

# Phrase Structure and Parsing as Search

## Informatics 2A: Lecture 19

Shay Cohen

School of Informatics  
University of Edinburgh

30 October 2017

Part-of-speech tagging and its applications

The use of hidden Markov models for POS tagging

The Viterbi algorithm

Weighted finite-state automata and their use in applications like speech recognition, machine translation (similarly: optical character recognition, predictive text, poetry generation ...)

## 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



A well-studied, difficult, and unsolved problem.

Fortunately, we know enough to have made partial progress (Watson won).

Over the next few weeks, we will work up to the study of systems that can assign **logical forms** that mathematically state the meaning of a sentence, so that they can be processed by machines.

Our first stop will be **natural language syntax**.

Syntax provides the scaffolding for semantic composition.

The brown dog on the mat saw the striped cat through the window.

Syntax provides the scaffolding for semantic composition.

The brown dog on the mat saw the striped cat through the window.

The brown cat saw the striped dog through the window on the mat.

Syntax provides the scaffolding for semantic composition.

The brown dog on the mat saw the striped cat through the window.

The brown cat saw the striped dog through the window on the mat.

Do the two sentences above mean the same thing? What is the process by which you computed their meanings?

Words in a sentence often form groupings that can combine with other units to produce meaning. These groupings, called **constituents** can often be identified by substitution tests (much like parts of speech!)

Kim [read a book], [gave it to Sandy], and [left]

You said I should read the book and [read it] I did.

Kim read [a very interesting book about grammar].



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 **part of speech** that a word belongs to is closely linked to the type of constituent that it is associated with.
- In a X-phrase (eg **NP**), the key occurrence of X (eg **N**) is called the **head**, and controls how the phrase interacts (both syntactically and semantically) with the rest of the sentence.
- In English, the head tends to appear in the middle of a phrase.

# Constituents have structure

English NPs are commonly of the form:

(Det) Adj\* **Noun** (PP | RelClause)\*

**NP**: *the angry duck that tried to bite me,*

VPs are commonly of the form:

(Aux) Adv\* **Verb** Arg\* Adjunct\*

Arg → NP | PP

Adjunct → PP | AdvP | ...

**VP**: *usually eats artichokes for dinner,*

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.

# Constituents have structure

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*, .

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.

# Constituents have structure

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 we'll 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.

# 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 <b>walked</b> in the park.	NP,V,PP
The man <b>walked</b> with a duck.	NP,V,PP
You <b>made</b> a duck.	Pro,V,NP
You <b>made</b> her duck.	? Pro,V,NP
A man with a telescope <b>saw</b> you.	NP,PP,V,Pro
A man <b>saw</b> you with a telescope.	NP,V,Pro,PP
You <b>saw</b> 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.



# 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.

# 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 <b>a duck</b> .	Pro,V,NP
You made <b>her duck</b> .	? Pro,V,NP
A man with a telescope saw <b>you</b> .	NP,PP,V,Pro
A man saw <b>you</b> with a telescope.	NP,V,Pro,PP
You saw <b>a man</b> 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.

# 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 <b>in the park</b> .	NP,V,PP
The man walked <b>with a duck</b> .	NP,V,PP
You made a duck.	Pro,V,NP
You made her duck.	? Pro,V,NP
A man <b>with a telescope</b> saw you.	NP,PP,V,Pro
A man saw you <b>with a telescope</b> .	NP,V,Pro,PP
You saw a man <b>with a telescope</b> .	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 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** 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:

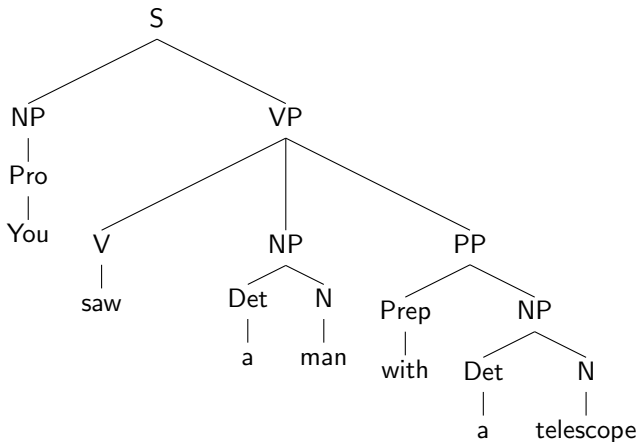
VP  $\rightarrow$  VP Conj VP  
Conj  $\rightarrow$  and | or

In **indirect recursion**, some non-terminal can be expanded (via several steps) to a sequence of symbols containing that non-terminal:

NP  $\rightarrow$  Det N PP  
PP  $\rightarrow$  Prep NP

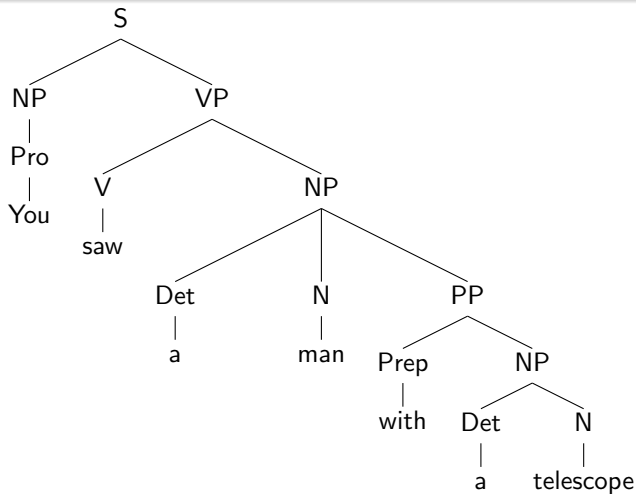
# Structural Ambiguity

You saw a man with a telescope.



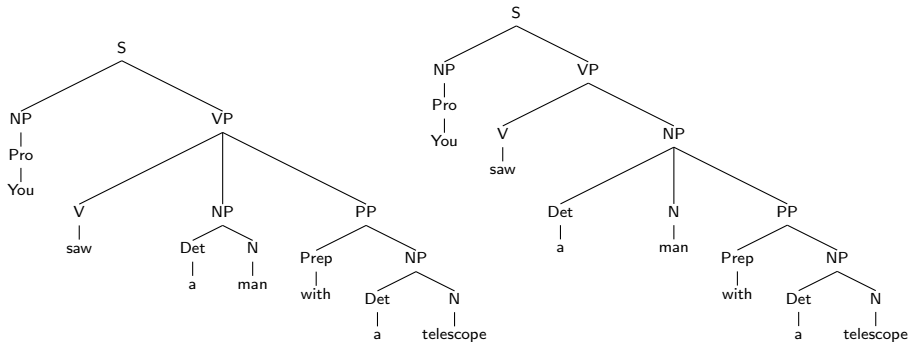
# Structural Ambiguity

You saw a man with a telescope.



# Structural Ambiguity

You saw a man with a telescope.

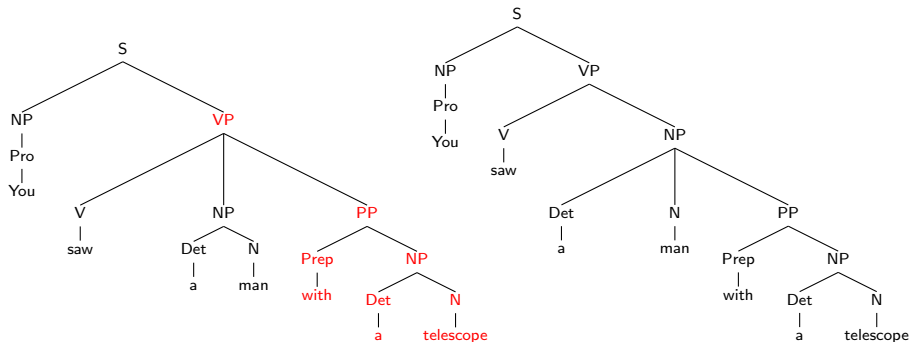


This illustrates **attachment ambiguity**: the PP can be a part of the VP or of the NP. Note that there's no **POS ambiguity** here.



# Structural Ambiguity

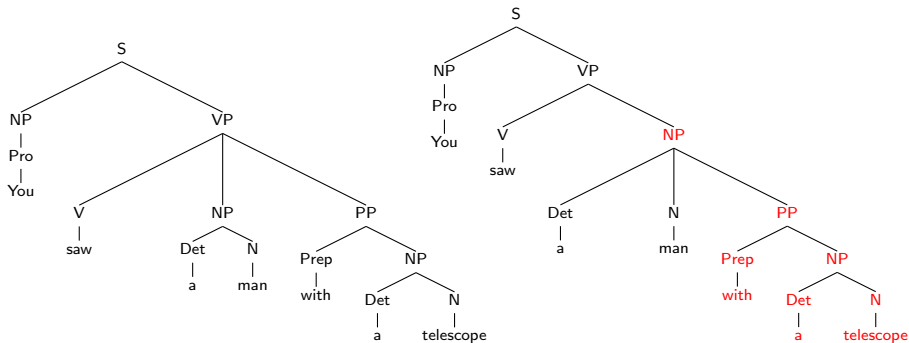
You saw a man with a telescope.



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

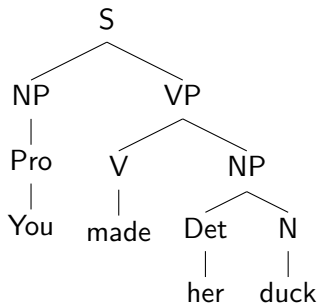
You saw a man with a telescope.



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.

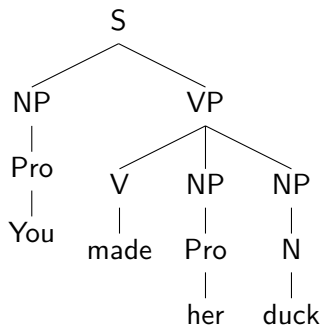
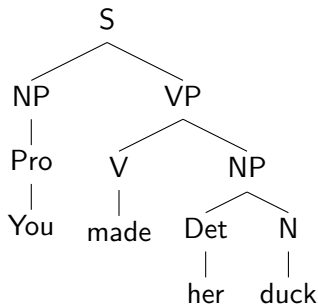
# Structural Ambiguity

For this alternative, G1 also needs rules like:

NP  $\rightarrow$  N

VP  $\rightarrow$  V NP NP

Pro  $\rightarrow$  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.

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.:

$VP \rightarrow V S1$

$S1 \rightarrow NP(\text{acc}) VP(\text{untensed})$

$NP(\text{acc}) \rightarrow \text{her} \mid \text{him} \mid \text{them}$

# 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.

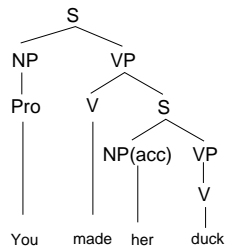
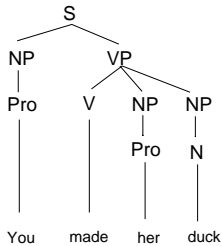
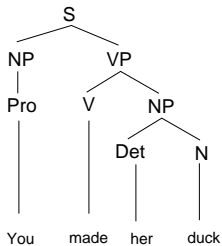
G1 needs a rule that requires accusative case-marking on the subject of a small clause and no tense on its verb.:

$VP \rightarrow V S1$

$S1 \rightarrow NP(\text{acc}) VP(\text{untensed})$

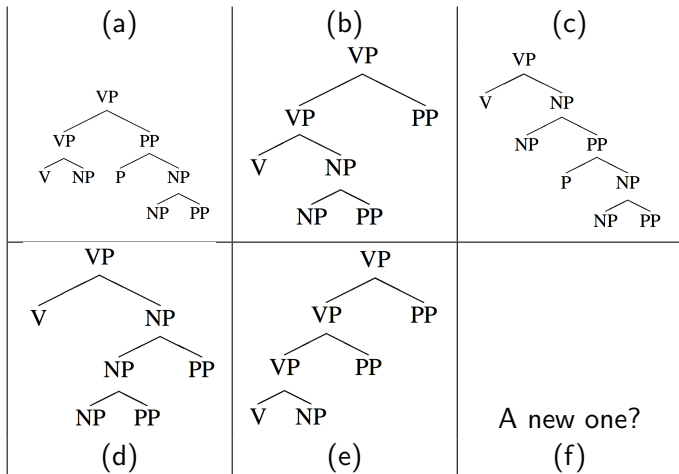
$NP(\text{acc}) \rightarrow \text{her} \mid \text{him} \mid \text{them}$

Now we have three analyses for *you made her duck*:



How can we compute these analyses automatically?

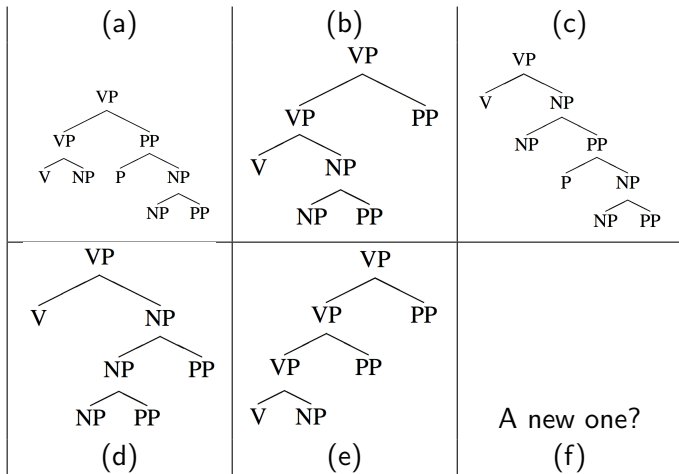
# A Fun Exercise - Which is the VP?



watched the train from the window with my binoculars



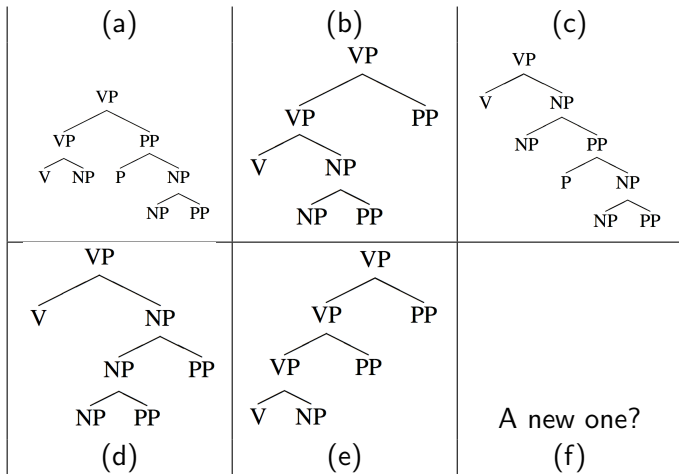
# A Fun Exercise - Which is the VP?



watched the train from the window with my binoculars

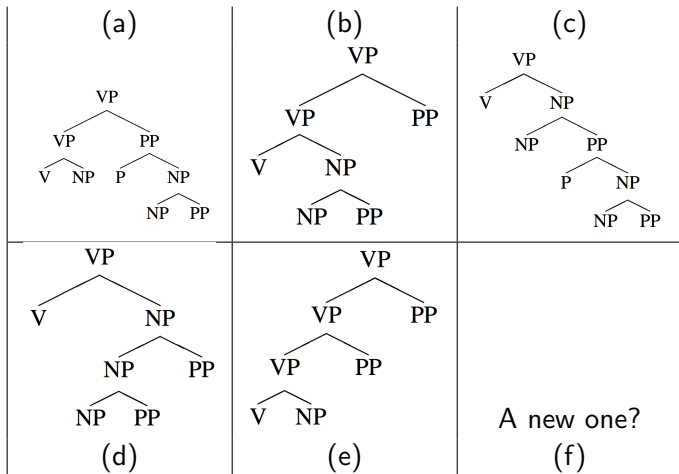
E

# A Fun Exercise - Which is the VP?



watched the train from the window on the wall

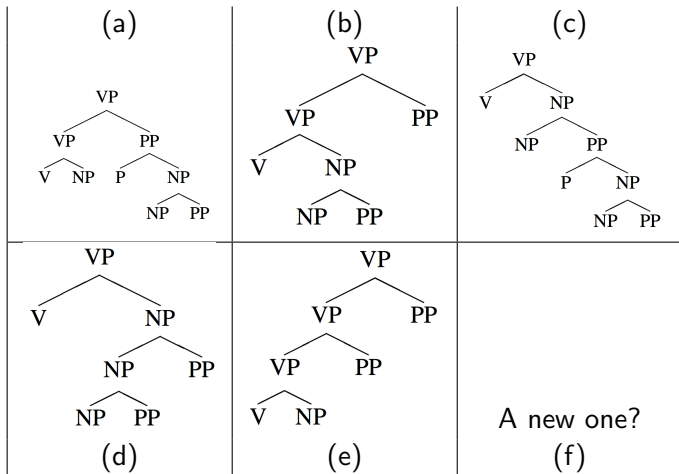
# A Fun Exercise - Which is the VP?



watched the train from the window on the wall

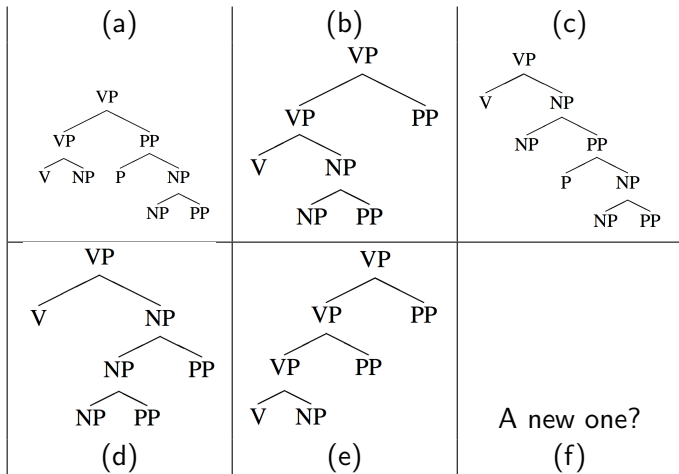
A

# A Fun Exercise - Which is the VP?



watched a show by Netflix about the starship

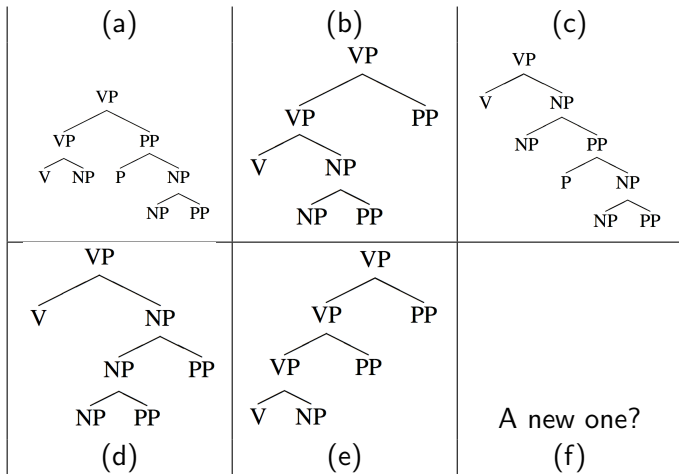
# A Fun Exercise - Which is the VP?



watched a show by Netflix about the starship

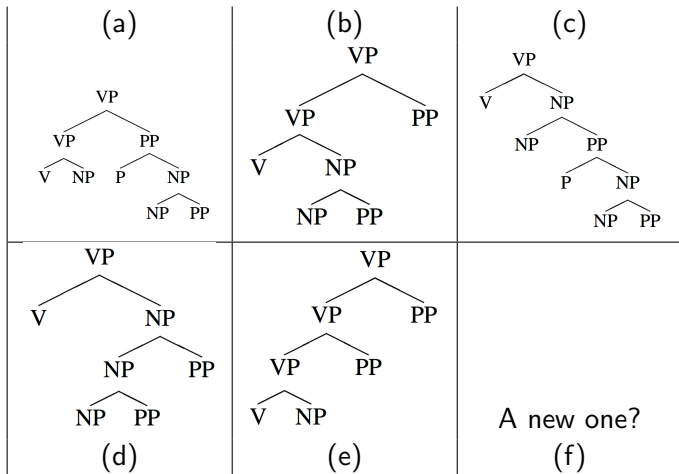
D

# A Fun Exercise - Which is the VP?



watched a show about the expedition to space

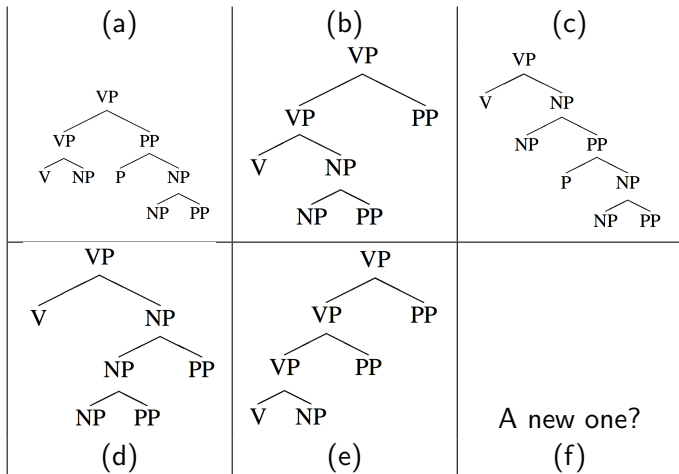
# A Fun Exercise - Which is the VP?



watched a show about the expedition to space

C

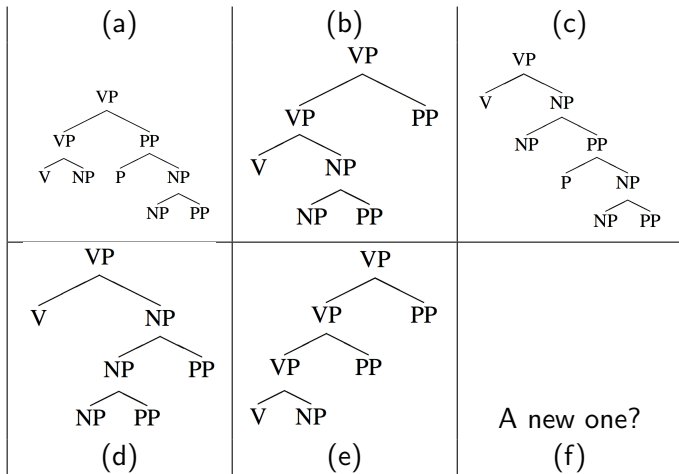
# A Fun Exercise - Which is the VP?



watched a video about the comet on my mobile



# A Fun Exercise - Which is the VP?



watched a video about the commet on my mobile

B

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

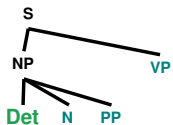
# Recursive Descent Parsing



---

the dog saw a man in the park

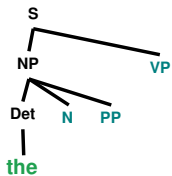
# Recursive Descent Parsing



.....  
the dog saw a man in the park

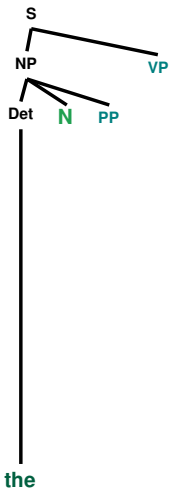


# Recursive Descent Parsing



.....  
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



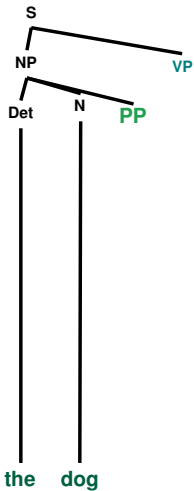
the dog saw a man in the park

# Recursive Descent Parsing



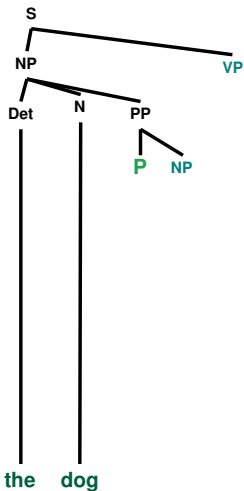
the dog saw a man in the park

# Recursive Descent Parsing



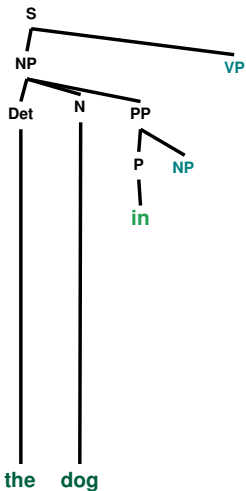
the dog saw a man in the park

# Recursive Descent Parsing



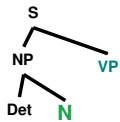
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



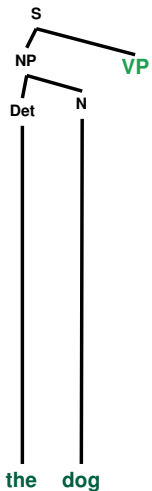
the

---

the dog saw a man in the park

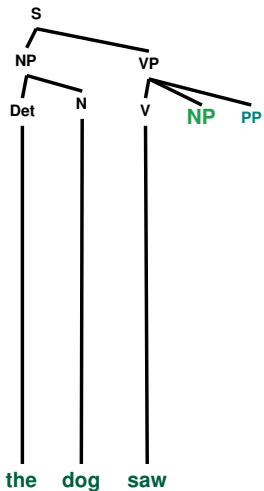


# Recursive Descent Parsing



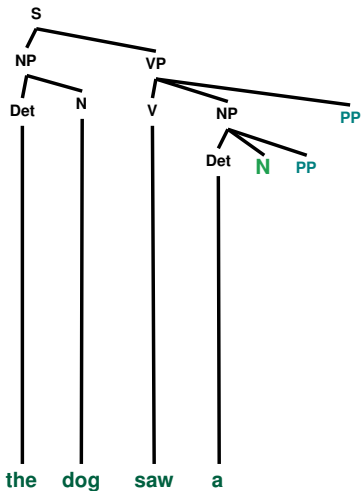
the dog saw a man in the park

# Recursive Descent Parsing



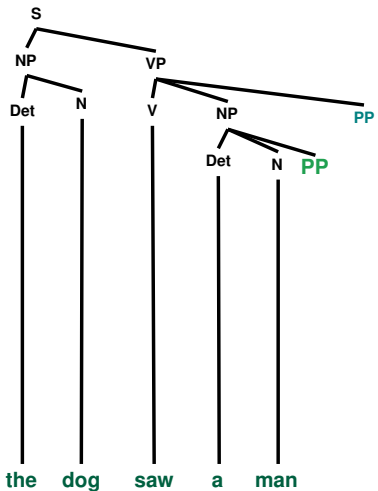
the dog saw a man in the park

# Recursive Descent Parsing



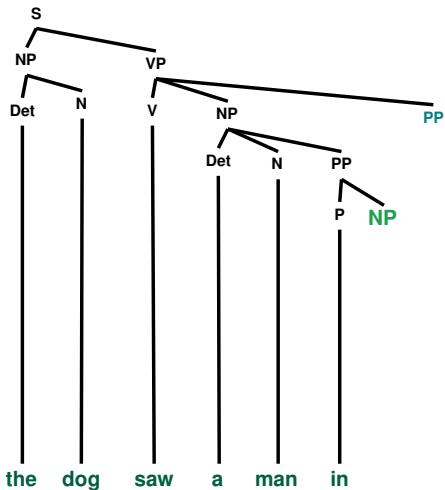
the dog saw a man in the park

# Recursive Descent Parsing



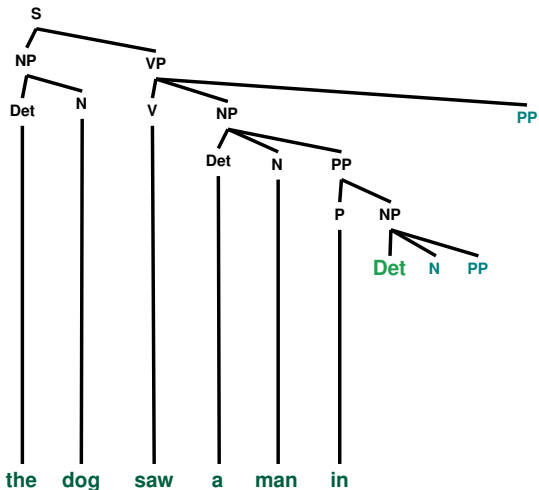
the dog saw a man in the park

# Recursive Descent Parsing



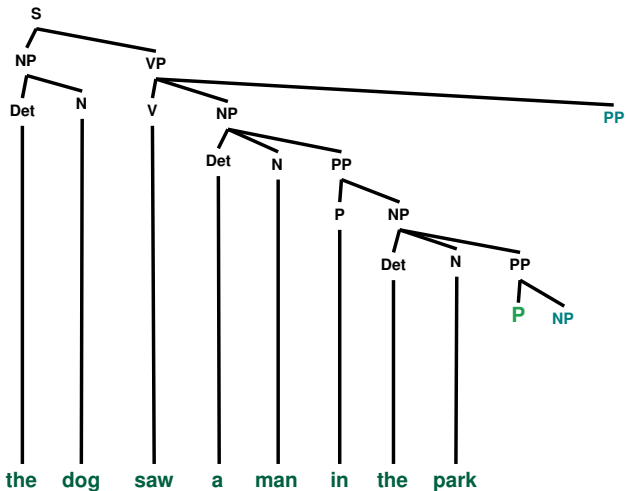
the dog saw a man in the park

# Recursive Descent Parsing



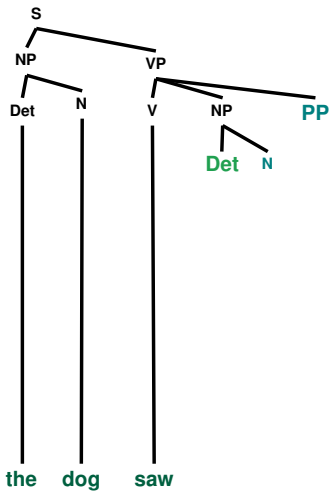
the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

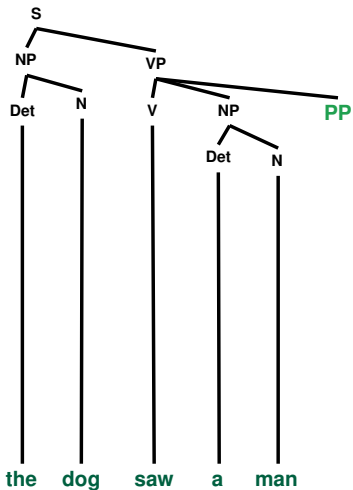
# Recursive Descent Parsing



the dog saw a man in the park

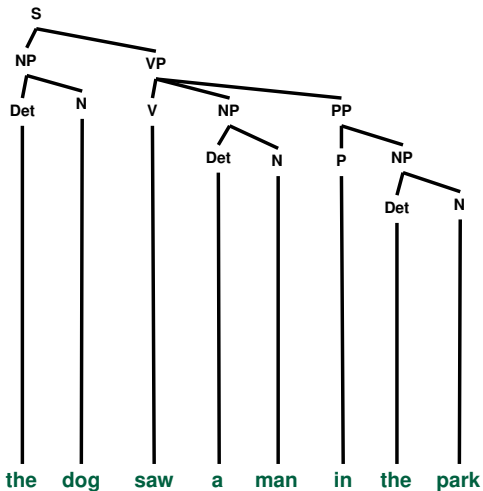


# Recursive Descent Parsing



the dog saw a man in the park

# Recursive Descent Parsing



the dog saw a man in the park

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

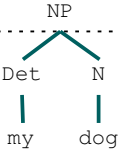
# Shift-Reduce Parsing

Stack	Remaining
Det	dog saw a man in the park
 my	

# Shift-Reduce Parsing

Stack	Remaining
Det      N	saw a man in the park
 my    dog	

# Shift-Reduce Parsing

Stack	Remaining
 <pre>graph TD; NP --&gt; Det; NP --&gt; N; Det --&gt; my; N --&gt; dog;</pre>	saw a man in the park



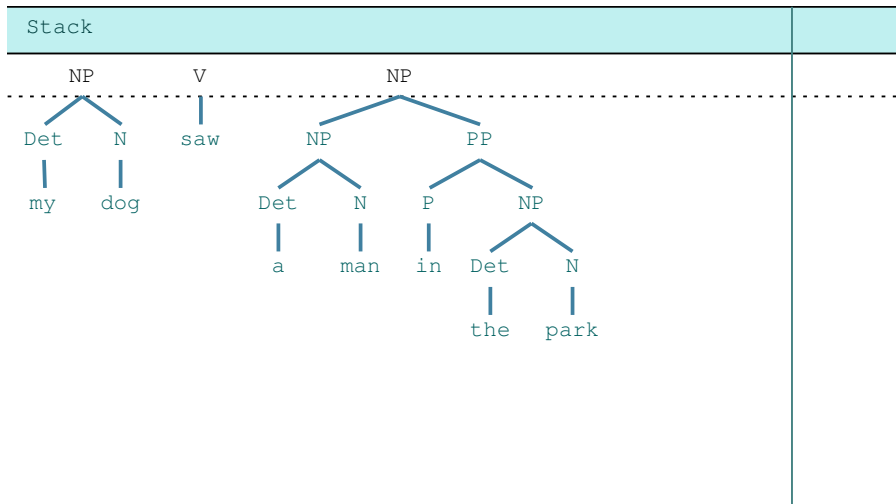
# Shift-Reduce Parsing

Stack	Remaining
<p data-bbox="193 252 230 277">NP</p> <p data-bbox="367 252 389 277">V</p> <p data-bbox="526 252 563 277">NP</p> <hr data-bbox="107 294 1366 298"/> <p data-bbox="128 342 186 367">Det</p> <p data-bbox="252 342 271 367">N</p> <p data-bbox="351 342 408 367">saw</p> <p data-bbox="460 342 518 367">Det</p> <p data-bbox="595 342 614 367">N</p> <p data-bbox="134 384 153 409"> </p> <p data-bbox="257 384 277 409"> </p> <p data-bbox="134 434 178 458">my</p> <p data-bbox="238 434 296 458">dog</p> <p data-bbox="482 384 502 409"> </p> <p data-bbox="606 384 625 409"> </p> <p data-bbox="482 434 502 458">a</p> <p data-bbox="578 434 636 458">man</p>	<p data-bbox="965 252 1190 277">in the park</p>

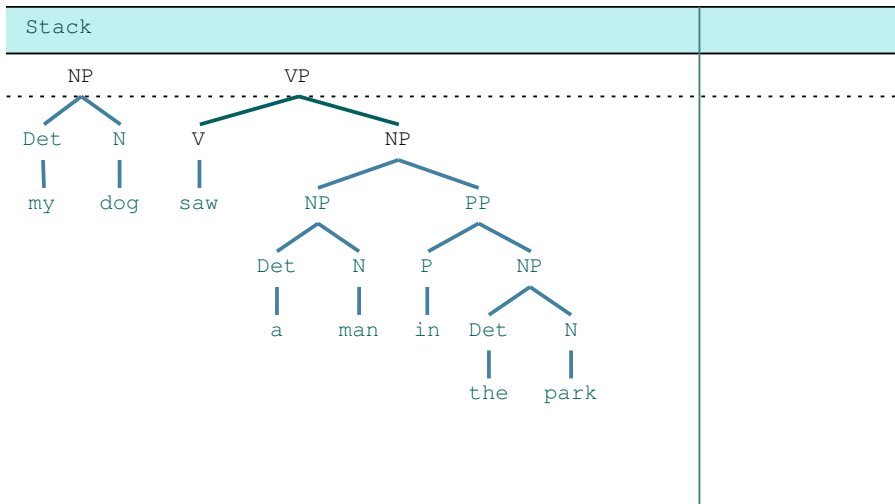
# Shift-Reduce Parsing

Stack	Remaining
<p data-bbox="193 254 230 277">NP</p> <pre data-bbox="128 291 296 462">graph TD   NP1[NP] --- Det1[Det]   NP1 --- N1[N]   Det1 --- my[my]   N1 --- dog[dog]</pre> <p data-bbox="367 254 388 277">V</p> <p data-bbox="351 346 408 368">saw</p> <p data-bbox="526 254 563 277">NP</p> <pre data-bbox="460 291 628 462">graph TD   NP2[NP] --- Det2[Det]   NP2 --- N2[N]   Det2 --- a[a]   N2 --- man[man]</pre> <p data-bbox="755 254 793 277">PP</p> <pre data-bbox="690 291 943 555">graph TD   PP[PP] --- P[P]   PP --- NP3[NP]   P --- in[in]   NP3 --- Det3[Det]   NP3 --- N3[N]   Det3 --- the[the]   N3 --- park[park]</pre>	

# Shift-Reduce Parsing

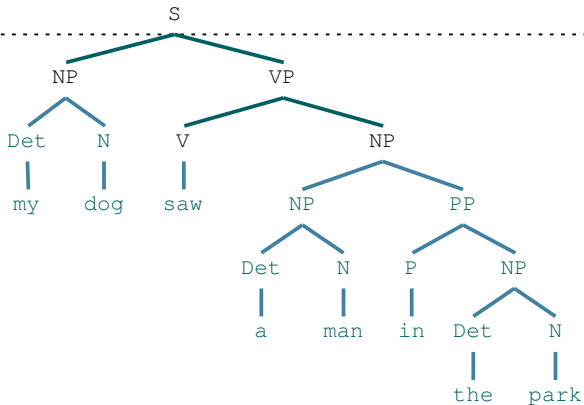


# Shift-Reduce Parsing



# Shift-Reduce Parsing

Stack



# 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 they parse, 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 parse tree 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



## 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