

Phrase Structure and Parsing as Search

Informatics 2A: Lecture 17

John Longley

School of Informatics
University of Edinburgh

25 October 2013

- 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

Heads and Phrases

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.

Heads and Phrases

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 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:

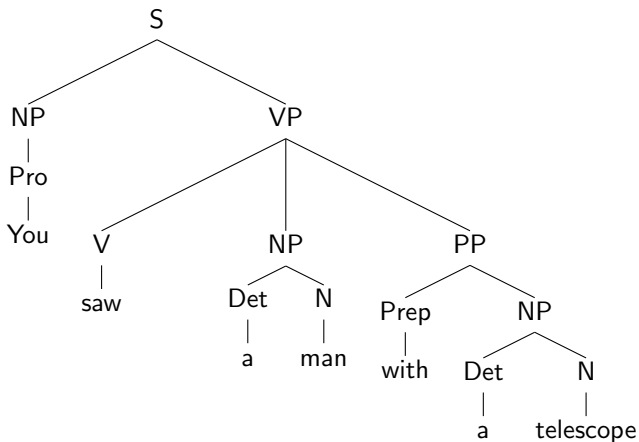
VP \rightarrow VP Conj VP
Conj \rightarrow and | or

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

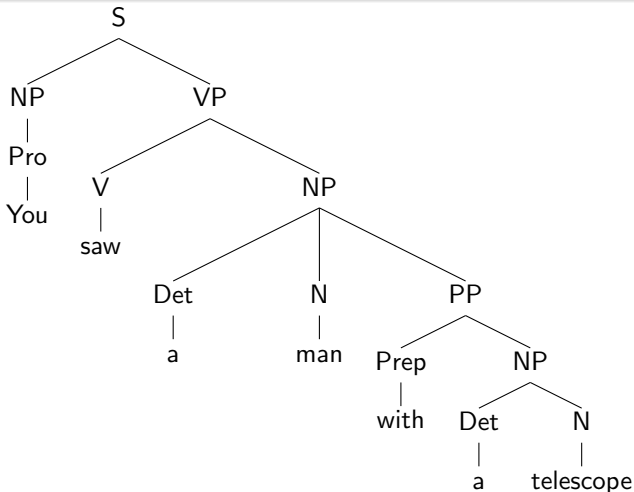
Structural Ambiguity

You saw a man with a telescope.



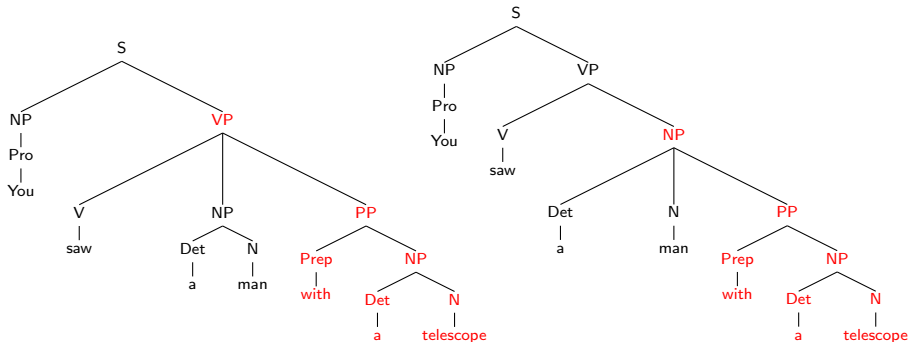
Structural Ambiguity

You saw a man with a telescope.



Structural Ambiguity

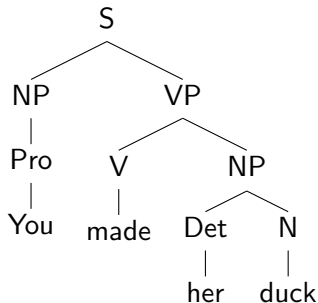
You saw a man with a telescope.



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

Grammar G1 only gives us one analysis of *you made her duck*.



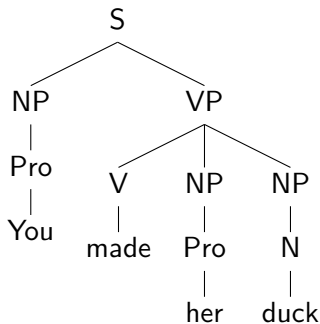
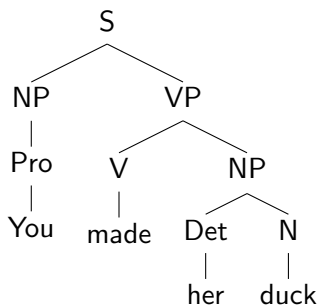
There is another, ditransitive (i.e., two-object) analysis of this sentence – one that underlies the pair:

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. (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.:

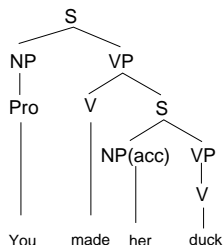
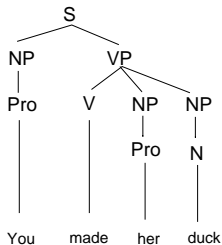
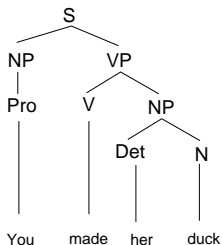
$VP \rightarrow V S1$

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

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

Structural Ambiguity

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



How can we compute these analyses automatically?

Parsing Algorithms

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.

Recursive Descent Parsing

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

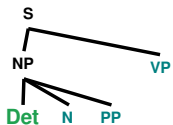
.....
the dog saw a man in the park

Recursive Descent Parsing

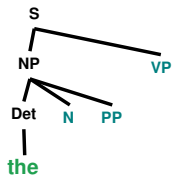


.....
the dog saw a man in the park

Recursive Descent Parsing



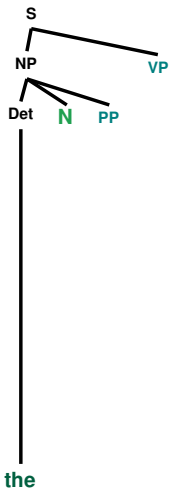
Recursive Descent Parsing



.....

the dog saw a man in the park

Recursive Descent Parsing



the dog saw a man in the park

Recursive Descent Parsing



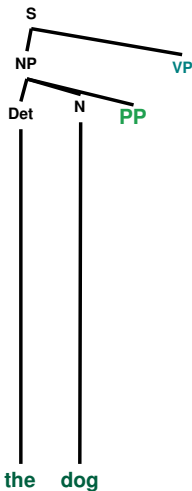
the dog saw a man in the park

Recursive Descent Parsing



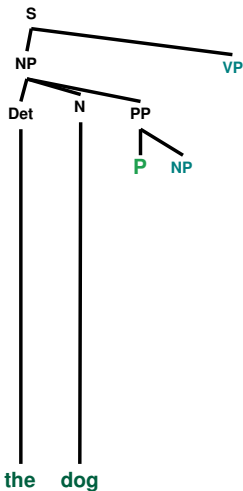
the dog saw a man in the park

Recursive Descent Parsing



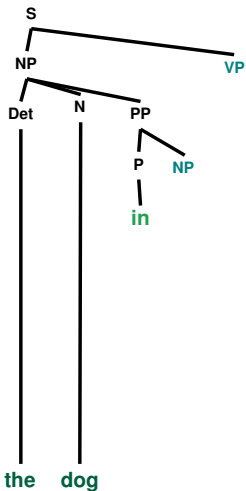
the dog saw a man in the park

Recursive Descent Parsing



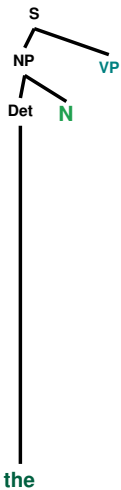
the dog saw a man in the park

Recursive Descent Parsing



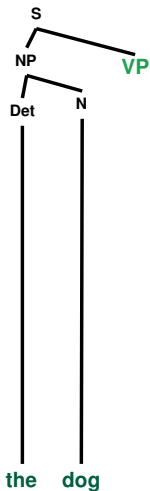
the dog saw a man in the park

Recursive Descent Parsing



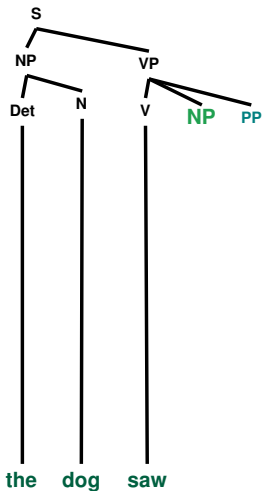
the dog saw a man in the park

Recursive Descent Parsing



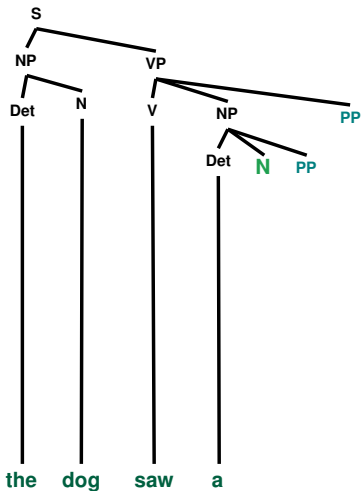
the dog saw a man in the park

Recursive Descent Parsing



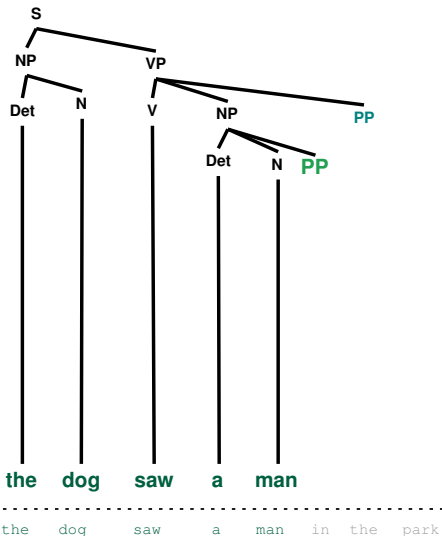
the dog saw a man in the park

Recursive Descent Parsing

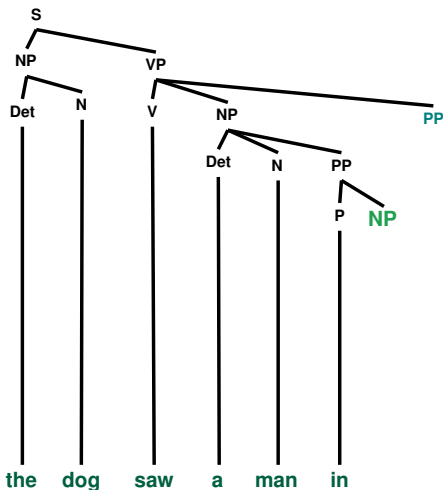


the dog saw a man in the park

Recursive Descent Parsing

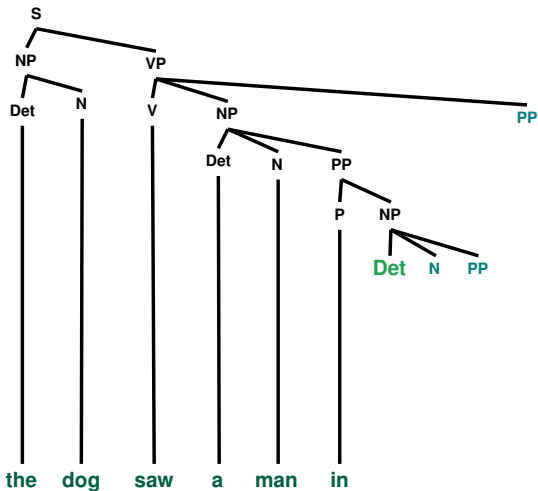


Recursive Descent Parsing



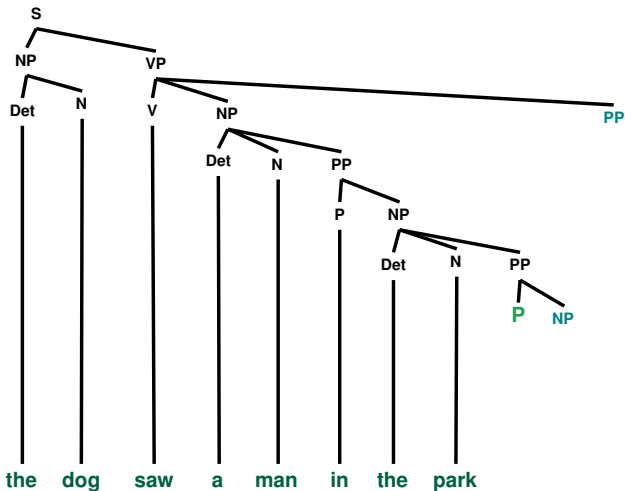
the dog saw a man in the park

Recursive Descent Parsing



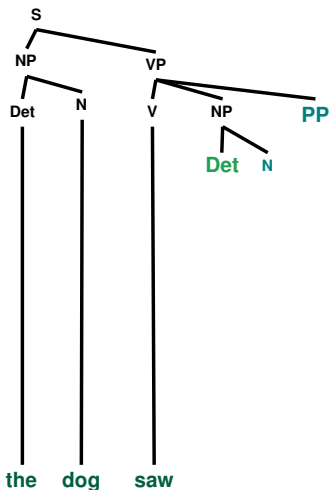
the dog saw a man in the park

Recursive Descent Parsing



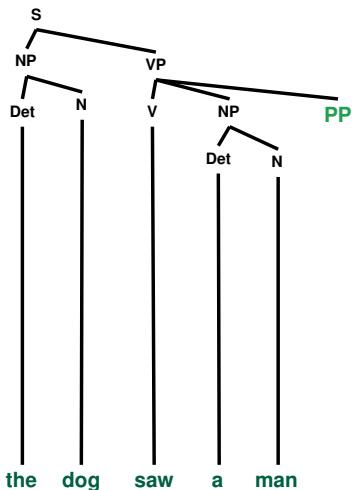
the dog saw a man in the park

Recursive Descent Parsing



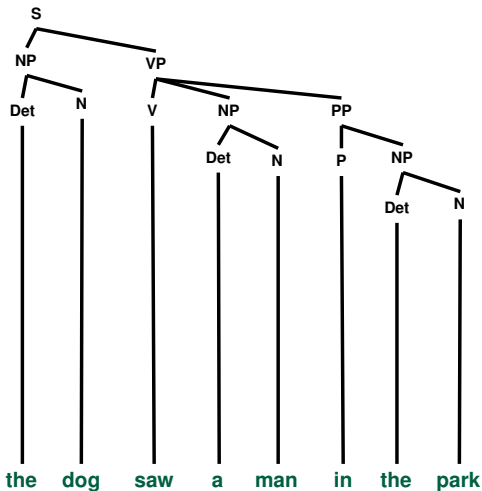
the dog saw a man in the park

Recursive Descent Parsing



the dog saw a man in the park

Recursive Descent Parsing



the dog saw a man in the park

Shift-Reduce 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 <hr style="border-top: 1px dashed black;"/>

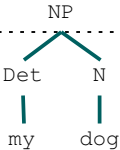
Shift-Reduce Parsing

Stack	Remaining
Det	dog saw a man in the park
<div style="border-left: 1px solid black; border-right: 1px solid black; border-bottom: 1px solid black; padding: 5px;"> my </div>	

Shift-Reduce Parsing

Stack	Remaining
<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Det</p> <p> </p> <p>my</p> </div> <div style="text-align: center;"> <p>N</p> <p> </p> <p>dog</p> </div> </div>	<p>saw a man in the park</p>

Shift-Reduce Parsing

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

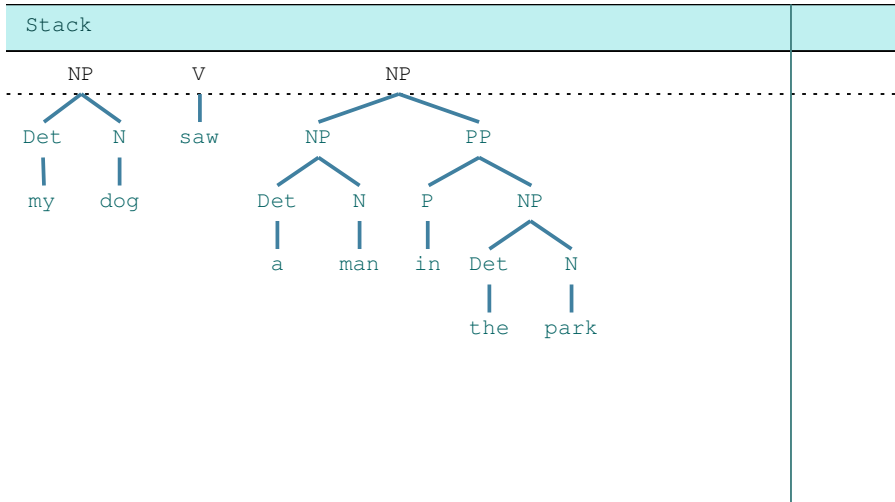
Shift-Reduce Parsing

Stack	Remaining
<p>NP V NP</p> <p> / \ / \ Det N saw Det N my dog a man</p> <hr style="border-top: 1px dashed black;"/>	<p>in the park</p>

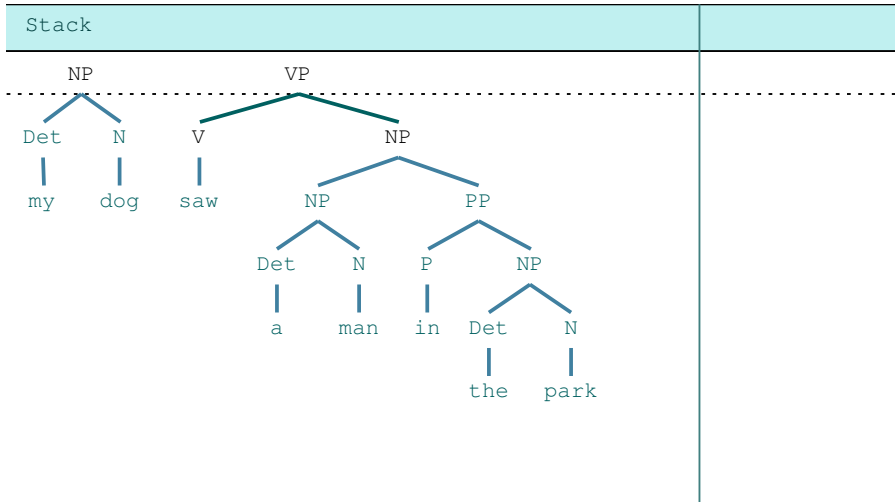
Shift-Reduce Parsing

Stack	Remaining
<p>my dog saw a man in the park</p>	

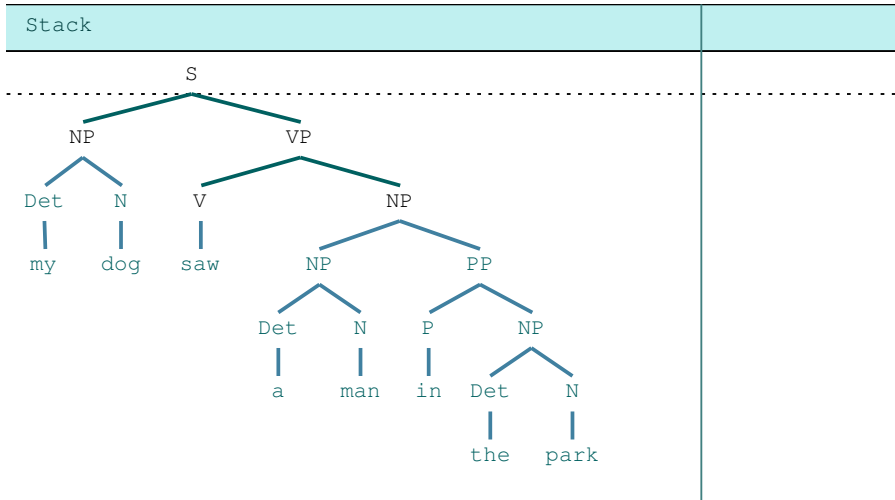
Shift-Reduce Parsing



Shift-Reduce Parsing



Shift-Reduce Parsing



Try it out Yourself!

Recursive Decent Parser

```
>>> from nltk.app import rdparser  
>>> rdparser()
```

Shift-Reduce Parser

```
>>> from nltk.app import srparser  
>>> srparser()
```

Summary

- 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