Recap: Syntax

Two reasons to care about syntactic structure (parse tree):

- As a guide to the semantic interpretation of the sentence
- As a way to prove whether a sentence is grammatical or not

But having a grammar isn’t enough.
We also need a parsing algorithm to compute the parse tree for a given input string and grammar.

Global and local ambiguity

- We’ve already seen examples of global ambiguity: multiple analyses for a full sentence, like *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.

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.)

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

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 → 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:
  
  \[
  \begin{align*}
  S & \rightarrow NP \ VP \\
  NP & \rightarrow DT \ NN \\
  VP & \rightarrow V \ NN \\
  V & \rightarrow bit \\
  NN & \rightarrow dog \\
  DT & \rightarrow the \\
  
  \end{align*}
  \]

• The input sequence is the dog bit

Operations:

– Expand (E)
– Match (M)
– Backtrack to step n (Bn)

<table>
<thead>
<tr>
<th>Step</th>
<th>Op.</th>
<th>Subgoals</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>S</td>
<td>the dog bit</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>NP VP</td>
<td>the dog bit</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>DT NN VP</td>
<td>the dog bit</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>the NN VP</td>
<td>the dog bit</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>NN VP</td>
<td>dog bit</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>bit VP</td>
<td>dog bit</td>
</tr>
<tr>
<td>6</td>
<td>B4</td>
<td>NN VP</td>
<td>dog bit</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>dog VP</td>
<td>dog bit</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
<td>VP</td>
<td>bit</td>
</tr>
<tr>
<td>9</td>
<td>E</td>
<td>V</td>
<td>bit</td>
</tr>
<tr>
<td>10</td>
<td>E</td>
<td>bit</td>
<td>bit</td>
</tr>
<tr>
<td>11</td>
<td>M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 → 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.

### 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...)

### Shift-reduce example

- Same example grammar and sentence.
- Operations:
  - Reduce (R)
  - Shift (S)
  - Backtrack to step \( n \) (\( B_n \))
- 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.

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

### 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.
  
  \[ \text{[he]_{np} [saw her duck]_{vp}} \]

- **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:
  \[ \begin{array}{c}
  0 & \text{he} & 1 & \text{saw} & 2 & \text{her} & 3 & \text{duck} & 4 \\
  \end{array} \]
- 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.

CKY Algorithm

CKY (Cocke, Kasami, Younger) is a bottom-up, breadth-first parsing algorithm.

- Original version assumes grammar in Chomsky Normal Form.
- Add constituent \(A\) in cell \((i,j)\) if there is:
  - a rule \(A \rightarrow B\), and a \(B\) in cell \((i,j)\), or
  - a rule \(A \rightarrow B \ C\), and a \(B\) in cell \((i,k)\) and a \(C\) in cell \((k,j)\).
CKY Algorithm

CKY (Cocke, Kasami, Younger) is a **bottom-up, breadth-first** parsing algorithm.

- Original version assumes grammar in Chomsky Normal Form.
- Add constituent $A$ in cell $(i,j)$ if there is:
  - a rule $A \rightarrow B$, and a $B$ in cell $(i,j)$, or
  - a rule $A \rightarrow B C$, and a $B$ in cell $(i,k)$ and a $C$ in cell $(k,j)$.
- Fills chart in order: only looks for rules that use a constituent from $i$ to $j$ after finding all constituents ending at $i$. So, guaranteed to find all possible parses.

Alex Lascarides  ANLP Lecture 12  20

CKY Pseudocode

- Assume input sentence with indices 0 to $n$, and chart $c$.
  
  ```plaintext
  for len = 1 to n:  #number of words in constituent
      for i = 0 to n-len:  #start position
          j = i+len #end position
          #process unary rules
          foreach $A \rightarrow B$ where $c[i,j]$ has $B$
              add $A$ to $c[i,j]$ with a pointer to $B$
          for $k = i+1$ to $j-1$  #mid position
              #process binary rules
              foreach $A \rightarrow B C$ where $c[i,k]$ has $B$ and $c[k,j]$ has $C$
                  add $A$ to $c[i,j]$ with pointers to $B$ and $C$
  ```
  
  - Takes time $O(Gn^3)$, where $G$ is the number of grammar rules.

Alex Lascarides  ANLP Lecture 12  21

CKY Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>POS ambiguities</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Pro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Vt,Vp,N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pro, PosPro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N,Vi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We’ve added all POSs that are allowed for each word.

Alex Lascarides  ANLP Lecture 12  22
CKY Example

Larger Constituents: Unary rule $NP \rightarrow Pro$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pro, NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Vt, Vp, N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pro, PosPro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N, Vi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• red shows which children create which parents.
• Normally we’d add pointers from parent to child to store this info permanently, but we’d end up with too many arrows here to see what’s going on!

Alex Lascarides ANLP Lecture 12 24

---

CKY Example

Larger Constituents: Binary rule $NP \rightarrow D \ N$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pro, NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Vt, Vp, N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pro, PosPro, D \ NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N, Vi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• red shows which children create which parents.
• Normally we’d add pointers from parent to child to store this info permanently, but we’d end up with too many arrows here to see what’s going on!

Alex Lascarides ANLP Lecture 12 26

---

CKY Example

Larger Constituents: Binary rule $VP \rightarrow Vt \ NP$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pro, NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Vt, Vp, N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pro, PosPro, D \ NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N, Vi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Vt from (1,2) plus NP from (2,4) makes a VP from (1,4).
• For cell (1,4) we also consider (1,3) plus (3,4) but there’s nothing in those cells that can combine to make a larger phrase.

Alex Lascarides ANLP Lecture 12 27
**A note about CKY ordering**

- Notice that to fill cell \((i,j)\), we use a cell from row \(i\) and a cell from column \(j\).
- So, we must fill in all cells down and left of \((i,j)\) before filling \((i,j)\).
- Here, we filled in all short entries, then longer ones, but other orders can work (e.g., J&M fill in all spans ending at \(j\), then increment \(j\).)

**CKY Example**

Larger Constituents: alternate parses

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pro, NP</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Vt, Vp, N</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pro, PosPro, D</td>
<td>NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N, Vi</td>
<td>he1</td>
<td>saw2</td>
<td>her3</td>
</tr>
</tbody>
</table>

- We also have another way to build the same VP (1,4). Add more pointers to remember this new analysis.
- (Not standard CKY because we used a ternary rule! In reality we would have converted this rule into CNF, but still ended up with two parses for VP.)

**CKY in practice**

- Avoids re-computing substructures, so much more efficient than depth-first parsers (in worst case).
- Still may compute a lot of unnecessary partial parses.
- Simple version requires converting the grammar to CNF (may cause blowup: remember time depends on grammar too!).

Various other chart parsing methods avoid these issues by combining top-down and bottom-up approaches (see J&M2).

But rather than going that way, next time we’ll focus on **statistical parsing** which can help with both ambiguity and efficiency issues.
Questions for review/practice

• What is the difference between global and local ambiguity? Construct example sentences illustrating each type of ambiguity (different from those given in the lecture).

• What are two examples of depth-first parsing algorithms, and what is the difference between them?

• Add a $VP \rightarrow V \ NP$ rule to the grammar on slide 10, and hand-simulate the different parsing strategies on the inputs the dog bit, the dog dog, and the dog bit the dog.

• What is the big disadvantage of (non-probabilistic) depth-first parsers? How does CKY avoid this disadvantage?