FNLP Lecture 11
Syntax and parsing

Henry S. Thompson
(including slides from Sharon Goldwater, Alex Lascarides, Mark Steedman and Philipp Koehn)

28 February 2017

Modelling word behaviour

We've seen various ways to model word behaviour.

- Bag-of-words models: ignore word order entirely
- N-gram models: capture a fixed-length history to predict word sequences.
- HMMs: also capture fixed-length history, using latent variables.

Useful for various tasks, but a really accurate model of language needs more than a fixed-length history!

Long-range dependencies

The form of one word often depends on (agrees with) another, even when arbitrarily long material intervenes.

Sam/Dogs sleeps/sleep soundly
Sam, who is my cousin, sleeps soundly
Dogs often stay at my house and sleep soundly
Sam, the man with red hair who is my cousin, sleeps soundly

We want models that can capture these dependencies.

Phrasal categories

We may also want to capture **substitutability** at the phrasal level.

- POS categories indicate which *words* are substitutable. For example, substituting **adjectives**:

  I saw a red cat
  I saw a former cat
  I saw a billowy cat

- **Phrasal categories** indicate which *phrases* are substitutable. For example, substituting **noun phrase**:

  Dogs sleep soundly
  My next-door neighbours sleep soundly
  Green ideas sleep soundly

Theories of syntax

A **theory of syntax** should explain which sentences are **well-formed** (grammatical) and which are not.

- Note that **well-formed** is distinct from **meaningful**.
- Famous example from Chomsky:
  Colorless green ideas sleep furiously
- However we'll see shortly that the reason we care about syntax is mainly for interpreting meaning.

Theories of syntax

We'll look at two theories of syntax to handle one or both phenomena above (long-range dependencies, phrasal substitutability):

- **Context-free grammar** (and variants): today, next class
- **Dependency grammar**: following class

These can be viewed as different models of language behaviour. As with other models, we will look at

- What each model can capture, and what it cannot.
- Algorithms that provide syntactic analyses for sentences using these models (i.e., **syntactic parsers**).

Reminder: Context-free grammar

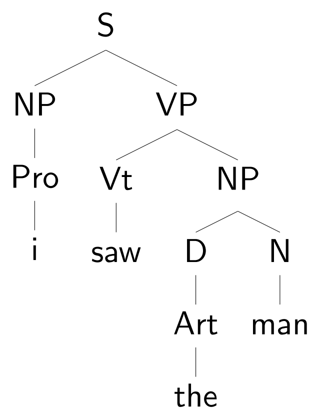
- Two types of grammar symbols:
  - **terminals** (t): words.
  - **Non-terminals** (NT): phrasal categories like `S`, `NP`, `VP`, `PP`, with `S` being the **Start symbol**. In practice, we sometimes distinguish **pre-terminals** (POS tags), a type of NT.
- Rules of the form $\mathtt{NT} \rightarrow \beta$, where β is any string of NT's and t's.
  - Strictly speaking, that's a *notation* for a rule.
  - There's also an abbreviated notation for sets of rules with same LHS: $\mathtt{NT} \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$
- A CFG in **Chomsky Normal Form** only has rules of the form $\mathtt{NT}_i \rightarrow \mathtt{NT}_j\ \mathtt{NT}_k$ or $\mathtt{NT}_i \rightarrow \mathtt{t}_j$

CFG example

S → NP VP	**(Sentences)**
NP → D N | Pro | PropN	**(Noun phrases)**
D → PosPro | Art | NP 's	**(Determiners)**
VP → Vi | Vt NP | Vp NP VP	**(Verb phrases)**
Pro → i | we | you | he | she | him | her	**(Pronouns)**
PosPro → my | our | your | his | her	**(Possessive pronouns)**
PropN → Robin | Jo	**(Proper nouns)**
Art → a | an | the	**(Articles)**
N → man | duck | saw | park | telescope	**(Nouns)**
Vi → sleep | run | duck	**(Intransitive verbs)**
Vt → eat | break | see | saw	**(Transitive verbs)**
Vp → see | saw | heard	**(Verbs with NP VP args)**

Example syntactic analysis

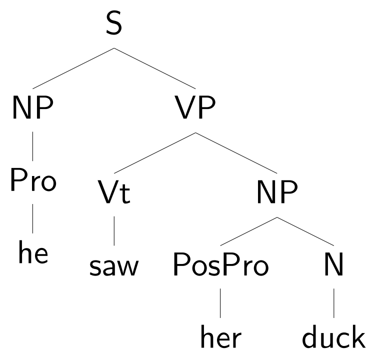
To show that a sentence is well-formed under this CFG, we must provide a parse. One way to do this is by drawing a tree:

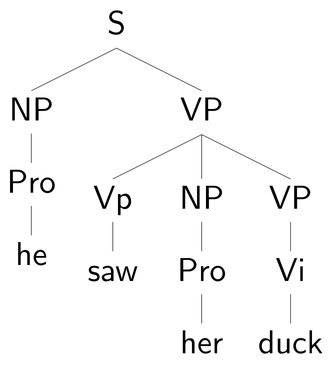


You can think of a tree like this as *proving* that its leaves are in the language generated by the grammar.

Structural Ambiguity

Some sentences have more than one parse: **structural ambiguity**.





Here, the **structural** ambiguity is caused by **POS** ambiguity in several of the words. (Both are types of **syntactic** ambiguity.)

Attachment ambiguity

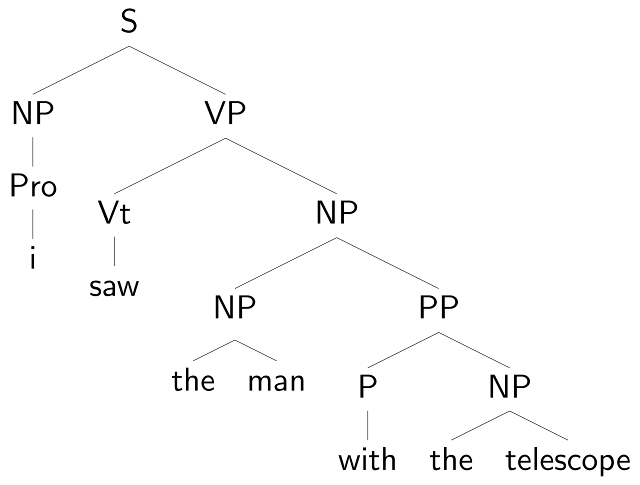
Some sentences have structural ambiguity even **without** part-of-speech ambiguity. This is called **attachment ambiguity**.

- Depends on where different phrases attach in the tree.
- Different attachments have different meanings:

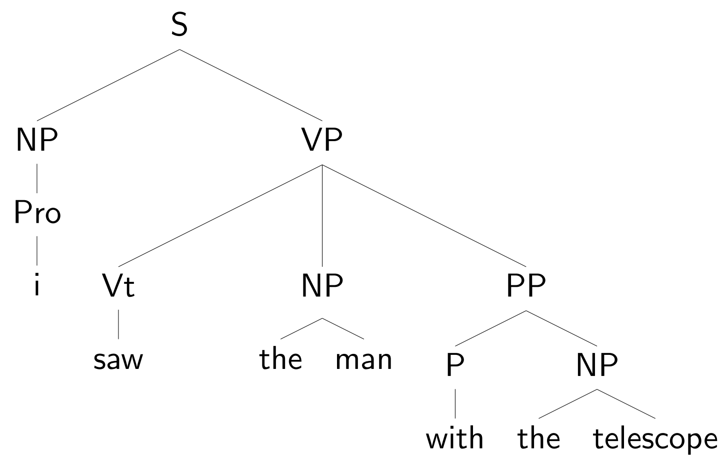
  I saw the man with the telescope
  She ate the pizza on the floor
  Good boys and girls get presents from Santa

- Next slides show trees for the first example: prepositional phrase (PP) attachment ambiguity. (Trees slightly abbreviated...)

Attachment ambiguity



Attachment ambiguity



Parsing algorithms

Goal: compute the structure(s) for an input string given a grammar.

- Ultimately, want to use the structure to interpret meaning.
- As usual, ambiguity is a huge problem.
  - For correctness: need to find the right structure to get the right meaning.
  - For efficiency: searching all possible structures can be very slow; want to use parsing for large-scale language tasks (e.g., used to create Google's "infoboxes").

Global and local ambiguity

- We've already seen examples of **global ambiguity**: multiple analyses for a full sentence, such as I saw the man with the telescope
- But **local ambiguity** is also a big problem: multiple analyses for parts of sentence.
  - the dog bit the child: first three words could be NP (but aren't).
  - Building useless partial structures wastes time.
  - Avoiding useless computation is a major issue in parsing.
- Syntactic ambiguity is rampant; humans usually don't even notice because we are good at using context/semantics to disambiguate.

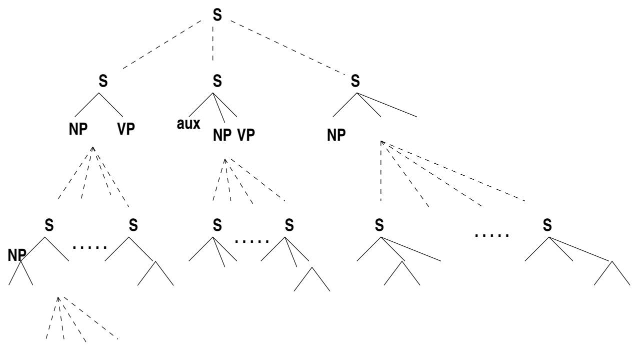
Parser properties

All parsers have two fundamental properties:

- **Directionality**: the sequence in which the structures are constructed.
  - **top-down**: start with root category (S), choose expansions, build down to words.
  - **bottom-up**: build subtrees over words, build up to S.
  - **Mixed** strategies also possible (e.g., left corner parsers)
- **Search strategy**: the order in which the search space of possible analyses is explored.

Example: search space for top-down parser

- Start with S node.
- Choose one of many possible expansions.
- Each of which has children with many possible expansions...
- etc



Search strategies

- **depth-first search**: explore one branch of the search space at a time, as far as possible. If this branch is a dead-end, parser needs to **backtrack.**
- **breadth-first search**: expand all possible branches in parallel (or simulated parallel). Requires storing many incomplete parses in memory at once.
- **best-first search**: score each partial parse and pursue the highest-scoring options first. (Will get back to this when discussing statistical parsing.)

Recursive Descent Parsing

- A **recursive descent** parser treats a grammar as a specification of how to break down a top-level goal (find `S`) into subgoals (find `NP VP`).
- It is a **top-down**, **depth-first** parser:
  - Blindly expand nonterminals until reaching a terminal (word).
  - If multiple options available, choose one but store current state as a backtrack point (in a **stack** to ensure depth-first.)
  - If terminal matches next input word, continue; else, backtrack.

RD Parsing algorithm

Start with subgoal = `S`, then repeat until input/subgoals are empty:

- If first subgoal in list is a **non-terminal** `A`, then pick an expansion $A \rightarrow B\ C$ from grammar and replace `A` in subgoal list with `B C`
- If first subgoal in list is a **terminal** `w`:
  - If input is empty, backtrack.
  - If next input word is different from `w`, backtrack.
  - If next input word is `w`, match! i.e., consume input word `w` and subgoal `w` and move to next subgoal.

If we run out of backtrack points but not input, no parse is possible.

Recursive descent example

Consider a very simple example:

- Grammar contains only these rules:

`S` $\rightarrow$ `NP VP`	`VP` $\rightarrow$ `V`	`NN` $\rightarrow$ `bit`	`V` $\rightarrow$ `bit`
`NP` $\rightarrow$ `DT NN`	`DT` $\rightarrow$ `the`	`NN` $\rightarrow$ `dog`	`V` $\rightarrow$ `dog`

- The input sequence is the dog bit

Recursive descent example

- Operations:
  - Expand (E)
  - Match (M)
  - Backtrack to step n (Bn)

Step	Op.	Subgoals	Input
0		`S`	the dog bit
1	E	`NP VP`	the dog bit
2	E	`DT NN VP`	the dog bit
3	E	`the NN VP`	the dog bit
4	M	`NN VP`	dog bit
5	E	`bit VP`	dog bit
6	B4	`NN VP`	dog bit
7	E	`dog VP`	dog bit
8	M	`VP`	bit
9	E	`V`	bit
10	E	`bit`	bit
11	M		

Further notes

- The above sketch is actually a **recognizer**: it tells us whether the sentence has a valid parse, but not what the parse is. For a parser, we'd need more details to store the structure as it is built.
- We only had one backtrack, but in general things can be much worse!
  - See Inf2a Lecture 17 for a much longer example showing inefficiency.
  - If we have left-recursive rules like `NP` $\rightarrow$ `NP PP`, we get an infinite loop!

Shift-Reduce Parsing

- Search strategy and directionality are orthogonal properties.
- **Shift-reduce** parsing is **depth-first** (like RD) but **bottom-up** (unlike RD).
- Basic shift-reduce recognizer repeatedly:
  - Whenever possible, **reduces** one or more items from top of stack that match RHS of rule, replacing with LHS of rule.
  - When that's not possible, **shifts** an input symbol onto a stack.
- Like RD parser, needs to maintain backtrack points.

Shift-reduce example

- Same example grammar and sentence.
- Operations:
  - Reduce (R)
  - Shift (S)
  - Backtrack to step n (Bn)
- Note that at 9 and 11 we skipped over backtracking to 7 and 5 respectively as there were actually no choices to be made at those points.

Step	Op.	Stack	Input
0			the dog bit
1	S	the	dog bit
2	R	DT	dog bit
3	S	DT dog	bit
4	R	DT V	bit
5	R	DT VP	bit
6	S	DT VP bit	
7	R	DT VP V	
8	R	DT VP VP	
9	B6	DT VP bit	
10	R	DT VP NN	
11	B4	DT V	bit
12	S	DT V bit	
13	R	DT V V	
14	R	DT V VP	
15	B3	DT dog	bit
16	R	DT NN	bit
17	R	NP	bit
...			

Depth-first parsing in practice

- Depth-first parsers are very efficient for unambiguous structures.
  - Widely used to parse/compile programming languages, which are constructed to be unambiguous.
- But can be massively inefficient (exponential in sentence length) if faced with local ambiguity.
  - Blind backtracking may require re-building the same structure over and over: so, simple depth-first parsers are not used in NLP.
  - But: if we use a probabilistic model to **learn** which choices to make, we can do very well in practice (coming next week...)

Breadth-first search using dynamic programming

- With a CFG, you should be able to avoid re-analysing any substring because its analysis is **independent** of the rest of the parse.

[he]$_{\mathrm{np}}$ [saw her duck]$_{\mathrm{vp}}$

- **chart parsing** algorithms exploit this fact.
  - use dynamic programming to store and reuse sub-parses, composing them into a full solution.
  - So multiple potential parses are explored at once: a breadth-first strategy.

Parsing as dynamic programming

- For parsing, subproblems are analyses of substrings, memoized in **chart** (aka **well-formed substring table**, WFST).
- Chart entries are indexed by *start* and *end* positions in the sentence, and correspond to:
  - either a complete **constituent** (sub-tree) spanning those positions (if working bottom-up),
  - **or** a **prediction** about what complete constituent might be found (if working top-down).

What's in the chart?

- We assume **indices** between each word in the sentence:

  $_0$ he $_1$ saw $_2$ her $_3$ duck $_4$

- The chart is a matrix where cell $[i, j]$ holds information about the word span from position i to position j:
  - The root node of any constituent(s) spanning those words
  - Pointers to its sub-constituents
  - (Depending on parsing method,) predictions about what constituents might follow the substring.

Algorithms for Chart Parsing

Many different chart parsing algorithms, including

- the **CKY algorithm**, which memoizes only complete constituents
- various algorithms that also memoize predictions/partial constituents
  - often using mixed bottom-up and top-down approaches, e.g., the Earley algorithm described in J&M, and left-corner parsing.