

LLyfr cyntaf Moses yr hwn a elwir GENESIS.

PENNOD I.

Creadwriaeth y nef, a'r ddaiar, 2 Y goleuni a'r ty-
wyllwch, 8 Y ffurfafen, 16 Y pysc, yr adar, a'r ani-
feiliaid, 26 A dyn. 29 LLynnfaeth dyn ac anifail.



12 y dechreuad y * cre-
awdd Duw y nefoedd
a'r ddaiar,

2 Y ddaiar oedd af-

wrth eu rhywogaeth : a Duw a welodd mai da
oedd,

13 Felly yz hwy a fu, a'r borau a fu, y try-
dydd dydd.

14 Duw hefyd a ddywedodd, * bydded psal.136.7.
goleuadau yn ffurfafen y nefoedd i wahanu deut.4.19.
rhwng y dydd a'r nôs : a byddant yn arwydd-
ion, ac yn dymmozau, ac yn ddyddiau, a bly.

Natural Language Processing

Info 159/259

Lecture 17: Dependency parsing (March 19, 2020)

David Bamman, UC Berkeley

Announcements

- Project midterm reports due Tuesday, Mar. 31
- Spring break next week — no office hours; expect reduced Piazza response times
- Quiz 8 out today, due after spring break (3/31)
- Homework 5 out today, due Monday April 6 (2.5 weeks from today)

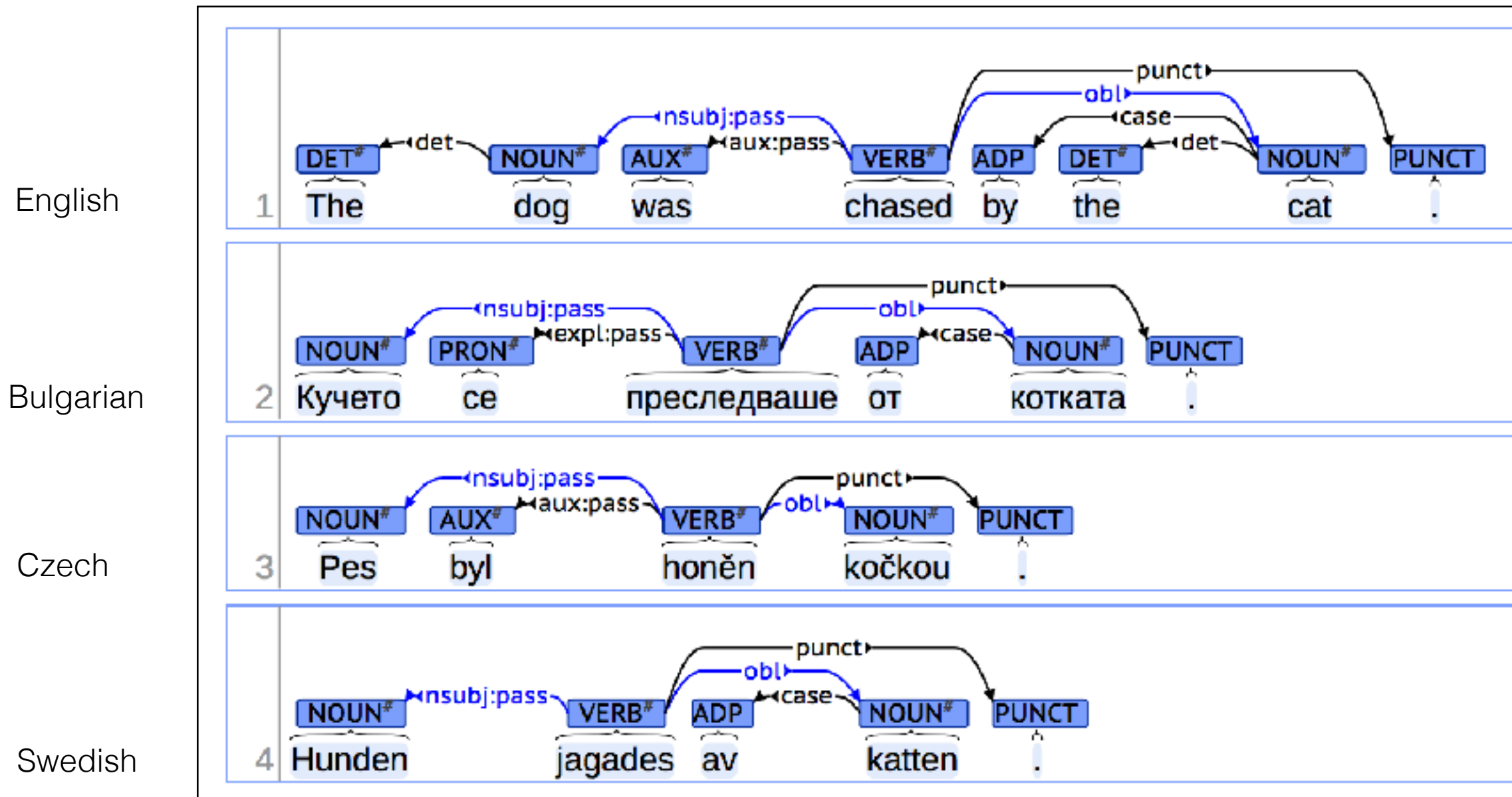
Dependency syntax

- Syntactic structure = *asymmetric*, *binary* relations between words.

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**)

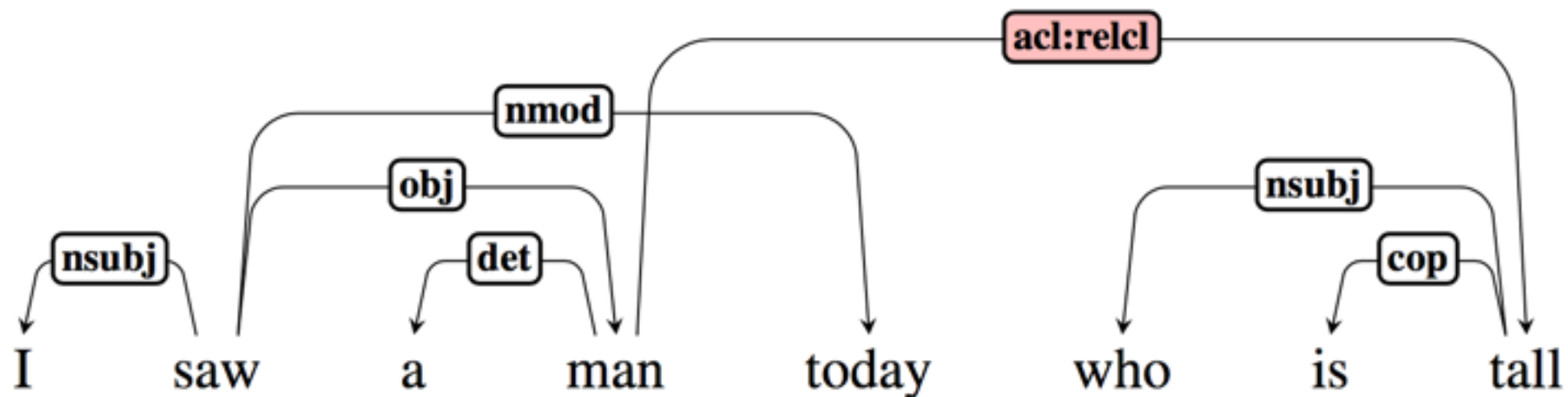
Universal Dependencies



Dependency parsing

- Transition-based parsing
 - $O(n)$
 - Only projective structures (pseudo-projective [Nivre and Nilsson 2005])
- Graph-based parsing
 - $O(n^2)$
 - Projective and non-projective trees

Projectivity



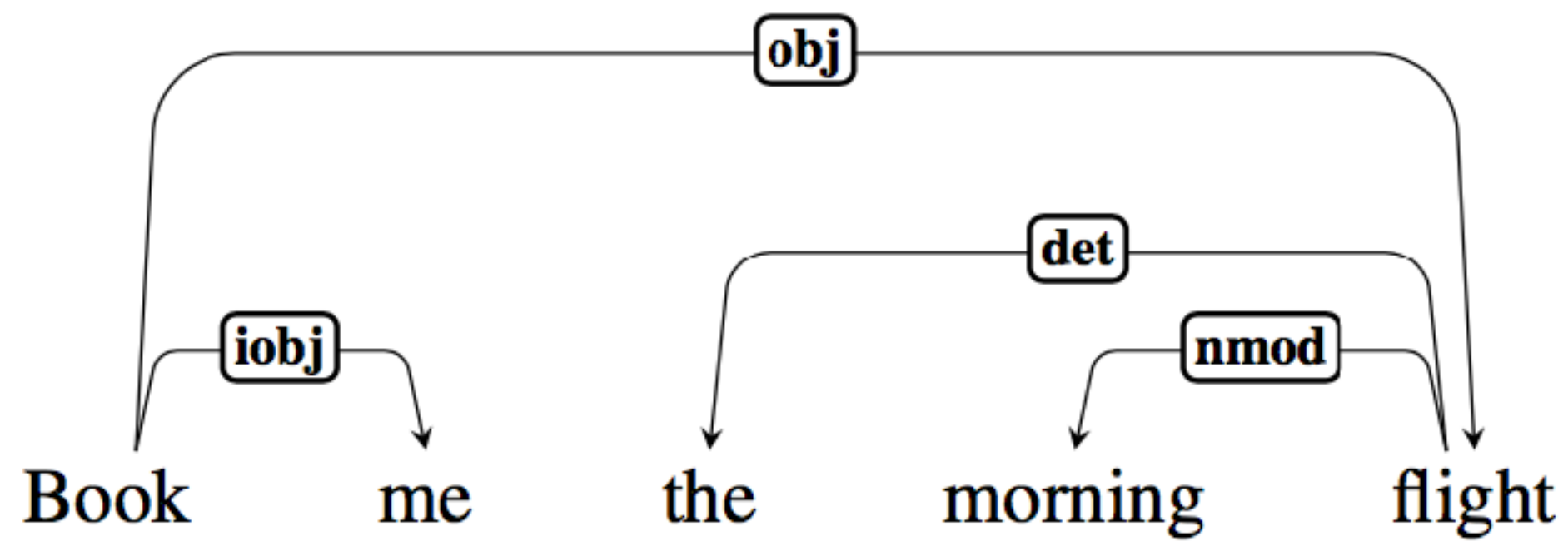
- An arc between a head and dependent is projective if there is a path from the head to every word between the head and dependent. Every word between head and dependent is a descendent of the head.

Transition-based parsing

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

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 ($\emptyset$)
and dependent at stack_2 :
remove stack_2

RightArc(label): assert
relation between head at
 stack_2 and dependent at
 stack_1 ($\emptyset$); remove stack_1 ($\emptyset$)

$\emptyset$



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

action

arc

book

∅

LeftArc(label): assert relation between head at stack₁ (book) and dependent at stack₂ (∅); remove stack₂ (∅)

RightArc(label): assert relation between head at stack₂ (∅) and dependent at stack₁ (book); remove stack₁ (book)



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

the morning flight

stack

action

arc

iobj(book, me)

LeftArc(label): assert relation
between head at stack₁ (me)
and dependent at stack₂
(book); remove stack₂ (book)

me



RightArc(label): assert
relation between head at
stack₂ (book) and
dependent at stack₁ (me);
remove stack₁ (me)

book

∅

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

the morning flight

stack

action

arc

iobj(book, me)

LeftArc(label): assert relation between head at stack₁ (book) and dependent at stack₂ (∅); remove stack₂ (∅)

RightArc(label): assert relation between head at stack₂ (∅) and dependent at stack₁ (book); remove stack₁ (book)

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

book

∅

morning flight

stack

action

arc

the

book

∅

LeftArc(label): assert relation
between head at stack₁ (the)
and dependent at stack₂
(book): remove stack₂ (book)

RightArc(label): assert
relation between head at
stack₂ (book) and
dependent at stack₁ (the);
remove stack₁ (the)



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

iobj(book, me)

flight

stack

action

arc

morning

the

book

∅

LeftArc(label): assert relation between head at stack₁ (morning) and dependent at stack₂ (the): remove stack₂ (the)

RightArc(label): assert relation between head at stack₂ (the) and dependent at stack₁ (morning); remove stack₁ (morning)



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

iobj(book, me)

stack		action	arc
flight		LeftArc(label): assert relation between head at stack ₁ (flight) and dependent at stack ₂ (morning); remove stack ₂ (morning)	<i>iobj(book, me)</i>
morning			<i>nmod(flight, morning)</i>
the		RightArc(label): assert relation between head at stack ₂ (morning) and dependent at stack ₁ (flight); remove stack ₁ (flight)	
book			
∅		~~Shift: Remove word from front of input buffer and push it onto stack~~	

stack		action	arc
flight		LeftArc(label): assert relation between head at stack ₁ (flight) and dependent at stack ₂ (the); remove stack ₂ (the)	<i>iobj(book, me)</i> <i>nmod(flight, morning)</i> <i>det(flight, the)</i>
the		RightArc(label): assert relation between head at stack ₂ (the) and dependent at stack ₁ (flight); remove stack ₁ (flight)	
book			
∅		~~Shift: Remove word from front of input buffer and push it onto stack~~	

stack		action	arc
flight		LeftArc(label): assert relation between head at stack ₁ (flight) and dependent at stack ₂ (book); remove stack ₂ (book)	<i>iobj(book, me)</i> <i>nmod(flight, morning)</i> <i>det(flight, the)</i>
book	☞	RightArc(label): assert relation between head at stack ₂ (book) and dependent at stack ₁ (flight); remove stack ₁ (flight)	<i>obj(book, flight)</i>
∅		~~Shift: Remove word from front of input buffer and push it onto stack~~	

This is our parse

stack

action

arc

LeftArc(label): assert relation between head at stack₁ (book) and dependent at stack₂ (∅); remove stack₂ (∅)

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)



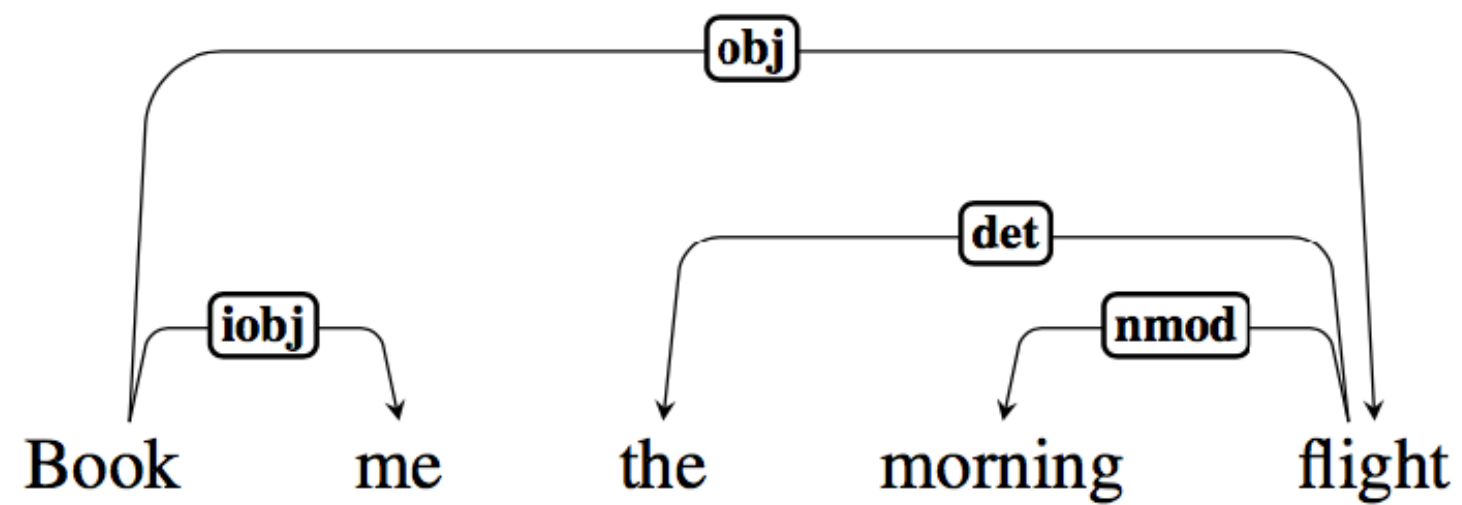
RightArc(label): assert relation between head at stack₂ (∅) and dependent at stack₁ (book); remove stack₁ (book)

book

∅

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

This is our parse



arc

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root(∅, book)

Let's go back to this earlier configuration

the morning flight

stack

action

arc

me

book

∅

LeftArc(label): assert relation between head at stack₁ (me) and dependent at stack₂ (book); remove stack₂ (book)

RightArc(label): assert relation between head at stack₂ (book) and dependent at stack₁ (me); remove stack₁ (me)

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

Output space $\mathbf{y} =$

- 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?

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

Features are scoped over the stack,
buffer, and arcs created so far

stack

me

book

buffer

the morning flight

arc

feature	example
stack ₁ = me	1
stack ₂ = book	1
stack ₁ POS = PRP	1
buffer ₁ = the	1
buffer ₂ = morning	1
buffer ₁ = today	0
buffer ₁ POS = RB	0
stack ₁ = me AND stack ₂ = book	1
stack ₁ = PRP AND stack ₂ = VB	1
iobj(book,*) in arcs	0

Use any multiclass classification model

- Logistic regression
- SVM
- NB
- Neural network

feature	example	β
stack ₁ = me	1	0.7
stack ₂ = book	1	1.3
stack ₁ POS = PRP	1	6.4
buffer ₁ = the	1	-1.3
buffer ₂ = morning	1	-0.07
buffer ₁ = today	0	0.52
buffer ₁ POS = RB	0	-2.1
stack ₁ = me AND stack ₂ =	1	0
stack ₁ = PRP AND stack ₂ =	1	-0.1
iobj(book,*) in arcs	0	3.2

Training

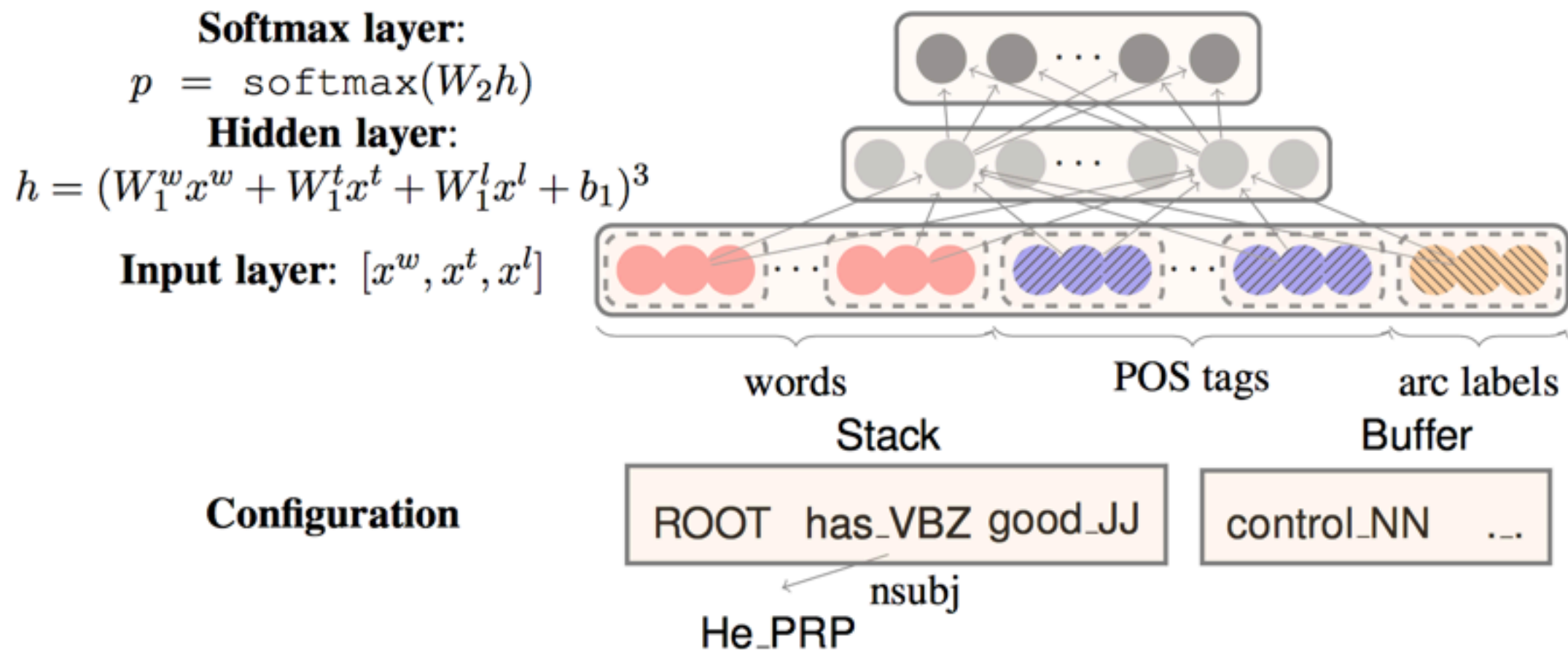
We're training to predict the parser action
—Shift, RightArc(label), LeftArc(label)—
given the featurized configuration

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)

Neural Shift-Reduce Parsing

- We can train a neural shift-reduce parser by just changing how we:
 - represent the configuration
 - predict the label from that representation
- Otherwise training and prediction remains the same.

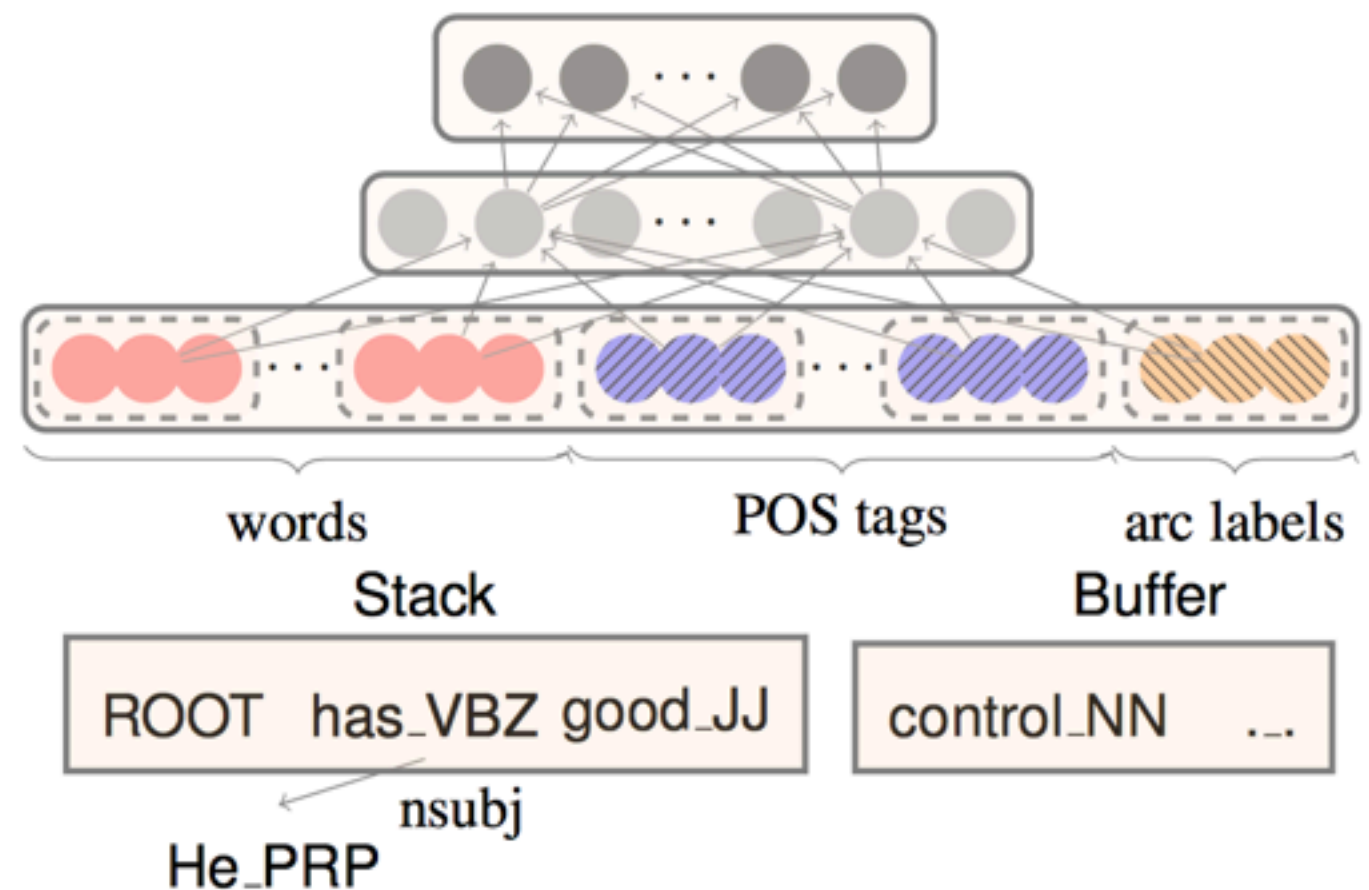
Neural Shift-Reduce Parsing



Neural Shift-Reduce Parsing

Representation for configuration:

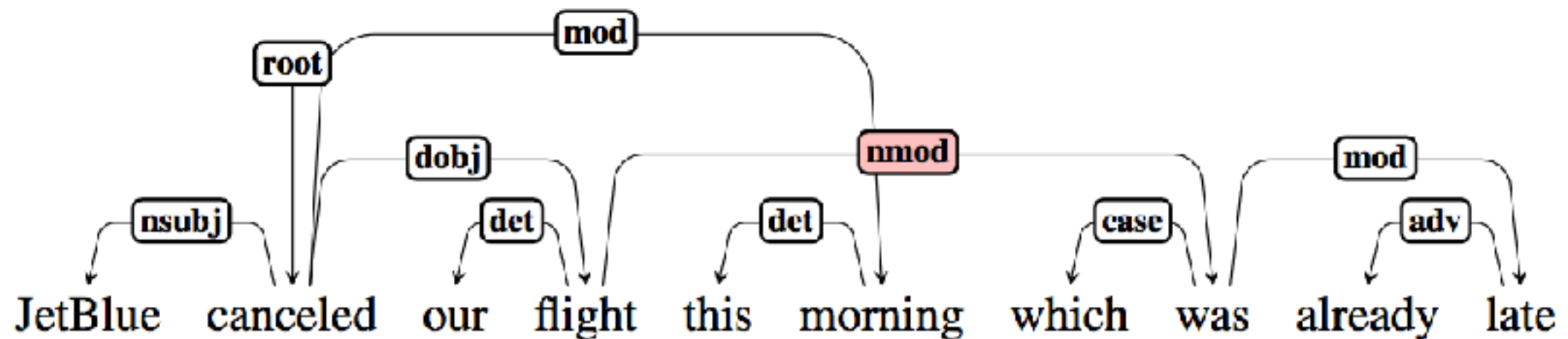
- Embeddings for words/POS tags on top of stack
- Embeddings for words/POS tags at front of buffer
- Embeddings for existing arc labels at specific positions



Classifier:

- Feed-forward neural network (input representation has a fixed dimensionality)

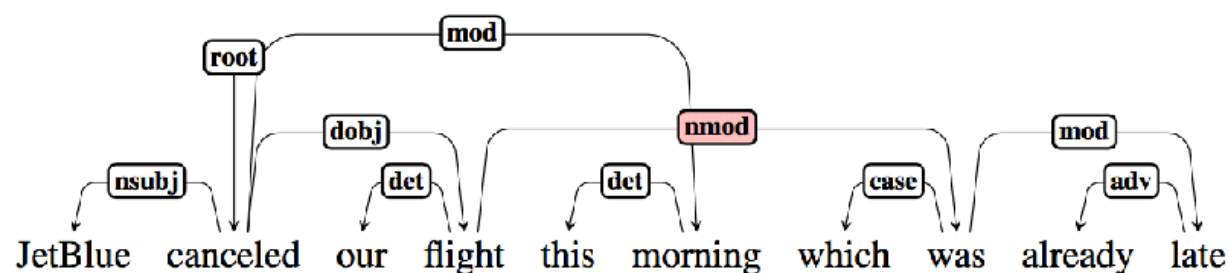
Training data



Our training data comes from treebanks
(native dependency syntax or converted to
dependency trees).

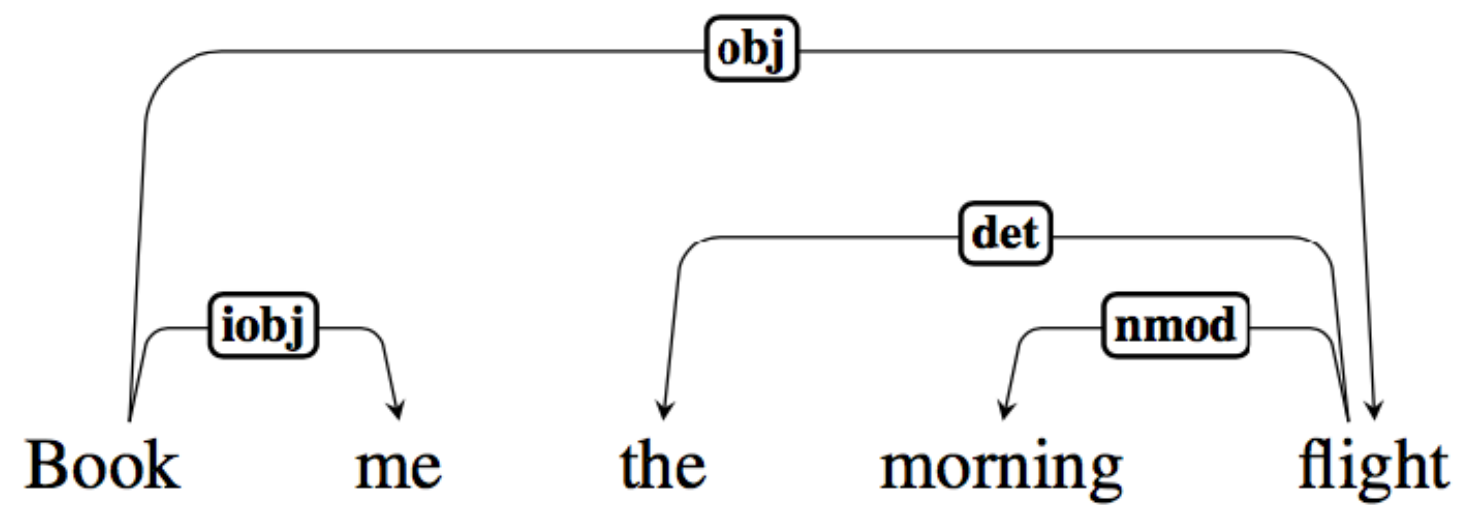
Oracle

- 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

This is our parse



arc

iobj(book, me)

nmod(flight, morning)

det(flight, the)

obj(book, flight)

root($\emptyset$, 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(stack₁, stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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($\emptyset$, book) exists but book
has dependents in gold tree!

book me the morning flight

stack

action

gold tree

Choose LeftArc(label) if
 $\text{label}(\text{stack}_1, \text{stack}_2)$ exists in
gold tree. Remove stack_2 .

Else choose RightArc(label)
if $\text{label}(\text{stack}_2, \text{stack}_1)$ exists
in gold tree and all arcs
 $\text{label}(\text{stack}_1, *)$ 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($\emptyset$, book)

$\emptyset$

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

me the morning flight

stack

action

gold tree

Choose LeftArc(label) if
label(stack₁, stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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)

book

∅

the morning flight

stack

action

gold tree

me

book

∅

Choose LeftArc(label) if
label(stack₁, stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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)

morning flight

stack

action

gold tree

the

book

∅

Choose LeftArc(label) if
label(stack₁, stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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)

flight

stack

action

gold tree

morning

the

book

∅

Choose LeftArc(label) if
label(stack₁, stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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)

nmod(flight,morning)

stack

flight

morning

the

book

∅

action

Choose LeftArc(label) if
label(stack₁,stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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(stack₁,stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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(stack₁,stack₂) exists in
gold tree. Remove stack₂.

Else choose RightArc(label)
if *label(stack₂, stack₁)* exists
in gold tree and all arcs
*label(stack₁, *)*. have been
generated. Remove stack₁

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($\emptyset$, book) *and* book has no more dependents we haven't seen

stack

action

gold tree

book

$\emptyset$

Choose LeftArc(label) if $\text{label}(\text{stack}_1, \text{stack}_2)$ exists in gold tree. Remove stack_2 .

Else choose RightArc(label) if $\text{label}(\text{stack}_2, \text{stack}_1)$ exists in gold tree and all arcs $\text{label}(\text{stack}_1, *)$ have been generated. Remove stack_1

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

✓ $\text{iobj}(\text{book}, \text{me})$

✓ $\text{nmod}(\text{flight}, \text{morning})$

✓ $\text{det}(\text{flight}, \text{the})$

✓ $\text{obj}(\text{book}, \text{flight})$

✓ $\text{root}(\emptyset, \text{book})$

With only $\emptyset$ left on the stack and nothing in the buffer, we're done

stack

action

gold tree

$\emptyset$

Choose LeftArc(label) if $\text{label}(\text{stack}_1, \text{stack}_2)$ exists in gold tree. Remove stack_2 .

Else choose RightArc(label) if $\text{label}(\text{stack}_2, \text{stack}_1)$ exists in gold tree and all arcs $\text{label}(\text{stack}_1, *)$ have been generated. Remove stack_1

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

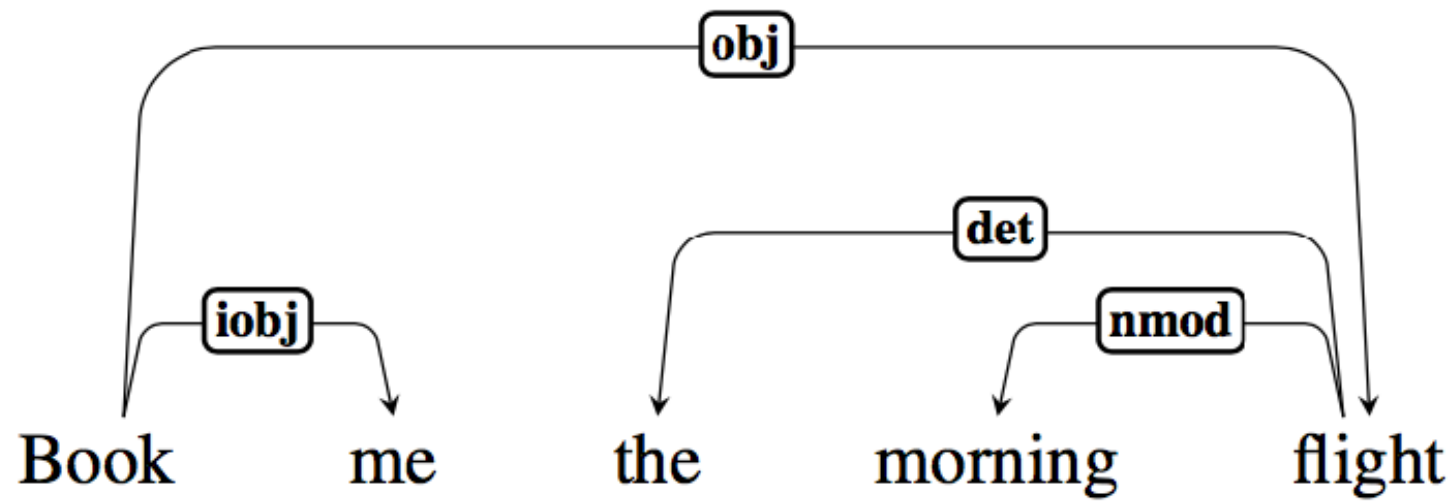
✓ *iobj(book, me)*

✓ *nmod(flight, morning)*

✓ *det(flight, the)*

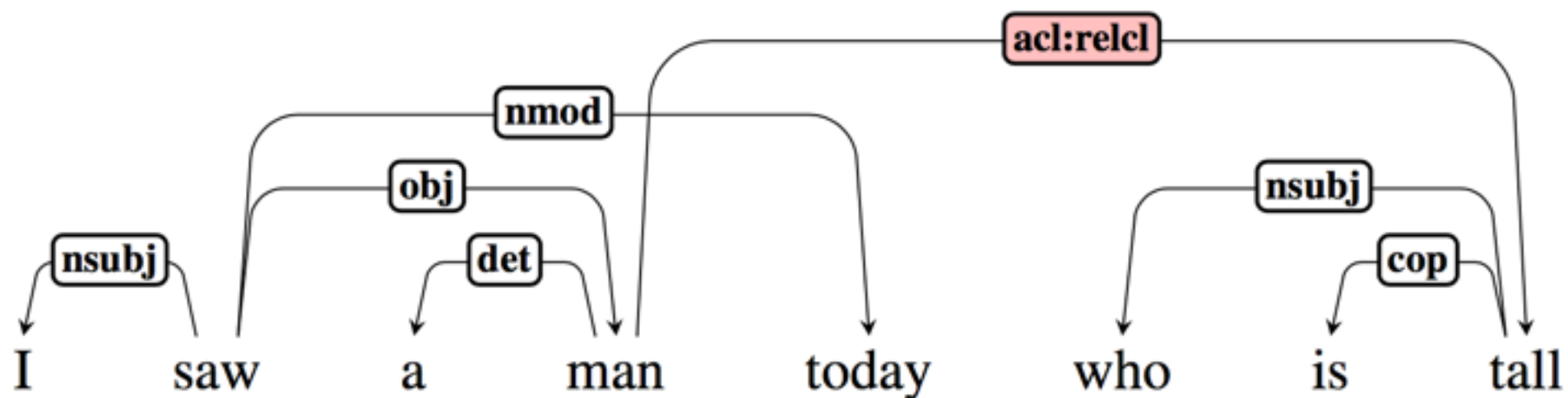
✓ *obj(book, flight)*

✓ *root($\emptyset$, book)*



Shift
Shift
Shift
RightArc(iobj)
Shift
Shift
Shift
LeftArc(nmod)
LeftArc(det)
RightArc(obj)
RightArc(root)

Projectivity



- What happens if you run an oracle on a sentence with a non-projective parse tree?

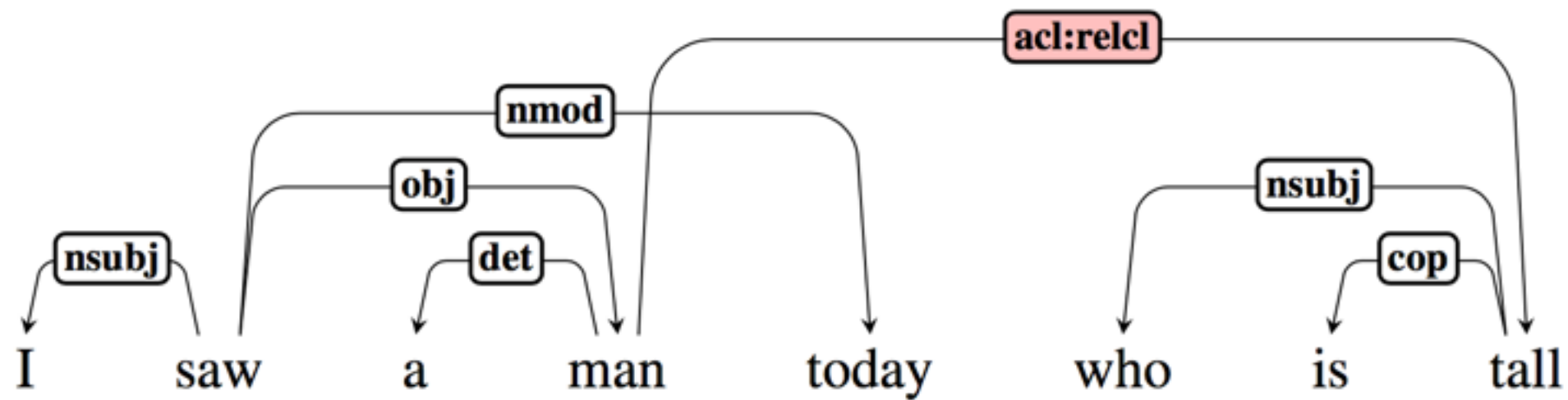
Graph-based parsing

- For a given sentence S , we want to find the highest-scoring tree among all possible trees for that sentence $\mathcal{G}_S$

$$\hat{T}(S) = \arg \max_{t \in \mathcal{G}_S} \text{score}(t, S)$$

- Edge-factored scoring: the total score of a tree is the sum of the scores for all of its edges (arcs):

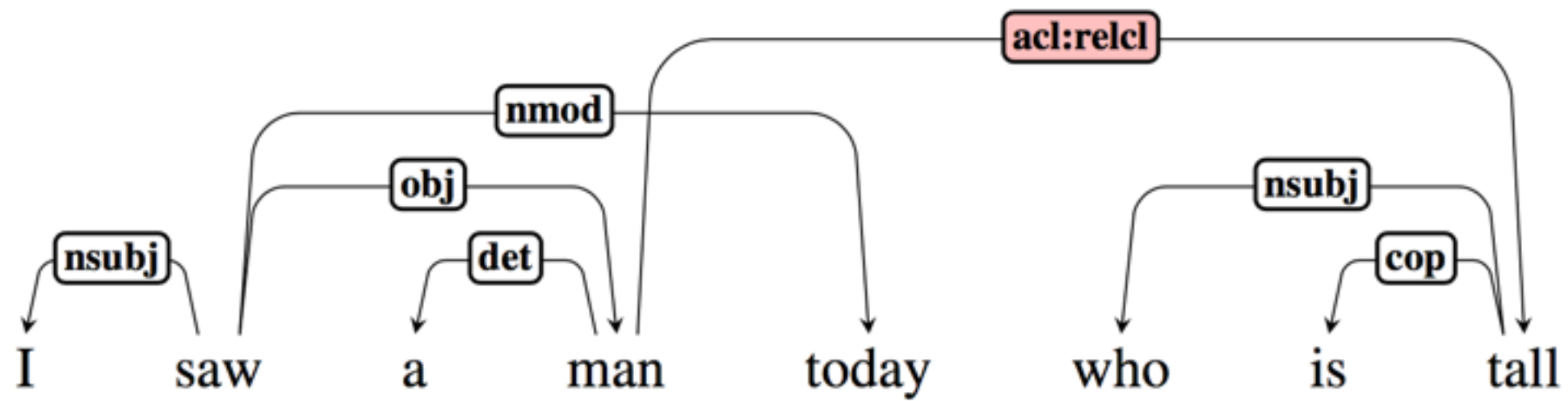
$$\text{score}(t, S) = \sum_{e \in t} \text{score}(e)$$



Edge-factored features

- Word form of head/dependent
- POS tag of head/dependent
- Distributed representation of h/d
- Distance between h/d
- POS tags between h/d
- Head to left of dependent?

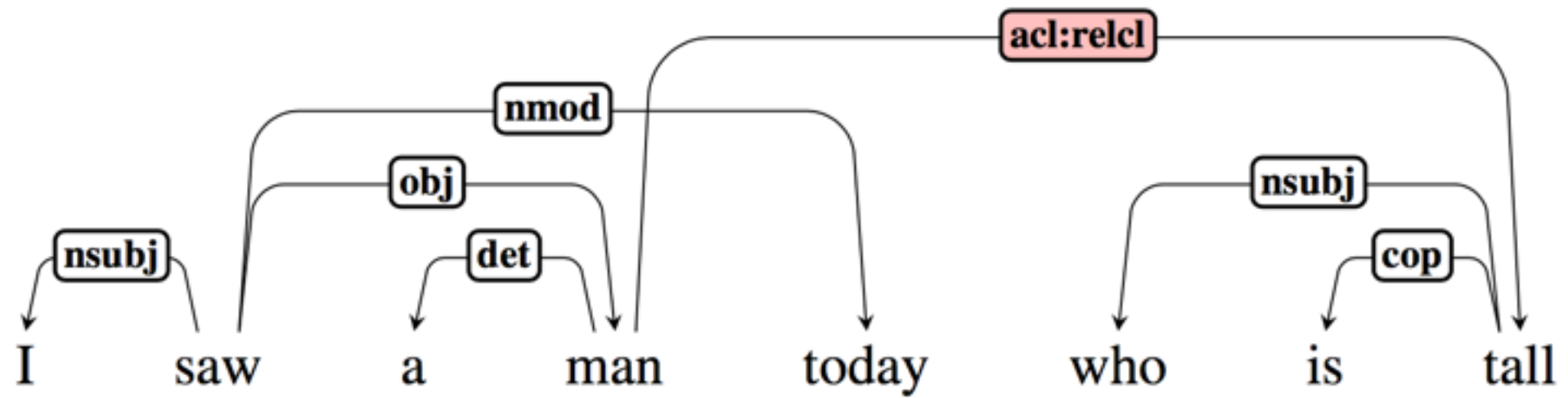
head _t = man	1
head _{pos} = NN	1
distance	4
child _{pos} = JJ and head _{pos} = NN	1
child _{pos} = NN and head _{pos} = JJ	0



$$\text{score}(e) = \sum_{i=1}^F x_i \beta_i$$

Feature value

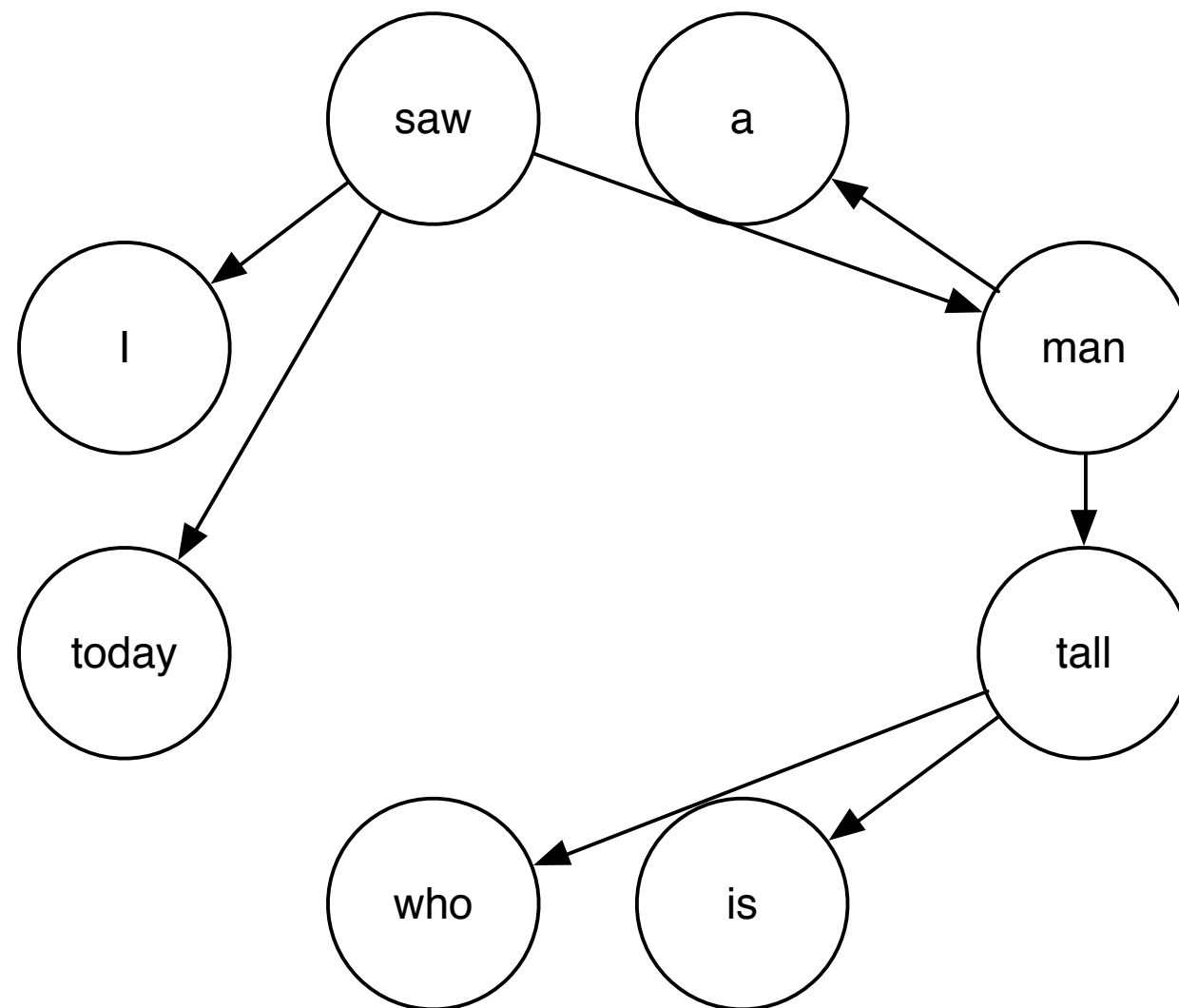
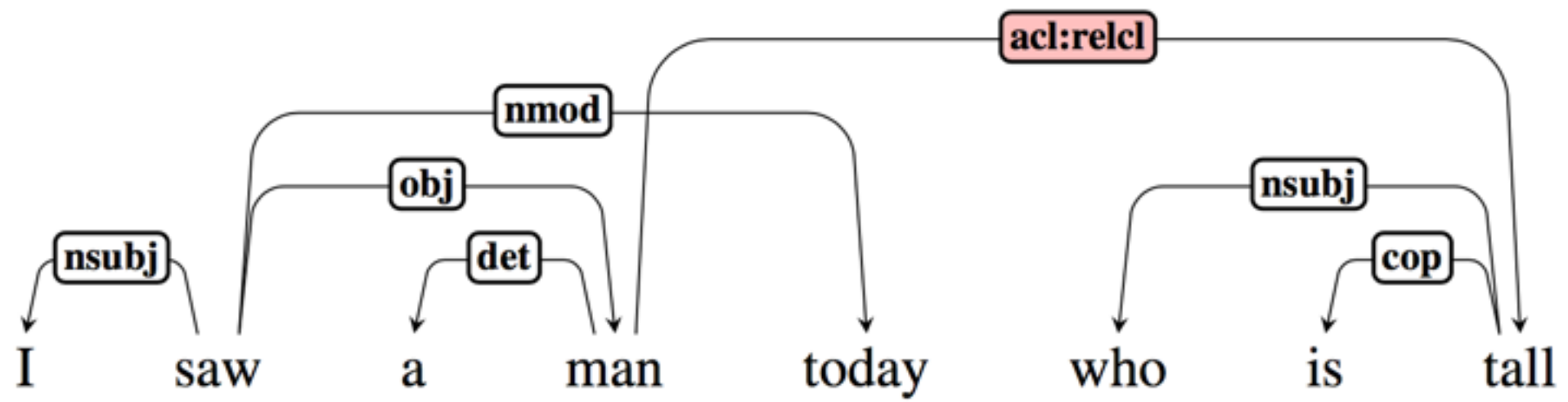
Learned coefficient for that
feature



$$\text{score}(e) = \sum_{i=1}^F x_i \beta_i$$

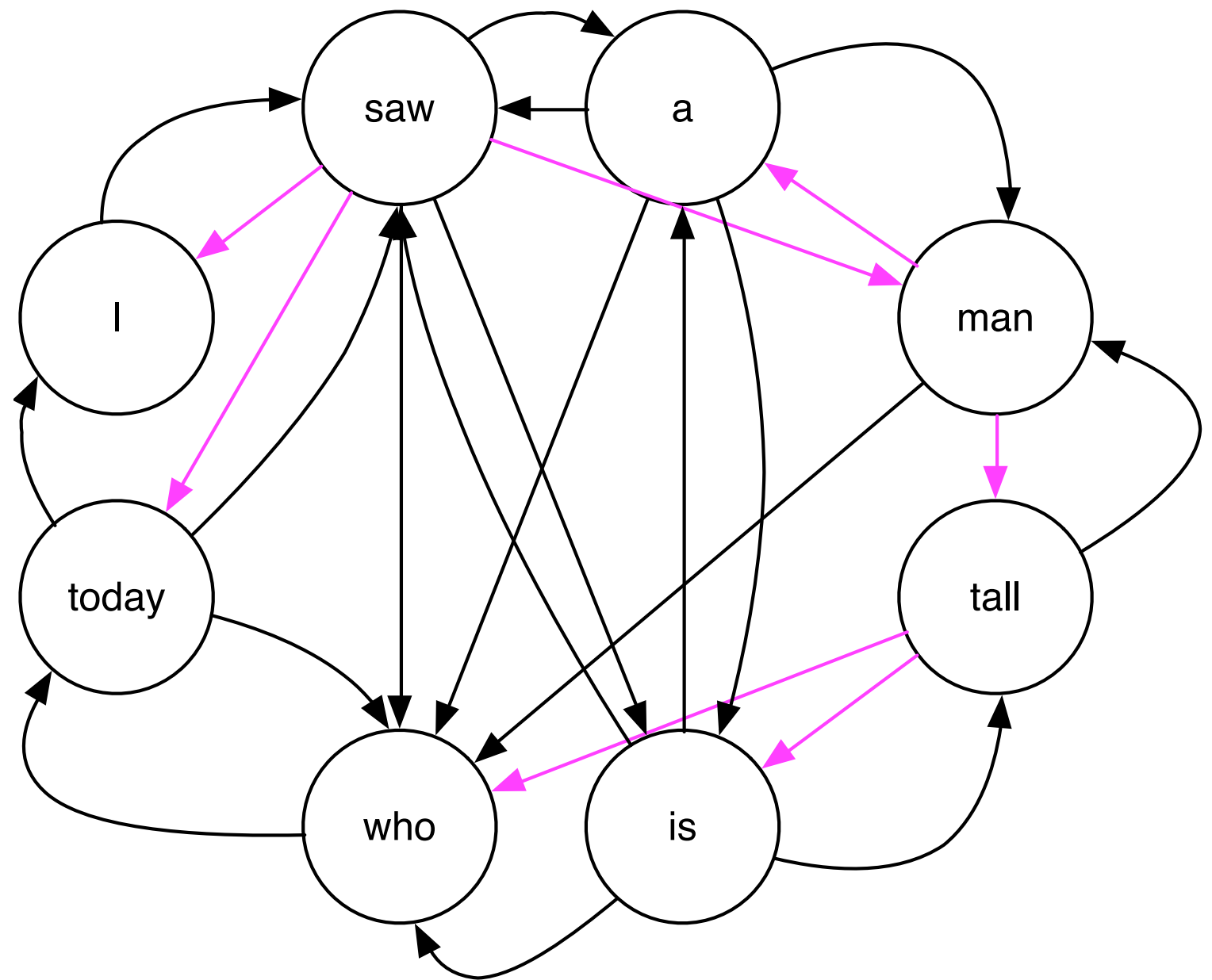
	x	β
head _t = man	1	3.7
head _t = man	1	1.3
distance	4	0.7
child _{pos} = JJ and	1	0.3
child _{pos} = NN and	0	-2.7

$$\text{score}(e) = 8.1$$



MST Parsing

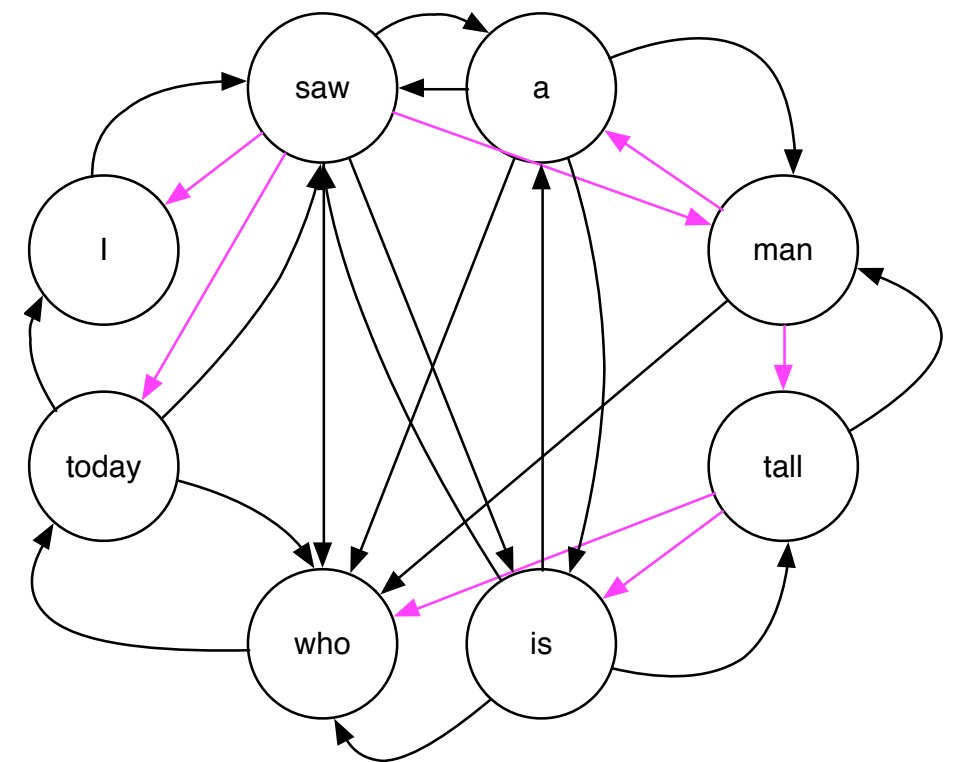
- We start out with a fully connected graph with a score for each edge
- N^2 edges total



(Assume one edge connects each node as dependent and node as head, N^2 total)

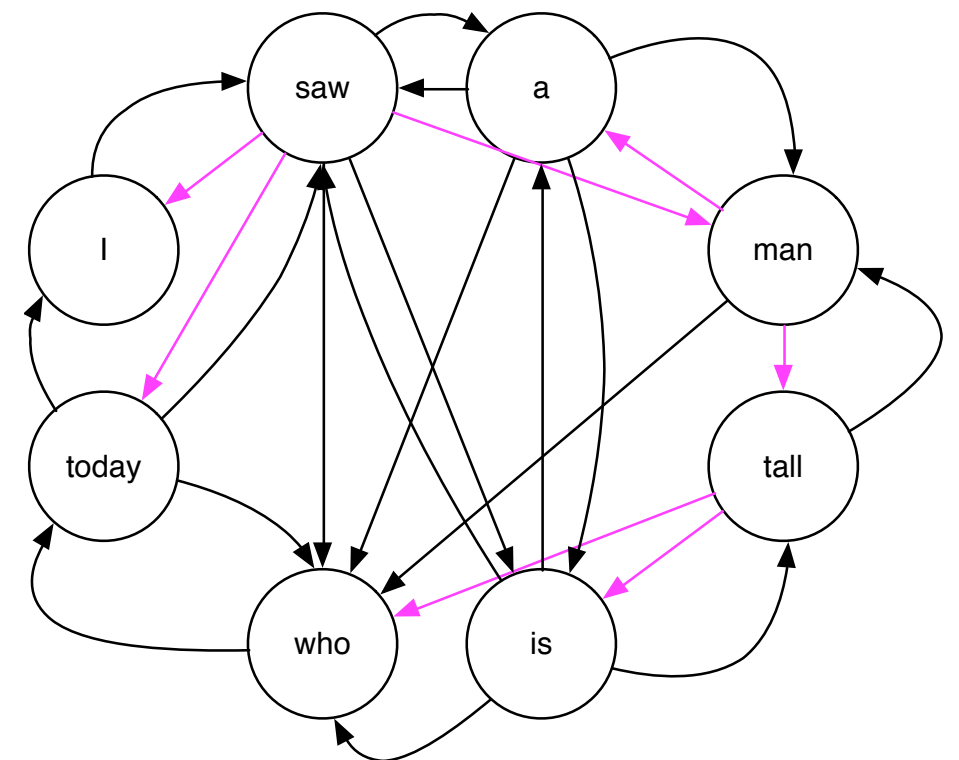
MST Parsing

- From this graph G , we want to find a **spanning tree** (tree that spans G [includes all the vertices in G])
- If the edges have weights, the best parse is the **maximal spanning tree** (the spanning tree with the highest total weight).



MST Parsing

- To find the MST of any graph, we can use the Chu-Liu-Edmonds algorithm in $O(n^3)$ time.
- More efficient Gabow et al. find the MST in $O(n^2 + n \log n)$



Learning

$$\hat{T}(S) = \arg \max_{t \in \mathcal{G}_S} \text{score}(t, S)$$

ϕ is our feature vector scoped over the source dependent, target head and entire sentence x

both are vectors

$$\hat{T}(S) = \arg \max_{t \in \mathcal{G}_S} \sum_{e \in E} \phi(e, x)^\top \beta$$

$$\hat{T}(S) = \arg \max_{t \in \mathcal{G}_S} \left[\sum_{e \in E} \phi(e, x) \right]^\top \beta$$

Learning

$$\hat{T}(S) = \arg \max_{t \in \mathcal{G}_S} \text{score}(t, S)$$

- Given this formulation, we want to learn weights for β that make the score for the gold tree higher than for all other possible trees.
- That's expensive, so let's just try to make the score for the gold tree higher than **the single best tree** we predict (if it's wrong)

Learning

$$\left[\sum_{e \in \textcolor{violet}{E}} \phi(e, x) \right]^\top \beta = \Phi_{gold}(E, x)^\top \beta$$

score for gold tree in treebank

$$\left[\sum_{e \in \textcolor{violet}{\hat{E}}} \phi(e, x) \right]^\top \beta = \hat{\Phi}_{gold}(\hat{E}, x)^\top \beta$$

score for argmax tree in our model

Learning

- We can optimize this using SGD by taking the derivative with respect to the difference in scores.

$$\begin{aligned} & \Phi_{gold}(E, x)^\top \beta - \hat{\Phi}_{pred}(\hat{E}, x)^\top \beta \\ &= \left[\Phi_{gold}(E, x) - \hat{\Phi}_{pred}(\hat{E}, x) \right]^\top \beta \end{aligned}$$

$$\frac{\partial}{\partial \beta} \left[\Phi_{gold}(E, x) - \hat{\Phi}_{pred}(\hat{E}, x) \right]^\top \beta = \Phi_{gold}(E, x) - \hat{\Phi}_{pred}(\hat{E}, x)$$

Structured Perceptron

Algorithm 1 Structured perceptron

```
1: function PERCEPTRONUPDATE( $x, E, \beta$ )  
2:    $\Phi_{gold}(E, x) \leftarrow \text{createFeatures}(x, E)$   
3:    $\hat{\Phi}_{pred}(\hat{E}, x) \leftarrow \text{createFeatures}(x, \hat{E})$   
4:    $\hat{E} \leftarrow \text{CLU}(x, \beta)$   
5:    $\beta \leftarrow \beta + \Phi_{gold}(E, x) - \hat{\Phi}_{pred}(\hat{E}, x)$   
6: end function
```

Create feature vector from true tree

Use CLU to find best tree given scores from current β

Update β with the difference between the feature vectors