ANLP Lecture 14
Dependency syntax and parsing

Sharon Goldwater
(based on slides from Nathan Schneider)

19 October, 2017
Recap: lexicalized PCFGs

We saw that adding **lexical head** of the phrase can help choose the right parse:

![Parse Tree]

In fact, heads are so important that in some theories, they are primary.
Today’s Lecture

1. What is dependency syntax and why use it?
2. How can we transform constituency \( \rightarrow \) dependency parse?
3. What is a (non-)projective dependency structure and why does projectivity matter?
4. How can we parse sentences to get dependency trees?
Dependency syntax

An alternative approach to sentence structure.

- A fully lexicalized formalism: no phrasal categories.

- Assumes *binary, asymmetric* grammatical relations between words: **head-dependent** relations, shown as directed edges:

  - Here, edges point from heads to their dependents.
Dependency trees

A valid dependency tree for a sentence requires:

- A single distinguished root word.
- All other words have exactly one incoming edge.
- A unique path from the root to each other word.
It really is a tree!

- The usual way to show dependency trees is with edges over ordered sentences.

- But the edge structure (without word order) can also be shown as a more obvious tree:
**Labelled dependencies**

It is often useful to distinguish different kinds of head → modifier relations, by labelling edges:

- Historically, different treebanks/languages used different labels.
- Now, the **Universal Dependencies** project aims to standardize labels and annotation conventions, bringing together annotated corpora from over 50 languages.
- Labels in this example (and in textbook) are from UD.
Why dependencies??

Consider these sentences. Two ways to say the same thing:
Why dependencies??

Consider these sentences. Two ways to say the same thing:

- We only need a few phrase structure rules:
  
  \[
  S \rightarrow NP \ VP \\
  VP \rightarrow V \ NP \ NP \\
  VP \rightarrow V \ NP \ PP \\
  \]

  plus rules for NP and PP.
Equivalent sentences in Russian

- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).

- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke
Equivalent sentences in Russian

• Russian uses morphology to mark relations between words:
  – knigu means book (kniga) as a direct object.
  – devochke means girl (devochka) as indirect object (to the girl).

• So we can have the same word orders as English:
  – Sasha dal devochke knigu
  – Sasha dal knigu devochke

• But also many others!
  – Sasha devochke dal knigu
  – Devochke dal Sasha knigu
  – Knigu dal Sasha devochke
In languages with free word order, phrase structure (constituency) grammars don’t make as much sense.

- E.g., we would need both $S \rightarrow NP \ VP$ and $S \rightarrow VP \ NP$, etc. Not very informative about what’s really going on.
Phrase structure vs dependencies

- In languages with **free word order**, phrase structure (constituency) grammars don’t make as much sense.
  - E.g., we would need both $S \rightarrow NP \ VP$ and $S \rightarrow VP \ NP$, etc. Not very informative about what’s really going on.

- In contrast, the dependency relations stay constant:
Phrase structure vs dependencies

- Even more obvious if we just look at the trees without word order:
Pros and cons

- Sensible framework for free word order languages.

- Identifies syntactic relations directly. (using CFG, how would you identify the subject of a sentence?)

- Dependency pairs/chains can make good features in classifiers, for information extraction, etc.

- Parsers can be very fast (coming up...)

But

- The assumption of asymmetric binary relations isn’t always right... e.g., how to parse dogs and cats?
How do we annotate dependencies?

Two options:

1. Annotate dependencies directly.

2. Convert phrase structure annotations to dependencies. (Convenient if we already have a phrase structure treebank.)

Next slides show how to convert, assuming we have head-finding rules for our phrase structure trees.
Lexicalized Constituency Parse

S-saw
  NP-kids
    kids
  VP-saw
    V-saw
      saw
    NP-birds
      NP-birds
        birds
      PP-fish
        P-with
          with
        NP-fish
          fish
... remove the phrasal categories...
... remove the (duplicated) terminals. ...

```
  saw
 /   \
|     |
| kids |
|      |  saw
|      | /     \
|      | birds
|      /   \
| saw    fish
|       /   \
| birds
|     /   \
| with fish
```
... and collapse chains of duplicates. ...

```
      saw
     /   \
   kids   saw
         /   \
       saw   birds
             /   \
           birds   fish
               /   \
             with   fish
```
... and collapse chains of duplicates. . .

```
saw
  kids saw
    saw birds
      birds fish
      with
```
... and collapse chains of duplicates. ...
and collapse chains of duplicates.
and collapse chains of duplicates.
... done!

saw

kids   birds
     |   |
     fish
     |   |
     with
Constituency Tree $\rightarrow$ Dependency Tree

We saw how the **lexical head** of the phrase can be used to collapse down to a dependency tree:

```
S-saw
    /
   /   
NP-kids  VP-saw
        /
   kids
VP-saw
    /
  V-saw  NP-birds
   /
  saw  birds
PP-binoculars
    /
P-with  NP-binoculars
       /
   with  binoculars
```

- But how can we find each phrase’s head in the first place?
The standard solution is to use **head rules**: for every non-unary (P)CFG production, designate one RHS nonterminal as containing the head. $S \rightarrow NP\ VP$, $VP \rightarrow VP\ PP$, $PP \rightarrow P\ NP$ (content head), etc.

- Heuristics to scale this to large grammars: e.g., within an NP, last immediate N child is the head.
Head Rules

Then, propagate heads up the tree:

```
S
  NP-kids  VP
    kids        

VP
  V-saw  NP-birds
    saw        

PP
  P-with  NP-binoculars
    with  binoculars
```
Head Rules

Then, propagate heads up the tree:

\[
S \rightarrow \text{NP-kids} \rightarrow \text{kids} \rightarrow \text{VP-saw} \rightarrow \text{V-saw} \rightarrow \text{saw} \rightarrow \text{NP-birds} \rightarrow \text{birds} \rightarrow \text{PP} \rightarrow \text{P-with} \rightarrow \text{with} \rightarrow \text{NP-binoculars} \rightarrow \text{binoculars}
\]
Head Rules

Then, propagate heads up the tree:

S

NP-kids
  kids

VP

Saw
  NP-birds
    saw
    birds

P-with
  NP-binoculars
    with
    binoculars

Sharon Goldwater ANLP Lecture 15 29
Head Rules

Then, propagate heads up the tree:

$$S$$

- **NP-kids**
  - kids

- **VP-saw**
  - **VP-saw**
    - **V-saw**
      - saw
    - **NP-birds**
      - birds
  - **PP-binoculars**
    - **P-with**
      - with
    - **NP-binoculars**
      - binoculars
Then, propagate heads up the tree:

- **S**-saw
- **NP**-kids
  - kids
- **VP**-saw
  - **VP**-saw
  - **V**-saw
  - saw
  - **NP**-birds
  - birds
- **PP**-binoculars
  - **P**-with
  - with
  - **NP**-binoculars
  - binoculars
Projectivity

If we convert constituency parses to dependencies, all the resulting trees will be **projective**.

- Every subtree (node and all its descendants) occupies a *contiguous span* of the sentence.
- = the parse can be drawn over the sentence w/ no crossing edges.
But some sentences are **nonprojective**.

- We’ll only get these annotations right if we directly annotate the sentences (or correct the converted parses).
- Nonprojectivity is rare in English, but common in many languages.
- Nonprojectivity presents problems for parsing algorithms.
Some of the algorithms you have seen for PCFGs can be adapted to dependency parsing.

- **CKY** can be adapted, though efficiency is a concern: obvious approach is $O(Gn^5)$; Eisner algorithm brings it down to $O(Gn^3)$
  

- **Shift-reduce**: more efficient, doesn't even require a grammar!
Recall: shift-reduce parser with CFG

- 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.

<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></td>
<td>the dog bit</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>the</td>
<td>dog bit</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>DT</td>
<td>dog bit</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>DT dog</td>
<td>bit</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>DT V</td>
<td>bit</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>DT VP</td>
<td>bit</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td>DT VP bit</td>
<td>bit</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>B6</td>
<td>DT VP bit</td>
<td>bit</td>
</tr>
<tr>
<td>10</td>
<td>R</td>
<td>DT VP NN</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>S</td>
<td>DT V bit</td>
<td>bit</td>
</tr>
<tr>
<td>13</td>
<td>R</td>
<td>DT V V</td>
<td>bit</td>
</tr>
<tr>
<td>14</td>
<td>R</td>
<td>DT V VP</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>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Transition-based Dependency Parsing

The **arc-standard** approach parses input sentence \(w_1 \ldots w_N\) using two types of **reduce** actions (three actions altogether):

- **Shift**: Read next word \(w_i\) from input and push onto the stack.

- **LeftArc**: Assign head-dependent relation \(s_2 \leftarrow s_1\); pop \(s_2\)

- **RightArc**: Assign head-dependent relation \(s_2 \rightarrow s_1\); pop \(s_1\)

where \(s_1\) and \(s_2\) are the top and second item on the stack, respectively. (So, \(s_2\) *preceded* \(s_1\) in the input sentence.)
### Example

**Parsing** *Kim saw Sandy:*

<table>
<thead>
<tr>
<th>Step</th>
<th>←bot. Stack</th>
<th>top→</th>
<th>Word List</th>
<th>Action</th>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td></td>
<td>[Kim,saw,Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[root,Kim]</td>
<td></td>
<td>[saw,Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[root,Kim,saw]</td>
<td></td>
<td>[Sandy]</td>
<td>LeftArc</td>
<td>Kim←saw</td>
</tr>
<tr>
<td>3</td>
<td>[root,saw]</td>
<td></td>
<td>[Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[root,saw,Sandy]</td>
<td></td>
<td>[]</td>
<td>RightArc</td>
<td>saw→Sandy</td>
</tr>
<tr>
<td>5</td>
<td>[root,saw]</td>
<td></td>
<td>[]</td>
<td>RightArc</td>
<td>root→saw</td>
</tr>
<tr>
<td>6</td>
<td>[root]</td>
<td></td>
<td>[]</td>
<td>(done)</td>
<td></td>
</tr>
</tbody>
</table>

- Here, top two words on stack are also always adjacent in sentence. Not true in general! (See longer example in JM3.)
Labelled dependency parsing

- These parsing actions produce **unlabelled** dependencies (left).

- For **labelled** dependencies (right), just use more actions: LeftArc(NSUBJ), RightArc(NSUBJ), LeftArc(DOBJ), . . .
Differences to constituency parsing

- Shift-reduce parser for CFG: not all sequences of actions lead to valid parses. Choose incorrect action $\rightarrow$ may need to backtrack.

- Here, all valid action sequences lead to valid parses.
  - Invalid actions: can’t apply LeftArc with root as dependent; can’t apply RightArc with root as head unless input is empty.
  - Other actions may lead to incorrect parses, but still valid.

- So, parser doesn’t backtrack. Instead, tries to greedily predict the correct action at each step.
  - Therefore, dependency parsers can be very fast (linear time).
  - But need a good way to predict correct actions (next lecture).
Notions of validity

• In constituency parsing, valid parse = grammatical parse.
  – That is, we first define a grammar, then use it for parsing.

• In dependency parsing, we don’t normally define a grammar.
  – Valid parses are those with the properties on slide 4.
Summary: Transition-based Parsing

- **arc-standard** approach is based on simple shift-reduce idea.

- Can do labelled or unlabelled parsing, but need to train a **classifier** to predict next action, as we’ll see next time.

- Greedy algorithm means time complexity is linear in sentence length.

- Only finds **projective** trees (without special extensions)

- Pioneering system: Nivre’s **MALT**Parser.
Alternative: Graph-based Parsing

- Global algorithm: From the fully connected directed graph of all possible edges, choose the best ones that form a tree.

- **Edge-factored** models: Classifier assigns a nonnegative score to each possible edge; **maximum spanning tree** algorithm finds the spanning tree with highest total score in $O(n^2)$ time.

- Pioneering work: McDonald’s **MSTParser**

- Can be formulated as constraint-satisfaction with **integer linear programming** (Martins’s **TURBOParser**)

- Details in JM3, Ch 14.5 (optional).
**Graph-based vs. Transition-based vs. Conversion-based**

- **TB**: Features in scoring function can look at any part of the stack; no optimality guarantees for search; linear-time; (classically) projective only

- **GB**: Features in scoring function limited by factorization; optimal search within that model; quadratic-time; no projectivity constraint

- **CB**: In terms of accuracy, sometimes best to first constituency-parse, then convert to dependencies (e.g., *[Stanford Parser]*). Slower than direct methods.
Choosing a Parser: Criteria

- Target representation: constituency or dependency?
- Efficiency? In practice, both runtime and memory use.
- Incrementality: parse the whole sentence at once, or obtain partial left-to-right analyses/expectations?
- Retrainable system?
- Accuracy?
Summary

- Constituency syntax: hierarchically nested phrases with categories like NP.

- Dependency syntax: trees whose edges connect words in the sentence. Edges often labeled with relations like nsubj.

- Can convert constituency to dependency parse using head rules.

- For projective trees, transition-based parsing is very fast and can be very accurate.

- Google “online dependency parser”.
  Try out the Stanford parser and SEMAFOR!