mass** (things we can't count): rice, milk

```
hamburger <: sandwich <: food item <: food  
<: substance <: matter <: physical entity <: entity
```

- To deal with meanings in NL, more fine-grained classifications (of varying levels of specificity) are often useful.
- There are also many other more abstract types of entities to which a NL expression may refer: e.g., locations, points in time, time spans, events, beliefs, desires, possibilities, ...
- This leads to a vast system of subtypes capturing information about real-world concepts and their relationships.  
(Cf. the **WordNet** database.)

## Selectional restrictions

We can often characterize verbs and other predicates in terms of their **selectional restrictions** — constraints on the type of entities or expression can serve as their arguments.

- I want to eat somewhere close to Appleton Tower.
- I want to eat something close to Thai food.

How do we know that *Thai food* is the **object** of the eating event in the second sentence, and that *somewhere close to AT* is the **location** of the eating event in the first?

- The **object** of eating is usually something *edible*: Its semantic type is *edible things*.
- The **location** of an event is usually a *place*: Its semantic type is *location*.

## Selectional restrictions

Selectional restrictions are associated with word **senses**, not words:

- Do any international airlines serve vegan meals?  
(ie, *provide food or drink*)
- Do any international airlines serve Edinburgh?
- (ie, *provide a service*)
- ?? Do any international airlines serve Edinburgh and vegan meals?

Selectional restrictions vary in their specificity:

OBJECT(imagine): a situation

OBJECT(diagonalise): a matrix

⇒ Verbs vary in the specificity of their argument types.

# Selectional restrictions and type coercion

Selectional restrictions can change the way we interpret a term:

- Jane Austen wrote 'Emma'.
- I used to read Jane Austen a lot.

- The chicken was domesticated in Asia.
- The chicken was overcooked.

**Metonymy** is when the referent of a term changes to a related entity, often associated with the demands of a verb's **selectional restrictions**.

- Many agreement phenomena in NL can be modelled using CFGs with attributes.
- Type systems are also useful in semantic modelling.
- To capture selectional restrictions associated with verb arguments, a very rich system of subtypes is desirable.
- Type coercion is common in Natural Language: changing the type (and often the referent) of an expression to one that fits the verb (predicate) to which it serves as an argument.

# Semantics for Natural Languages

## Informatics 2A: Lecture 23

Shay Cohen

13 November 2015

## 1 Introduction

- Syntax and Semantics
- Compositionality
- Desiderata for Meaning Representation

## 2 Logical Representations

- Propositional Logic
- Predicate Logic

## 3 Semantic Composition

- Compositionality
- Lambda Expressions

**Syntax** is concerned with which expressions in a language are well-formed or grammatically correct. This can largely be described by rules that make no reference to *meaning*.

**Semantics** is concerned with the meaning of expressions: i.e. how they relate to 'the world'. This includes both their

- **denotation** (literal meaning)
- **connotation** (other associations)

When we say a sentence is ambiguous, we usually mean it has more than one 'meaning'. (So what exactly are **meanings**?)

We've already encountered **word sense ambiguity** and **structural ambiguity**. We'll also meet another kind of semantic ambiguity, called **scope ambiguity**. (This already shows that the meaning of a sentence can't be equated with its parse tree.)

Providing a **semantics** for a language (natural or formal) involves giving a **systematic mapping from** the structure underlying a string (e.g. syntax tree) **to** its 'meaning'.

Whilst the kinds of meaning conveyed by NL are much more complex than those conveyed by FLs, they both broadly adhere to a principle called **compositionality**.

# Compositionality

**Compositionality:** The meaning of a complex expression is a function of the meaning of its parts and of the rules by which they are combined.

While formal languages are designed for **compositionality**, the meaning of NL utterances can often (not always) be derived compositionally as well.

Compare:

purple armadillo      hot dog

# Other desiderata for Meaning Representation

**Verifiability:** One must be able to use the meaning representation of a sentence to determine whether the sentence is **true** with respect to some given model of the world.

Example: given an exhaustive table of ‘who loves whom’ relations (a world model), the meaning of a sentence like *everybody loves Mary* can be established by checking it against this model.

# Desiderata for Meaning Representation

**Unambiguity:** a meaning representation should be unambiguous, with one and only one interpretation. If a sentence is ambiguous, there should be a different meaning representation for each sense.

Example: each interpretation of *I made her duck* or *time flies like an arrow* should have a distinct meaning representation.

# Desiderata for Meaning Representation

**Canonical form:** the meaning representations for sentences with the same meaning should (ideally) both be convertible into the same canonical form, that shows their equivalence.

Example: the sentence *I filled the room with balloons* should ideally have the same canonical form with *I put enough balloons in the room to fill it from floor to ceiling*.

(The kind of formal semantics we discuss won't achieve this particularly well!)

# Desiderata for Meaning Representation

**Logical inference:** A good meaning representation should come with a set of rules for logical inference or deduction, showing which truths imply which other truths.

E.g. from

Zoot is an armadillo.

Zoot is purple.

Every purple armadillo sneezes.

we should be able to deduce

Zoot sneezes.

**Propositional logic** is a very simple system for meaning representation and reasoning in which expressions comprise:

- **atomic sentences** ( $P$ ,  $Q$ , etc.);
- **complex sentences** built up from atomic sentences and logical connectives (and, or, not, implies).

# Propositional Logic

Why not use propositional logic as a meaning representation system for NL? E.g.

Fred ate lentils or he ate rice. ( $P \vee Q$ )

Fred ate lentils or John ate lentils ( $P \vee R$ )

- We're unable to represent the internal structure of the proposition 'Fred ate lentils' (e.g. how its meaning is derived from that of 'Fred', 'ate', 'lentils').
- We're unable to express e.g.

Everyone ate lentils.

Someone ate lentils.

# Predicate Logic

First-order predicate logic (FOPL) let us do a lot more (though still only accounts for a tiny part of NL).

Sentences in FOPL are built up from terms made from:

- constant and variable symbols that represent entities;
- predicate symbols that represent properties of entities and relations that hold between entities;
- function symbols (won't bother with these here).

which are combined into simple sentences (predicate-argument structures) and complex sentences through:

quantifiers ( $\forall, \exists$ )	disjunction ( $\vee$ )
negation ( $\neg$ )	implication ( $\Rightarrow$ )
conjunction ( $\wedge$ )	equality ( $=$ )

# Predicate Logic Example

## Constant symbols:

- **Each constant symbol denotes one and only one entity:**  
Scotland, Aviemore, EU, Barack Obama, 2007
- **Not all entities have a constant that denotes them:**  
Barack Obama's right knee, this piece of chalk
- **Several constant symbols may denote the same entity:**  
The Morning Star  $\equiv$  The Evening Star  $\equiv$  Venus  
National Insurance number, Student ID, your name

## Predicate symbols:

- Every predicate has a specific **arity**. E.g. brown/1, country/1, live\_in/2, give/3.
- A predicate symbol of arity  $n$  is interpreted as a set of  $n$ -tuples of entities that satisfy it.
- Predicates of arity 1 denote properties: brown/1.
- Predicates of arity  $> 1$  denote relations: live\_in/2, give/3.

Variable symbols: x, y, z:

- Variable symbols range over entities.
- An atomic sentence with a variable among its arguments, e.g., `Part_of(x, EU)`, only has a truth value if that variable is **bound** by a quantifier.

# Universal Quantifier ( $\forall$ )

Universal quantifiers can be used to express **general truths**:

- *Cats are mammals*
- $\forall x.\text{Cat}(x) \Rightarrow \text{Mammal}(x)$

Intuitively, a universally quantified sentence corresponds to a (possibly infinite) **conjunction** of sentences:

$\text{Cat}(\text{sam}) \Rightarrow \text{Mammal}(\text{sam}) \wedge \text{Cat}(\text{zoot}) \Rightarrow \text{Mammal}(\text{zoot})$   
 $\wedge \text{Cat}(\text{fritz}) \Rightarrow \text{Mammal}(\text{fritz}) \wedge \dots$

A quantifier has a **scope**, analogous to scope of PL variables.

# Existential Quantifier ( $\exists$ )

Existential quantifiers are used to express the **existence** of an entity with a given property, without specifying which entity:

- *I have a cat*
- $\exists x. \text{Cat}(x) \wedge \text{Own}(I, x)$

An existentially quantified sentence corresponds intuitively to a **disjunction** of sentences:

$$\begin{aligned} & (\text{Cat(Josephine)} \wedge \text{Own}(I, \text{Josephine})) \vee \\ & (\text{Cat(Zoot)} \wedge \text{Own}(I, \text{Zoot})) \vee \\ & (\text{Cat(Malcolm)} \wedge \text{Own}(I, \text{Malcolm})) \vee \\ & (\text{Cat(John)} \wedge \text{Own}(I, \text{John})) \vee \dots \end{aligned}$$

## Existential Quantifier ( $\exists$ )

Why do we use “ $\wedge$ ” rather than “ $\Rightarrow$ ” with the existential quantifier? What would the following correspond to?

$$\exists x. \text{Cat}(x) \Rightarrow \text{Own}(i, x)$$

- (a) I own a cat
- (b) There's something such that if it's a cat, I own it.

What if that something isn't a cat?

- The proposition formed by connecting two propositions with  $\Rightarrow$  is true if the antecedent (the left of the  $\Rightarrow$ ) is false.
- So this proposition is true if there is something that's e.g. a laptop. But “I own a cat” shouldn't be true simply for this reason.

# Abstract syntax of FOPL

The language of first-order predicate logic can be defined by the following CFG (think of it as a grammar for **abstract syntax trees**). We write F for formulae, AF for atomic formulae, t for terms, v for variables, c for constants.

$$\begin{array}{lcl} F & \rightarrow & AF \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \mid \neg F \\ & & \mid \forall v.F \mid \exists v.F \\ AF & \rightarrow & t=t \mid \text{UnaryPred}(t) \mid \text{BinaryPred}(t,t) \mid \dots \\ t & \rightarrow & v \mid c \end{array}$$

## Question

Which captures the meaning of *Every dog has a bone?*

- ①  $\forall x. \exists y. (\text{dog}(x) \wedge \text{bone}(y) \wedge \text{has}(x, y))$
- ②  $\forall x. (\text{dog}(x) \Rightarrow \exists y. (\text{bone}(y) \wedge \text{has}(x, y)))$
- ③  $\forall x. \exists y. \text{bone}(y) \wedge (\text{dog}(x) \Rightarrow \text{has}(x, y))$
- ④  $\exists y. \forall x. (\text{dog}(x) \Rightarrow (\text{bone}(y) \wedge \text{has}(x, y)))$

(N.B. The logical form looks structurally quite different from the parse tree for the original sentence. So there's some real work to be done!)

# Compositionality

**Compositionality:** The meaning of a complex expression is a function of the meaning of its parts and of the rules by which they are combined.

Do we have sufficient tools to **systematically compute** meaning representations according to this principle?

- The meaning of a complete sentence will hopefully be a FOPL formula, which we consider as having type  $t$  (truth values).
- But the meaning of smaller fragments of the sentence will have other types. E.g.

*has a bone*     $\langle e, t \rangle$

*every dog*     $\langle\langle e, t \rangle, t \rangle$

- The idea is to show how to associate a meaning with such fragments, and how these meanings combine.
- To do this, we need to extend our language of FOPL with  **$\lambda$  expressions** ( $\lambda$  = lambda; written as  $\backslash$  in Haskell).

# Lambda ( $\lambda$ ) Expressions

**$\lambda$ -expressions** are an extension to FOPL that allows us to work with '**partially constructed**' **formulae**. A  $\lambda$ -expression consists of:

- the Greek letter  $\lambda$ , followed by a variable (**formal parameter**);
- a FOPL expression that may involve that variable.

$\lambda x.sleep(x) : < e, t >$

'The function that takes an entity  $x$  to the statement  $sleep(x)$ '

$$\underbrace{(\lambda x.sleep(x))}_{\text{function}} \underbrace{(Mary)}_{\text{argument}} : t$$

A  $\lambda$ -expression can be **applied** to a term.

(The above has the same truth value as  $sleep(Mary)$ .)

Lambda expressions can be **nested**. We can use nesting to create functions of several arguments that accept their arguments one at a time.

$\lambda y. \lambda x. \text{love}(x,y) : < e, < e, t >>$

'The function that takes y to  
(the function that takes x to the statement  $\text{love}(x,y)$ )'

$\lambda z. \lambda y. \lambda x. \text{give}(x,y,z) : < e, < e, < e, t >>>$

'The function that takes z to  
(the function that takes y to  
(the function that takes x to the statement  $\text{give}(x,y,z)$ ))'

# Beta Reduction

When a lambda expression **applies** to a term, a reduction operation (**beta ( $\beta$ ) reduction**) can be used to replace its formal parameter with the term and simplify the result. In general:

$$(\lambda x.M)N \Rightarrow_{\beta} M[x \mapsto N] \quad (M \text{ with } N \text{ substituted for } x)$$

$$\underbrace{(\lambda x.\text{sleep}(x))}_{\text{function}} \underbrace{(\text{Mary})}_{\text{argument}} \Rightarrow_{\beta} \text{sleep}(\text{Mary})$$

$$\underbrace{(\lambda y.\lambda x.\text{love}(x,y))}_{\text{function}} \underbrace{(\text{crabapples})}_{\text{argument}} \Rightarrow_{\beta} \lambda x.\text{love}(x,\text{crabapples})$$

$$\underbrace{(\lambda x.\text{love}(x,\text{crabapples}))}_{\text{function}} \underbrace{(\text{Mary})}_{\text{argument}} \Rightarrow_{\beta} \text{love}(\text{Mary},\text{crabapples})$$

- Principle of compositionality: the meaning of an complex expression is a function of the meaning of its parts.
- Predicate logic can be used to give meaning representations for a certain portion of natural language.
- $\lambda$ -expressions can be used to represent meanings for *fragments of sentences*.
- In the next lecture, we'll see how the meaning of sentences can be systematically derived in a compositional way using such  $\lambda$ -expressions.

# Computing Natural Language Semantics

## Informatics 2A: Lecture 24

Shay Cohen

17 November 2015

## 1 Semantic Composition for NL

- Logical Form

## 2 Semantic Composition for NL

## 3 Semantic (Scope) Ambiguity

- Definition
- Semantic Scope
- Approaches to Scope Ambiguity
- Underspecification: General Idea

# Compositional Semantics: the key idea

## Grammar I

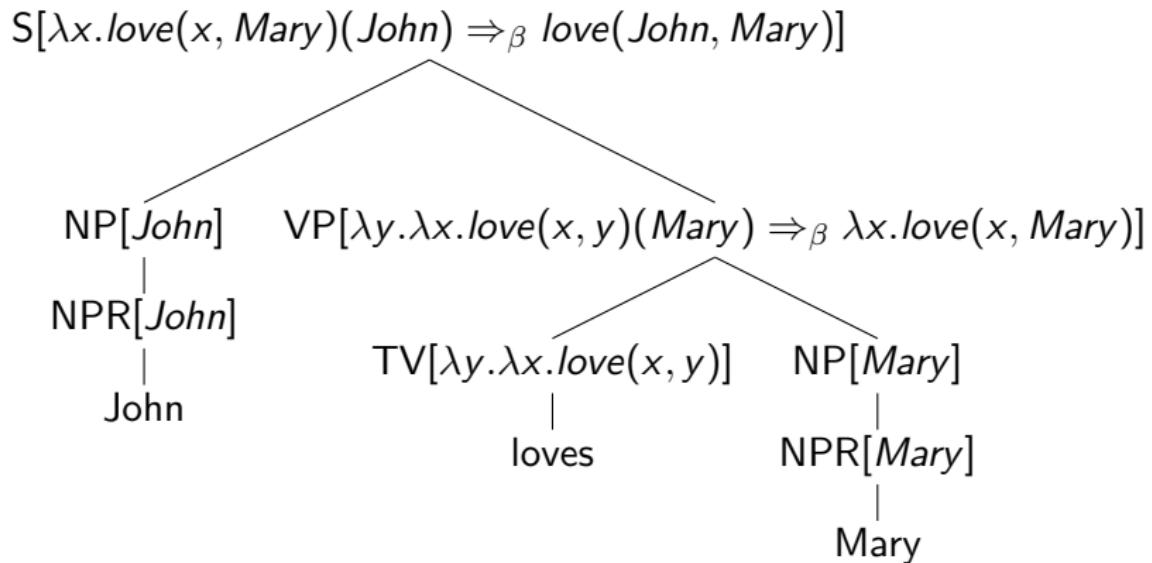
$S \rightarrow NP\ VP$	$\{VP.Sem(NP.Sem)\}$	$t$
$VP \rightarrow TV\ NP$	$\{TV.Sem(NP.Sem)\}$	$< e, t >$
$NP \rightarrow NPR$	$\{NPR.Sem\}$	$e$
$TV \rightarrow \text{loves}$	$\{\lambda y.\lambda x.\text{love}(x,y)\}$	$< e, < e, t >>$
$NPR \rightarrow \text{Mary}$	$\{\text{Mary}\}$	$e$
$NPR \rightarrow \text{John}$	$\{\text{John}\}$	$e$

- To build a compositional semantics for NL, we attach **valuation functions** to grammar rules (**semantic attachments**).
- These show how to compute the interpretation of the LHS of the rule from the interpretations of its RHS components.
- For example,  $VP.Sem(NP.Sem)$  means **apply** the interpretation of the VP to the interpretation of the NP.
- Types** have been added to ease understanding.

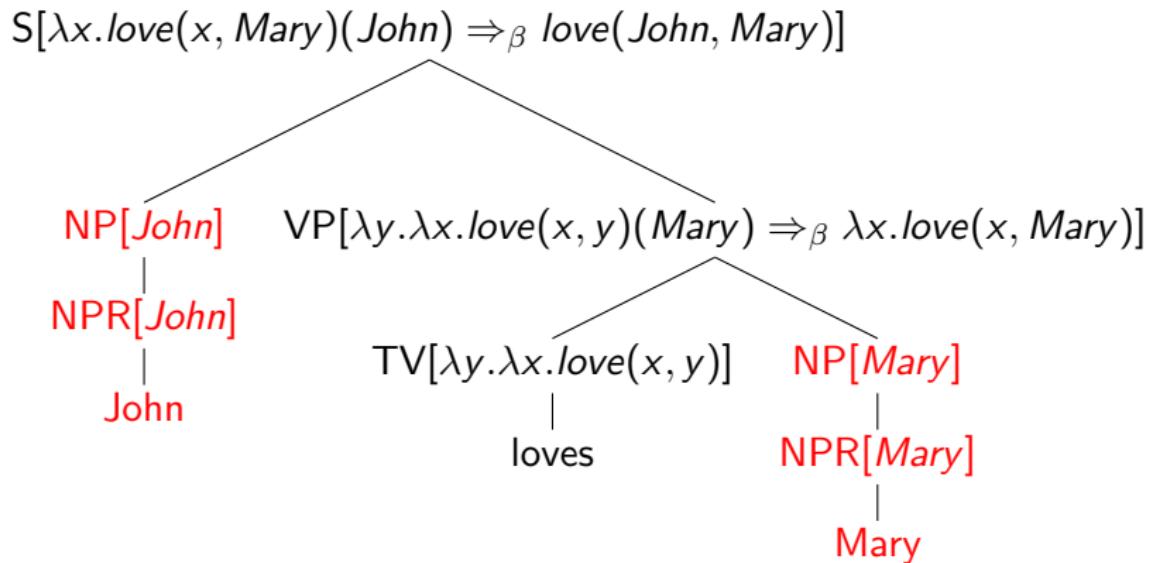
## Example

The semantics of “John loves Mary”:

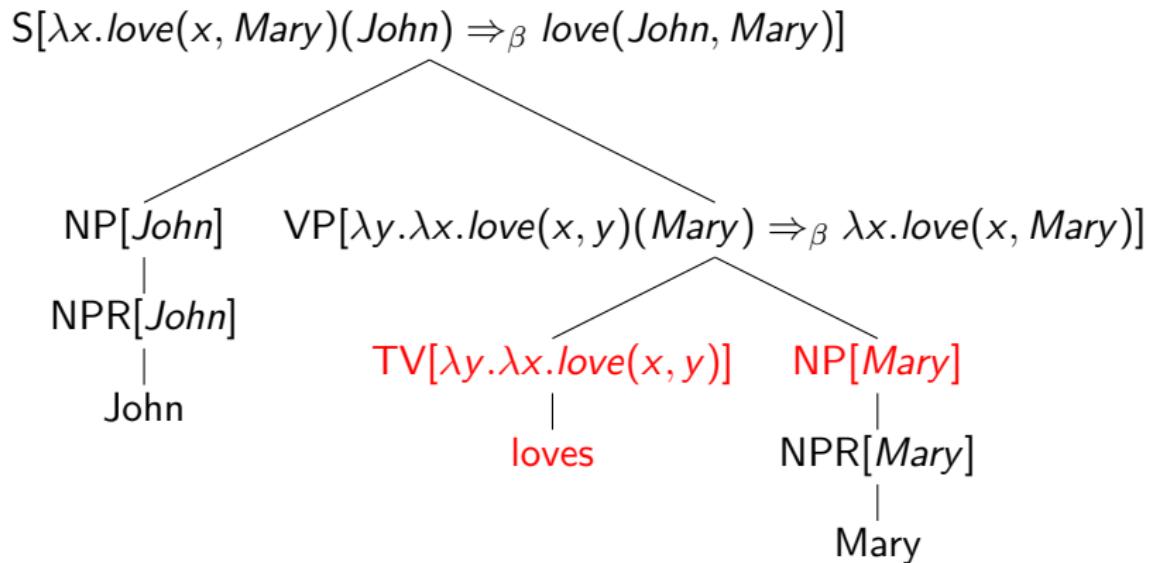
## Compositional Semantics: example



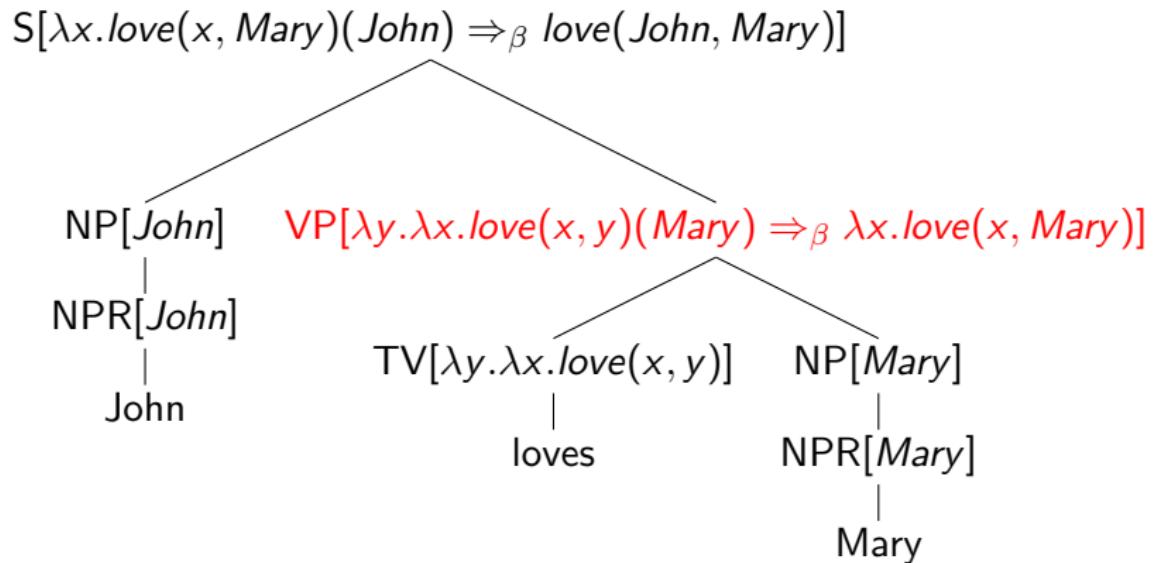
# Compositional Semantics: example



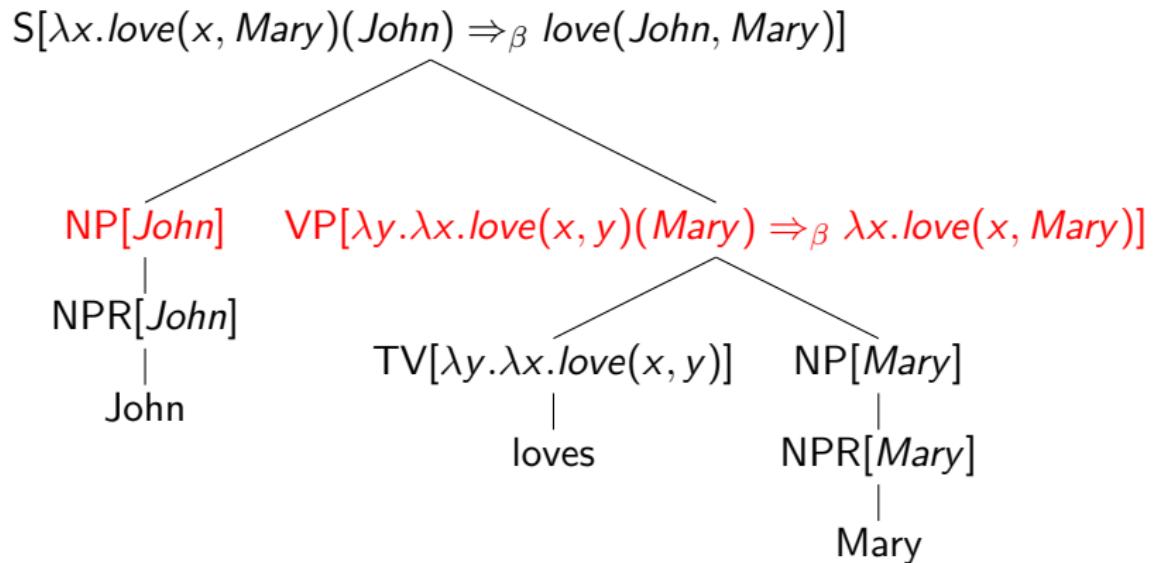
## Compositional Semantics: example



## Compositional Semantics: example

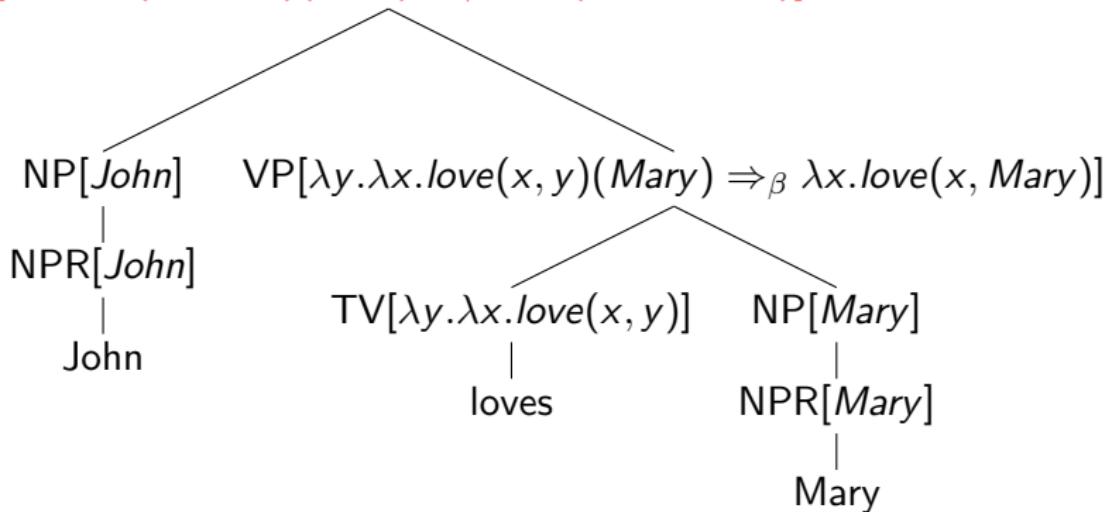


## Compositional Semantics: example



## Compositional Semantics: example

$S[\lambda x.love(x, Mary)(John) \Rightarrow_{\beta} love(John, Mary)]$



# A minor variation

The following alternative semantics assigns the same overall meaning to sentences. Only the treatment of the arguments of 'love' is different.

## Grammar I

$S \rightarrow NP\ VP$	$\{VP.Sem(NP.Sem)\}$	$t$
$VP \rightarrow TV\ NP$	$\{\lambda x.TV.Sem(x)(NP.Sem)\}$	$< e, t >$
$NP \rightarrow NPR$	$\{NPR.Sem\}$	$e$
$TV \rightarrow loves$	$\{\lambda x.\lambda y.love(x,y)\}$	$< e, < e, t >>$
$NPR \rightarrow Mary$	$\{Mary\}$	$e$
$NPR \rightarrow John$	$\{John\}$	$e$

## Compositional Semantics, continued

What about the interpretation of an NP other than a proper name? The FOPL interpretation should often contain an existential ( $\exists$ ) or a universal ( $\forall$ ) quantifier:

John has access to a computer.

$$\exists x(\text{computer}(x) \wedge \text{have\_access\_to}(\text{john}, x))$$

Every student has access to a computer.

$$\forall x(\text{student}(x) \rightarrow \exists y(\text{computer}(y) \wedge \text{have\_access\_to}(x, y)))$$

Can we build such interpretations up from their component parts in the same way as with proper names?

# A halfway stage.

## Grammar II

$S \rightarrow \text{NPR VP}$	$\{ \text{VP.Sem}(\text{NPR.Sem}) \}$	$t$
$\text{VP} \rightarrow \text{TV a Nom}$	$\{ \lambda x. \exists y. \text{Nom.Sem}(y) \& \text{TV.Sem}(y)(x) \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{N}$	$\{ \text{N.Sem} \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{A Nom}$	$\{ \lambda x. \text{Nom.Sem}(x) \& \text{A.Sem}(x) \}$	$\langle e, t \rangle$
$\text{NPR} \rightarrow \text{John}$	$\{ \text{John} \}$	$e$
$\text{TV} \rightarrow \text{loves}$	$\{ \lambda y. \lambda x. \text{love}(x, y) \}$	$\langle e, \langle e, t \rangle \rangle$
$\text{N} \rightarrow \text{girl}$	$\{ \lambda z. \text{girl}(z) \}$	$\langle e, t \rangle$
$\text{A} \rightarrow \text{tall}$	$\{ \lambda z. \text{tall}(z) \}$	$\langle e, t \rangle$

- Note we haven't given a meaning here to **a tall girl**.
- Could take this to have the same meaning as **tall girl**.
- This would be fine for this example (also in Assignment 2).  
But what about **every tall girl**?

# A halfway stage.

## Grammar II

$S \rightarrow \text{NPR VP}$	$\{ \text{VP.Sem}(\text{NPR.Sem}) \}$	$t$
$\text{VP} \rightarrow \text{TV a Nom}$	$\{ \lambda x. \exists y. \text{Nom.Sem}(y) \& \text{TV.Sem}(y)(x) \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{N}$	$\{ \text{N.Sem} \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{A Nom}$	$\{ \lambda x. \text{Nom.Sem}(x) \& \text{A.Sem}(x) \}$	$\langle e, t \rangle$
$\text{NPR} \rightarrow \text{John}$	$\{ \text{John} \}$	$e$
$\text{TV} \rightarrow \text{loves}$	$\{ \lambda y. \lambda x. \text{love}(x, y) \}$	$\langle e, \langle e, t \rangle \rangle$
$\text{N} \rightarrow \text{girl}$	$\{ \lambda z. \text{girl}(z) \}$	$\langle e, t \rangle$
$\text{A} \rightarrow \text{tall}$	$\{ \lambda z. \text{tall}(z) \}$	$\langle e, t \rangle$

- Note we haven't given a meaning here to **a tall girl**.
- Could take this to have the same meaning as **tall girl**.
- This would be fine for this example (also in Assignment 2).  
But what about **every tall girl**?

# A halfway stage.

## Grammar II

$S \rightarrow \text{NPR VP}$	$\{ \text{VP.Sem}(\text{NPR.Sem}) \}$	$t$
$\text{VP} \rightarrow \text{TV a Nom}$	$\{ \lambda x. \exists y. \text{Nom.Sem}(y) \& \text{TV.Sem}(y)(x) \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{N}$	$\{ \text{N.Sem} \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{A Nom}$	$\{ \lambda x. \text{Nom.Sem}(x) \& \text{A.Sem}(x) \}$	$\langle e, t \rangle$
$\text{NPR} \rightarrow \text{John}$	$\{ \text{John} \}$	$e$
$\text{TV} \rightarrow \text{loves}$	$\{ \lambda y. \lambda x. \text{love}(x, y) \}$	$\langle e, \langle e, t \rangle \rangle$
$\text{N} \rightarrow \text{girl}$	$\{ \lambda z. \text{girl}(z) \}$	$\langle e, t \rangle$
$\text{A} \rightarrow \text{tall}$	$\{ \lambda z. \text{tall}(z) \}$	$\langle e, t \rangle$

- Note we haven't given a meaning here to **a tall girl**.
- Could take this to have the same meaning as **tall girl**.
- This would be fine for this example (also in Assignment 2).  
But what about **every tall girl**?

# A halfway stage.

## Grammar II

$S \rightarrow \text{NPR VP}$	$\{ \text{VP.Sem}(\text{NPR.Sem}) \}$	$t$
$\text{VP} \rightarrow \text{TV a Nom}$	$\{ \lambda x. \exists y. \text{Nom.Sem}(y) \& \text{TV.Sem}(y)(x) \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{N}$	$\{ \text{N.Sem} \}$	$\langle e, t \rangle$
$\text{Nom} \rightarrow \text{A Nom}$	$\{ \lambda x. \text{Nom.Sem}(x) \& \text{A.Sem}(x) \}$	$\langle e, t \rangle$
$\text{NPR} \rightarrow \text{John}$	$\{ \text{John} \}$	$e$
$\text{TV} \rightarrow \text{loves}$	$\{ \lambda y. \lambda x. \text{love}(x, y) \}$	$\langle e, \langle e, t \rangle \rangle$
$\text{N} \rightarrow \text{girl}$	$\{ \lambda z. \text{girl}(z) \}$	$\langle e, t \rangle$
$\text{A} \rightarrow \text{tall}$	$\{ \lambda z. \text{tall}(z) \}$	$\langle e, t \rangle$

- Note we haven't given a meaning here to **a tall girl**.
- Could take this to have the same meaning as **tall girl**.
- This would be fine for this example (also in Assignment 2).  
But what about **every tall girl**?

## An example with Grammar II

The semantics of “John loves a tall girl”:

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Computing semantics with Grammar II

Before we add more, let's use Grammar II to compute the semantics of John loves a tall girl.

loves	TV	$\lambda yx. \text{love}(x, y)$
tall girl	Nom	$\lambda x. (\lambda z. \text{girl}(z))(x) \ \& \ (\lambda z. \text{tall}(z))(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \text{girl}(x) \ \& \ \text{tall}(x)$
loves a tall girl	VP	$\lambda x. \exists y. (\lambda x. \text{girl}(x) \ \& \ \text{tall}(x))(y) \ \&$ $(\lambda yx. \text{love}(x, y))(y)(x)$
	$\Rightarrow_{\beta}$	$\lambda x. \exists y. (\text{girl}(y) \ \& \ \text{tall}(y)) \ \&$ $\text{love}(x, y)$
John loves a tall girl	S	$(\lambda x. \exists y. \dots)(\text{John})$
	$\Rightarrow_{\beta}$	$\exists y. \text{girl}(y) \ \& \ \text{tall}(y) \ \& \ \text{love}(\text{John}, y)$

# Type raising

- We've given **John**, **Mary** the semantic type  $e$ , and **girl** the semantic type  $\langle e, t \rangle$ .
- But what type should **some girl** or **every girl** have?
- **Idea:** Since we wish to combine an NP.Sem with a VP.Sem (of type  $\langle e, t \rangle$ ) to get an S.Sem (of type  $t$ ), let's try again with NP.Sem having type  $\langle\langle e, t \rangle, t \rangle$ .

John       $\lambda P. P(John)$     (type raising)  
every girl     $\lambda P. \forall x. girl(x) \Rightarrow P(x)$

The appropriate semantic attachment for NP VP is then

$S \rightarrow NP\ VP \quad \{NP.Sem\ (VP.Sem)\}$

# Semantics of determiners

- Using this approach, we can also derive the semantics of 'every girl' from that of 'every' and 'girl'.
- We've seen that 'girl' has semantic type  $\langle e, t \rangle$ , and 'every girl' has semantic type  $\langle\langle e, t \rangle, t \rangle$ .
- So the interpretation of 'every' should have type  $\langle\langle e, t \rangle, \langle\langle e, t \rangle, t \rangle\rangle$ . Similarly for other *determiners* (e.g. every, a, no, not every).

girl	$\lambda x. \text{girl}(x)$	$\langle e, t \rangle$
every	$\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x)$	$\langle\langle e, t \rangle, \langle\langle e, t \rangle, t \rangle\rangle$
a	$\lambda Q. \lambda P. \exists x. Q(x) \wedge P(x)$	$\langle\langle e, t \rangle, \langle\langle e, t \rangle, t \rangle\rangle$
$\text{NP} \rightarrow \text{Det N}$	{ Det.Sem (N.Sem) }	$\langle\langle e, t \rangle, t \rangle$

We can now compute the semantics of 'every girl' and check that it  $\beta$ -reduces to  $\lambda P. \forall x. \text{girl}(x) \Rightarrow P(x)$ .

## Example

The semantics of “every girl”:

## More on type raising

- The natural rule for VP is now  $\text{VP} \rightarrow \text{TV NP}$ .
- Since the semantic type for NP has now been **raised** to  $\langle\langle e, t \rangle, t \rangle$ , and we want VP to have semantic type  $\langle e, t \rangle$ , what should the semantic type for TV be?

## More on type raising

- The natural rule for VP is now  $\text{VP} \rightarrow \text{TV NP}$ .
- Since the semantic type for NP has now been **raised** to  $\langle\langle e, t \rangle, t \rangle$ , and we want VP to have semantic type  $\langle e, t \rangle$ , what should the semantic type for TV be?

It had better be  $\langle\langle\langle e, t \rangle, t \rangle, \langle e, t \rangle\rangle$ .  
(A 3rd order function type!)

$$\begin{array}{ll} \text{TV} \rightarrow \text{loves} & \{\lambda R^{\langle\langle e, t \rangle, t \rangle}. \lambda z^e. R(\lambda w^e. \text{loves}(z, w))\} \\ \text{VP} \rightarrow \text{TV NP} & \{\text{TV.Sem}(\text{NP.Sem})\} \end{array}$$

To summarize where we've got to:

### Grammar III

$S \rightarrow NP\ VP$	{ NP.Sem(VP.Sem) }	$t$
$VP \rightarrow TV\ NP$	{ TV.Sem(NP.Sem) }	$< e, t >$
$NP \rightarrow John$	{ $\lambda P.P(John)$ }	$<< e, t >, t >$
$NP \rightarrow Det\ Nom$	{ Det.Sem(Nom.Sem) }	$<< e, t >, t >$
$Det \rightarrow a$	{ $\lambda Q.\lambda P.\exists x.Q(x) \wedge P(x)$ }	$<< e, t >, << e, t >, t >>>$
$Det \rightarrow every$	{ $\lambda Q.\lambda P.\forall x.Q(x) \Rightarrow P(x)$ }	$<< e, t >, << e, t >, t >>>$
$Nom \rightarrow N$	{ N.Sem }	$< e, t >$
$Nom \rightarrow A\ Nom$	{ $\lambda x.Nom.Sem(x) \& A.Sem(x)$ }	$< e, t >$
$TV \rightarrow loves$	{ { $\lambda R.\lambda z.R(\lambda w.loves(z, w))$ } }	$<<< e, t >, t >, < e, t >>$
$N \rightarrow girl$	{ $\lambda z.girl(z)$ }	$< e, t >$
$A \rightarrow tall$	{ $\lambda z.tall(z)$ }	$< e, t >$

Can add similar entries for 'student', 'computer', 'has access to'.

## Example

The semantics for 'every student has access to a computer'.

## Example

The semantics for 'every student has access to a computer'.

$$\begin{aligned} \text{every student } & (\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x))(\lambda x. \text{student}(x)) \\ & \rightarrow_{\beta} \lambda P. \forall x. \text{student}(x) \Rightarrow P(x) \end{aligned}$$

## Example

The semantics for 'every student has access to a computer'.

$$\begin{aligned} \text{every student } & (\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x))(\lambda x. \text{student}(x)) \\ & \rightarrow_{\beta} \lambda P. \forall x. \text{student}(x) \Rightarrow P(x) \end{aligned}$$

$$\begin{aligned} \text{a computer } & (\lambda Q. \lambda P. \exists x. Q(x) \wedge P(x))(\lambda x. \text{computer}(x)) \\ & \rightarrow_{\beta} \lambda P. \exists x. \text{computer}(x) \wedge P(x) \end{aligned}$$

## Example

The semantics for 'every student has access to a computer'.

$$\begin{aligned} \text{every student } & (\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x))(\lambda x. \text{student}(x)) \\ & \rightarrow_{\beta} \lambda P. \forall x. \text{student}(x) \Rightarrow P(x) \end{aligned}$$

$$\begin{aligned} \text{a computer } & (\lambda Q. \lambda P. \exists x. Q(x) \wedge P(x))(\lambda x. \text{computer}(x)) \\ & \rightarrow_{\beta} \lambda P. \exists x. \text{computer}(x) \wedge P(x) \end{aligned}$$

$$\begin{aligned} \text{h.a.t. a computer } & \cdots \rightarrow_{\beta} \cdots \\ & \rightarrow_{\beta} \lambda z. \exists x. \text{computer}(x) \wedge \text{h\_a\_t}(z, x) \end{aligned}$$

## Example

The semantics for 'every student has access to a computer'.

$$\begin{aligned} \text{every student } & (\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x))(\lambda x. \text{student}(x)) \\ & \rightarrow_{\beta} \lambda P. \forall x. \text{student}(x) \Rightarrow P(x) \end{aligned}$$

$$\begin{aligned} \text{a computer } & (\lambda Q. \lambda P. \exists x. Q(x) \wedge P(x))(\lambda x. \text{computer}(x)) \\ & \rightarrow_{\beta} \lambda P. \exists x. \text{computer}(x) \wedge P(x) \end{aligned}$$

$$\begin{aligned} \text{h.a.t. a computer } & \cdots \rightarrow_{\beta} \cdots \\ & \rightarrow_{\beta} \lambda z. \exists x. \text{computer}(x) \wedge \text{h\_a\_t}(z, x) \end{aligned}$$

$$\begin{aligned} (\text{whole sentence}) \cdots & \rightarrow_{\beta} \cdots \\ & \rightarrow_{\beta} \forall x. \text{student}(x) \Rightarrow \exists y. \text{computer}(y) \wedge \text{h\_a\_t}(x, y) \end{aligned}$$

## Example

The semantics for 'every student has access to a computer'.

$$\begin{aligned} \text{every student } & (\lambda Q. \lambda P. \forall x. Q(x) \Rightarrow P(x))(\lambda x. \text{student}(x)) \\ & \rightarrow_{\beta} \lambda P. \forall x. \text{student}(x) \Rightarrow P(x) \end{aligned}$$

$$\begin{aligned} \text{a computer } & (\lambda Q. \lambda P. \exists x. Q(x) \wedge P(x))(\lambda x. \text{computer}(x)) \\ & \rightarrow_{\beta} \lambda P. \exists x. \text{computer}(x) \wedge P(x) \end{aligned}$$

$$\begin{aligned} \text{h.a.t. a computer } & \cdots \rightarrow_{\beta} \cdots \\ & \rightarrow_{\beta} \lambda z. \exists x. \text{computer}(x) \wedge h\_a\_t(z, x) \end{aligned}$$

$$\begin{aligned} (\text{whole sentence}) \cdots & \rightarrow_{\beta} \cdots \\ & \rightarrow_{\beta} \forall x. \text{student}(x) \Rightarrow \exists y. \text{computer}(y) \wedge h\_a\_t(x, y) \end{aligned}$$

Note: In the last  $\beta$ -step, we've renamed 'x' to 'y' to avoid capture.

## Question

Suppose that the predicate  $L(x, y)$  means  $x$  loves  $y$ . Which of the following is **not** a possible representation of the meaning of *Everybody loves somebody*?

- ①  $\forall x. \exists y. L(x, y)$
- ②  $(\lambda P. \forall x. \exists y. P(x, y))(\lambda x. \lambda y. L(x, y))$
- ③  $(\lambda P. \forall x. \exists y. P(x, y))(\lambda x. \lambda y. L(y, x))$
- ④  $(\lambda P. \forall x. \exists y. P(y, x))(\lambda x. \lambda y. L(y, x))$

# Semantic Ambiguity

Whilst **every student has access to a computer** is neither syntactically nor lexically ambiguous, it has **two different interpretations** because of its determiners:

- **every**: interpreted as  $\forall$  (*universal quantifier*)
- **a**: interpreted as  $\exists$  (*existential quantifier*)

## Meaning 1

Possibly a different computer per student

$$\forall x(\text{student}(x) \rightarrow \exists y(\text{computer}(y) \wedge \text{have\_access\_to}(x, y)))$$

## Meaning 2

Possibly the same computer for all students

$$\exists y(\text{computer}(y) \wedge \forall x(\text{student}(x) \rightarrow \text{have\_access\_to}(x, y)))$$

The ambiguity arises because **every** and **a** each has its own **scope**:

Interpretation 1: **every has scope over a**

Interpretation 2: **a has scope over every**

- Scope is not uniquely determined either by left-to-right order, or by position in the parse tree.
- We therefore need other mechanisms to ensure that the ambiguity is reflected by there being multiple interpretations assigned to S.

# Scope ambiguity, continued

The number of interpretations grows exponentially with the number of scope operators:

**Every student at **some** university has access to a laptop.**

1. Not necessarily same laptop, not necessarily same university

$$\forall x(\text{stud}(x) \wedge \exists y(\text{univ}(y) \wedge \text{at}(x, y)) \rightarrow \exists z(\text{laptop}(z) \wedge \text{have\_access}(x, z)))$$

2. Same laptop, not necessarily same university

$$\exists z(\text{laptop}(z) \wedge \forall x(\text{stud}(x) \wedge \exists y(\text{univ}(y) \wedge \text{at}(x, y)) \rightarrow \text{have\_access}(x, z)))$$

3. Not necessarily same laptop, same university

$$\exists y(\text{univ}(y) \wedge \forall x((\text{stud}(x) \wedge \text{at}(x, y)) \rightarrow \exists z(\text{laptop}(z) \wedge \text{have\_access}(x, z))))$$

4. Same university, same laptop

$$\exists y(\text{univ}(y) \wedge \exists z(\text{laptop}(z) \wedge \forall x((\text{stud}(x) \wedge \text{at}(x, y)) \rightarrow \text{have\_access}(x, z))))$$

5. Same laptop, same university

$$\exists z(\text{laptop}(z) \wedge \exists y(\text{univ}(y) \wedge \forall x((\text{stud}(x) \wedge \text{at}(x, y)) \rightarrow \text{have\_access}(x, z))))$$

where 4 & 5 are equivalent

**Every student at **some** university does **not** have access to a computer.**

→ 18 interpretations

- ① **Enumerate all interpretations.** Computationally unattractive!
- ② Use an **underspecified representation** that can be further specified to each of the multiple interpretations on demand.

Sometimes the surrounding context will help us choose between interpretations:

Every student has access to a computer. It can be borrowed from the ITO. (⇒ Meaning 2)

- The idea in underspecified representations is that instead of trying to associate **a single FOPL formula** with a sentence, we associate **fragments of formulae** with various parts of the sentence.
- These fragments can have **holes** into which other fragments can be plugged. Since there may be some freedom in the order of plugging, the same bunch of fragments can give rise to several formulae with different scoping orders.
- There may also be **constraints** on the order of plugging, corresponding to partial information about the intended interpretation derived e.g. from the discourse context.

See J&M Chapter 18.3 for more on this.

- Syntax guides semantic composition in a systematic way.
- Lambda expressions facilitate the construction of compositional semantic interpretations.
- Logical forms can be constructed by attaching valuation functions to grammar rules.
- However, this approach is not adequate enough for quantified NPs, as LFs are not always isomorphic with syntax.
- We can elegantly handle scope by building an abstract underspecified representation and disambiguate on demand.

# Complexity and Character of Human Languages

Informatics 2A: Lecture 25

Shay Cohen

19 November 2015

## 1 Human Language Complexity

- Chomsky Hierarchy
- Strong and Weak Adequacy

## 2 Mildly Context-Sensitive Grammars

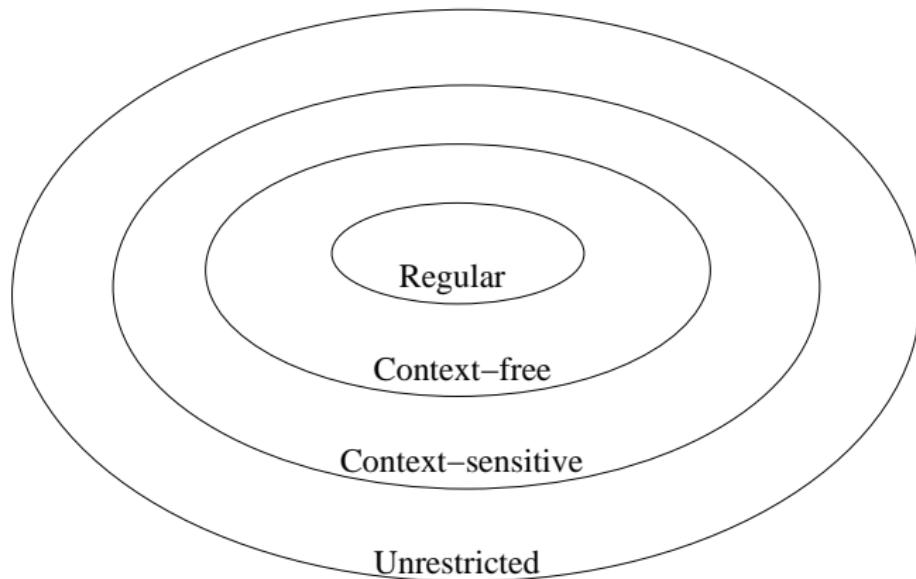
Reading: J&M. Chapter 16.3–16.4.

## Review

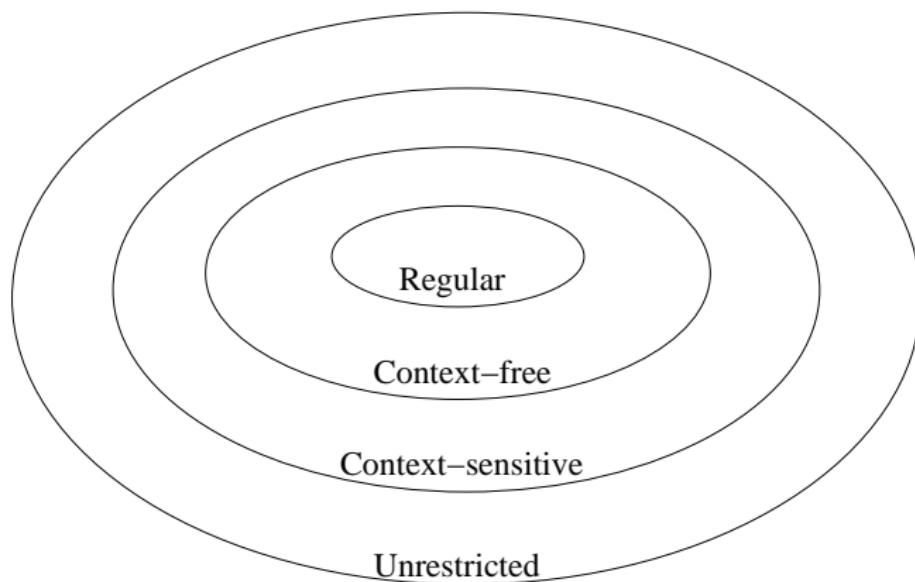
Chomsky Hierarchy: classifies languages on scale of complexity:

- **Regular** languages: those whose phrases can be 'recognized' by a finite state machine.
- **Context-free** languages: the set of languages accepted by pushdown automata. Many aspects of PLs and NLs can be described at this level;
- **Context-sensitive** languages: equivalent with a linear bounded nondeterministic Turing machine, also called a linear bounded automaton. Need this to capture e.g. *typing rules* in PLs.
- **Unrestricted** languages: *all* languages that can in principle be defined via mechanical rules.

# Review



## Review



Where do human languages fit within this complexity hierarchy?

# Recursion

The potential infiniteness of the language faculty has been recognized by Galileo, Descartes, von Humboldt.

## Discrete Infinity

- Sentences are built up by discrete units
- There are 6-word sentences, and 7-word sentences, but no 6.5 word sentences
- **There is no longest sentence!**
- **There is no non-arbitrary upper bound to sentence length!**

Mary thinks that John thinks that George thinks that Mary thinks that this course is boring!

I ate lunch and slept and watched tv and went to the bathroom and had a coffee and got dressed ...

# Strong and Weak Adequacy

Questions about the formal complexity of language are about the computational power of syntax, as represented by a grammar that's **adequate** for it.

## A strongly adequate grammar

- generates all and only the strings of the language;
- assigns them the “right” structures — ones that support a correct representation of meaning. (See previous lecture.)

## A weakly adequate grammar

generates all and only the strings of a language but doesn't necessarily give a correct (insightful) account of their structures.

# Is Natural Language Regular?

It is generally agreed that NLs are not (in principle) regular.

## Centre-embedding

[The cat<sub>1</sub> likes tuna fish<sub>1</sub>].

[The cat<sub>1</sub> [the dog<sub>2</sub> chased<sub>2</sub>] likes tuna fish<sub>1</sub>].

[The cat<sub>1</sub> [the dog<sub>2</sub> [the rat<sub>3</sub> bit<sub>3</sub>] chased<sub>2</sub>] likes tuna fish<sub>1</sub>].

# Is Natural Language Regular?

It is generally agreed that NLs are not (in principle) regular.

## Centre-embedding

[The cat<sub>1</sub> likes tuna fish<sub>1</sub>].

[The cat<sub>1</sub> [the dog<sub>2</sub> chased<sub>2</sub>] likes tuna fish<sub>1</sub>].

[The cat<sub>1</sub> [the dog<sub>2</sub> [the rat<sub>3</sub> bit<sub>3</sub>] chased<sub>2</sub>] likes tuna fish<sub>1</sub>].

## Idea of proof

(the+noun)<sup>n</sup> (transitive verb)<sup>n-1</sup> likes tuna fish.

A = { the cat, the dog, the rat, the elephant, the kangaroo ... }

B = { chased, bit, admired, ate, befriended ... }

Intersect /A\* B\* likes tuna fish/ with English

$L = x^n y^{n-1}$  likes tuna fish,  $x \in A, y \in B$

Use pumping lemma to show  $L$  is not regular

## Another example

Courtesy of an anonymous Inf2a student in the 2012 exam ...

John, Andrew and Mark were wearing T-shirts  
that were red, blue and yellow respectively.

Using this idea, can encode the language  $\{a^n b^n \mid n \geq 2\}$ .

# Is Natural Language Context Free?

It seems NLs aren't always context free! E.g. in Swiss German, some verbs (e.g. *let*, *paint*) take an object in **accusative form**, while others (e.g. *help*) take it in **dative form**.

## Crossing dependencies

... das mer	d'chind	em Hans	es huus	lönd	hälfte	aastriiche
... that we	the children	Hans	the house	let	help	paint

NP-ACC                    NP-DAT                    NP-ACC                    V-ACC                    V-DAT                    V-ACC

... that we let the children help Hans paint the house

Abstracting out the key feature here, we see that the same sequence over  $\{a, d\}$  (in this case *ada*) must 'appear twice'.

But it turns out that  $\{ss \mid s \in \{a, d\}^*\}$  isn't context-free (see a later lecture). Hence neither is Swiss German!

## Weaker examples

These ‘crossing dependencies’ are non-context-free in a very strong sense: no CFG is even **weakly adequate** for modelling them.

Other phenomena can *in theory* be modelled using CFGs, though it seems unnatural to do so. E.g. **a** versus **an** in English.

**a** banana            **an** apple

**a** large apple      **an** exceptionally large banana

Over-simplifying a bit: **a** before consonants, **an** before vowels.

In theory, we could use a **context-free** grammar:

$$\begin{array}{ll} \text{NP} \rightarrow \text{a NP}^c & \text{NP} \rightarrow \text{an NP}^v \\ \text{NP}^c \rightarrow \text{N}^c \mid \text{AP}^c \text{ NP}^1 & \text{NP}^v \rightarrow \text{N}^v \mid \text{AP}^v \text{ NP}^1 \\ \text{AP}^c \rightarrow \text{A}^c \mid \text{Adv}^c \text{ AP} & \text{AP}^v \rightarrow \text{A}^v \mid \text{Adv}^v \text{ AP} \end{array}$$

But more natural to use **context-sensitive** rules, e.g.

DET [c-word]  $\rightarrow$  **a** [c-word]

DET [v-word]  $\rightarrow$  **an** [v-word]

# Mild context sensitivity

A set  $\mathcal{L}$  of languages is mildly context-sensitive if:

- $\mathcal{L}$  contains all context-free languages.
- $\mathcal{L}$  can describe cross-serial dependencies. There is an  $n \geq 2$  such that  $\{w^k | w \in T^*\} \in \mathcal{L}$  for all  $k \leq n$ .
- The languages in  $\mathcal{L}$  are polynomially parsable.
- The languages in  $\mathcal{L}$  have the constant growth property.

Let  $X$  be an alphabet and  $L \subseteq X^*$ .  $L$  has constant growth property iff there is a constant  $c_0 > 0$  and a finite set of constants  $C \subset \mathbb{N} \setminus \{0\}$  such that for all  $w \in L$  with  $|w| > c_0$ , there is a  $w' \in L$  with  $|w| = |w'| + c$  for some  $c \in C$

Example: the language  $\{a^{2^n} | n \in \mathbb{N}\}$  does not have the constant growth property.

# Combinatory Categorial Grammars

CCGs are more powerful than CFGs, but less powerful than arbitrary CSGs.

They satisfy the criteria for mildly context-sensitive languages, i.e. the set of languages defined by CCGs is mildly context-sensitive.

The set of categories (nonterminals) in CCG is compositional, defined by a set of atomic units such as  $S$ ,  $NP$  and  $PP$ .

There are combination rules that tell us how to generate new categories from older ones in a derivation.

# Linear Indexed Grammars

Linear indexed grammars (LIGs) are more powerful than CFGs, but much less powerful than an arbitrary CSGs. Think of them as mildly context sensitive grammars. These seem to suffice for NL phenomena.

## Definition

An indexed grammar has three disjoint sets of symbols: terminals, non-terminals and indices.

An index is a stack of symbols that can be passed from the LHS of a rule to its RHS, allowing counting and recording what rules were applied in what order.

## Summary

- The ‘narrow’ language faculty involves a computational system that generates syntactic representations that can be mapped onto meanings.
- This raises the question of the complexity of this system (its position in the Chomsky hierarchy).
- A weakly adequate grammar generates the correct strings, while a strongly adequate one also generates the correct structures.
- NLs appear to surpass the power of context-free languages, but only just.
- The mild form of context-sensitivity captured by LIGs seems weakly adequate for NL structures.

**Next Lecture:** Models of human parsing.

# Models of Human Parsing

## Informatics 2A: Lecture 26

John Longley

20 November 2015

## 1 Human Parsing

- Cognitive Constraints
- Garden Paths
- Dimensions of Parsing

## 2 Experimental Data

- Eye-tracking
- Reading

## 3 Bottom-Up Parser

- Parallel Parsing
- Properties

## 4 Left Corner Parser

- Left Corner Chart
- Serial Parsing
- Properties

**Reading:** J&M, ch. 9 (pp. 350–352), ch. 12 (pp. 467–473), ch. 13 (pp. 491–496).

# Human Parsing

So far, we looked at parsing from an engineering perspective.  
However, **humans** also do parsing to understand language.

The mathematical and algorithmic tools in this course can be used to analyze **human parsing** (human sentence processing). This is the domain of **psycholinguistics**.

# Human Parsing

So far, we looked at parsing from an engineering perspective.  
However, **humans** also do parsing to understand language.

The mathematical and algorithmic tools in this course can be used to analyze **human parsing** (human sentence processing). This is the domain of **psycholinguistics**.

To study human parsing, we need:

- **experimental data** that tell us how humans parse;

# Human Parsing

So far, we looked at parsing from an engineering perspective. However, **humans** also do parsing to understand language.

The mathematical and algorithmic tools in this course can be used to analyze **human parsing** (human sentence processing). This is the domain of **psycholinguistics**.

To study human parsing, we need:

- **experimental data** that tell us how humans parse;
- **cognitive constraints** derived from these data (e.g., incrementality, garden paths, memory limitations);

# Human Parsing

So far, we looked at parsing from an engineering perspective. However, **humans** also do parsing to understand language.

The mathematical and algorithmic tools in this course can be used to analyze **human parsing** (human sentence processing). This is the domain of **psycholinguistics**.

To study human parsing, we need:

- **experimental data** that tell us how humans parse;
- **cognitive constraints** derived from these data (e.g., incrementality, garden paths, memory limitations);
- **parsing models** (and algorithms that implement them) that respect these constraints;

# Human Parsing

So far, we looked at parsing from an engineering perspective. However, **humans** also do parsing to understand language.

The mathematical and algorithmic tools in this course can be used to analyze **human parsing** (human sentence processing). This is the domain of **psycholinguistics**.

To study human parsing, we need:

- **experimental data** that tell us how humans parse;
- **cognitive constraints** derived from these data (e.g., incrementality, garden paths, memory limitations);
- **parsing models** (and algorithms that implement them) that respect these constraints;
- an **evaluation** of the models against the data.

## Incrementality

**Parsing:** extracting syntactic structure from a string; prerequisite for assigning a meaning to the string.

The human parser builds structures **incrementally** (word by word) as the input comes in.

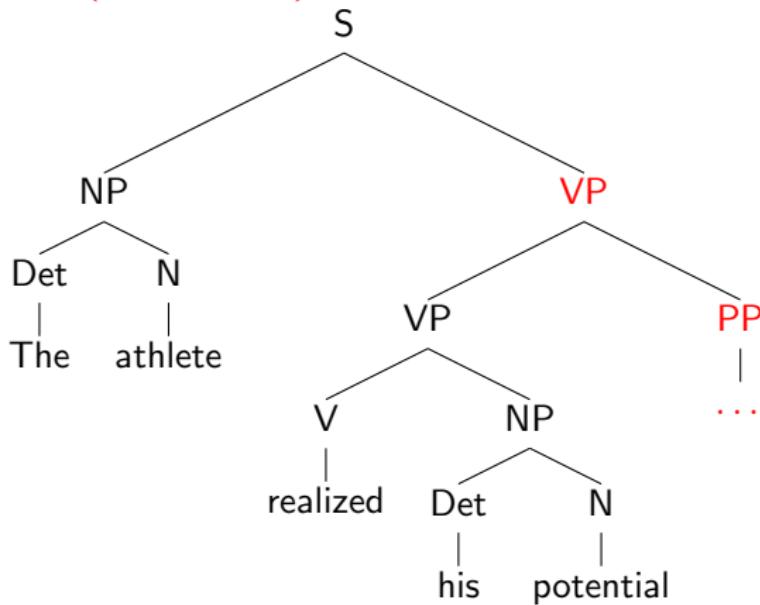
This can lead to **local ambiguity**.

Example:

- (1)     The athlete realized his potential ...
  - a.     ... at the competition.
  - b.     ... would make him a world-class sprinter.

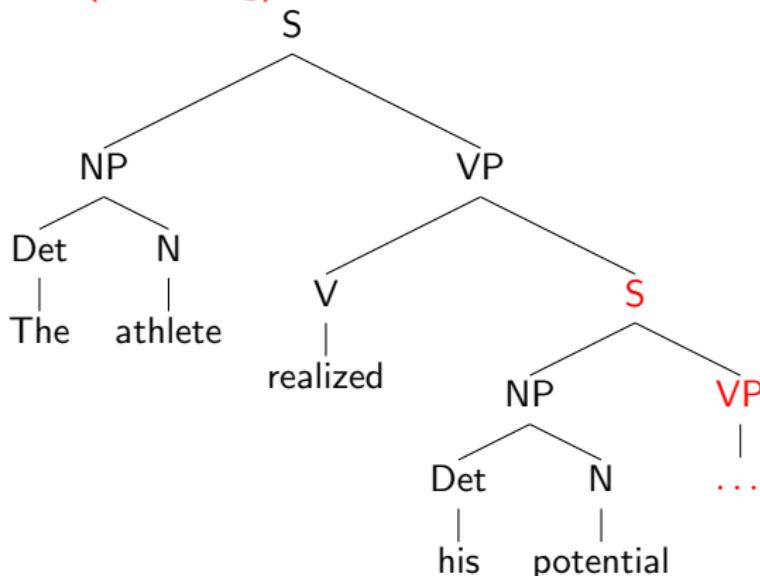
# Incrementality

Structure 1 (NP reading):



# Incrementality

Structure 2 (S reading):



# Garden Paths

- **Early commitment:** when it reaches *potential*, the processor has to decide which structure to build.
- If the parser makes the wrong choice (e.g., NP reading for sentence (1-b)) it needs to backtrack and revise the structure.
- A **garden path** occurs, which typically results in longer reading times (and reverse eye-movements).
- Some garden paths are so strong that the parser fails to recover from them.

# Garden Paths

More examples of garden paths:

- (2) a. The horse raced past the barn fell.  
b. I convinced her children are noisy.  
c. Until the police arrest the drug dealers control the street.  
d. The old man the boat.  
e. We painted the wall with cracks.  
f. Fat people eat accumulates.  
g. The cotton clothing is usually made of grows in Mississippi.  
h. The prime number few.

## Dimensions of Parsing

In addition to incrementality, a number of properties are important when designing a model of the human parser:

- **Directionality:** the parser can process sentence bottom-up (from the words up) or top-down (from the non-terminals down). Evidence that the human parser combines both strategies.
- **Parallelism:** a serial parser maintains only one structure at a time; a parallel parser pursues all possible structures. Controversial issue; evidence for both serialism and limited parallelism.
- **Interactivity:** the parser can be encapsulated (only access to syntactic information) or interactive (access to semantic information, context). Evidence for limited interactivity.

# Eye-tracking

An **eye-tracker** makes it possible to record the eye-movements of subjects while they are performing a cognitive task:

- looking at a scene;
- driving a vehicle;
- using a computer;
- reading a text.

**Mind's Eye Hypothesis:** where subjects are looking indicates what they are processing. How long they are looking at it indicates how much processing effort is needed.

# Eye-tracking



A head-mounted, video-based eye-tracker.



# Eye-movements and Reading

Let's look at eye-tracking data for **reading** in detail:

- eye-movements are recorded while subjects read texts;
- very high spatial ( $0.15^\circ$  visual angle) and temporal (1 ms) accuracy;
- eye movements in reading are saccadic: a series of relatively stationary periods (**fixations**) between very fast movements (**saccades**);
- average fixation time is about 250 ms; can be longer or shorter, depending on ease or difficulty of processing;
- typically test a number of subjects, with a number of test sentences, and statistical analysis done on results.

## Eye-movements and Reading

Buck did not read the newspapers, or he would have known that trouble was brewing, not alone for himself, but for every tide-water dog, strong of muscle and with warm, long hair, from Puget Sound to San Diego. Because men, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were booming the find, thousands of men were rushing into the Northland. These men wanted dogs, and the dogs they wanted were heavy dogs, with strong muscles by which to toil, and furry coats to protect them from the frost.

Buck lived at a big house in the sun-kissed Santa Clara Valley. Judge Miller's place, it was called. It stood back from the road, half hidden among the trees, through which glimpses could be caught of the wide cool veranda that ran around its four sides.

## Eye-movements and Reading

Buck did not read the newspapers, or he would have known that trouble was brewing, not alone for himself, but for every tide-water dog, strong of muscle and with warm, long hair, from Puget Sound to San Diego. Because men, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were booming the find, thousands of men were rushing into the Northland. These men wanted dogs, and the dogs they wanted were heavy dogs, with strong muscles by which to toil, and furry coats to protect them from the frost.

Buck lived at a big house in the sun-kissed Santa Clara Valley. Judge Miller's place, it was called. It stood back from the road, half hidden among the trees, through which glimpses could be caught of the wide cool veranda that ran around its four sides.

## Eye-movements and Reading

Buck did not read the newspapers, or he would have known that trouble was brewing, not alone for himself, but for every tide-water dog strong of muscle and with warm, long hair, from Puget Sound to San Diego. Because man, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were beginning to stir, thousands of men were rushing into the Northland. These men wanted dogs, and the dogs they wanted were heavy dogs, with strong muscles by which to toil, and fury coats to protect them from the frost.

Buck lived at a big house in the sun-kissed Santa Clara Valley. Judge Miller's place, it was called. It stood back from the road, half hidden among the trees, through which glimpses could be caught of the wide cool veranda that ran around its four sides.

## Eye-movements and Reading

We can use the data generated by eye-tracking experiments to investigate how the human parser works. For example:

- evidence for **garden paths** comes from increased reading times, and more reverse saccades, when reading certain words;
- evidence for **incrementality** comes from studies where participants view visual scenes while listening to sentences;
- evidence for **interactivity** comes from the fact that semantic properties of words influence reading times in the same way as syntactic ones.

We will look at how to **model** these properties by building a parser that mimics human parsing behavior.

## Clicker Question

Which of the following sentences in **not** a garden path?

- ① The man returned to his house was happy.
- ② The complex houses married and single soldiers and their families.
- ③ The tomcat that curled up on the cushion seemed friendly.
- ④ The sour drink from the ocean.

# A Small Grammar of English

We need a grammar of English to study our parsing models.

**Phrasal categories:**

S: sentence, NP: noun phrase, VP: verb phrase

**Syntactic categories** (aka parts of speech):

Det: determiner, CN: common noun, TV: transitive verb

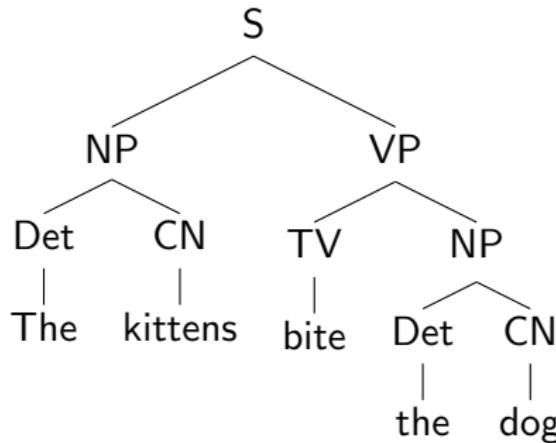
**Phrase structure rules:**

S	→	NP VP	Det	→	the
NP	→	Det CN	CN	→	kittens
VP	→	TV NP	TV	→	bite

CN → dog

# Syntax Tree

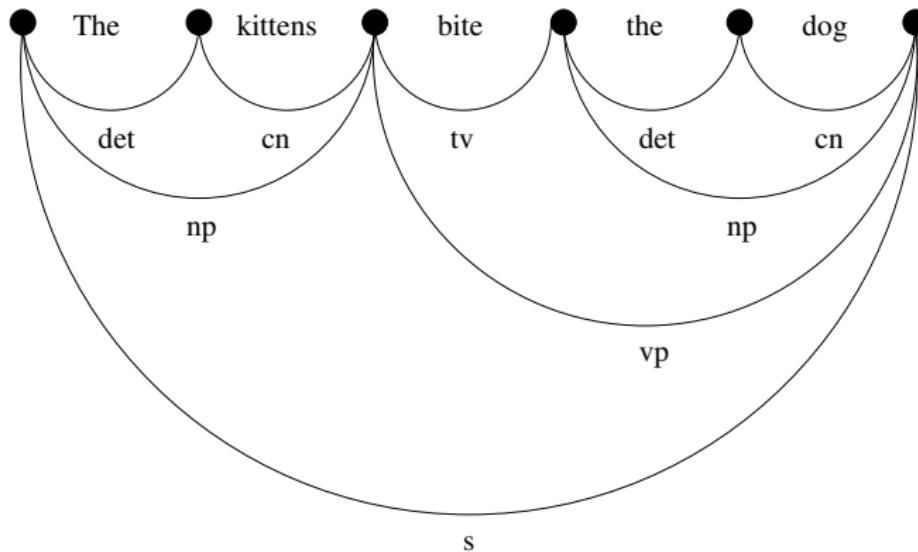
The syntax tree for the sentence *the kittens bite the dog*:



Let's assume that this tree is constructed a **CYK parser** as discussed in lecture 16.

# A Bottom-Up Parallel Parser

Example of a CYK chart (in graph notation):



## Properties of the Model

This offers a simple model of the human parser with the following properties:

- **bottom-up**: parsing is driven by the addition of words to the chart; chart is expanded upwards from lexical to phrasal categories;
- **limited incrementality**: when a new word appears, all possible edges are added to the chart; then the system waits for the next word;
- **parallelism**: all chart edges are added at the same time; multiple analyses are pursued.

## Properties of the Model

This offers a simple model of the human parser with the following properties:

- **bottom-up**: parsing is driven by the addition of words to the chart; chart is expanded upwards from lexical to phrasal categories;
- **limited incrementality**: when a new word appears, all possible edges are added to the chart; then the system waits for the next word;
- **parallelism**: all chart edges are added at the same time; multiple analyses are pursued.

Doesn't fit with observed data. Processing time isn't correlated to number of edges to be added. And on the above model, garden paths wouldn't be a particular problem.

# Left Corner Parsing

**Cognitively plausible incrementality:** each word is integrated into the structure as it appears (no unconnected words).

This can be achieved using **left corner** chart parsing. This is similar to **Earley parsing**, but with a more ‘bottom-up’ Predictor:

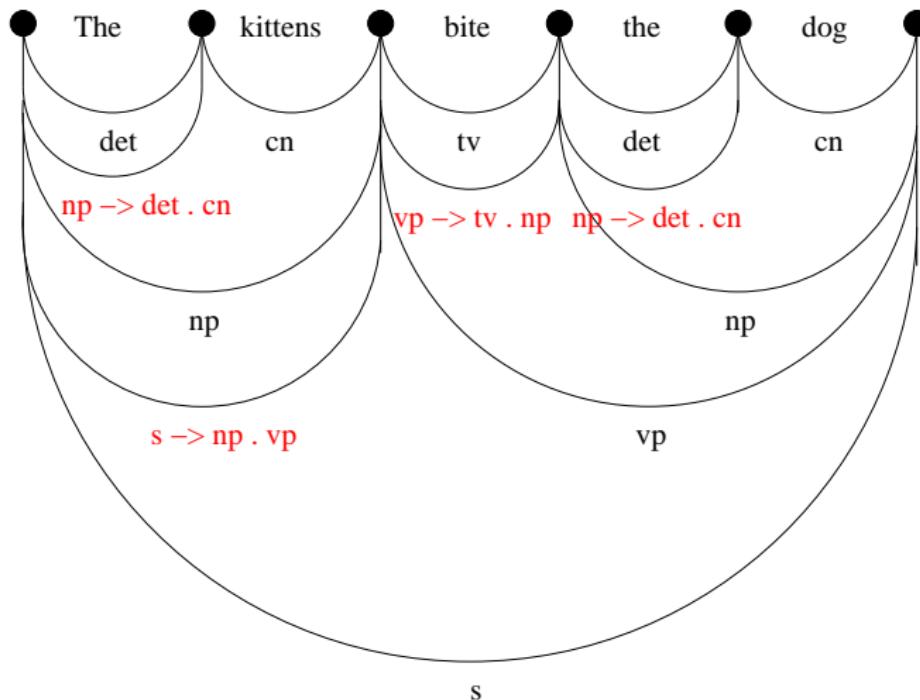
PREDICTOR ( $A \rightarrow \alpha \bullet, [i, k]$ )

for each  $(B \rightarrow A\beta)$  in Grammar\_Rules\_For( $B, grammar$ ):

enqueue( $(B \rightarrow A \bullet \beta, [i, k]), chart[i]$ )

Completer and Scanner are basically the same as in Earley (Scanner gets to start).

## Example of a Left Corner Chart



# Serial Parsing

If parsing was fully parallel, all analyses of a sentence would be equally available; there would be no garden paths.

In the literature, two types of models have been assumed:

- **ranked parallel**: multiple structures are pursued in parallel; they are ranked in order of preference; garden paths occur if a low-ranked structure turns out to be correct;
- **serial**: only one structure is pursued; if it turns out to be incorrect, then a garden path occurs.

Let's assume a **serial left-corner parser with backtracking**.

# Serial Parsing

Serial left-corner parser with backtracking:

- At each point of ambiguity, the parser has to choose one structure, instead of adding all possible structures to the chart;
- if this structure turns out to be incorrect; the parser has to backtrack;
- at the last point of ambiguity, the incorrect structure is removed from the chart, and an alternative one is added to the chart instead;
- this requires additional data structures to keep track of the choice points and the choices made.

## Properties of the Model

Properties of the left corner model:

- this model will parse garden path sentences such as *the horse raced past the barn fell*;
- extensive backtracking will occur for such sentences; only possible if the stack size of the choice point stack is sufficient;

Potential issues:

- backtracking requires that parse failure is detected; requires that the parser knows where the sentence boundaries are;
- processing order is fixed; context or experience is not taken into account; no attempt to minimize backtracking.

## Summary

- The human parser builds syntactic structure in response to strings of words;
- parsing models have to capture the incrementality of human parsing and account for ambiguity resolution (garden paths);
- parsing models can be implemented using a chart (representing partial syntactic structure);
- a simple bottom-up parser assumes limited incrementality, full parallelism: not cognitively plausible;
- left-corner parsing models achieves a higher degree of incrementality;
- to model garden paths, we can assume serial parsing with backtracking.

# Semantics of programming languages

## Informatics 2A: Lecture 27

John Longley

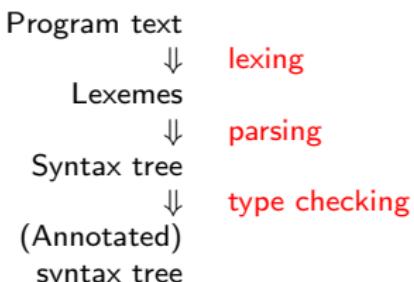
School of Informatics  
University of Edinburgh  
`als@inf.ed.ac.uk`

24 November 2015

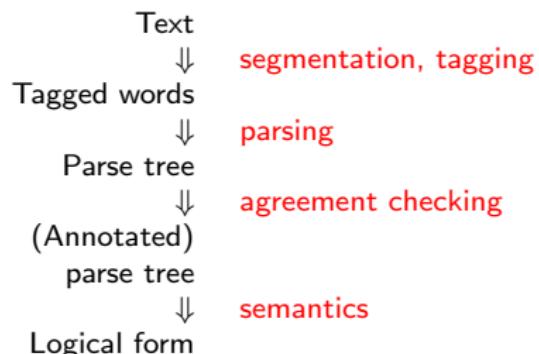
# Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

## Formal Language



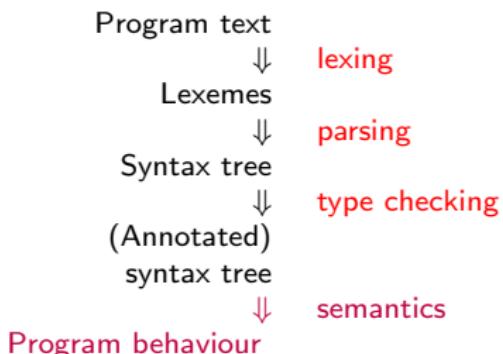
## Natural Language



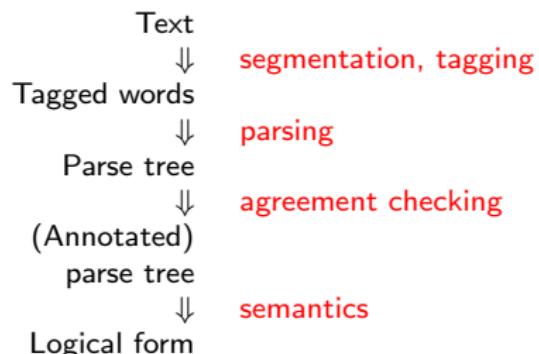
# Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

## Formal Language

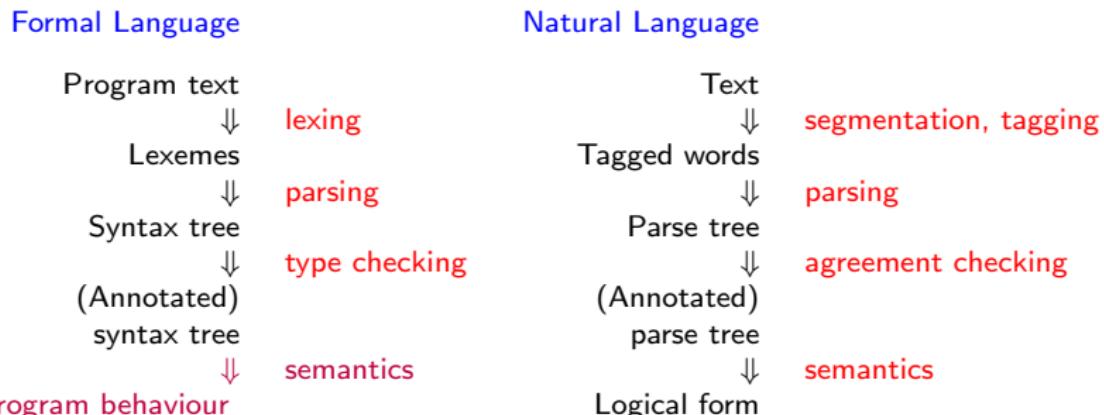


## Natural Language



# Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.



Today we look at methods of specifying program behaviour.

# Semantics for programming languages

The **syntax** of NLs (as described by CFGs etc.) is concerned with what sentences are **grammatical** and what structure they have, whilst their **semantics** are concerned with what sentences **mean**.

A similar distinction can be made for programming languages. Rules associated with lexing, parsing and typechecking concern the form and structure of legal programs, but say nothing about what programs should **do** when you run them.

The latter is what **programming language semantics** is about. It thus concerns the later stages of the **formal language processing pipeline**.

# Specification vs. implementation

In principle, one way to give a semantics (or ‘meaning’) for a programming language is to provide a [working implementation](#) of it, e.g. an interpreter or compiler for the language.

However, such an implementation will probably consist of thousands of lines of code, and so isn’t very suitable as a readable definition or [reference specification](#) of the language.

The latter is what we’re interested in here. In other words, we want to fill the blank in the following table:

	Specification	Implementation
Lexical structure	Regular exprs.	Lexer impl.
Grammatical structure	CFGs	Parser impl.
Execution behaviour	???	Interpreter/compiler

# Semantic paradigms

We'll look at two styles of formal programming language semantics:

- **Operational semantics.** Typically consists of a bunch of rules for 'executing' programs given by syntax trees. Oriented towards **implementations** of the language. indeed, an op. sem. often gives rise immediately to a 'toy implementation'.
- **Denotational semantics.** Typically consists of a **compositional** description of the meaning of program phrases (close in spirit to what we've seen for NLs). Oriented towards **mathematical reasoning** about the language and about programs written in it. May be 'executable' or not.

These two styles are complementary: ideally, it's nice to have both. There are also other styles (e.g. **axiomatic semantics**), but we won't discuss them here.

## Micro-Haskell: recap

We use Micro-Haskell (recall Lecture 13 and Assignment 1) as a vehicle for introducing the methods of operational and denotational semantics.

The format of MH declarations is illustrated by:

```
div :: Integer -> Integer -> Integer ;
div x y = if x < y then 0 else 1 + div (x + -y) y ;
```

This declares a function `div`, of the type specified, such that, when applied to two (non-negative) integer literals  $\overline{m}$  and  $\overline{n}$ , the function application

`div  $\overline{m}$   $\overline{n}$`

returns, as result, the integer literal representing the integer division of  $m$  by  $n$ .

## Semantic paradigms in the case of MH

### Operational semantics:

This explains the **computational process** by which MH calculates the value of a function application, such as  $\text{div } \overline{m} \overline{n}$ .

### Denotational semantics:

This defines a mathematical **denotation**

$$[\![\text{div}]\!] \in [\![\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}]\!]$$

**Roughly**,  $[\![\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}]\!]$  is the set of binary functions on integers, and  $[\![\text{div}]\!]$  is the integer-division function.

**In reality**, denotational semantics is more complicated than this.

# Operational semantics

We model the execution behaviour of programs as a series of **reduction steps**.

E.g. for Micro-Haskell:

```
if 4+5 <= 8 then 4 else 6+7
→ if 9 <= 8 then 4 else 6+7
→ if False then 4 else 6+7
→ 6+7
→ 13
```

A (small-step) operational semantics is basically a bunch of rules for performing such reductions.

## More complex example

Consider the Micro-Haskell declaration

$$f \ x \ y = x+y+x ;$$

This effectively introduces the definition

$$f = \lambda x. \lambda y. \ x+y+x$$

Now consider the evaluation of  $f \ 3 \ 4$ :

$$\begin{aligned} f \ 3 \ 4 &\rightarrow (\lambda x. \lambda y. \ x+y+x) \ 3 \ 4 \\ &\rightarrow (\lambda y. \ 3+y+3) \ 4 \\ &\rightarrow 3+4+3 \\ &\rightarrow 7 + 3 \\ &\rightarrow 10 \end{aligned}$$

Notice that two of these steps are  **$\beta$ -reductions!**

## Operational semantics for Micro-Haskell: general rules

Suppose  $E$  is a **runtime environment** associating a definition to each function symbol, e.g.  $E(f) = \lambda x. \lambda y. x + y + x$ .

Also let  $v$  range over **variables** of MH, and write  $\bar{n}$  to mean the integer literal for  $n$ .

Relative to  $E$ , we can define  $\rightarrow$  as follows:

- $v \rightarrow E(v)$  ( $v$  a variable defined in  $E$ )
- $(\lambda v. M)N \rightarrow M[v \mapsto N]$  ( $\beta$ -reduction)
- $\bar{m} + \bar{n} \rightarrow \overline{m + n}$ , and similarly for other infixes.
- if True then  $M$  else  $N \rightarrow M$
- if False then  $M$  else  $N \rightarrow N$

Continued on next slide ...

## Operational semantics for Micro-Haskell (continued)

Let's say a term  $M$  is a **value** if it's an integer literal, a boolean literal, or a  $\lambda$ -abstraction. Let  $V$  range over values,

**Intuition:** values are terms that can't be reduced any further. We try to reduce all other terms to values.

To complete the definition of  $\rightarrow\!\!\!\rightarrow$ , we decree that if  $M \rightarrow\!\!\!\rightarrow M'$  then:

- $MN \rightarrow\!\!\!\rightarrow M'N$
- $M \odot N \rightarrow\!\!\!\rightarrow M' \odot N$  ( $\odot$  any infix symbol)
- $V \odot M \rightarrow\!\!\!\rightarrow V \odot M'$  (ditto)
- if  $M$  then  $N$  else  $P \rightarrow\!\!\!\rightarrow$  if  $M'$  then  $N$  else  $P$

We then say  $M \rightarrow\!\!\!\rightarrow^* V$  (" $M$  evaluates to  $V$ ") if there's a sequence

$$M \equiv M_0 \rightarrow\!\!\!\rightarrow M_1 \rightarrow\!\!\!\rightarrow \cdots \rightarrow\!\!\!\rightarrow M_r \equiv V$$

That defines the intended behaviour of Micro-Haskell programs.  
It's also how the Assignment 1 **evaluator** for MH works.

## Two questions

Consider the following MH program.

```
e :: Integer ;  
e = e ;  
  
f :: Integer -> Integer ;  
f x = if True then 2+3 else 1+x ;
```

What happens if the expressions below are evaluated?

e                    f e

- ➊ There is a parse error.
- ➋ There is a type error.
- ➌ The evaluator goes into a loop.
- ➍ The evaluator terminates and produces an integer as result.
- ➎ The evaluator terminates and produces a different result.

## Operational semantics: further remarks

What happens if we encounter an expression that isn't a value but can't be reduced? E.g.  $5\ \text{True}$ , or  $(\lambda x.x)+4$  ?

!!! If our original program typechecks, this can never happen !!!

Indeed, it can be proved that:

- if  $M$  can be typed, either it's a value or it can be reduced;
- if  $M$  has type  $t$  and  $M \rightarrow M'$ , then  $M'$  has type  $t$ .

That's one reason why type systems are so valuable: they can guarantee programs won't derail at runtime.

The general form of operational semantics we've described is immensely flexible. It works beautifully for functional languages like MH. But it can also be adapted to most other kinds of programming language.

## Denotational semantics

An operational semantics provides a kind of idealized implementation of the language in terms of symbolic rules.

That's fine, but doesn't give much 'structural' understanding. Conceptually and mathematically, it is more satisfying to assign **meaning** to (parts of) a program — in roughly the way that mathematical expressions (or indeed NL expressions) have meaning.

This is the idea behind denotational semantics: associate a **denotation**  $\llbracket P \rrbracket$  to each program phrase  $P$  in a compositional way.

## Denotational semantics for MH: first attempt

Let's try interpreting MH types by **sets** in a natural way:

$$\llbracket \text{Integer} \rrbracket = \mathbb{Z} \quad \llbracket \text{Bool} \rrbracket = \mathbb{B} = \{T, F\}$$

$$\llbracket \sigma \rightarrow \tau \rrbracket = \llbracket \tau \rrbracket^{\llbracket \sigma \rrbracket} \quad (\text{set of all functions from } \llbracket \sigma \rrbracket \text{ to } \llbracket \tau \rrbracket)$$

A term  $M :: \tau$  will receive a denotation  $\llbracket M \rrbracket \in \llbracket \tau \rrbracket$ . More accurately, if  $M :: \tau$  is a term in the **type environment**  
 $\Gamma = \langle x_1 :: \sigma_1, \dots, x_n :: \sigma_n \rangle$ , its denotation will be a function

$$\llbracket M \rrbracket_{\Gamma} : \llbracket \sigma_1 \rrbracket \times \cdots \times \llbracket \sigma_n \rrbracket \rightarrow \llbracket \tau \rrbracket$$

We define  $\llbracket M \rrbracket_{\Gamma}$  **compositionally**. E.g. writing  $\vec{a}$  for  $\langle a_1, \dots, a_n \rangle$ :

$$\llbracket \bar{n} \rrbracket_{\Gamma} : \vec{a} \mapsto n$$

$$\llbracket x_i \rrbracket_{\Gamma} : \vec{a} \mapsto a_i$$

$$\llbracket M + N \rrbracket_{\Gamma} : \vec{a} \mapsto \llbracket M \rrbracket_{\Gamma}(\vec{a}) + \llbracket N \rrbracket_{\Gamma}(\vec{a})$$

$$\llbracket MN \rrbracket_{\Gamma} : \vec{a} \mapsto \llbracket M \rrbracket_{\Gamma}(\vec{a})(\llbracket N \rrbracket_{\Gamma}(\vec{a})), \quad \text{etc.}$$

## Denotational semantics for MH: the challenge

That works well as far as it goes. The problem comes when we try to interpret **recursive** definitions, e.g.

```
div = λx.λy.if x < y then 0 else 1 + div (x + -y) y ;
```

Here we'd end up trying to define  $\llbracket \text{div} \rrbracket$  in terms of itself!

Our simple ‘set-theoretic’ interpretation can’t make sense of this. One needs an alternative interpretation that builds in some deeper mathematical properties in order to allow ‘circular definitions’. E.g. one can interpret the whole of MH using

- **complete partial orders** (non-executable semantics)
- **game models** (executable semantics)

This is where denotational semantics gets interesting, and where, reluctantly, we move on from MH to something simpler ...

## Denotational semantics for regular expressions

Let's turn to an easier example. Recall our (meta)language of regular expressions:

$$R \rightarrow \epsilon \mid \emptyset \mid a \mid RR \mid R + R \mid R^*$$

In fact, we've already met two good den. sems. for this!

- $\llbracket R \rrbracket_1 = \mathcal{L}(R)$ , the language (i.e. set of strings) defined by  $R$ .
- $\llbracket R \rrbracket_2$  = the particular ( $\epsilon$ -)NFA for  $R$  constructed by the methods of Lecture 5.

Both of these are defined compositionally: e.g.  $\mathcal{L}(R + R')$  is defined as  $\mathcal{L}(R) \cup \mathcal{L}(R')$ , and the standard NFA for  $R + R'$  is constructed out of NFAs for  $R$  and  $R'$ . Note that:

- $\llbracket - \rrbracket_1$  is more abstract than  $\llbracket - \rrbracket_2$ : can have  $\llbracket R \rrbracket_2 \neq \llbracket R' \rrbracket_2$  but  $\llbracket R \rrbracket_1 = \llbracket R' \rrbracket_1$ . So  $\llbracket - \rrbracket_1$  is more useful for arguing that two regular expressions are 'equivalent'.
- However,  $\llbracket - \rrbracket_2$  is naturally executable, while  $\llbracket - \rrbracket_1$  is not.

## Summary

- Formal semantics can be used to give a concise and precise **reference specification** for the intended behaviour of programs.
- Operational semantics is nowadays widely used. Denotational semantics gets quite mathematical and is at present more of a research topic.
- Operational semantics, and some kinds of denotational semantics, also offer a starting-point for building working **implementations** of the language.
- Denotational semantics also offers a framework for **proving** things about programs. E.g. if  $\llbracket P \rrbracket = \llbracket P' \rrbracket$ , that shows that  $P$  can be replaced by  $P'$  *in any program context* without changing the program's behaviour.
- Ideas from both op. and den. semantics have had a significant effect on the **design** of programming languages.

# Context-sensitive languages

Informatics 2A: Lecture 28

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

26 November 2015

## Recap: context-sensitivity in natural language

An example of **context sensitivity** in natural language was presented in Lecture 25:

- Crossing dependencies in Swiss German (and Dutch).

There are other phenomena that are most naturally described in a 'context-sensitive' way (e.g. choice between the determiners *a* and *an*).

Such phenomena take natural languages outside the **context-free** level of the Chomsky hierarchy.

It is believed that natural languages naturally live (comfortably) within the **context-sensitive** level of the Chomsky hierarchy.

# In today's lecture . . .

. . . we look at what lies beyond context-free languages from a formal language viewpoint.

- How to show that a language is not context free.
- Defining the notion of **context-sensitive language** using **context-sensitive grammars**.
- An alternative characterisation of **context-sensitive languages** using **noncontracting grammars**.
- The notion of **unrestricted grammar**, and the associated **recursively-enumerable languages**.

## Non-context-free languages

We saw in Lecture 8 that the **pumping lemma** can be used to show a language isn't regular.

There's also a context-free version of this lemma, which can be used to show that a language isn't even context-free:

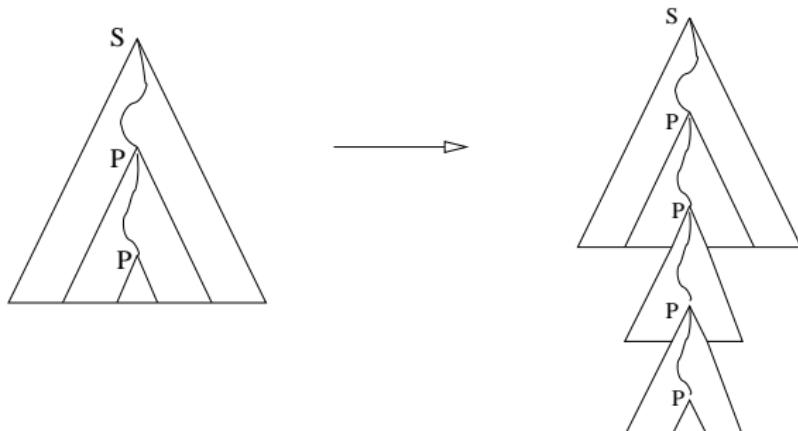
**Pumping Lemma for context-free languages.** Suppose  $L$  is a context-free language. Then  $L$  has the following property.

(P) *There exists  $k \geq 0$  such that every  $z \in L$  with  $|z| \geq k$  can be broken up into five substrings,  $z = uvwxy$ , such that  $|vx| \geq 1$ ,  $|vwx| \leq k$  and  $uv^iwx^iy \in L$  for all  $i \geq 0$ .*

## Context-free pumping lemma: the idea

In the regular case, the key point is that any sufficiently long string will **visit the same state twice**.

In the context-free case, we note that any sufficiently large syntax tree will have a downward path that **visits the same non-terminal twice**. We can then 'pump in' extra copies of the relevant subtree and remain within the language:



## Context-free pumping lemma: continued

More precisely, suppose  $L$  has a CFG in CNF with  $m$  non-terminals.

Then take  $k$  so large that every syntax tree for a string of length  $\geq k$  contains a path of length  $> m + 1$ .

Such a path (even with the root node removed, which means the remaining path has length  $> m$ ) is guaranteed to visit the same nonterminal twice.

To show that a language  $L$  is **not** context free, we just need to prove that it satisfies the negation ( $\neg P$ ) of the property ( $P$ ):

$(\neg P)$  For every  $k \geq 0$ , there exists  $z \in L$  with  $|z| \geq k$  such that, for every decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ , there exists  $i \geq 0$  such that  $uv^iwx^iy \notin L$ .

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b^k c^k$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b^k c^k$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

Suppose we have a decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ .

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b^k c^k$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

Suppose we have a decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ .

Since  $|vwx| \leq k$ , the string  $vwx$  contains at most two different letters. So there must be some letter  $d \in \{a, b, c\}$  that does not occur in  $vwx$ .

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b^k c^k$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

Suppose we have a decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ .

Since  $|vwx| \leq k$ , the string  $vwx$  contains at most two different letters. So there must be some letter  $d \in \{a, b, c\}$  that does not occur in  $vwx$ .

But then  $uw \notin L$  because at least one character different from  $d$  now occurs  $< k$  times, whereas  $d$  still occurs  $k$  times.

## Standard example 1

The language  $L = \{a^n b^n c^n \mid n \geq 0\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b^k c^k$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

Suppose we have a decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ .

Since  $|vwx| \leq k$ , the string  $vwx$  contains at most two different letters. So there must be some letter  $d \in \{a, b, c\}$  that does not occur in  $vwx$ .

But then  $uw \notin L$  because at least one character different from  $d$  now occurs  $< k$  times, whereas  $d$  still occurs  $k$  times.

We have shown that  $(\neg P)$  holds with  $i = 0$ .

## Standard example 2

The language  $L = \{ss \mid s \in \{a, b\}^*\}$  isn't context-free!

We prove that  $(\neg P)$  holds for  $L$ :

Suppose  $k \geq 0$ .

We choose  $z = a^k b a^k b a^k b a^k b$ . Then indeed  $z \in L$  and  $|z| \geq k$ .

Suppose we have a decomposition  $z = uvwxy$  with  $|vx| \geq 1$  and  $|vwx| \leq k$ . Since  $|vwx| \leq k$ , the string  $vwx$  contains at most one  $b$ .

There are two main cases:

- $vx$  contains  $b$ , in which case  $uwy$  contains exactly 3  $b$ 's.
- Otherwise  $uwy$  has the form  $z = a^g b a^h b a^i b a^j b$  where either:
  - exactly two adjacent numbers from  $g, h, i, j$  are  $< k$  (this happens if  $w$  contains  $b$  and  $|v| \geq 1 \leq |x|$ ), or
  - exactly one of  $g, h, i, j$  is  $< k$  (this happens if  $w$  contains  $b$  and one of  $v, x$  is empty, or if  $vwx$  does not contain  $b$ ).

In each case, we have  $uwy \notin L$ . So  $(\neg P)$  holds with  $i = 0$ .

# Complementation

Consider the language  $L'$  defined by:

$$\{a, b\}^* - \{ss \mid s \in \{a, b\}^*\}$$

This **is** context free.

Idea: If  $t = t_1 \dots t_{2n} \in L'$ , there's some  $i \leq n$  such that  $t_i \neq t_{n+i}$ .

This means that  $t$  has the form  $waxybz$  or  $wbxyz$ , where  $|w| = |x|$  and  $|y| = |z|$ . Not hard to give a CFG that generates all such strings. (See Kozen p. 155).

The complement of  $L'$  is

$$\{a, b\}^* - L' = \{ss \mid s \in \{a, b\}^*\}$$

which, as we've seen, is *not* context-free.

So **context-free languages are not closed under complementation**.

## Context sensitive grammars

A **Context Sensitive Grammar** has productions of the form

$$\alpha X \gamma \rightarrow \alpha \beta \gamma$$

where  $X$  is a nonterminal, and  $\alpha, \beta, \gamma$  are sequences of terminals and nonterminals (i.e.,  $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ ) with the requirement that  $\beta$  is **nonempty**.

So the rules for expanding  $X$  can be **sensitive to the context** in which the  $X$  occurs (contrasts with **context free**).

**Minor wrinkle:** The nonempty restriction on  $\beta$  disallows rules with right-hand side  $\epsilon$ . To remedy this, we also permit the special rule

$$S \rightarrow \epsilon$$

where  $S$  is the start symbol, and with the restriction that this rule is only allowed to occur if the nonterminal  $S$  does not appear on the right-hand-side of any productions.

# Context sensitive languages

A language is **context sensitive** if it can be generated by a context sensitive grammar.

The non-context-free languages:

$$\begin{aligned} & \{a^n b^n c^n \mid n \geq 0\} \\ & \{ss \mid s \in \{a, b\}^*\} \end{aligned}$$

are both context sensitive.

In practice, it can be quite an effort to produce context sensitive grammars, according to the definition above.

It is often more convenient to work with a more liberal notion of grammar for generating context-sensitive languages.

## General and noncontracting grammars

In a **general** or **unrestricted grammar**, we allow productions of the form

$$\alpha \rightarrow \beta$$

where  $\alpha, \beta$  are sequences of terminals and nonterminals, i.e.,  $\alpha, \beta \in (N \cup \Sigma)^*$ , with  $\alpha$  containing at least one nonterminal.

In a **noncontracting grammar**, we restrict productions to the form

$$\alpha \rightarrow \beta$$

with  $\alpha, \beta$  as above, subject to the additional requirement that  $|\alpha| \leq |\beta|$  (i.e., the sequence  $\beta$  is at least as long as  $\alpha$ ).

In a noncontracting grammar also permit the special production

$$S \rightarrow \epsilon$$

where  $S$  is the start symbol, as long as  $S$  does not appear on the right-hand-side of any productions.

## Example noncontracting grammar

Consider the noncontracting grammar with start symbol  $S$ :

$$\begin{array}{lcl} S & \rightarrow & abc \\ S & \rightarrow & aSBc \\ cB & \rightarrow & Bc \\ bB & \rightarrow & bb \end{array}$$

Example derivation (underlining the sequence to be expanded):

$$\underline{S} \Rightarrow a\underline{SBc} \Rightarrow aabc\underline{Bc} \Rightarrow aab\underline{Bcc} \Rightarrow aabbcc$$

**Exercise:** Convince yourself that this grammar generates exactly the strings  $a^n b^n c^n$  where  $n > 0$ .

(N.B. With noncontracting grammars and CSGs, need to think in terms of **derivations**, not **syntax trees**.)

# Noncontracting = Context sensitive

**Theorem.** A language is context sensitive if and only if it can be generated by a noncontracting grammar.

That every context-sensitive language can be generated by a noncontracting grammar is immediate, since context-sensitive grammars are, by definition, noncontracting.

The proof that every noncontracting grammar can be turned into a context sensitive one is intricate, and beyond the scope of the course.

Sometimes (e.g., in Kozen) noncontracting grammars are called context sensitive grammars; but this terminology is not faithful to Chomsky's original definition.

# The Chomsky Hierarchy

At this point, we have a fairly complete understanding of the machinery associated with the different levels of the Chomsky hierarchy.

- **Regular languages:** DFAs, NFAs, regular expressions, regular grammars.
- **Context-free languages:** context-free grammars, nondeterministic pushdown automata.
- **Context-sensitive languages:** context-sensitive grammars, noncontracting grammars.
- **Recursively enumerable languages:** unrestricted grammars.

# Context-sensitivity in programming languages

Some aspects of typical programming languages can't be captured by context-free grammars, e.g.

- Typing rules
- Scoping rules (e.g. variables can only be used in contexts where they have been 'declared')
- Access constraints (e.g. use of public vs. private methods in Java).

The usual approach is to give a CFG that's a bit 'too generous', and then **separately** describe these additional rules.  
(E.g. typechecking done as a separate stage after parsing.)

In principle, though, all the above features fall within what can be captured by **context-sensitive** grammars. In fact, **no** programming language known to humankind contains anything that can't.

## Scoping constraints aren't context-free

Consider the simple language  $L_1$  given by

$$S \rightarrow \epsilon \mid \text{declare } v; S \mid \text{use } v; S$$

where  $v$  stands for a lexical class of variables. Let  $L_2$  be the language consisting of strings of  $L_1$  in which variables must be declared before use.

Assuming there are infinitely many possible variables, it can be shown that  $L_2$  is not context-free, but is context-sensitive.

(If there are just  $n$  possible variables, we could in theory give a CFG for  $L_2$  with around  $2^n$  nonterminals — but that's obviously silly...)

# Summary

- Context-sensitive languages are a big step up from context-free languages in terms of their power and generality.
- Natural languages have features that can't be captured conveniently (or at all) by context-free grammars. However, it appears that NLs are only **mildly context-sensitive** — they only exploit the low end of the power offered by CSGs.
- Programming languages contain non-context-free features (**typing**, **scoping** etc.), but all these fall comfortably within the realm of context-sensitive languages.
- **Next time:** what kinds of **machines** are needed to recognize context-sensitive languages?

# Turing machines and linear bounded automata

Informatics 2A: Lecture 29

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

27 November 2015

# The Chomsky hierarchy: summary

Level	Language type	Grammars	Accepting machines
3	Regular	$X \rightarrow \epsilon$ , $X \rightarrow Y$ , $X \rightarrow aY$ (regular)	NFAs (or DFAs)
2	Context-free	$X \rightarrow \beta$ (context-free)	NPDAs
1	Context-sensitive	$\alpha \rightarrow \beta$ with $ \alpha  \leq  \beta $ (noncontracting)	Nondet. linear bounded automata
0	Recursively enumerable	$\alpha \rightarrow \beta$ (unrestricted)	Turing machines

The material in red will be introduced today.

# The length restriction in noncontracting grammars

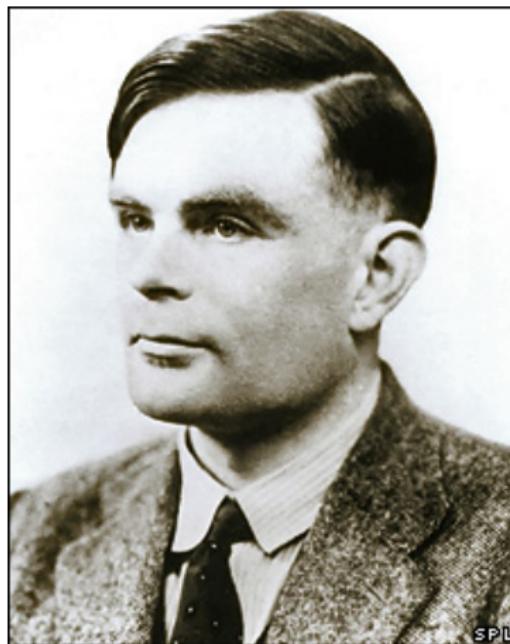
What's the effect of the restriction  $|\alpha| \leq |\beta|$  in noncontracting grammar rules?

**Idea:** in a noncontracting derivation  $S \Rightarrow \dots \Rightarrow \dots \Rightarrow s$  of a nonempty string  $s$ , all the sentential forms are of length at most  $|s|$ .

This means that if  $L$  is context-sensitive, and we're trying to decide whether  $s \in L$ , we only need to consider possible sentential forms of length  $\leq |s|$ . So intuitively, we have the problem **under control**, at least in principle.

By contrast, without the length restriction, there's no upper limit on the length of intermediate forms that might appear in a derivation of  $s$ . So if we're searching for a derivation for  $s$ , how do we know when to stop looking? Intuitively, the problem here is **wild** and **out of control**. (This will be made more precise next lecture.)

# Alan Turing (1912–1954)



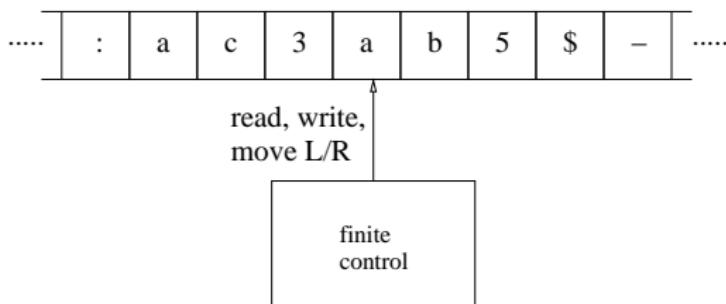
SPL

# Turing machines

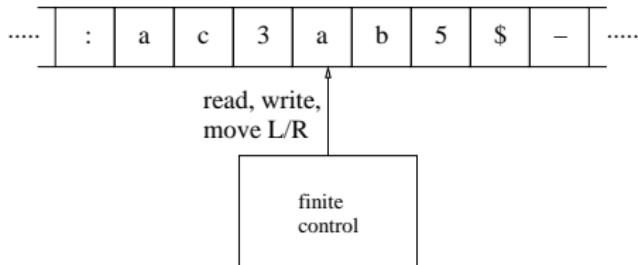
Recall that NFAs are ‘essentially memoryless’, whilst NPDAs are equipped with memory in the form of a stack.

To find the right kinds of machines for the top two Chomsky levels, we need to allow more general manipulation of memory.

A **Turing machine** consists of a **finite-state control unit**, equipped with a **memory tape**, infinite in both directions. Each cell on the tape contains a symbol drawn from a **finite alphabet**  $\Gamma$ .



# Turing machines, continued



At each step, the behaviour of the machine can depend on

- the current state of the control unit,
- the tape symbol at the current read position.

Depending on these things, the machine may then

- overwrite the current tape symbol with a new symbol,
- shift the tape left or right by one cell,
- jump to a new control state.

This happens repeatedly until (if ever) the control unit enters some identified **final state**.

# Turing machines, formally

A **Turing machine**  $T$  consists of:

- A set  $Q$  of **control states**
- An **initial state**  $i \in Q$
- A **final (accepting) state**  $f \in Q$
- A **tape alphabet**  $\Gamma$
- An **input alphabet**  $\Sigma \subseteq \Gamma$
- A **blank symbol**  $- \in \Gamma - \Sigma$
- A **transition function**  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ .

A **nondeterministic Turing machine** replaces the transition function  $\delta$  with a **transition relation**  $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ .

(Numerous variant definitions of Turing machine are possible. All lead to notions of TM of equivalent power.)

# Turing machines as acceptors

To use a Turing machine  $T$  as an acceptor for a language over  $\Sigma$ , assume  $\Sigma \subseteq \Gamma$ , and set up the tape with the test string  $s \in \Sigma^*$  written left-to-right starting at the read position, and with blank symbols everywhere else.

Then let the machine run (maybe overwriting  $s$ ), and if it enters the final state, declare that the original string  $s$  is accepted.

The language accepted by  $T$  (written  $\mathcal{L}(T)$ ) consists of all strings  $s$  that are accepted in this way.

**Theorem:** A set  $L \subseteq \Sigma^*$  is generated by some unrestricted (Type 0) grammar if and only if  $L = \mathcal{L}(T)$  for some Turing machine  $T$ .

So both Type 0 grammars and Turing machines lead to the same class of recursively enumerable languages.

# Questions

**Q1.** Which is the most powerful form of language acceptor (i.e., accepts the widest class of languages)?

- ➊ DFAs
- ➋ NPDAs
- ➌ Turing machines
- ➍ Your laptop

# Questions

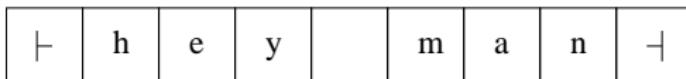
**Q1.** Which is the most powerful form of language acceptor (i.e., accepts the widest class of languages)?

**Q2.** Which is the least powerful form of language acceptor (i.e., accepts the narrowest class of languages)?

- ➊ DFAs
- ➋ NPDAs
- ➌ Turing machines
- ➍ Your laptop

## Linear bounded automata

Suppose we modify our model to allow just a **finite** tape, initially containing just the test string  $s$  with **endmarkers** on either side:



The machine therefore has just a **finite** amount of memory, determined by the length of the input string. We call this a **linear bounded automaton**.

(LBAs are sometimes defined as having tape length bounded by a **constant multiple** of length of input string. In fact, this doesn't make any difference.)

**Theorem:** A language  $L \subseteq \Sigma^*$  is context-sensitive if and only if  $L = \mathcal{L}(T)$  for some **non-deterministic** linear bounded automaton  $T$ .

**Rough idea:** we can guess at a derivation for  $s$ . We can check each step since each sentential form fits within the tape.

## Example: a non-context-sensitive language

Sentences of **first-order predicate logic** can be regarded as strings over a certain alphabet, with symbols like  $\forall, \wedge, x, =, (, )$  etc.

Let  $P$  be the language consisting of all such sentences that are **provable** using the rules of FOPL. E.g. the sentence

$$(\forall x. A(x)) \wedge (\forall x. B(x)) \Rightarrow (\forall x. A(x) \wedge B(x))$$

is in  $P$ . But the sentence

$$(\exists x. A(x)) \wedge (\exists x. B(x)) \Rightarrow (\exists x. A(x) \wedge B(x))$$

is not in  $P$ , even though it's syntactically well-formed.

**Theorem:**  $P$  is recursively enumerable, but not context-sensitive.

Intuition: to show  $s \in P$ , we'd in effect have to construct a **proof** of  $s$ . But in general, a proof of  $s$  might involve formulae much, much longer than  $s$  itself, which wouldn't fit on the LBA tape. Put another way, **mathematics itself** is wild and out of control!

## Determinism vs. non-determinism: a curiosity

- At the bottom level of the Chomsky hierarchy, it makes no difference: every NFA can be simulated by a DFA.
- At the top level, the same happens. Any nondeterministic Turing machine can be simulated by a deterministic one.
- At the context-free level, there **is** a difference: we need NPDAs to account for all context-free languages.  
*(Example:  $\Sigma^* - \{ss \mid s \in \Sigma^*\}$  is a context-free language whose complement isn't context-free, see last lecture. However, if  $L$  is accepted by a DPDA then so is its complement.)*
- What about the context-sensitive level? Are NLBAs strictly more powerful than DLBAs? Asked in 1964, and **still open!!**  
*(Can't use the context-free argument because CSLs are closed under complementation — shown in 1988.)*

## Detecting non-acceptance: LBAs versus TMs

Suppose  $T$  is an LBA. How might we detect that  $s$  is **not** in  $\mathcal{L}(T)$ ?

Clearly, if there's an accepting computation for  $s$ , there's one that doesn't pass through exactly the same machine configuration twice (if it did, we could shorten it).

Since the tape is finite, the total number of machine configurations is finite (though large). So in theory, if  $T$  runs for long enough without reaching the final state, it will enter the same configuration twice, and we may as well abort.

Note that on this view, repeated configurations would be spotted not by  $T$  itself, but by 'us watching', or perhaps by some super-machine spying on  $T$ .

For Turing machines with unlimited tape space, this reasoning doesn't work. **Is there some general way of spotting that a computation isn't going to terminate ??** See next lecture ...

# Wider significance of Turing machines

Turing machines are important because (it's generally believed that) any symbolic computation that can be done by any **mechanical procedure or algorithm** can in principle be done by a Turing machine. This is called the **Church-Turing Thesis**.

E.g.:

- Any language  $L \subseteq \Sigma^*$  that can be 'recognized' by some mechanical procedure can be recognized by a TM.
- Any mathematical function  $f : \mathbb{N} \rightarrow \mathbb{N}$  that can be computed by a mechanical procedure can be computed by a TM (e.g. representing integers in binary, and requiring the TM to write the result onto the tape.)

## Status of Church-Turing Thesis

The CT Thesis is a somewhat informal statement insofar as the general notion of a **mechanical procedure** isn't formally defined (although we have a pretty good idea of what we mean by it).

Although a certain amount of philosophical hair-splitting is possible, the broad idea behind CTT is generally accepted.

At any rate, anything that can be done on any present-day computer (even disregarding time/memory limitations) can in principle be done on a TM.

So if we buy into CTT, theorems about what TMs can/can't do can be interpreted as fundamental statements about what can/can't be accomplished by **mechanical computation** in general.

We'll see some examples of such theorems next time.

# Undecidability

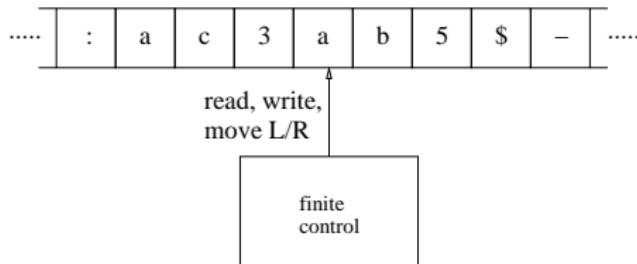
Informatics 2A: Lecture 30

John Longley

School of Informatics  
University of Edinburgh  
[jrl@inf.ed.ac.uk](mailto:jrl@inf.ed.ac.uk)

1 December 2015

## Recap: Turing machines



- If  $|\Sigma| \geq 2$ , any kind of 'finite data' can be coded up as a string in  $\Sigma^*$ , which can then be written onto a Turing machine tape. (E.g. natural numbers could be written in binary.)
- According to the **Church-Turing thesis (CTT)**, any '**mechanical computation**' that can be performed on finite data can be performed in principle by a Turing machine.
- Any decent programming language (and even Micro-Haskell!) has the same computational power in principle as a Turing machine.

# Universal Turing machines

Consider any Turing machine with input alphabet  $\Sigma$ .

Such a machine  $T$  is itself specified by a **finite amount of information**, so can in principle be 'coded up' by a string  $\overline{T} \in \Sigma^*$ . (Details don't matter).

So one can imagine a **universal Turing machine**  $U$  which:

- Takes as its input a coded description  $\overline{T}$  of some TM  $T$ , along with an input string  $s$ , separated by a blank symbol.
- **Simulates** the behaviour of  $T$  on the input string  $s$ .  
(N.B. a single step of  $T$  may require many steps of  $U$ ).
  - If  $T$  ever halts (i.e. enters final state),  $U$  will halt.
  - If  $T$  runs forever,  $U$  will run forever.

If we believe CTT, such a  $U$  must exist — but in any case, it's possible to construct one explicitly.

## The concept of a general-purpose computer

Alan Turing's discovery of the existence of a **universal** Turing machine (1936) was in some sense the fundamental insight that gave us the general-purpose (programmable) computer.

In most areas of life, we have different machines for different jobs. So it's quite remarkable that a **single** physical machine can be persuaded to perform as many different tasks as a computer can . . . just by feeding it with a cunning sequence of 0's and 1's!

# The halting problem

The universal machine  $U$  in effect serves as a recognizer for the set

$$\{\overline{T} \_ s \mid T \text{ halts on input } s\}$$

But is there also a machine  $V$  that recognizes the set

$$\{\overline{T} \_ s \mid T \text{ doesn't halt on input } s\} ?$$

If there were, then given any  $T$  and  $s$ , we could run  $U$  and  $V$  in parallel, and we'd eventually get an answer to the question "does  $T$  halt on input  $s$ ?"

Conversely, if there were a machine that answered this question, we could construct a machine  $V$  with the above property.

**Theorem: There is no such Turing machine  $V$ !**

In other words, the halting problem is **undecidable**.

# Proof of undecidability

Why is the halting problem undecidable?

Suppose  $V$  existed. Then we could easily make a Turing machine  $W$  that recognised the set  $L$  defined by:

$$L = \{s \in \Sigma^* \mid \text{the TM coded by } s \text{ runs forever on the input } s\}$$

( $W$  could just write two copies of its input string  $s$ , separated by a blank, and thereafter behave as  $V$ .)

Now consider what  $W$  does when given the string  $\overline{W}$  as input.  
That is, the input to  $W$  is the string that encodes  $W$  itself.

- $W$  accepts  $\overline{W}$  iff  $W$  runs forever on  $\overline{W}$  (since  $W$  recognises  $L$ )
- but  $W$  accepts  $\overline{W}$  iff  $W$  halts on  $\overline{W}$  (definition of acceptance)

**Contradiction!!!** So  $V$  can't exist after all!

## Precursor: Russell's paradox (1901)

Define  $R$  to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does  $R$  contain itself, i.e. is  $R \in R$ ?

**Russell's analogy:** The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

## Precursor: Russell's paradox (1901)

Define  $R$  to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does  $R$  contain itself, i.e. is  $R \in R$ ?

**Conclusion:** no such set  $R$  exists.

**Russell's analogy:** The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

## Precursor: Russell's paradox (1901)

Define  $R$  to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does  $R$  contain itself, i.e. is  $R \in R$ ?

**Conclusion:** no such set  $R$  exists.

**Russell's analogy:** The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

**Conclusion:** no man exists in the village with the property identified by Russell.

## Decidable vs. semidecidable sets

In general, a set  $S$  (e.g.  $\subseteq \Sigma^*$ ) is called **decidable** if there's a mechanical procedure which, given  $s \in \Sigma^*$ , will always return a yes/no answer to the question “Is  $s \in S$ ?”.

E.g. the set  $\{s \mid s \text{ represents a prime number}\}$  is decidable.

We say  $S$  is **semidecidable** if there's a mechanical procedure which will return ‘yes’ precisely when  $s \in S$  (it isn't obliged to return anything if  $s \notin S$ ).

Semidecidable sets coincide with **recursively enumerable** (i.e., Type 0) **languages** as defined in lectures 28–9

The **halting set**  $\{\overline{T}_s \mid T \text{ halts on input } s\}$  is an example a semidecidable set that isn't decidable. So there exist Type 0 languages for which membership is undecidable.

## Separating Type 0 and Type 1

Every Type 1 (context-sensitive) language is decidable.  
(The argument was outlined in Lecture 29.)

As we have seen, the halting set

$$\{\overline{T} \_ s \mid T \text{ halts on input } s\}$$

is an undecidable Type 0 language.

So the halting set is an example of a Type 0 language that is not a Type 1 language.

(Last lecture, we saw another example: the set of provable sentences of FOPL. This too is an undecidable Type 0 language.)

## Undecidable problems in mathematics

The existence of ‘mechanically unsolvable’ mathematical problems was in itself a major breakthrough in mathematical logic: until about 1930, some people (the mathematician [David Hilbert](#) in particular) hoped there might be a single **killer algorithm** that could solve all mathematical problems!

Once we have **one** example of an unsolvable problem (the halting problem), we can use it to obtain others — typically by showing “the halting problem can be **reduced** to problem X.” (If we had a mechanical procedure for solving X, we could use it to solve the halting problem.)

## Example: Provability of theorems

Let  $M$  be some reasonable (consistent) **formal logical system** for proving mathematical theorems (something like **Peano arithmetic** or **Zermelo-Fraenkel set theory**).

**Theorem:** The set of theorems provable in  $M$  is **semidecidable** (and hence is a Type 0 language), but not **decidable**.

**Proof:** Any reasonable system  $M$  will be able to prove all true statements of the form “ $T$  halts on input  $s$ ”. So if we could decide  $M$ -provability, we could solve the halting problem.

**Corollary (Gödel):** However strong  $M$  is, there are mathematical statements  $P$  such that neither  $P$  nor  $\neg P$  is provable in  $M$ .

**Proof:** Otherwise, given any  $P$  we could search through all possible  $M$ -proofs until either a proof of  $P$  or of  $\neg P$  showed up. This would give us an algorithm for deciding  $M$ -provability.

## Example: Diophantine equations

Suppose we're given a set of simultaneous equations involving polynomials in several variables with integer coefficients. E.g.

$$\begin{aligned}3xy + 4z + 5wx^2 &= 27 \\x^2 + y^3 - 9z &= 4 \\w^5 - z^4 &= 31 \\x^2 + y^2 + z^2 - w^2 &= 2536427\end{aligned}$$

**Hilbert's 10th Problem (1900):** Is there a mechanical procedure for determining whether a set of polynomial equations has an integer solution?

**Matiyasevich's Theorem (1970):** It is **undecidable** whether a given set of polynomial equations has an integer solution.

(By contrast, it's **decidable** whether there's a solution in real numbers!)

## BONUS TOPIC: Higher-Order Computability

In one sense, all reasonable prog. langs are **equally powerful**. E.g.

- They can compute the same class of functions  $\mathbb{Z} \rightarrow \mathbb{Z}$ . (In Micro-Haskell, these have type Integer->Integer).
- Any language can be implemented in any other. (E.g. you've implemented MH in Java.)

Indeed, there's only one reasonable mathematical class of 'computable' functions  $\mathbb{Z} \rightarrow \mathbb{Z}$  (the **Turing-computable** functions).

But what about **higher-order** functions, e.g. of type  $((\text{Integer}-\>\text{Integer})-\>\text{Integer})-\>\text{Integer}$  ?

- What does it mean for a function of this kind to be 'computable'?
- Are all reasonable languages 'equally powerful' when it comes to higher-order functions?

## Recursions at various types

One approach to this question focuses on **recursive definitions**.  
(Micro-)Haskell is a convenient language for discussing this.

For any recursive function definition in MH, we can consider the **type** of the entity being defined by recursion.

E.g. Here's a recursion at type Integer->Integer->Integer:

```
add :: Integer -> Integer -> Integer
add x y = if y==0 then x else add (suc x)(pre y)
```

In this case, the same function can be defined by a recursion at type Integer->Integer:

```
add' :: Integer -> Integer -> Integer
add' x y = if y==0 then x else suc (add' x (pre y))
```

## Recursions at higher types

Now here's a recursion at a second-order type:

```
exp2comp :: (Integer->Integer) -> Integer ->
            (Integer->Integer)
exp2comp f n = if n==0 then f
               else exp2comp (\x -> f(f x)) (n-1)
```

In this case, we can achieve the same effect with

```
exp2comp' f n x = iter f (exp2 n) x
```

using only recursions at type `Integer->Integer`.

We may say that `exp2comp` and `exp2comp'` are **equivalent** in that `exp2comp M N P`, `exp2comp' M N P` yield the same value (or lack of one) for any closed programs `M, N, P` of suitable type.

**Question:** Can all programs of Micro-Haskell be rewritten to equivalent ones that uses only first-order recursions?

## New result (May 2015, JL)

Let's write  $MH_k$  for the sublanguage of  $MH$  where we only allow recursions at types of order  $\leq k$ . So  $MH_1 \subseteq MH_2 \subseteq \dots \subseteq MH$ .

All of these languages are Turing-complete, i.e. they yield the same computable functions of type Integer->Integer. But they differ in the **higher-order** functions that they can compute:

**Theorem:** For each  $k$ , there are higher-order functions computable in  $MH_{k+1}$  but not in  $MH_k$ .

This is one of a bunch of results showing that the selection of features available in a programming language (e.g. recursion, exceptions, local state, concurrency, . . . ) makes a non-trivial difference to its 'expressive power'.

(The above theorem **just** made it into my book!)

That's all folks!

That concludes the course syllabus.

On Thursday, Shay and I will present a joint [revision lecture](#), in which we shall discuss:

- the exam structure
- examinable material
- pointers to UG3 (and upwards) Informatics courses that continue from this one

# **Informatics 2A 2015–16**

## Lecture 31

### Revision Lecture

John Longley

Shay Cohen

## **Reminder: pass criteria**

By 4pm tomorrow, you will have completed your coursework.  
This accounts for 25% of the course mark.

The remaining 75% of the course mark is provided by the exam.

For a **pass** in Inf2A, you need all of the following:

- At least 40% combined total mark.
- At least 35% in the exam.
- At least 25% on the assessed coursework.

## The 2015 Inf2A Exam

December exam time and location:

INFR08008 - Informatics 2A

Location: Patersons Land - G1 (Surname Ahari - Mikolajczak), Patersons Land - 1.18 (Surname Milloy - Roy), Patersons Land - 1.26 (Surname Saev - Vosloo), Patersons Land - G.21 (Surname Wallbridge - Zimmerman)

Date/Time: Saturday 12/12/2015, 09:30:00-11:30:00

This is copied from the Registry exam timetable

<http://www.scripts.sasg.ed.ac.uk/registry/examinations/>

which is the **official** exam timetable. Make sure that you use this link to double-check **all** your exam times (including Inf2A).

A resit exam will be held in **August 2016**.

## Exam structure

The exam is pen-and-paper, and lasts **2 hours**.

**Calculators** may be used, but you must bring your own. It must be one from an approved list of models specified by College:

[http://www.cltc.scieng.ed.ac.uk/docs/open/Paper\\_D.pdf](http://www.cltc.scieng.ed.ac.uk/docs/open/Paper_D.pdf)

The exam consists of:

- Part A: 5 compulsory short questions, worth 10% each.  
Guideline time per question: 10 minutes
- Part B: a choice of 2 out of 3 longer questions, worth 25% each.  
Guideline time per question: 30 minutes

The guideline times allow 10 minutes for reading and familiarising yourself with the exam paper.

## **Part A questions**

The 5 compulsory short questions were new in 2012 and replaced 20 multiple-choice questions in previous years.

The questions will be similar in style and length (but not necessarily in topic) to the questions on this week's Tutorial 9.

The multiple-choice questions of previous years still provide good revision material in terms of coverage of topics.

## **Revision office hours**

John Longley (IF 4.11): Wed 9 Dec, 10–12 am

Shay Cohen (IF 4.26): Thurs 10 Dec, 11 am–1 pm

Examinable material

## **Examinable material: formal language thread**

### Lectures 3–12

All of the material on regular and context-free languages (Lectures 3–12) should be considered as examinable.

### Lectures 13, 27

The details of Micro-Haskell (MH) (syntax, type-checking and semantics), covered in Lecture 13, are **not** examinable.

However, the general principles of types, type-checking and abstract syntax, from Lecture 13, **are** examinable.

The whole of Lecture 27 (semantics of programming languages, in particular MH) may be considered **non-examinable**.

## Examinable material: formal language thread (continued)

### Lectures 28–30

**Examinable:** Most of Lecture 28: Idea of context-free pumping lemma. The notions of context-sensitive, noncontracting and unrestricted grammar. Definitions of Chomsky levels 0 and 1. Standard examples of context-sensitive but not context-free languages (though not the proofs of these facts).

**Examinable:** Broad ideas from Lectures 29 and 30. Correspondence between linear-bounded automata and Type 1, and between Turing Machines and Type 0. Example of a Type 0 language that is not Type 1. General notions of decidable, semidecidable and undecidable problems.

**Non-examinable:** Technical details from lectures 29, 30. E.g. Detailed definitions of Turing machines and linearly bounded automata, proof of undecidability of halting problem, examples of undecidable problems in general mathematics.

## Kinds of exam question: formal language thread

Broadly speaking, 2 styles of question in exam.

**Algorithmic problems:** Minimizing a DFA, converting NFA to DFA, executing a PDA, LL(1) parsing using parse table, generating parse table from LL(1) grammar, . . .

When the algorithm is **complex** (e.g., minimization, calculating first and follow sets), it may be easier to work directly with the definitions rather than following the algorithm strictly.

**Non-algorithmic problems:** Converting DFA to regular expression, designing regular expression patterns, applying pumping lemma, designing CFGs, converting CFG to LL(1), parsing using CSG or noncontracting grammar, . . .

## **Examinable material: natural language thread**

The main thing being tested is your ability to apply *and understand* the methods for solving certain standard kinds of problems.

### **Algorithmic problems:**

- POS tagging via bigrams or Viterbi algorithm (lecture 16).
- CYK and Earley parsing (lectures 18, 19).
- Tree probabilities; probabilistic CYK; inferring probabilities from a corpus; lexicalization of rules (lectures 20, 21).
- Computing semantics, including  $\beta$ -reduction (lecture 24).

## **Examinable material: natural language thread (continued)**

### **Non-algorithmic problems (simple examples only!)**

- Design of a transducer for some morphology parsing task (lecture 14).
- Design of context-free rules for some feature of English. (Includes parameterized rules for agreement — lecture 22.)
- Adding semantic clauses to a given context-free grammar (lectures 23, 24).
- Converting an English sentence to a formula of FOPL (lecture 23).

## Examinable material: natural language thread (continued)

### General topics

- The language processing pipeline (lecture 2).
- Kinds of ambiguity (lectures 2, 15, 17, 24).
- The Chomsky hierarchy, and where human languages sit (lectures 2, 25).
- The *general idea* of parts of speech (lecture 16).
- Word distribution and Zipf's law (lecture 16).
- Very basic Python.

The ideas of recursive descent and shift-reduce parsing (lecture 17) are only **weakly examinable**.

## **Non-examinable material: natural language thread**

- Specific knowledge of linguistics (everything you need will be given in the question).
- Details of particular POS tagsets; ability to do POS tagging by hand (lecture 15).
- Fine-grained typing, e.g. selectional restrictions on verbs (lecture 22).
- Linear indexed grammars (lecture 25).
- Human parsing (lecture 26)

All natural language examples will be taken from English!

# Follow-on Informatics courses

## Compiling techniques (UG3)

Covers the entire language-processing pipeline for programming languages, aiming at effective **compilation**: translating code in a high-level source language (Java, C, Haskell, ...) to equivalent code in a low-level target language (machine code, bytecode)

Syllabus includes lexing and parsing from a more practical perspective than in Inf2A.

Majority of course focused on latter stages of language-processing pipeline. Converting lexed and parsed source-language code into equivalent target-language code.

## Introduction to theoretical computer science (UG3)

This will look at models of computation (register machines, Turing machines, lambda-calculus) and their different influences on computing practice.

One thread will address the boundaries between what is not computable at all (**undecidable** problems), what is computable in principle (**decidable** problems), and what is computable in practice (**tractable** problems). A major goal is to understand the famous **P = NP** question.

Another thread will look at the influence **lambda-calculus** has had, as a model of computation, on programming language design and practice, including LISP, OCaml, Haskell and Java.

## Natural Languages: what we've done, what we haven't.

NLs are endlessly complex and fascinating. In this course, we have barely scratched the surface.

There's a world of difference between doing NLP with small [toy grammars](#) (as in this course) and [wide-coverage](#) grammars intended to cope with real-world speech/text.

- Ambiguity is the norm rather than the exception.
- Empirical and statistical techniques (involving text corpora) come to the fore, as distinct from logical and symbolic ones.

Coping with the richness and complexity of real-world language is still a largely unsolved problem!

## Discourse structure.

In this course, we haven't considered any structure above the level of sentences. In practice, higher level discourse structure is crucial. E.g.

The Tin Man went to the Emerald City to see the Wizard of Oz and ask for a heart. Then **he** waited to see whether **he** would give **it** to **him**.

Or compare:

- John hid Bill's car keys. He was drunk.
- John hid Bill's car keys. He likes spinach. (??)

## Deep vs. shallow processing.

Roughly, the further we go along the NLP pipeline, the deeper our analysis.

- Many apparently ‘shallow’ NLP tasks (e.g. spell checking; speech transcription) can benefit from the use of ‘deeper’ techniques such as parsing.
- On the other hand, for many seemingly ‘deep’ tasks (e.g. machine translation), current state-of-the-art techniques are surprisingly ‘shallow’ (e.g. use of N-gram techniques with massive corpora).

## Follow-on courses in NLP

- **Foundations of Natural Language Processing** [UG3]. Empirical rather than theoretical in focus. Material on text corpora, N-grams, the ‘noisy channel’ model. A bit on the discourse level.
- **Machine Translation** [UG4]. Mainly on shallow techniques for MT: e.g. phrase-based models. Find out how Google Translate works!
- **Natural Language Understanding** [UG4]. Considers the LP pipeline much in the spirit of Inf2a, but including discourse level. Surveys both deep and shallow approaches.
- **Topics in Natural Language Processing** [UG4]. Get acquainted with state of the art in NLP and read cutting-edge research papers in NLP and machine learning.

# Thank you!!

Hope you've enjoyed Inf2A,  
and good luck with the exam!

Please complete the online course questionnaire when it  
becomes available.

# An Earley Parsing Example

Shay Cohen

Inf2a

November 4, 2015

The sentence we try to parse:

*“book that flight”*

Whenever we denote a span of words by  $[i,j]$ , it means it spans word  $i+1$  through  $j$ , because  $i$  and  $j$  index, between 0 and 3, the *spaces* between the words:

*0 book 1 that 2 flight 3*

## Grammar rules:

$S \rightarrow NP\ VP$

$S \rightarrow Aux\ NP\ VP$

$S \rightarrow VP$

$NP \rightarrow Pronoun$

$NP \rightarrow Proper-Noun$

$NP \rightarrow Det\ Nominal$

$Nominal \rightarrow Noun$

$Nominal \rightarrow Nominal\ Noun$

$Nominal \rightarrow Nominal\ PP$

$VP \rightarrow Verb$

$VP \rightarrow Verb\ NP$

$VP \rightarrow Verb\ NP\ PP$

$VP \rightarrow Verb\ PP$

$VP \rightarrow VP\ PP$

$PP \rightarrow Prep\ NP$

$Verb \rightarrow book \mid include \mid prefer$

$Noun \rightarrow book \mid flight \mid meal$

$Det \rightarrow that \mid this \mid these$

Start with Prediction for the S node:

$S \rightarrow . \text{NP VP} [0,0]$

$S \rightarrow . \text{Aux NP VP} [0,0]$

$S \rightarrow . \text{VP} [0,0]$

All of these elements are created because we just started parsing the sentence, and we expect an S to dominate the whole sentence

$\text{NP} \rightarrow . \text{Pronoun} [0,0]$

$\text{NP} \rightarrow . \text{Proper-Noun} [0,0]$

$\text{NP} \rightarrow . \text{Det Nominal} [0,0]$

$\text{VP} \rightarrow . \text{Verb} [0,0]$

$\text{VP} \rightarrow . \text{Verb NP} [0,0]$

$\text{VP} \rightarrow . \text{Verb NP PP} [0,0]$

$\text{VP} \rightarrow . \text{Verb PP} [0,0]$

$\text{VP} \rightarrow . \text{VP PP}$

Now we can apply PREDICTOR on the above S nodes! Note that PREDICTOR creates endpoints  $[i,j]$  such that  $i=j$  and  $i$  and  $j$  are the right-end points of the state from which the prediction was made

NOTE: For a PREDICTOR item, the dot is always in the beginning!

In the previous slide we had states of the following form:

$\text{VP} \rightarrow . \text{Verb NP} [0,0]$

$\text{VP} \rightarrow . \text{Verb NP PP} [0,0]$

$\text{VP} \rightarrow . \text{Verb PP} [0,0]$

Note that we now have a dot before a terminal.

We look at the right number of  $[i,j]$ , and we see that it is 0, so we will try to match the first word in the sentence being a verb. This is the job of the Scanner operation.

CHECK! We have a rule  $\text{Verb} \rightarrow \text{book}$ , so therefore, we can advance the dot for the above Verb rules and get the following new states:

$\text{VP} \rightarrow \text{Verb} . \text{NP} [0,1]$

$\text{VP} \rightarrow \text{Verb} . \text{NP PP} [0,1]$

$\text{VP} \rightarrow \text{Verb} . \text{PP} [0,1]$

Great. What does that mean now?

We can call PREDICTOR again, we have new nonterminals with a dot before them!

In the previous slide we had states of the following form:

VP -> Verb . NP [0,1]

VP → Verb . NP PP [0,1]

VP → Verb . PP [0,1]

We said we can now run PREDICTOR on them. What will this create?

For NP:

NP → . Pronoun [1,1]

NP → . Proper-Noun [1,1]

NP → . Det Nominal [1,1]

Note that now we are expecting a NP at position 1!

And also for PP:

PP → . Prep Nominal [1,1]

In the previous slide we created the following states:

$NP \rightarrow . \text{ Pronoun } [1,1]$

$NP \rightarrow . \text{ Proper-Noun } [1,1]$

$NP \rightarrow . \text{ Det Nominal } [1,1]$

$PP \rightarrow . \text{ Prep Nominal } [1,1]$

Now we have an opportunity to run SCANNER again on the second word in the sentence!  
Question: for which item above would we do that?

We would do that for  $NP \rightarrow . \text{ Det Nominal } [1,1]$ . “*that*” can only be a Det. So now we create a new item:

$NP \rightarrow \text{ Det } . \text{ Nominal } [1,2]$

Note that now  $[i,j]$  is such that it spans the second word (1 and 2 are the “indexed spaces” between the words before and after the second words)

In the previous slide, we added the state:  $NP \rightarrow Det . Nominal [1,2]$

Now PREDICTOR can kick in again, because Nominal is a nonterminal in a newly generated item in the chart.

What will PREDICTOR create? (Hint: PREDICTOR takes an item and adds new rules for all rules that have LHS like the nonterminal that appears after the dot.)

These are the new states that PREDICTOR will generate:

$Nominal \rightarrow . Noun [2,2]$

$Nominal \rightarrow .Nominal Noun [2,2]$

$Nominal \rightarrow .Nominal PP [2,2]$

Note that again, predictor always starts with  $i=j$  for the  $[i,j]$  spans.

Its interpretation is: there might be a Nominal nonterminal spanning a substring in the sentence that starts with the third word.

In the previous slide, we created an element of the form:

Nominal → . Noun [2,2]

Its interpretation is: “there might be a use of the rule Nominal → Noun, starting at the third word. To see if you can use it, you need to first check whether there is a Noun at the third position in the sentence.” We do! So SCANNER can kick in.

What will we get?

We now scanned the Noun, because we the word “flight” can be a Noun.

Nominal → Noun . [2,3]

That's nice, now we have a complete item. Can COMPLETER kick now into action?

We have to look for all items that we created so far that are expecting a Nominal starting at the second position.

In the previous slide, we created Nominal → Noun . [2,3] which is a complete item. Now we need to see whether we can apply completer on it.

Remember we created this previously?

NP → Det . Nominal [1,2]

Now we can apply COMPLETER on it in conjunction with Nominal → Noun . [2,3] and get:

NP → Det Nominal . [1,3]

Nice! This means we completed another item, and it means that we can create an NP that spans the second and the third word ("that flight") – that's indeed true if you take a look at the grammar.

In any case, now that we have completed an item, we need to see if we can complete other ones. The question we ask: is there any item that expects an NP (i.e. the dot appears before an NP) and the right-hand side of [i,j] is 1?

We actually had a couple of those:

$\text{VP} \rightarrow \text{Verb . NP } [0,1]$

$\text{VP} \rightarrow \text{Verb . NP PP } [0,1]$

They are waiting for an NP starting at the second word.

So we can use COMPLETER on them with the item  $\text{NP} \rightarrow \text{Det Nominal } [1,3]$  that we created in the previous slide.

So now we will have new items:

$\text{VP} \rightarrow \text{Verb NP . } [0,3]$

$\text{VP} \rightarrow \text{Verb NP . PP } [0,3]$

The first one is also a complete one! So maybe we can apply COMPLETE again?  
We need an item that expects a VP at position 0.

Let's try to remember if we had one of those...

We had one indeed:

$S \rightarrow . VP [0,0]$

That was one of the first few items we created, which is a good sign, it means we are creating now items that span the tree closer to the top node.

So now we can COMPLETE this node with the item  $VP \rightarrow Verb\ NP . [0,3]$  that we created in the previous slide.

What do we get?

We get the item:

$S \rightarrow VP . [0, 3]$

and that means we managed to create a full parse tree, we have an S that spans all words in the sentence.

How do we get a parse tree out of this? Back pointers...

Let's consider the “back-pointers” we created.

We created the node  $S \rightarrow VP . [0, 3]$  as a result of a COMPLETER on the item  $VP \rightarrow Verb\ NP . [0, 3]$ .

We created  $VP \rightarrow Verb\ NP . [0, 3]$  as a result of a COMPLETER on  $VP \rightarrow Verb . NP [0, 1]$  when we had an  $NP \rightarrow Det\ Nominal [1, 2]$ .

That means the tree has to look like:

