NLP 202: Dependency Parsing

Jeffrey Flanigan

Winter 2023

University of California Santa Cruz
jmflanig@ucsc.edu

Plan for Today

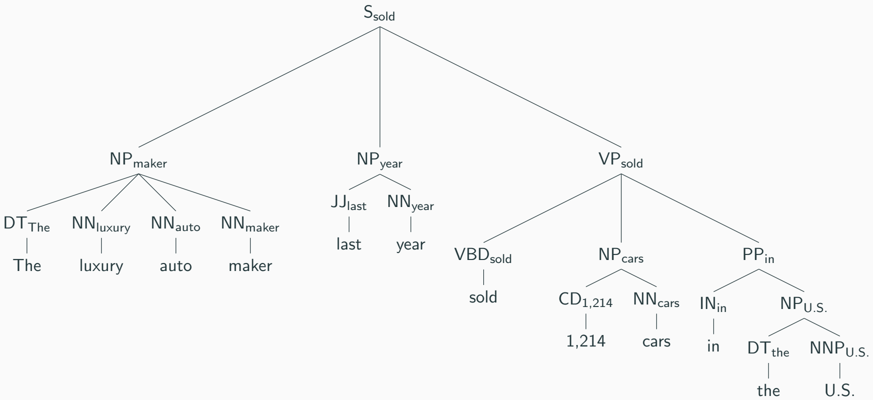
- Headness
- Dependencies and dependency trees
- Universal dependencies
- Transition-based dependency parsing
- Evaluation

Headedness

The **head** of a constituent is the main word for the phrase.

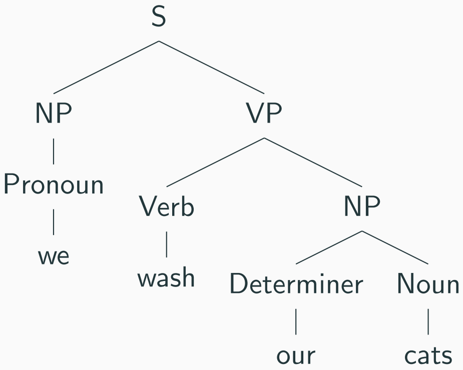
- John sees the [red truck]. **Head of “red truck” is truck**
- The [five quiet people] see the red truck. **Head of “five quiet people” is people**

Headness



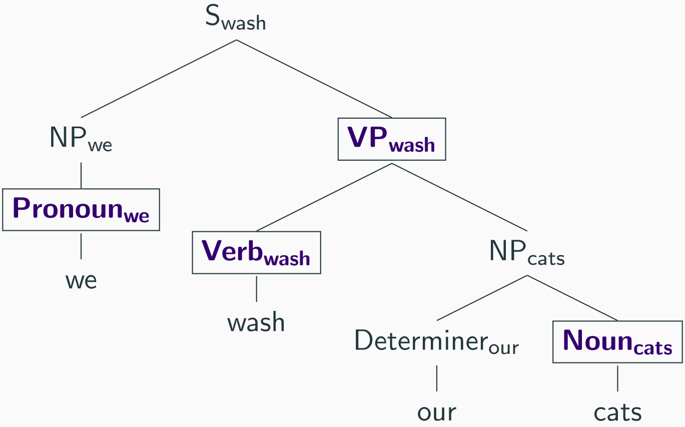
Each phrase has a **head** which is the “main” word of the phrase, which contains the important syntactic and semantic information of the phrase.

Example



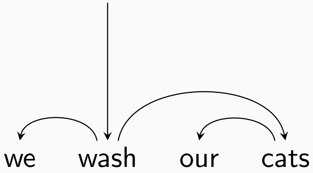
Phrase-structure tree.

Example


S_{wash}
NP_{we}
$\boxed{\textbf{VP}_{\textbf{wash}}}$
$\boxed{\textbf{Pronoun}_{\textbf{we}}}$
we
$\boxed{\textbf{Verb}_{\textbf{wash}}}$
wash
NP_{cats}
$Determiner_{our}$
our
$\boxed{\textbf{Noun}_{\textbf{cats}}}$
cats

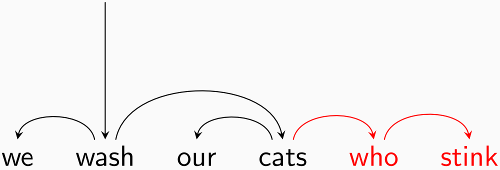
Phrase-structure tree with heads labeled.

Example



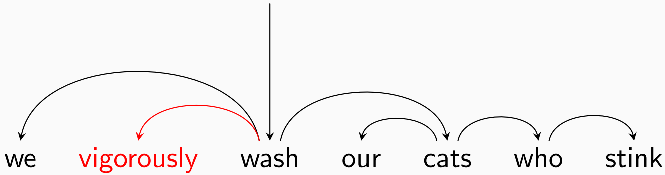
“Bare bones” dependency tree.

Example

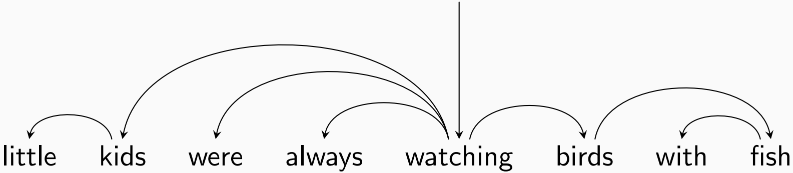


we wash our cats who stink

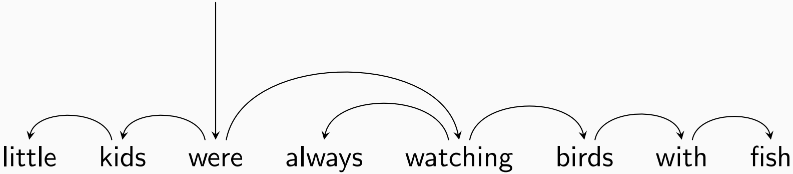
Example



Syntactic Heads vs. Semantic Heads

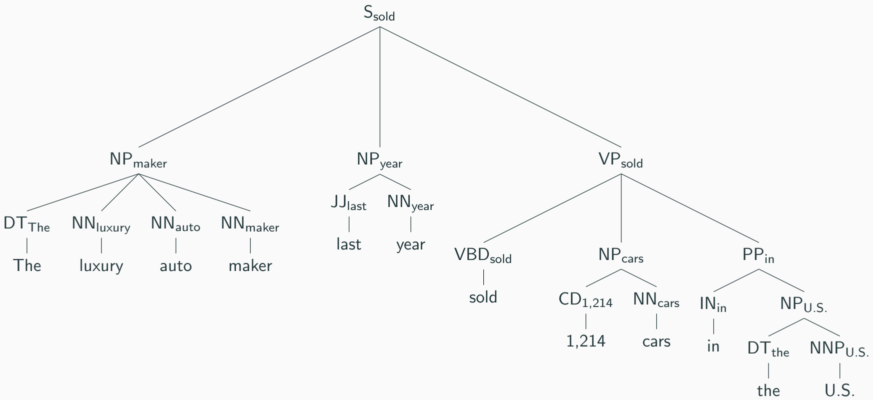


little kids were always watching birds with fish



little kids were always watching birds with fish

Headness



Each phrase has a **head** which is the "main" word of the phrase, which contains the important syntactic and semantic information of the phrase.

Dependency syntax

- Syntactic structure = asymmetric, binary relations between words.

Tesnier 1959; Nivre 2005

Dependency

- How do we decide which of a pair of words is the head and which is the dependent?

Dependency

- Many (conflicting) frameworks:
    - Head determines the syntactic category of a construction
    - Head is obligatory; dependents are optional
    - Head selects dependents and determines whether the dependent is required
    - The form of the dependent depends on the head (e.g., agreement between nouns/verbs, adjectives/nouns)
    - The linear position of a dependent is specified with respect to the head.

Trees

- A dependency structure is a directed graph G = (V,A) consisting of a set of vertices *V* and arcs *A* between them. Typically constrained to form a tree:
  - Single root vertex with no incoming arcs
  - Every vertex has exactly one incoming arc except root (single head constraint)
  - There is a unique path from the root to each vertex in V (acyclic constraint)

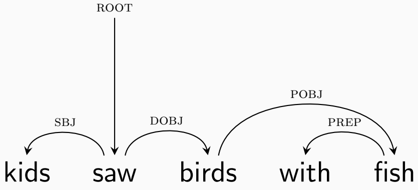
Dependency Tree: Definition

Let $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ be a sentence. Add a special $\textsc{root}$ symbol as "x_0."

A dependency tree consists of a set of tuples $\langle p, c, \ell \rangle$, where

- $p \in \{0, \ldots, n\}$ is the index of a parent
- $c \in \{1, \ldots, n\}$ is the index of a child
- $\ell \in \mathcal{Y}$ is a label
- The directed edges form an arborescence (directed tree) with x_0 as the root (sometimes denoted $\textsc{root}$).

Labels



Key dependency relations captured in the labels include: subject, direct object, preposition object, adjectival modifier, adverbial modifier.

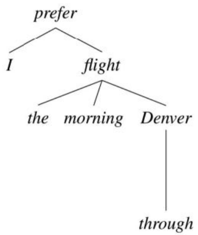
I sometimes won't include the labels to keep the algorithms simpler.

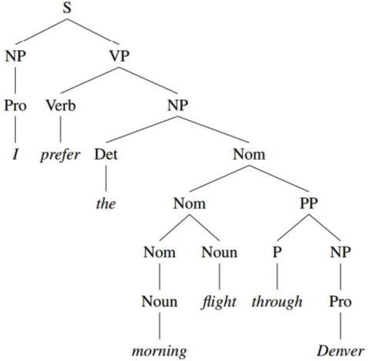
Dependency vs Constituency

- Constituency structures explicitly represent
  - Phrases (nonterminal nodes)
  - Structural categories (nonterminal labels)

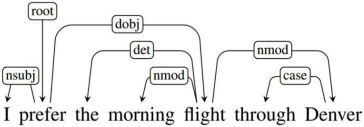
- Dependency structures explicitly represent
  - Head-dependent relations (directed arcs)
  - Functional categories (arc labels)
  - Possibly some structural categories (parts of speech)

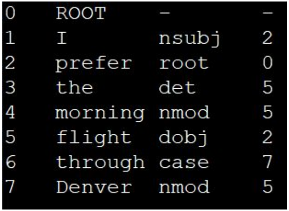
Dependency vs Constituency





Dependency Representation

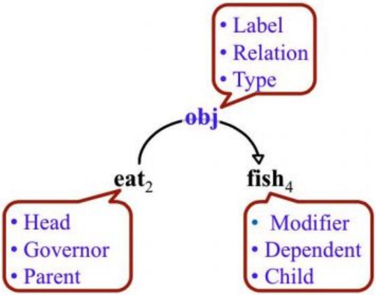




```
0    ROOT    -      -
1    I       nsubj  2
2    prefer  root   0
3    the     det    5
4    morning nmod   5
5    flight  dobj   2
6    through case   7
7    Denver  nmod   5
```

"CoNLL format"

Dependency Relations

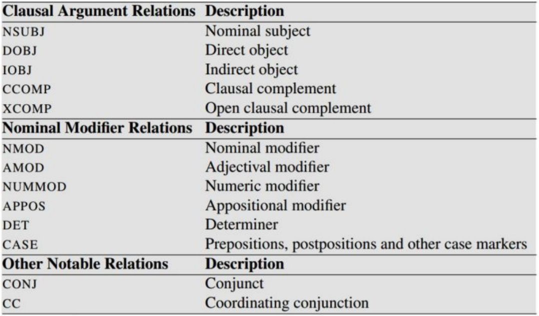


- Label
- Relation
- Type

- Head
- Governor
- Parent

- Modifier
- Dependent
- Child

Grammatical Functions

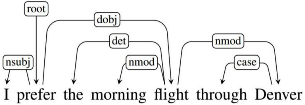


Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Selected dependency relations from the Universal Dependency Set

Dependency Constraints

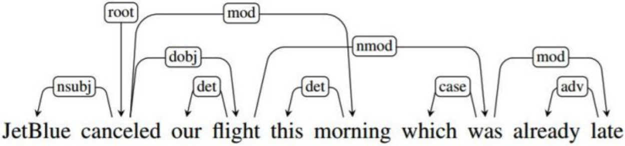
- Syntactic structure is complete (connectedness)
  - Connectedness can be enforced by adding a special root node
- Syntactic structure is hierarchical (acyclicity)
  - There is a unique pass from the root to each vertex
- Every word has at most one syntactic head (single-head constraint)
  - Except root that does not have incoming arcs



- This makes the dependencies a tree

Projectivity

- Projective parse
  - Arcs don’t across each other
  - Mostly true for English
- Non-projective structures are needed to account for
  - Long-distance dependencies
  - Flexible word order


root
mod
nmod
dobj
mod
nsubj
det
det
case
adv
JetBlue canceled our flight this morning which was already late

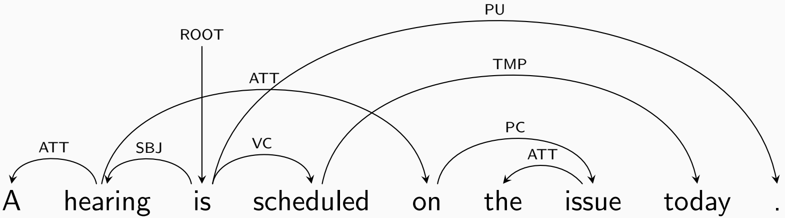
Projectivity

- Dependency grammars do not normally assume that all dependency-trees are projective, because some linguistic phenomena can only be achieved using non-projective trees.

- But a lot of parsers assume that the output trees are projective

- Reasons:
  - Conversion from constituency to dependency
  - The most widely used families of parsing algorithms impose projectivity

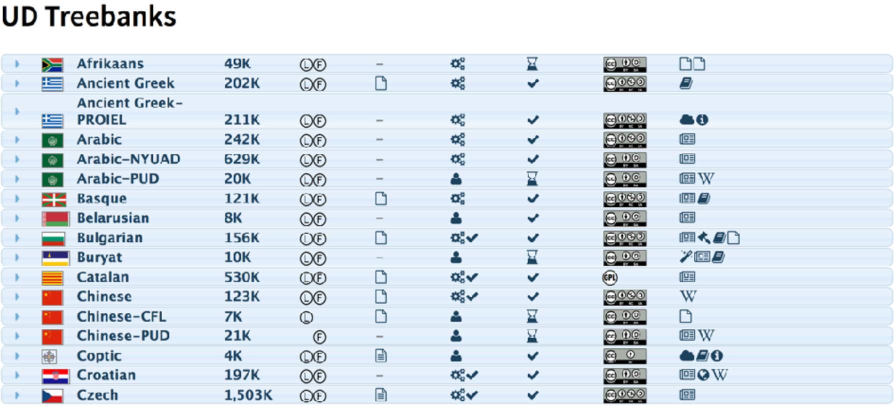
Nonprojective Example



Dependency Annotation

- Direct annotation.
- Transform the treebank: define “head rules” that can select the head child of any node in a phrase-structure tree and label the dependencies.
    - More powerful, less local rule sets, possibly collapsing some words into arc labels.
    - Stanford dependencies are a popular example (**?**).
    - Only results in projective trees.
- Rule based dependencies, followed by manual correction.

Universal Dependencies



UD Treebanks

▸	Afrikaans	49K	ⓁⒻ	–	⚙	⌛	cc BY SA	🗋🗋
▸	Ancient Greek	202K	ⓁⒻ	🗋	⚙	✓	cc BY NC SA	📘
▸	Ancient Greek-PROIEL	211K	ⓁⒻ	–	⚙	✓	cc BY NC SA	☁ℹ
▸	Arabic	242K	ⓁⒻ	–	⚙	✓	cc BY NC SA	📰
▸	Arabic-NYUAD	629K	ⓁⒻ	–	⚙	✓	cc BY SA	📰
▸	Arabic-PUD	20K	ⓁⒻ	–	👤	⌛	cc BY SA	📰W
▸	Basque	121K	ⓁⒻ	🗋	⚙	✓	cc BY NC SA	📰📘
▸	Belarusian	8K	ⓁⒻ	–	👤	✓	cc BY SA	📰
▸	Bulgarian	156K	ⓁⒻ	🗋	⚙✓	✓	cc BY NC SA	📰🔨📘🗋
▸	Buryat	10K	ⓁⒻ	–	👤	⌛	cc BY SA	🪄📰📘
▸	Catalan	530K	ⓁⒻ	🗋	⚙✓	✓	GPL	📰
▸	Chinese	123K	ⓁⒻ	🗋	⚙✓	✓	cc BY NC SA	W
▸	Chinese-CFL	7K	Ⓛ	🗋	👤	⌛	cc BY SA	🗋
▸	Chinese-PUD	21K	Ⓕ	–	👤	⌛	cc BY SA	📰W
▸	Coptic	4K	ⓁⒻ	🗎	👤	✓	cc BY	☁📘ℹ
▸	Croatian	197K	ⓁⒻ	–	⚙✓	✓	cc BY SA	📰🌐W
▸	Czech	1,503K	ⓁⒻ	🗎	⚙✓	✓	cc BY NC SA	📰

http://universaldependencies.org

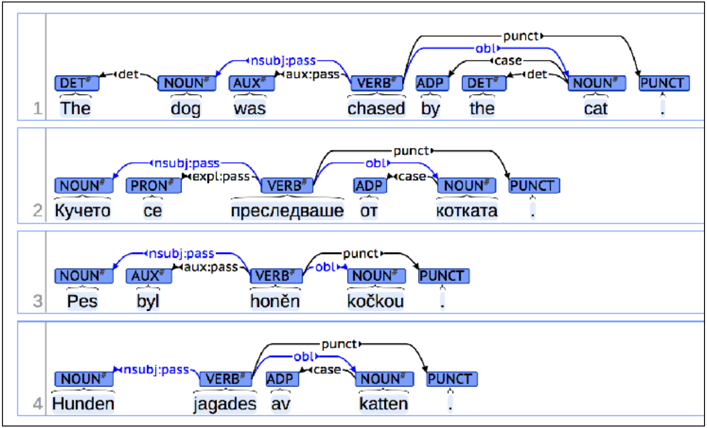
Universal Dependencies

- Developing cross-linguistically consistent treebank annotation for many languages

- Goals:
  - Facilitating multilingual parser development
  - Cross-lingual learning
  - Parsing research from a language typology perspective.

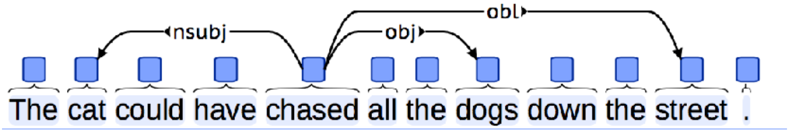
http://universaldependencies.org

Universal Dependencies



http://universaldependencies.org

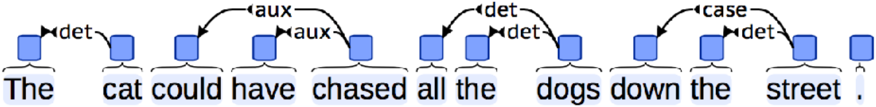
UD Principles



Dependency relations mainly hold between content words.

http://universaldependencies.org/u/overview/syntax.html

UD Principles

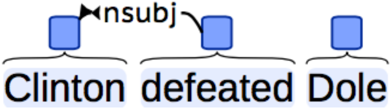


Function words dependent on closest related content word

http://universaldependencies.org/u/overview/syntax.html

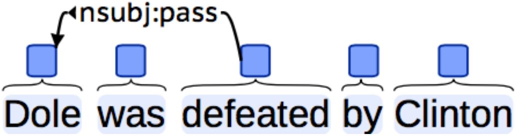
nsubj

- Syntactic subject of active verbs



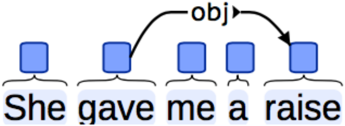
nsubj:pass

- Syntactic subject of passive verbs



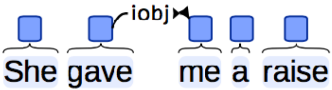
obj

- Generally, the entity that is acted upon as the direct object of the predicate.



iobj

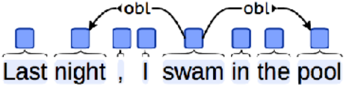
- Indirect object: recipients of ditransitive verbs of exchange (verbs requiring two objects)

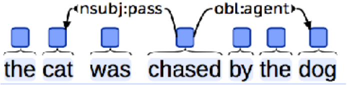


nsubj		iobj	obj
She	teaches	her daughters	math
She	told	her daughtesr	a story

obl

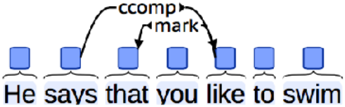
- Any nominal functioning as non-required argument or adjunct of a verb, including temporal and locational nominal modifiers and agents of passive verbs

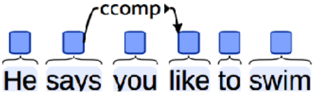




ccomp

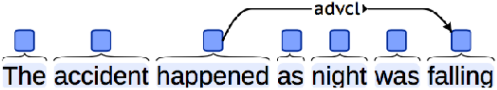
- Clausal complements, including dialogue

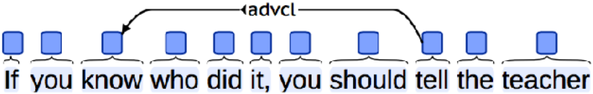




advcl

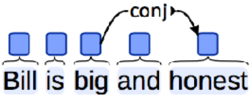
- A clause that modifies another predicate (temporal clauses, consequence, conditional clauses, purpose clauses)

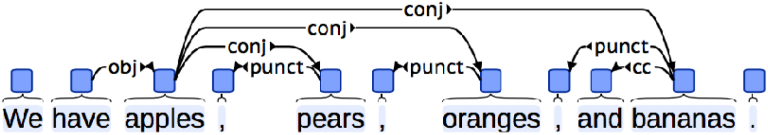




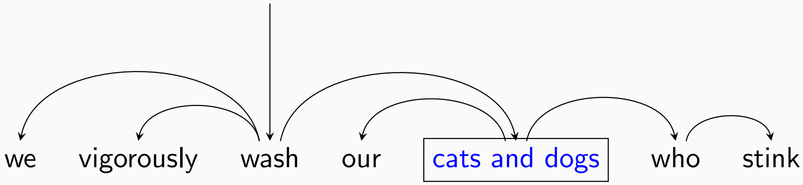
conj

- The elements that are coordinated; the head is the first conjunct



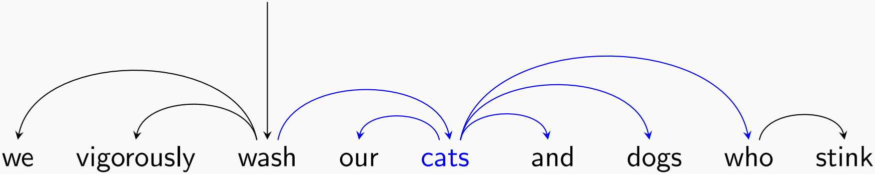


Aside: Coordination Structures



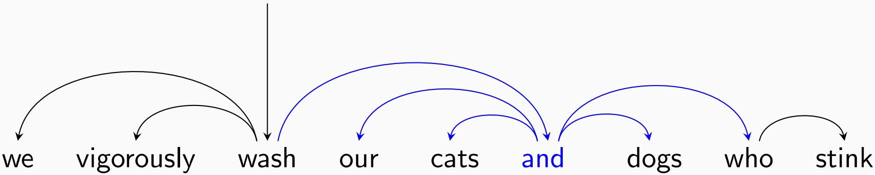
The bugbear of dependency syntax.

Coordination Structures



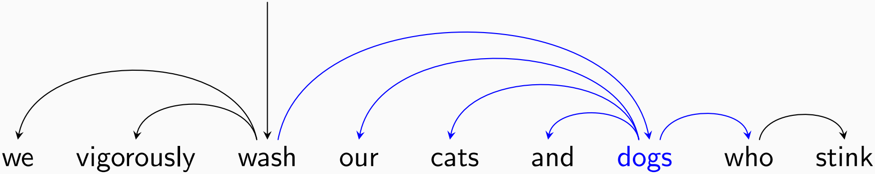
Make the first conjunct the head?

Coordination Structures



Make the coordinating conjunction the head?

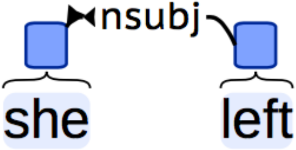
Coordination Structures



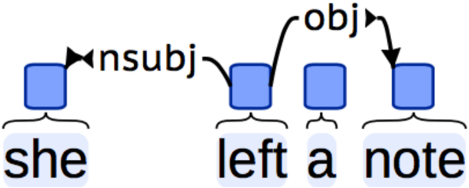
Make the second conjunct the head?

Universal dependencies: use the first conjunct as head

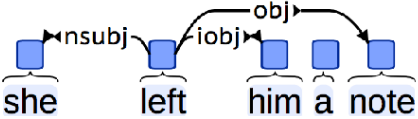
Intransitive verbs

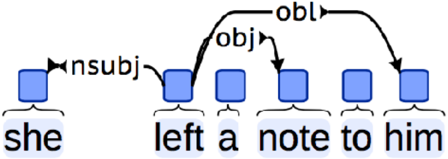


Transitive verbs

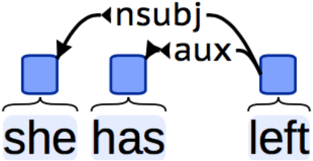


Ditransitive verbs



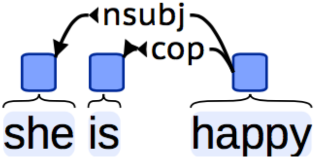


Aux



aux adds tense, aspect, mood, voice or evidentiality

cop



cop links a non-verbal predicate to subject

Approaches to Dependency Parsing

Today:

- **Transition-based parsing with a stack.**

In a couple weeks:

- Chu-Liu-Edmonds algorithm for arborescences (directed trees).
- Dynamic programming with the Eisner algorithm.

Transition-based parsing

- Basic idea: parse a sentence into a dependency by training a local classifier to predict a parser’s next action from its current configuration.

Transition-based Dependency Parsing

- We've seen transition-based parsing before: shift-reduce parsing for constituency parsing

Configuration

- Stack
- Input buffer of words
- Arcs in a parsed dependency tree
- Parsing = sequences of transitions through space of possible configurations

∅ book me the morning flight

stack	action	arc

∅ book me the morning flight

stack	action	arc

LeftArc(label): assert relation between head at stack$_1$ and dependent at stack$_{2:}$ remove stack$_2$

RightArc(label): assert relation between head at stack$_2$ and dependent at stack$_1$; remove stack$_1$

☞ Shift: Remove word from front of input buffer (∅) and push it onto stack book me the morning flight

stack		action	arc
		LeftArc(label): assert relation between head at stack$_1$ (∅) and dependent at stack$_2$: remove stack$_2$	
		RightArc(label): assert relation between head at stack$_2$ and dependent at stack$_1$ (∅); remove stack$_1$ (∅)	
∅	☞	Shift: Remove word from front of input buffer (book) and push it onto stack	

If we remove an element from the stack, it can't have any further dependents

me the morning flight

stack

book

∅

action

LeftArc(label): assert relation between head at stack$_1$ (book) and dependent at stack$_2$ (∅): remove stack$_2$ (∅)

RightArc(label): assert relation between head at stack$_2$ (∅) and dependent at stack$_1$ (book); remove stack$_1$ (book)

☞ Shift: Remove word from front of input buffer (me) and push it onto stack

arc

the morning flight

stack

me

book

∅

action

LeftArc(label): assert relation between head at stack$_1$ (me) and dependent at stack$_2$ (book): remove stack$_2$ (book)

☞ RightArc(label): assert relation between head at stack$_2$ (book) and dependent at stack$_1$ (me); remove stack$_1$ (me)

Shift: Remove word from front of input buffer (the) and push it onto stack

arc

iobj(book, me)

the morning flight

stack	action	arc
	LeftArc(label): assert relation between head at stack$_1$ (me) and dependent at stack$_2$ (book); remove stack$_2$ (book)	*iobj(book, me)*
book	RightArc(label): assert relation between head at stack$_2$ (book) and dependent at stack$_1$ (me); remove stack$_1$ (me)	
∅	Shift: Remove word from front of input buffer (the) and push it onto stack	

morning flight

stack

action

arc

LeftArc(label): assert relation between head at stack$_1$ (the) and dependent at stack$_2$ (book); remove stack$_2$ (book)

iobj(book, me)

RightArc(label): assert relation between head at stack$_2$ (book) and dependent at stack$_1$ (the); remove stack$_1$ (the)

the

book

∅

☞ Shift: Remove word from front of input buffer (morning) and push it onto stack flight

stack

morning

the

book

∅

action

LeftArc(label): assert relation between head at stack$_1$ (morning) and dependent at stack$_2$ (the); remove stack$_2$ (the)

RightArc(label): assert relation between head at stack$_2$ (the) and dependent at stack$_1$ (morning); remove stack$_1$ (morning)

☞ Shift: Remove word from front of input buffer (flight) and push it onto stack

arc

iobj(book, me)

stack

flight

morning

the

book

∅

action

☞ LeftArc(label): assert relation between head at stack$_1$ (flight) and dependent at stack$_2$ (morning): remove stack$_2$ (morning)

RightArc(label): assert relation between head at stack$_2$ (morning) and dependent at stack$_1$ (flight); remove stack$_1$ (flight)

~~Shift: Remove word from front of input buffer and push it onto stack~~

arc

iobj(book, me)

nmod(flight, morning)

stack

flight

the

book

∅

action

☞ LeftArc(label): assert relation between head at stack$_1$ (flight) and dependent at stack$_2$ (the): remove stack$_2$ (the)

RightArc(label): assert relation between head at stack$_2$ (the) and dependent at stack$_1$ (flight); remove stack$_1$ (flight)

~~Shift: Remove word from front of input buffer and push it onto stack~~

arc

iobj(book, me)

nmod(flight, morning)

det(flight, the)

stack

flight

book

∅

action

LeftArc(label): assert relation between head at stack$_1$ (flight) and dependent at stack$_2$ (book): remove stack$_2$ (book)

☞ RightArc(label): assert relation between head at stack$_2$ (book) and dependent at stack$_1$ (flight); remove stack$_1$ (flight)

~~Shift: Remove word from front of input buffer and push it onto stack~~

arc

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

This is our parse

stack

book

∅

action

LeftArc(label): assert relation between head at stack$_1$ (book) and dependent at stack$_2$ (∅): remove stack$_2$ (∅)

☞ RightArc(label): assert relation between head at stack$_2$ (∅) and dependent at stack$_1$ (book); remove stack$_1$ (book)

~~Shift: Remove word from front of input buffer and push it onto stack~~

arc

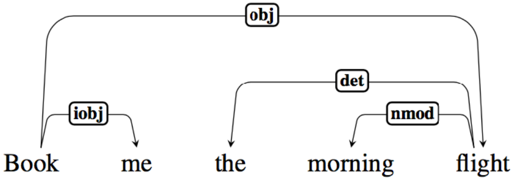
iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)



This is our parse

arc
iobj(book, me)
nmod(flight, morning)
det(flight, the)
obj(book, flight)
root(∅, book)

Output space $\mathcal{Y}$ =

Shift
LeftArc(nsubj)
RightArc(nsubj)
LeftArc(det)
RightArc(det)
LeftArc(obj)
RightArc(obj)
…

- This is a multi class classification problem: given the current configuration — i.e., the elements in the stack, the words in the buffer, and the arcs created so far, what’s the best transition?

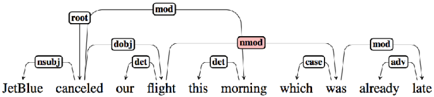
Training

We're training to predict the parser action (Shift, RightArc, LeftArc) given the featurized configuration

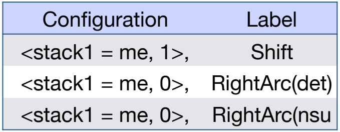
Configuration features	Label
<stack1 = me, 1>, <stack2 = book, 1>, <stack1 POS = PRP, 1>, <buffer1 = the, 1>,	Shift
<stack1 = me, 0>, <stack2 = book, 0>, <stack1 POS = PRP, 0>, <buffer1 = the, 0>,	RightArc(det)
<stack1 = me, 0>, <stack2 = book, 1>, <stack1 POS = PRP, 0>, <buffer1 = the, 0>,	RightArc(nsubj)

Oracle

- An algorithm for converting a gold-standard dependency tree into a series of actions a transition-based parser should follow to yield the tree.

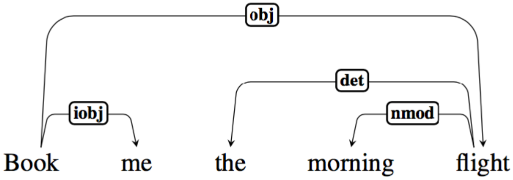


→



Configuration	Label
<stack1 = me, 1>,	Shift
<stack1 = me, 0>,	RightArc(det)
<stack1 = me, 0>,	RightArc(nsu

How to construct the oracle



This is our parse

arc

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

∅ book me the morning flight

stack	action	gold tree
		iobj(book, me)
		nmod(flight, morning)
		det(flight, the)
		obj(book, flight)
		root(∅, book)

∅ book me the morning flight

stack

action

gold tree

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

root(∅, book) exists but book has dependents in gold tree!

k me the morning flight

stack

∅

action

Choose LeftArc(label) if label(stack_1,stack_2) exists in gold tree. Remove stack_2.

Else choose RightArc(label) if label(stack_2, stack_1) exists in gold tree and all arcs label(stack_1, *). have been generated. Remove stack_1

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

iobj(book, me) exists and me has no dependents in gold tree

me the morning flight

stack

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

the morning flight

stack

me

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✓ *iobj(book, me)*

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

morning flight

stack

the

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✓ *iobj(book, me)*

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

flight

stack

morning

the

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✓ *iobj(book, me)*

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

nmod(flight,morning)

stack

flight

morning

the

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✓ *iobj(book, me)*

✓ *nmod(flight, morning)*

det(flight, the)

obj(book, flight)

root(∅, book)

det(flight,the)

stack

flight

the

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✅ *iobj(book, me)*

✅ *nmod(flight, morning)*

✅ *det(flight, the)*

obj(book, flight)

root(∅, book)

obj(book,flight)

stack

flight

book

∅

action

Choose LeftArc(label) if label(stack$_{1}$,stack$_{2}$) exists in gold tree. Remove stack$_{2}$.

Else choose RightArc(label) if label(stack$_{2}$, stack$_{1}$) exists in gold tree and all arcs label(stack$_{1}$, *). have been generated. Remove stack$_{1}$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✅ *iobj(book, me)*

✅ *nmod(flight, morning)*

✅ *det(flight, the)*

✅ *obj(book, flight)*

root(∅, book)

root(∅, book) *and* book has no more dependents we haven't seen

stack

book

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

✅ *iobj(book, me)*

✅ *nmod(flight, morning)*

✅ *det(flight, the)*

✅ *obj(book, flight)*

✅ *root(∅, book)*

With only ∅ left on the stack and nothing in the buffer, we're done

stack

∅

action

Choose LeftArc(label) if label(stack$_1$,stack$_2$) exists in gold tree. Remove stack$_2$.

Else choose RightArc(label) if label(stack$_2$, stack$_1$) exists in gold tree and all arcs label(stack$_1$, *). have been generated. Remove stack$_1$

Else shift: Remove word from front of input buffer and push it onto stack

gold tree

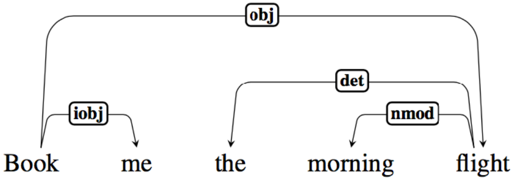
✅ *iobj(book, me)*

✅ *nmod(flight, morning)*

✅ *det(flight, the)*

✅ *obj(book, flight)*

✅ *root(∅, book)*



Shift

Shift

Shift

RightArc(iobj)

Shift

Shift

Shift

LeftArc(nmod)

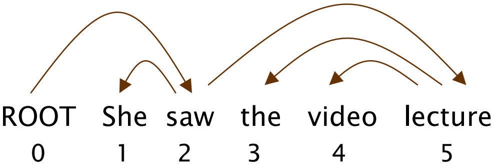
LeftArc(det)

RightArc(obj)

RightArc(root)

Evaluation

Evaluation of Dependency Parsing: (labeled) dependency accuracy



$$\text{Acc} = \frac{\text{\# correct deps}}{\text{\# of deps}}$$

UAS = 4 / 5 = 80%

LAS = 2 / 5 = 40%

Gold

1	2	She	nsubj
2	0	saw	root
3	5	the	det
4	5	video	nn
5	2	lecture	obj

Parsed

1	2	She	nsubj
2	0	saw	root
3	4	the	det
4	5	video	nsubj
5	2	lecture	ccomp