COMP5046: Parsing with Grammars

Joel Nothman

joel.nothman@sydney.edu.au

School of Information Technologies
University of Sydney

2018-05-15

Parsing: determining nested sentence structure

1. Structure
   - Why?
   - Applications
   - Grammars and formal languages
   - Context Free Grammars (CFGs)
   - Parsing algorithms
2. Statistics
   - Probabilistic Context Free Grammars (PCFGs)
   - Improving PCFG parsing
   - Dependency grammar
   - Dependency parsing
3. Semantics
   - Semantic Role Labelling

Part I

Introduction to Syntax, Ambiguity and Parsing

Why bother parsing?

- Why is John ate the apple
- different to The apple ate John
- and different to The apple John ate
- and different to ate the John apple

Why bother parsing?

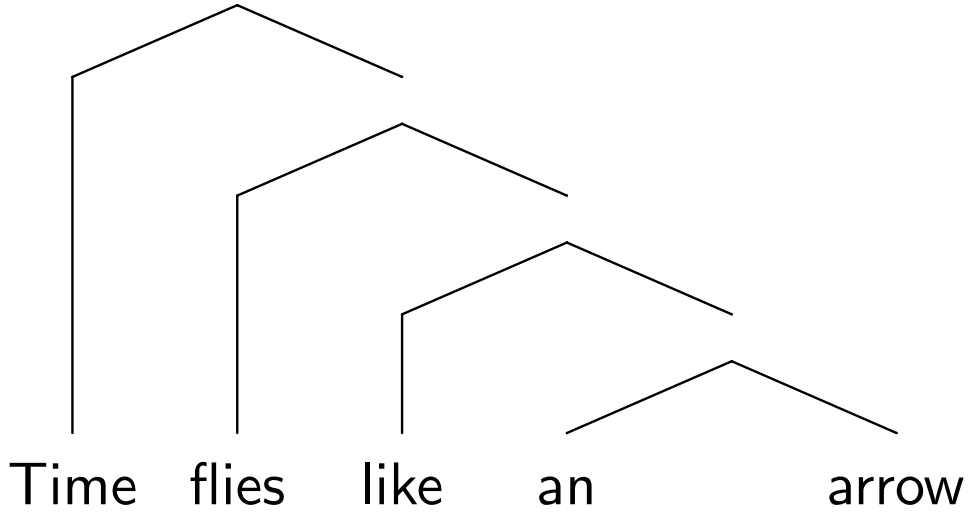
- Why is John ate the apple
- different to The apple ate John
- and different to The apple John ate
- and different to ate the John apple
- why doesn't the ate John apple make sense?

Why bother parsing?

- Why is John ate the apple
- different to The apple ate John
- and different to The apple John ate
- and different to ate the John apple
- why doesn't the ate John apple make sense?
- because there is more to language than just words!
- **language has structure**

Language structure *influences* our interpretation

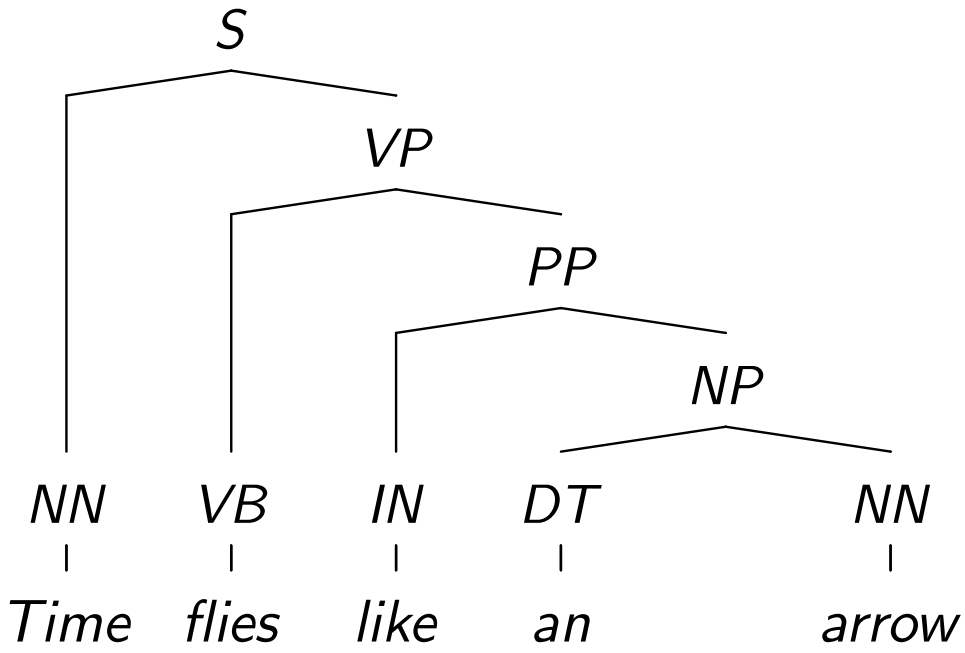
Time flies like an arrow
(Time (flies (like (an arrow))))



- words are grouped into **constituents**
- each constituent **dominates** the constituents under it
- the entire tree is a **derivation**

Language structure *influences* our interpretation

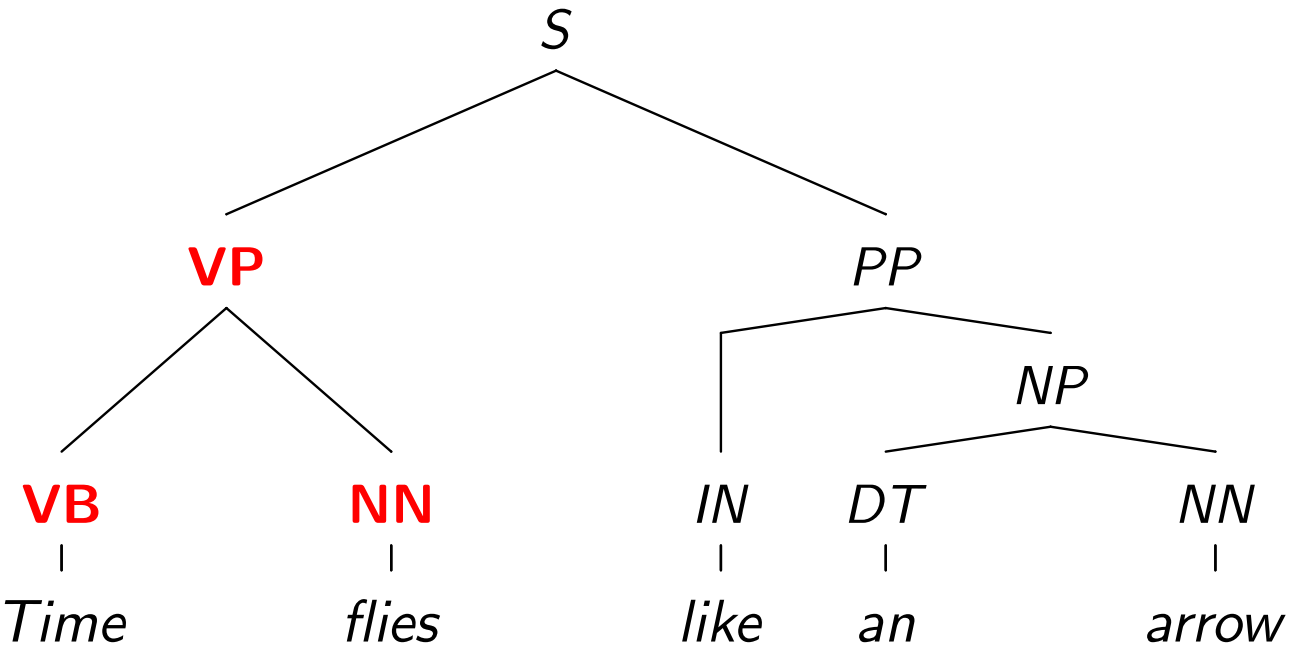
Time flies like an arrow



Time moves forwards quickly, irreversible etc

Language structure *influences* our interpretation

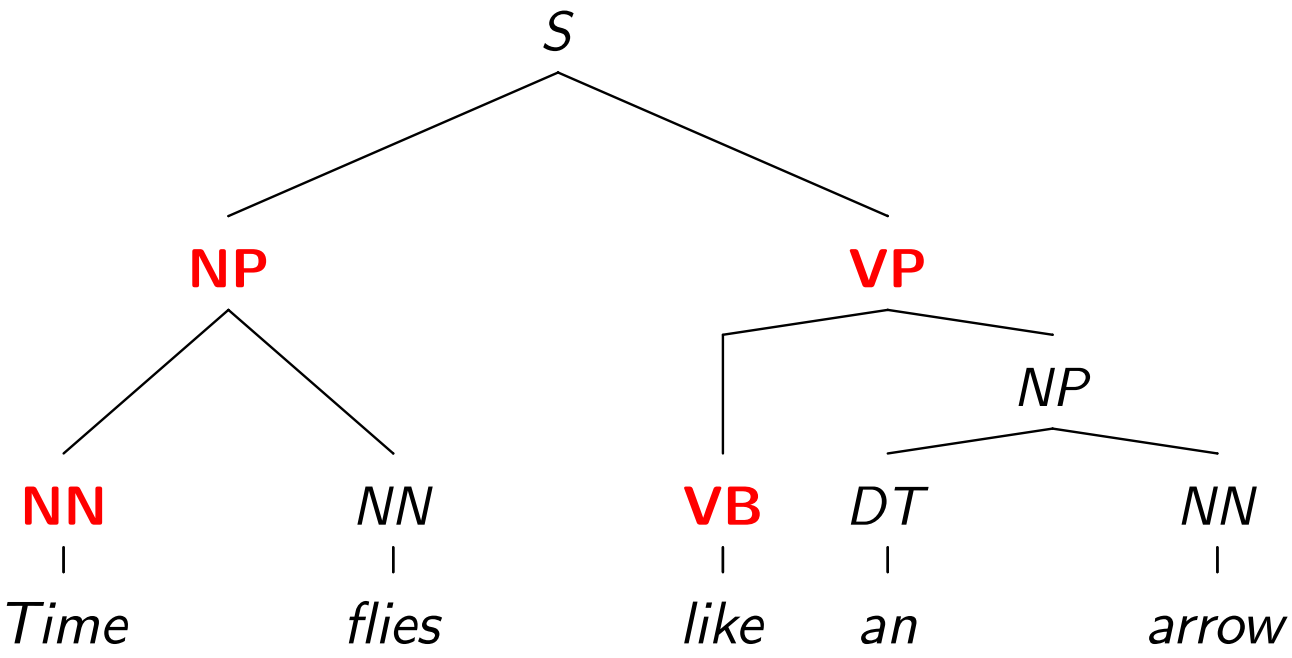
Time flies like an arrow



Time flies like you would time an arrow

Language structure *influences* our interpretation

Time flies like an arrow



cf. Fruit flies like a banana

Language structure *influences* our interpretation

Time flies like an arrow

but structure isn't all there is to it:

colourless green ideas sleep furiously

- some parses are more likely than others
- we are **not** dealing with the semantic meaning

Applications

- practically every task involving grammatical text...
  - language modelling (and speech recognition)
  - word sense disambiguation
  - information extraction
  - ...
  - question answering
  - automatic summarisation
  - (and making a comeback): machine translation
- doesn't work so well in noisy domains (the web?)
- a lot has to do with the training data available

Efficiency is often an issue

- we don't use parsers when speed is important
- phrase structure parsing in $O(n^3)$ time (with large constants)
- parsers really grind to a halt on long sentences

Sometimes we can't use the detail

- other tasks we don't know how to use that much info (yet)
- many of these tasks are currently treated as bag of words:
  - text categorisation
  - information retrieval

(Formal) Languages

- formal languages are sets of strings
- these sets can potentially be infinite in size
  - e.g. all words made of the letter *a*
  - *a*, *aa*, *aaa*, ...
- we want a way of compactly describing these sets
- we can describe methods of recognising (*accepting*) members
- these methods have different computational power (*expressiveness*)
- languages are grouped into *language classes*

Expressiveness of Language Classes

- how powerful is a language class?
- Chomsky hierarchy of language classes

LANGUAGE	AUTOMATA	
Regular	finite state	less complex
Context free	push-down	
Context sensitive	linear bounded	
Recursively enumerable	Turing machine	more complex

- where do natural (human) languages fit?
- what about programming languages?

Regular Expressions

- simple yet powerful language
  - tokenisation
  - morphological analysis
- can be accepted by Finite State Automata/Transducers
- what can it represent?
  - language of words made of the letter a
  - a^*
- what can't it represent?
  - recursion - balanced brackets
    - $\emptyset$, (), ()(), ((()), ...
  - expressions of the form $A^n B^n$
    - ab, $aabb$, $aaabbb$, ...

How do I parse HTML/XML with a regular expression?

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of regex **parsers for HTML will ins**tantly transport a *programmer's consciousness* into a world of ceaseless screaming, he comes, ~~the pestilent slithy~~ regex-infection will **devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fight* **he comes, his** unholy radiancé *destroying all enlightenment, HTML tags* ***leaking from your eyes like liquid*** pain, the song of regular expression ~~parsing~~ will *extinguish the voices of mortal man from the sp here I can see it can you see it it is beautiful* the f `inal snuf` fing *of the lies of Man ALL IS LOST* **ALL I S** LOST the *pony he comes* he ~~comes he comes~~ ***the ich***or permeates all MY FACE *MY FACE* ~~oh~~ *god no* ***NO NOOOO N***O stop *the an*~~*gles*~~ `are` **not real ZALGO IS** TONY THE PONY HE COMES

Have you tried using an XML parser instead?

http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

Regular languages and natural language

- **how do regular expressions apply to language?**
- we know language has recursive structure
- regular languages can only represent bounded depth
- in terms of natural languages it means you can't represent:
  - NP $\longrightarrow$ NP PP
  - I see [the girl] [on the hill] [with the telescope] [under the stars] [in the morning]...

Formal representation for Context Free Grammars

- a set of *terminal* symbols: $V = \{w_1, w_2, \ldots, w_n\}$
  w_i is a word in the vocabulary (or alphabet)
- a set of *non-terminal* symbols: $N = \{N_1, N_2, \ldots, N_m\}$
- a start symbol $N_s \subseteq N$ (often written as S in a grammar)
- rules of the form $N_i \longrightarrow \zeta_j$ where ζ_j is a sequence of terminals and nonterminals

- each rule application is dependent on a single symbol
- i.e. no rules of the form $XY \longrightarrow Z$

CFG examples

- language of words made of the letter a
  - terminal symbols: $\{a\}$
  - non-terminal symbols: $\{S\}$
  - start symbol: S
  - rules:
    - $S \longrightarrow Sa$
    - $S \longrightarrow \epsilon$
- balanced brackets
  - terminal symbols: $\{(,)\}$
  - non-terminal symbols: $\{S\}$
  - start symbol: S
  - rules:
    - $S \longrightarrow (S)S$
    - $S \longrightarrow \epsilon$

Context Free Grammars

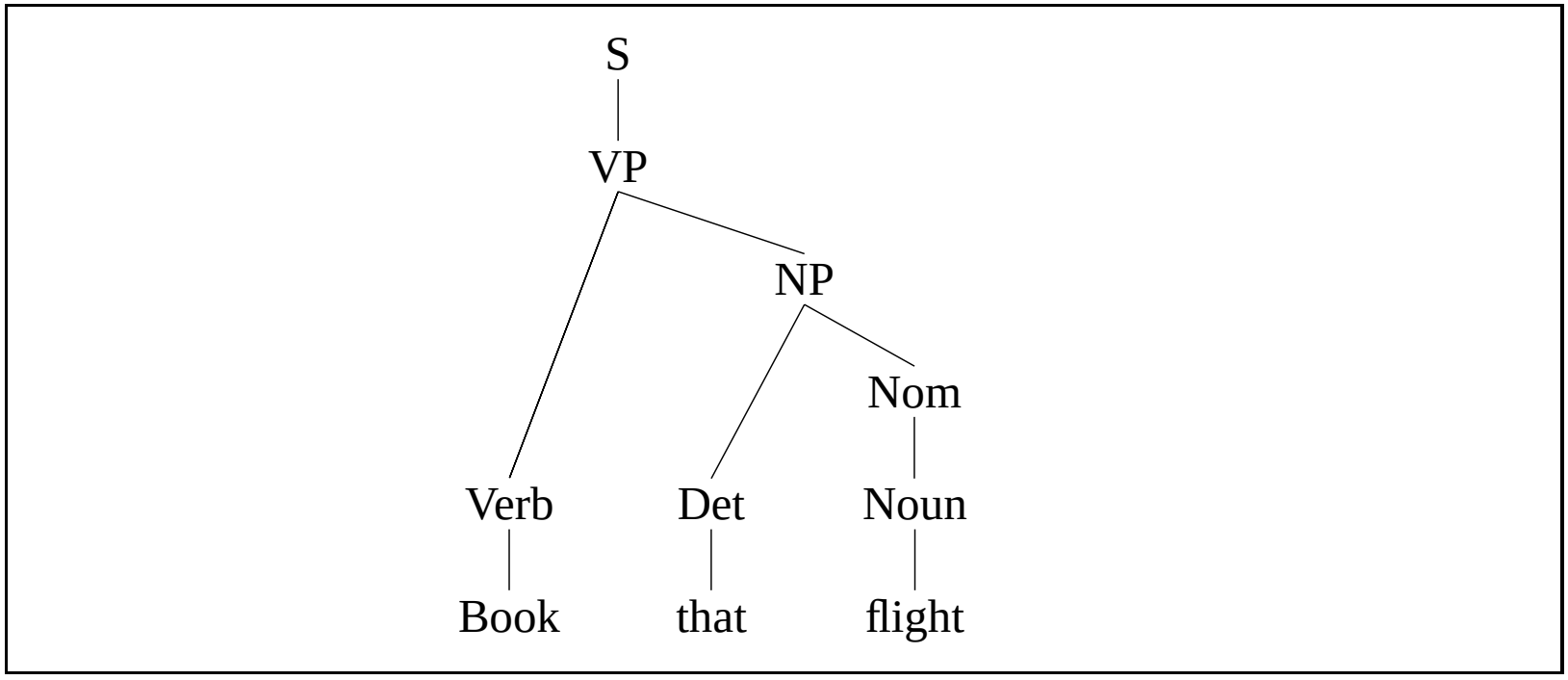
- representation of recursive structures
- can deal with balancing brackets problem
- cannot deal with expressions of the form $A^n B^n C^n$
- allows for most constructions in many natural languages (e.g. English)
- can be represented/executed by a pushdown automaton

An example grammar

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

Jurafsky and Martin, Figure 10.2

An example parse



Jurafsky and Martin, Figure 10.1

Context Sensitive Grammars

- can use surrounding categories to determine rule application
  $\zeta_i \longrightarrow \zeta_j$
- needed to handle some constructions in natural language (e.g. crossing dependencies in Dutch)
- cost of parsing with these grammars is painful
- try to minimise context dependence
  use **mildly context sensitive** grammars
- otherwise extremely expensive to parse

Part II

Parsing Algorithms

Parsing = search

- grammars are purely declarative
  - they don't say how to find a derivation
  - they don't say how to choose the best derivation
- the grammar defines the search space
- the constraints are the sentence and the grammar

Parsing Algorithms

- top down parsing
  - try to build the parse tree from the root downwards
  - hypothese all trees (smallest first)
- bottom up parsing
  - try to build the tree from the leaves (words) upwards
  - hypothesise all subtrees from the sentence
- Earley algorithm (top down + memoization)
- CKY - Cocke Younger Kasami (bottom up + memoization)

Top-down parsing

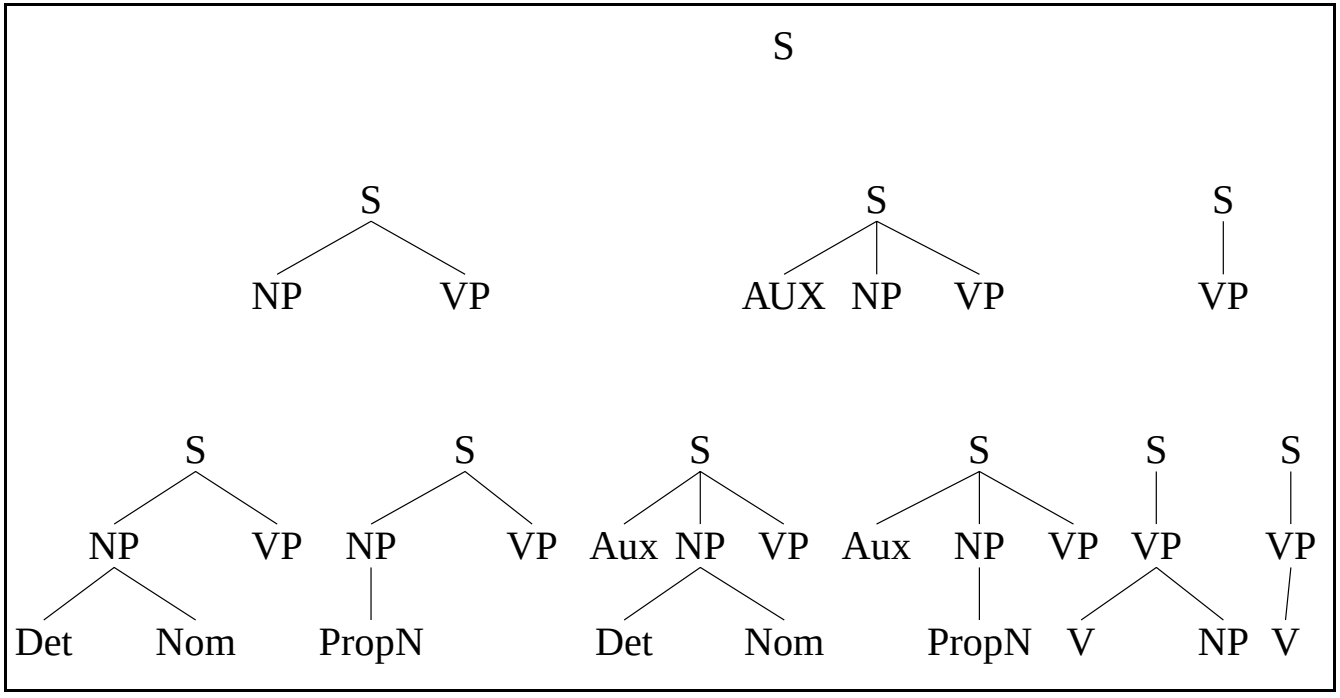
- try to build the parse tree from the root downwards
- hypothesising all trees (smallest first)

Here's the grammar again

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

Jurafsky and Martin, Figure 10.2

Top-down example

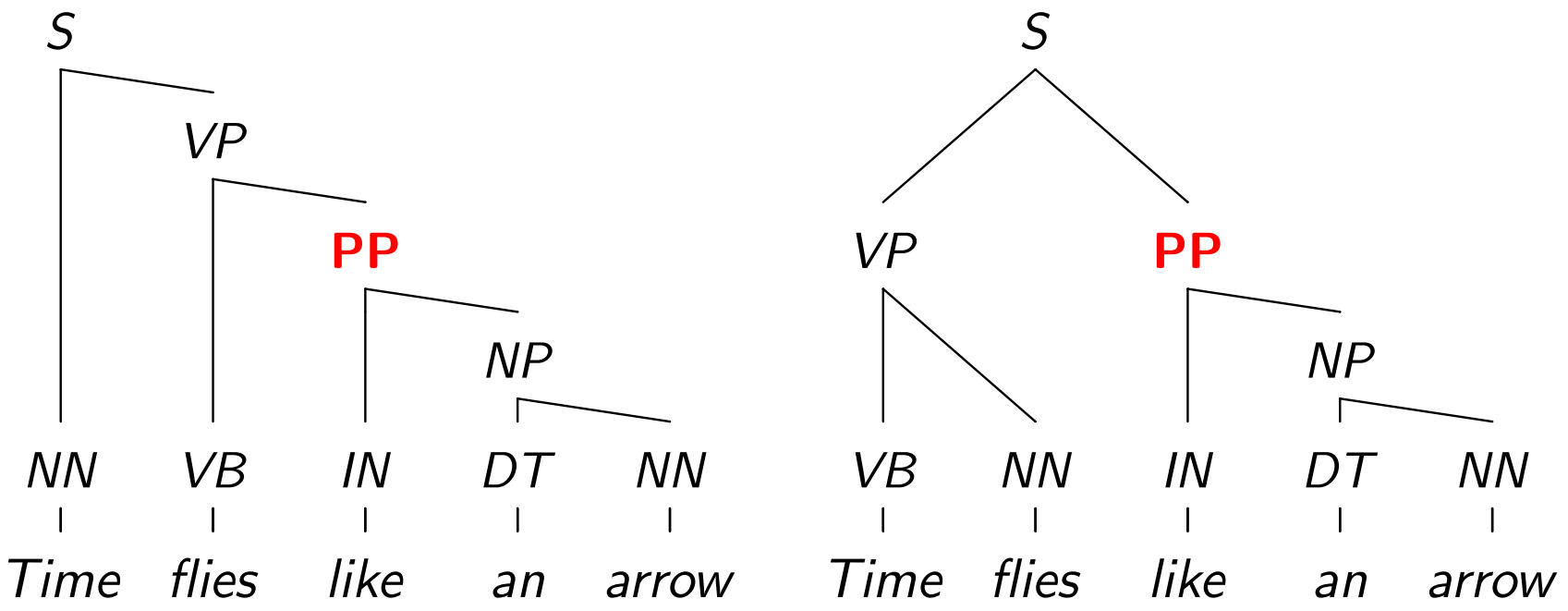


Jurafsky and Martin, Figure 10.3

Problems with top-down parsing

- search pays (almost) no attention to the input
- so we generate many many options which aren't possible
- ambiguity causes parts of the tree to be rebuilt
- this duplication is massive in useful grammars

Duplication



Bottom-up parsing

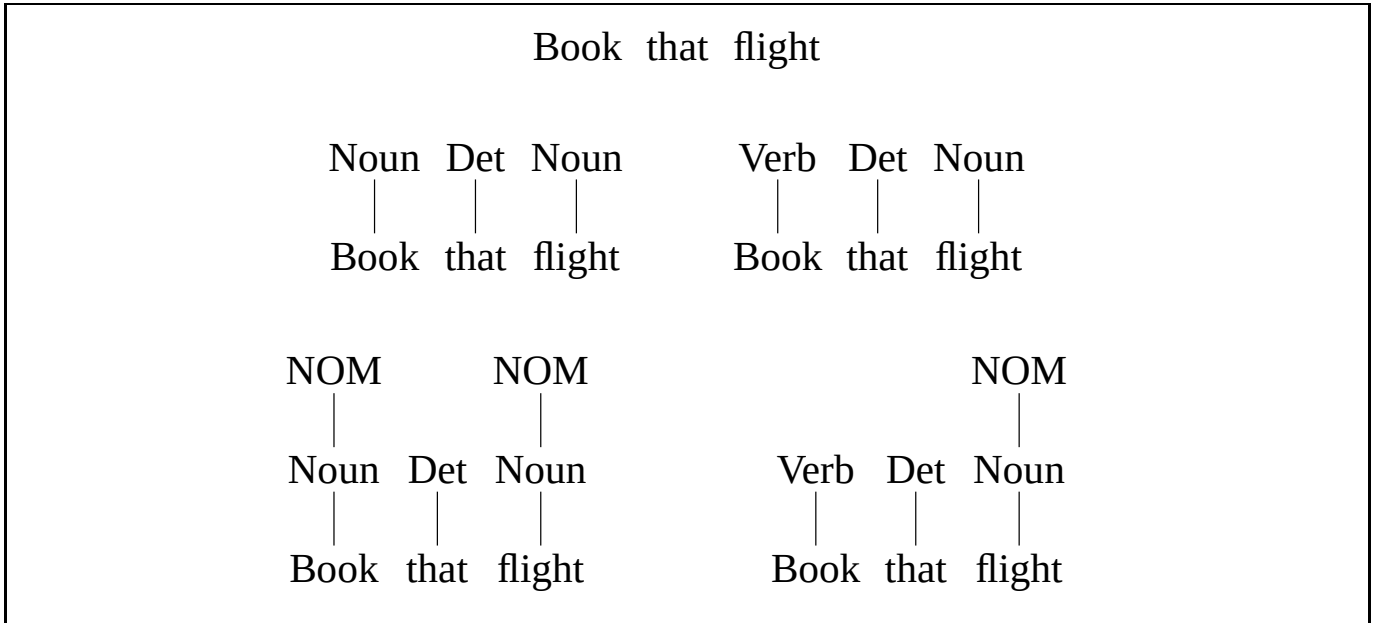
- try to build the tree from the leaves (words) upwards
- hypothesise all subtrees from the sentence

Here's the grammar again

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

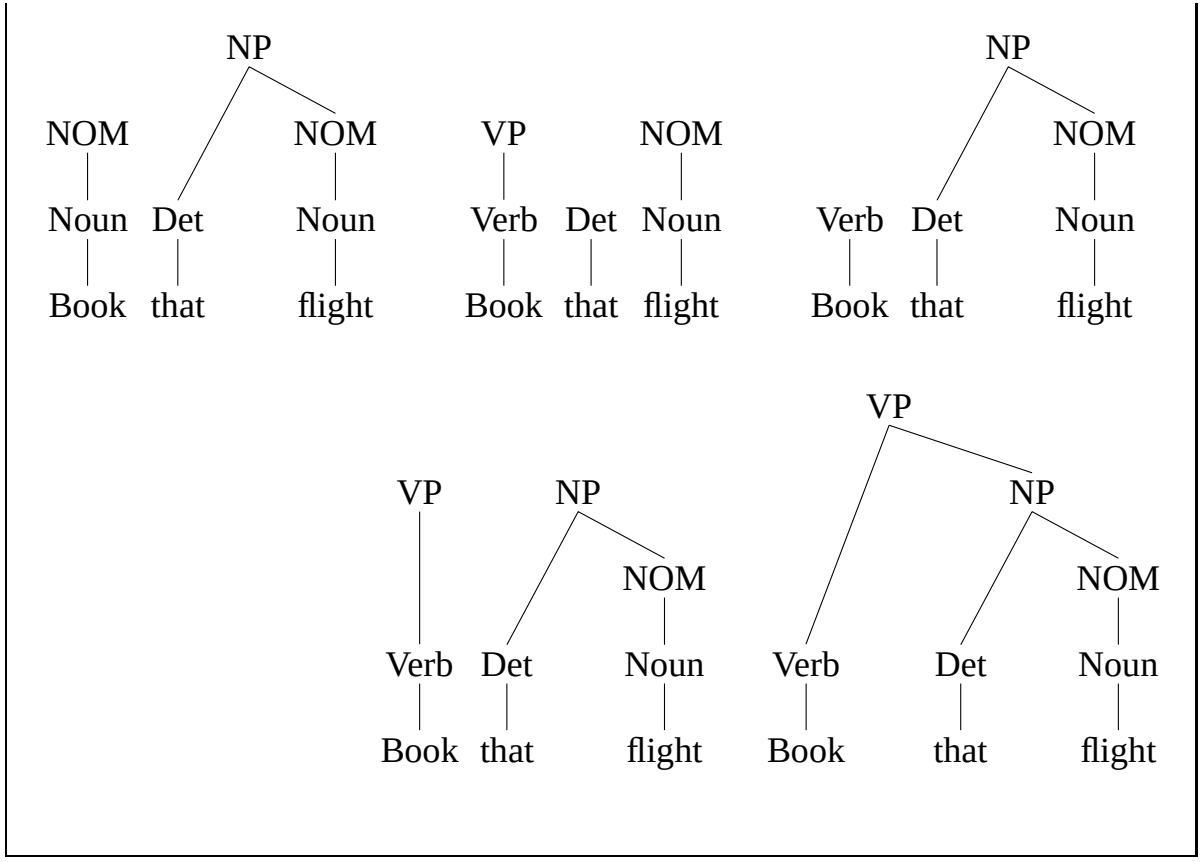
Jurafsky and Martin, Figure 10.2

Bottom-up example



Jurafsky and Martin, Figure 10.4

Bottom-up example



Jurafsky and Martin, Figure 10.4 (cont)

Problems with bottom-up parsing

- search pays no attention to the target (the root)
- generate many options which can/will never be used
- ambiguity causes parts of the tree to be rebuilt
- this duplication is massive in useful grammars

Top-down depth-first parsing

- in practice the space is not explored in parallel
- depth first allows you to reach the words themselves as soon as possible
  - this means you can throw away incorrect parses earlier
- when a tree no longer matches backtrack
- *recursive descent parsing*

Left-branching problem

- when a grammar rule is of the form:
  $X \longrightarrow X\ Y$
- for example prepositional phrases in NPs:
  $NP \longrightarrow NP\ PP$
- have to factor the rule into multiple rules
- these rules are no longer natural

Memoisation

- both top-down and bottom-up rebuild trees (over and over)
- **memoisation** involves saving previous calculations
- top-down + memoisation = Earley algorithm
- bottom-up + memoisation = CKY algorithm
- *chart* stores the cached calculations

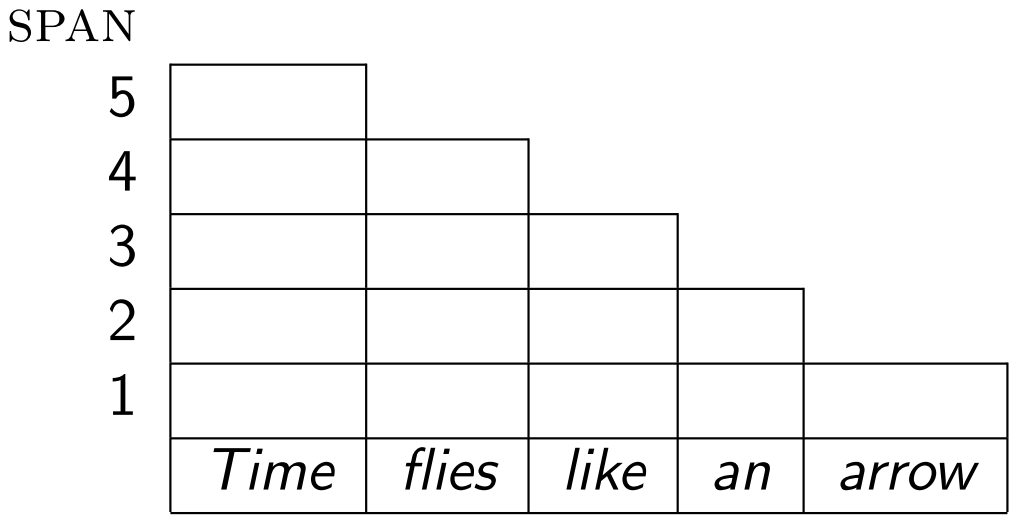
Charts

- represent/store the constructed subtrees
- later we will see that probabilities are stored too
- handy visualisation of alternative parses (see NLTK)
- spans are represented as two indices (beginning and end)
- the indices fall between the words starting from zero
- words are the first spans added

Cocke Kasami Younger (CKY) algorithm

- build all possible subtrees from the leaves
- store subtrees in the chart
- can build bottom to top or left to right
- time complexity is $O(n^3)$ in the sentence length
- we will examine the probabilistic case later

CKY algorithm example



CKY algorithm example

SPAN					
5					
4					
3					
2					
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

Fill in the bottom level POS tags

CKY algorithm example

SPAN					
5					
4					
3					
2	?				
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

Start moving up the chart, one cell at a time

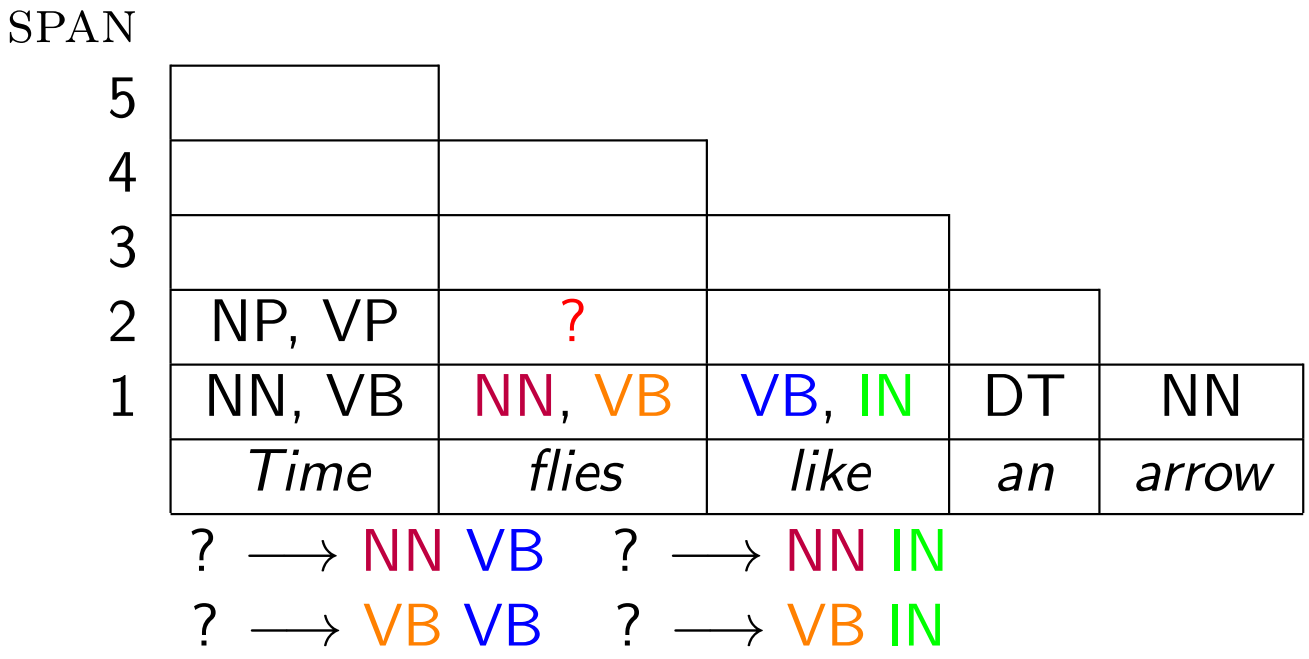
CKY algorithm example

SPAN					
5					
4					
3					
2	NP, VP				
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

NP $\longrightarrow$ NN NN ? $\longrightarrow$ NN VB

VP $\longrightarrow$ VB NN ? $\longrightarrow$ VB VB

CKY algorithm example

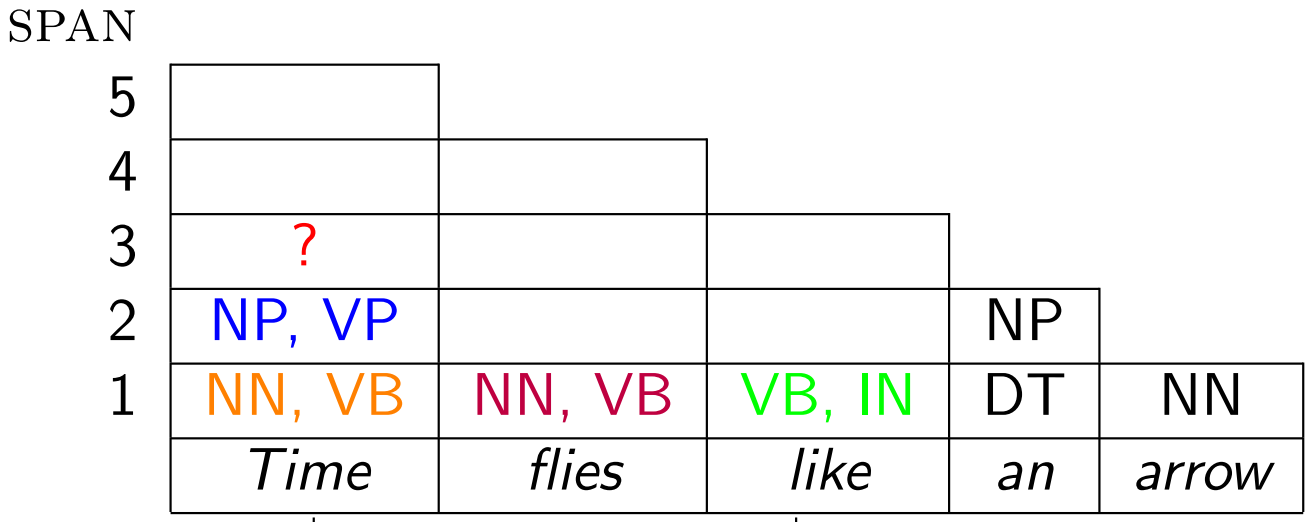


CKY algorithm example

SPAN					
5					
4					
3					
2	NP, VP		?	NP	
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

Filling in the rest of the row

CKY algorithm example



Time flies + *like*	*Time* + *flies like*	*Time* + *flies* + *like*
? ⟶ NP VB	None	? ⟶ NN NN VB
? ⟶ NP IN		? ⟶ NN NN IN
? ⟶ VP VB		? ⟶ NN VB IN
? ⟶ VP IN		...

CKY algorithm example

SPAN					
5					
4					
3			VP, PP		
2	NP, VP			NP	
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

VP $\longrightarrow$ VB NP

PP $\longrightarrow$ IN NP

CKY algorithm example

SPAN					
5					
4		S, VP			
3			VP, PP		
2	NP, VP			NP	
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

S $\longrightarrow$ NN VP

VP $\longrightarrow$ VB PP

CKY algorithm example

SPAN					
5	S				
4		VP			
3			VP, PP		
2	NP, VP			NP	
1	NN, VB	NN, VB	VB, IN	DT	NN
	Time	*flies*	*like*	*an*	*arrow*

S	$\longrightarrow$	NN VP	Time moves forwards quickly, irreversible etc
S	$\longrightarrow$	NP VP	cf. Fruit flies like a banana
S	$\longrightarrow$	VP PP	Time flies like you would time an arrow

Extra complications

- we have a recogniser, not a parser
  - need to keep back-pointers to remember the whole tree
- unary rules, need to be applied to every cell
- rules are often binarised to reduce the number of potential derivations
  - this also reduces the effectiveness of the rule probabilities
- where does the grammar come from?
  - hand-written
  - data-driven
- how do you choose between the final parse trees?

Take away

- Natural language is not regular
- What a CFG looks like
- Over-generation, under-generation
  - like regular expressions
- Syntactic ambiguity is everywhere
  - most often: preposition attachment, coordination, noun phrase
- PCFGs: probability of sentence = product of probability of production rules
- Dynamic programming gives $O(n^3)$ parse
  - may decide ambiguities