In the name of God
the Compassionate, the Merciful

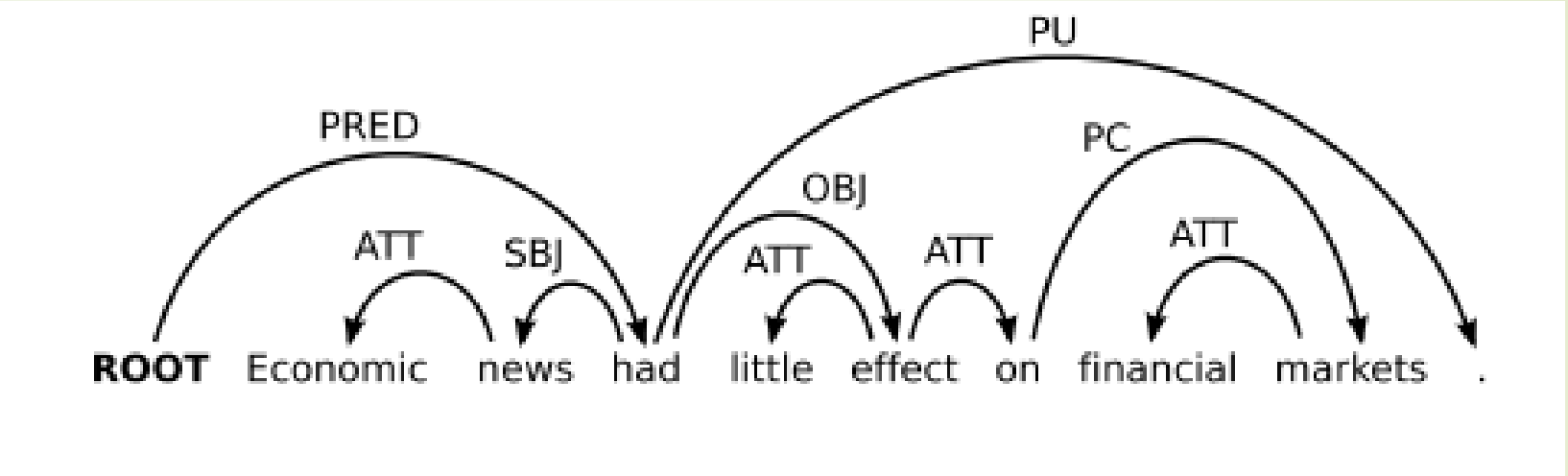
Dependency Parsing

Zahra Rahaie, PhD

Agenda

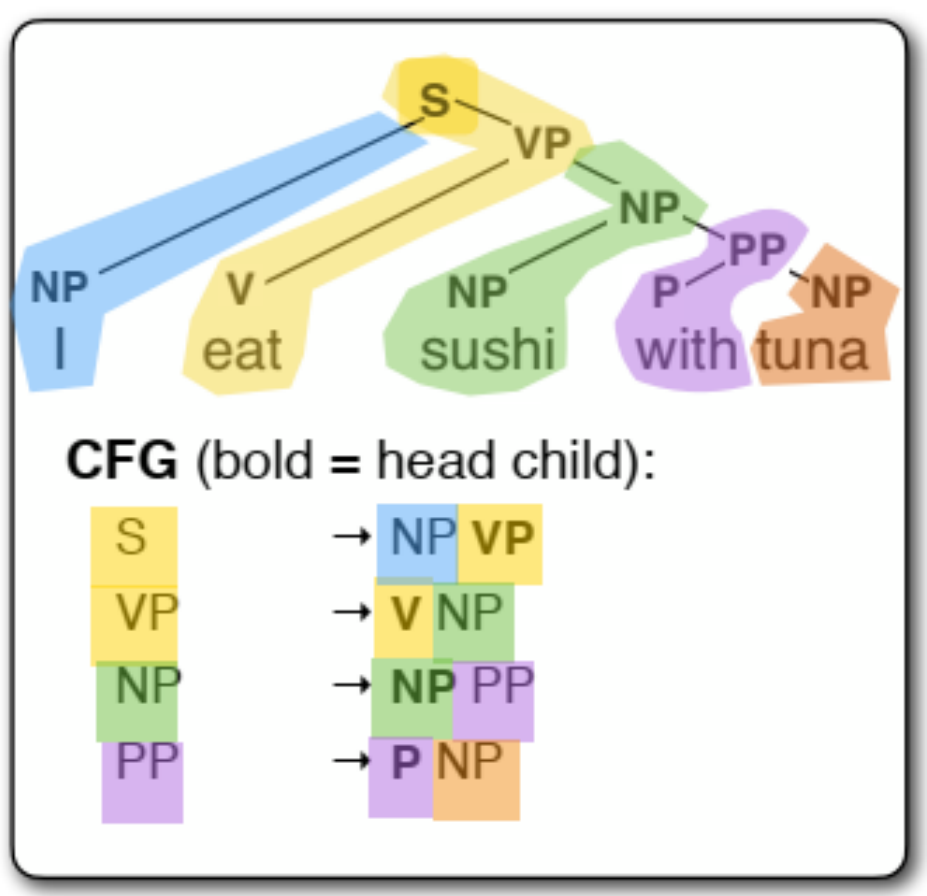
- Dependency Parsing
- Transition based Parsing
- An example

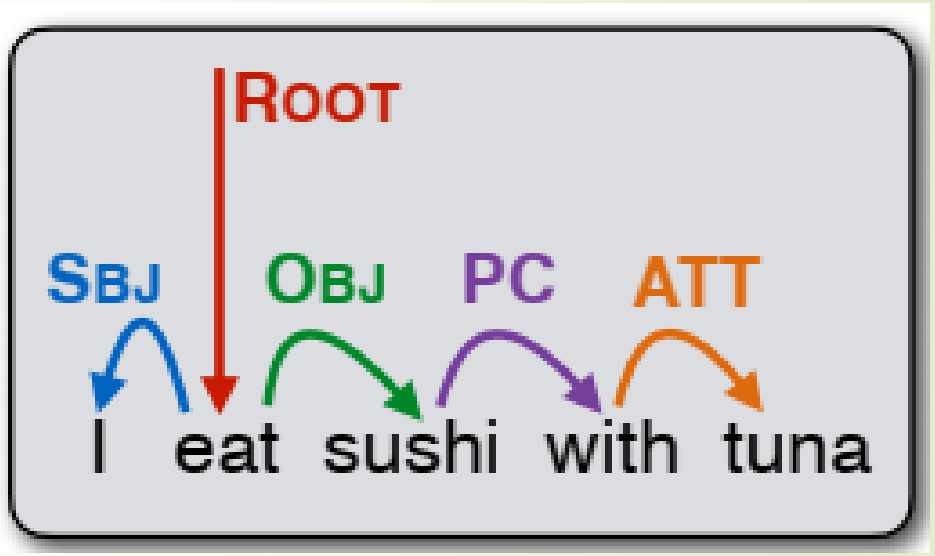
A dependency parse



- Dependencies are (labeled) asymmetrical binary relations between two lexical items (words)
- Dependencies form an acyclic graph over the words in a sentence
- Each word has only one parent
- The (non-root) nodes of a dependency tree are the words/tokens in the sentence.
- The root node of a dependency tree is a special token ROOT
- The edges of a dependency tree are dependencies from a head (parent) to a dependent (child)
- ROOT has exactly one child (the head of the sentence).
- Each non-root (i.e. each non-root node) has exactly one incoming edge (from its parent/head)
- A complete dependency tree (parse) for a a sentence includes all tokens in the sentence.

From CFGs to dependencies

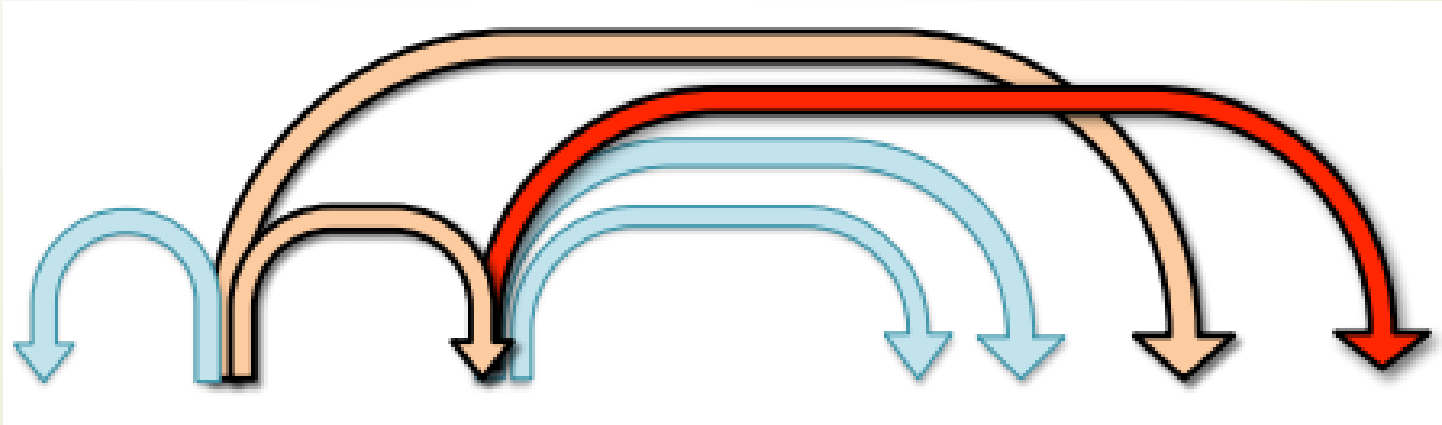




CFGs capture only **nested** dependencies
The dependency graph is a **tree**
The dependencies **do not cross**

Beyond CFGs: Nonprojective dependencies

- Dependencies: tree with crossing branches
  Arise in the following constructions
  - (Non-local) **scrambling** (free word order languages)
  Die Pizza hat Klaus **versprochen** zu **bringen**
  - **Extraposition** (The **guy** is **coming who is wearing a hat**)
  - **Topicalization** (**Cheeseburgers**, I **thought** he **likes**)



Notes

- Dependency treebanks exist for many languages
- Dependency grammar assumes that syntactic structure consists only of dependencies
- DG is purely descriptive (not generative)
- Universal Dependencies: 37 **syntactic** relations, intended to be applicable to all languages ("**universal**"), with slight modifications for each specific language, if necessary.

Different Types

Head-argument: *eat sushi*

Arguments may be obligatory, but can only occur once.
The head alone cannot necessarily replace the construction.

Head-modifier: *fresh sushi*

Modifiers are optional, and can occur more than once.
The head alone can replace the entire construction.

Head-specifier: *the sushi*

Between function words (e.g. prepositions, determiners) and their arguments. Here, syntactic head ≠ semantic head

Coordination: *sushi and sashimi*

Unclear where the head is.

Some Variations

Prepositional phrases (sushi [with wasabi])

Use the lexical head (the noun) as head (sushi→wasabi, wasabi→with),
or the functional head (thepreposition) (sushi→with, with→wasabi)

Verb clusters, complex tenses (I [will have done] this)

Which verb is the head? The main verb (done), or the auxiliaries?

Coordination (eat [sushi and sashimi], [sell and buy] shares)

eat→and, and→sushi, and→sashimi
or (e.g.) eat→sushi, sushi→and, sushi→sashimi, etc.

Relative clauses (the cat [that I thought I saw])

Two main types of parsers

'Transition-based' (shift-reduce) parsers:

Read the sentence incrementally, word by word

Actions (transitions):

Shift (read the next word)

Reduce (attach one word to another, i.e. add an arc)

Model: Predict the next action

Typically return a single, projective, dependency tree

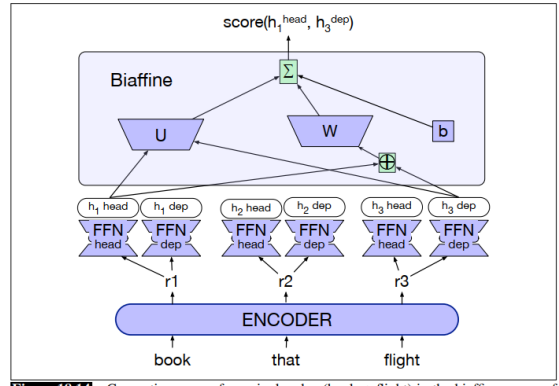


Figure 18.14 Computing scores for a single edge (book→ flight) in the biaffine parser of Dozat and Manning (2017); Dozat et al. (2017). The parser uses distinct feedforward networks to turn the encoder output for each word into a head and dependent representation for the word. The biaffine function turns the head embedding of the head and the dependent embedding of the dependent into a score for the dependency edge.

Graph-based' parsers:

Consider **all words** in the sentence at once.

Use a **minimum spanning tree algorithm** to find the best tree

Models: Score each dependency edge

May return the top *k* trees, including non-projective ones

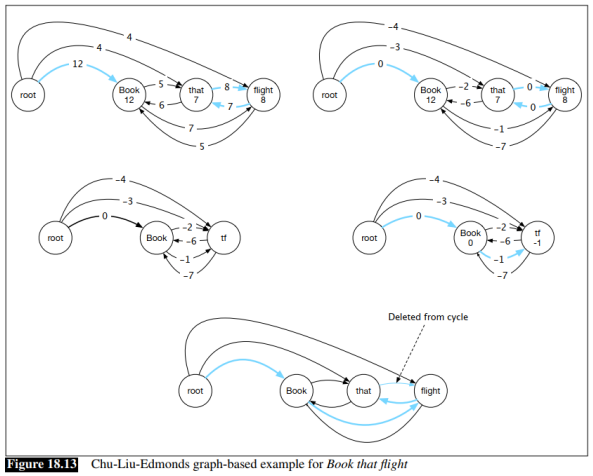


Figure 18.13 Chu-Liu-Edmonds graph-based example for *Book that flight*

Transition-based Parsing

- can only produce projective dependency trees (no crossing edges)

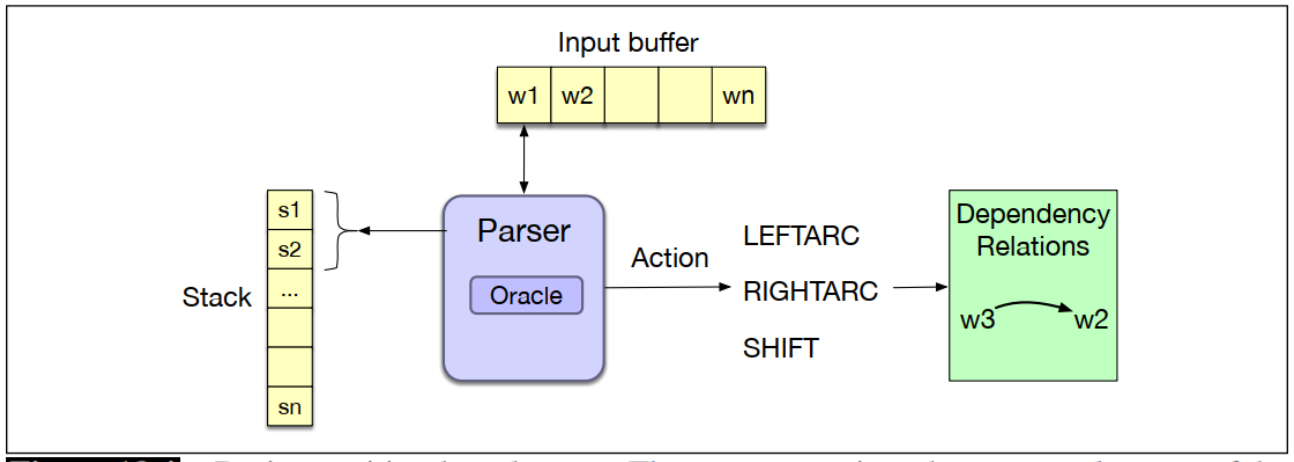
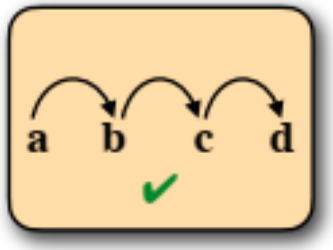


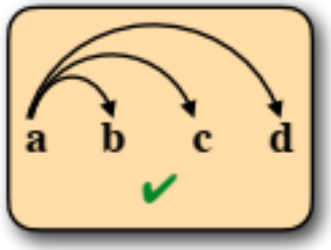
Figure 18.4 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action by consulting an oracle that examines the current configuration.

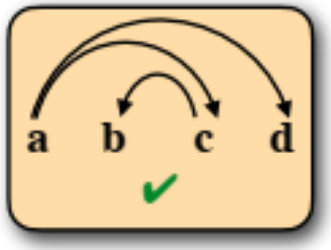
```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)         ; choose a transition operator to apply
    state ← APPLY(t, state)   ; apply it, creating a new state
  return state
```

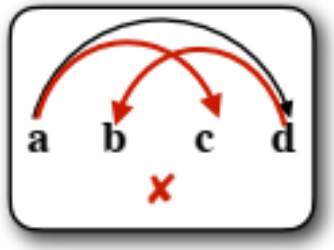
Figure 18.5 A generic transition-based dependency parser

Projective dependencies









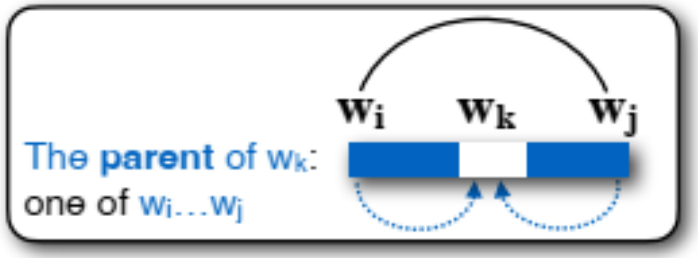
Projective = No crossing dependencies!

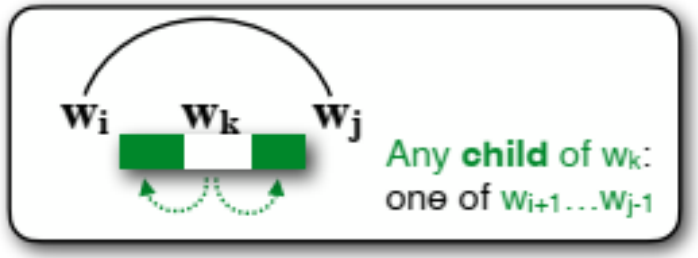
Projective dependencies: for any i, j, k with $i < k < j$:

if there is a dependency between w_i and w_j,

the **parent** of w_k is a word w_l between (possibly including) i and j: $i \leq l \leq j$,

while any **child** w_m of w_k has to occur between (excluding) i and j: $i<m<j$





Transition-based Parsing

The parser processes the sentence $\mathbf{S = w_0 w_1 ... w_n}$ from left to right ("**incremental parsing**")

The parser uses **three data structures**:

- σ: a **stack** of partially processed words $w_i \in T_S$
- β: a **buffer** of remaining input words $w_i \in T_S$
- A: a **set of dependency arcs** $(w_i, \ell, w_j) \in T_S \times L \times T_S$

w_0 is a special ROOT token.

$T_S = \{w_0, w_1, ..., w_n\}$ are the tokens in the input sentence

L is a predefined set of dependency relation labels

(w_i, ℓ, w_j) is a dependency with label ℓ from head w_i to w_j

Elements…

The stack σ is a list of partially processed words

We can *shift* the top word of β onto the top of σ (grow the stack by one) or remove one of the top two words from σ by attaching it to the other top word (this *reduces* the stack by one element)

- σ|w_j or σ|$w_i w_j$: w_j is the topmost word on the stack.
- σ|$w_i w_j$: w_i is the second top word on the stack.

The buffer β is the remaining input words

We read words from β (left-to-right) and push ('*shift*') them onto σ

w|β: w is on top of the buffer. w is the next word to be shifted onto σ

The set of arcs A defines the current tree.

We add a new arc to A by attaching the first word on top of the stack to the second word on top of the stack or vice versa

Parser actions (transitions)

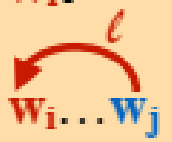
In any configuration (σ, β, A), take one of these actions:

1) **Shift** w_k from buffer β to stack σ:

Shift: $(\sigma, w_k|\beta, A) \Rightarrow (\sigma|w_k, \beta, A)$

2) Add a ***leftwards*** **dependency arc with label** ℓ from w_j to w_i:

LeftArc-ℓ : $(\sigma|w_i w_j, \beta, A) \Rightarrow (\sigma|w_j, \beta, A \cup \{(w_j, \ell, w_i)\})$



w_i (2nd on stack) is a dependent (with label ℓ) of head w_j (1st on stack)

w_i is removed from the stack σ: only do this if w_i has no further children to be attached

3) Add a ***rightwards*** **dependency arc with label** ℓ from w_i to w_j:

Right Arc-ℓ: $(\sigma|w_i w_j, w_k|\beta, A) \Rightarrow (\sigma|w_i, \beta, A \cup \{(w_i, \ell, w_j)\})$

w_j (1st on stack) is a dependent (with label ℓ) of head w_i (2nd on stack).

w_j is removed from the stack σ: only do this if w_j has no further children to be attached

Interpreting configuration

In the configuration $(\sigma|w_i w_j, w_k|\beta, A)$:

w_i and w_j are the top two elements of the stack.
Each may already have some dependents of their own

w_k (top of the buffer) does not have any dependents yet

w_i (2nd on stack) precedes w_j (top of stack): $i < j$

w_j (top of stack) precedes w_k (top of buffer): $j < k$

We have to either **attach** w_i **to** w_j (add a LeftArc),
attach w_j **to** w_i (add a RightArc), or **shift** w_k onto the stack

We can only reach $(\sigma|w_j, w_k|\beta, A)$ if all words w_l with $j < l < k$
have already been attached to their parent w_m with $j \leq m < k$

Configurations

We start in the **initial configuration** ($[w_0]$, $[w_1, \ldots, w_n]$, {})
(**Root token**, **Input Sentence**, **No tree**)

In the initial configuration, we can only **push w_1 onto the stack.**

We want to end in a **terminal configuration** ($[w_0]$, [], A)
(**Root token**, **Empty buffer**, **Complete tree**)

In a terminal configuration, we have **read all of the input words** (empty buffer) and we have **attached all input words**.

(The root w_0 is the only token that can't get attached to any other word)

In practice..

Which action should the parser take
in the current configuration?

We also need a **parsing model** that assigns a score to each possible action given a current configuration.

- **Possible actions:**
  SHIFT, and for any relation ℓ: LEFTARC-ℓ, or RIGHT-ARC-ℓ
- **Possible features of the current configuration**:
  The **top {1,2,3} words** on the **buffer** and on the **stack**, their **POS tags**, distances between the words, etc.

We can learn this model from a dependency treebank.

Economic news had little effect on financial markets .

Transition	Stack	Buffer	Arcs
	[**ROOT**]	[**Economic**, news, had, …]	{}
SHIFT	[ROOT, **Economic**]	[**news**, had, little, …]	
LA-amod	[ROOT, Economic, **news**]	[**had**, little, effect, …]	(news, amod, Economic)
SHIFT	[ROOT, **news**]	[**had**, little, effect, …]	
LA-nsubj	[ROOT, news, **had**]	[**little**, effect, on, …]	(had, nsubj, news)
SHIFT	[ROOT, **had**]	[**little**, effect, on, …]	
SHIFT	[ROOT, had, **little**]	[**effect**, on, financial, …]	
LA-amod	[ROOT, had, little, **effect**]	[**on**, financial, markets, …]	(effect, amod, little)
SHIFT	[ROOT, had, **effect**]	[**on**, financial, markets, …]	
SHIFT	[ROOT, had, effect, **on**]	[**financial**, markets, .]	
SHIFT	[ROOT, had, effect, on, **financial**]	[**markets**, .]	
LA-amod	[ROOT, had, effect, on, financial, **markets**]	[.]	(markets, amod, financial)
LA-case	[ROOT, had, effect, on, **markets**]	[.]	(markets, case, on)
RA-nmod	[ROOT, had, effect, **markets**]	[.]	(effect, nmod, markets)
RA-obj	[ROOT, had, **effect**]	[.]	(had, obj, effect)
SHIFT	[ROOT, **had**]	[.]	
RA-punct	[ROOT, had, .]	[]	(had, punct, .)
RA-root	[ROOT, **had**]	[]	(ROOT, root, had)
	[**ROOT**]	[]	

References

- Chapter 18 of NLP book

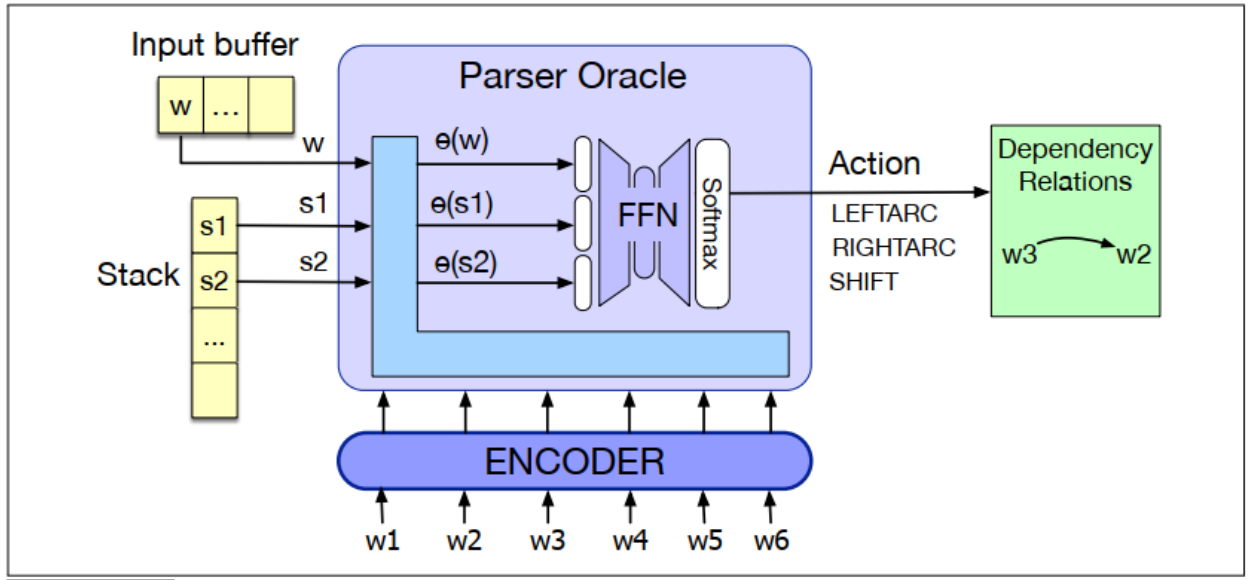


Figure 18.8 Neural classifier for the oracle for the transition-based parser. The parser takes the top 2 words on the stack and the first word of the buffer, represents them by their encodings (from running the whole sentence through the encoder), concatenates the embeddings and passes through a softmax to choose a parser action (transition).