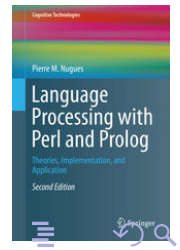
Language Processing with Perl and Prolog

Chapter 13: Dependency Parsing

Pierre Nugues

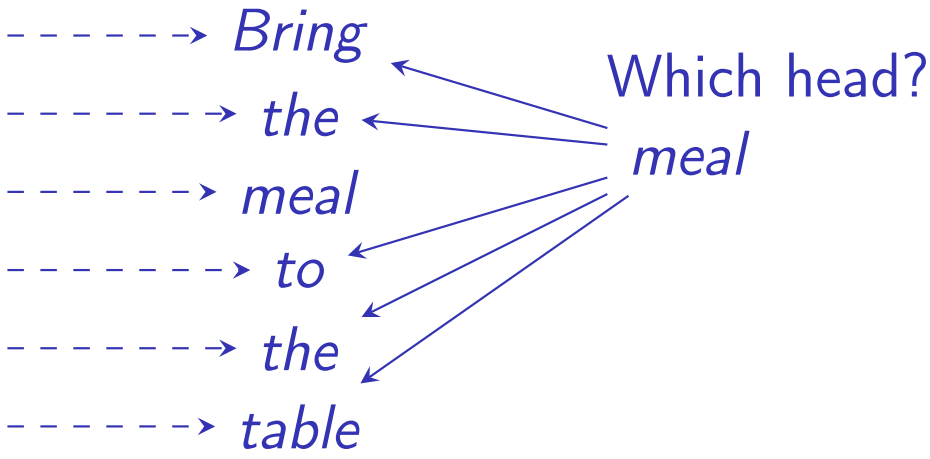
Lund University
Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

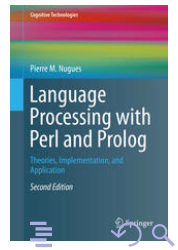


Parsing Dependencies

Generate all the pairs:

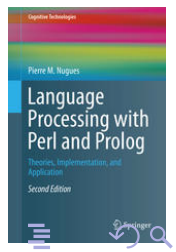
Which sentence root?





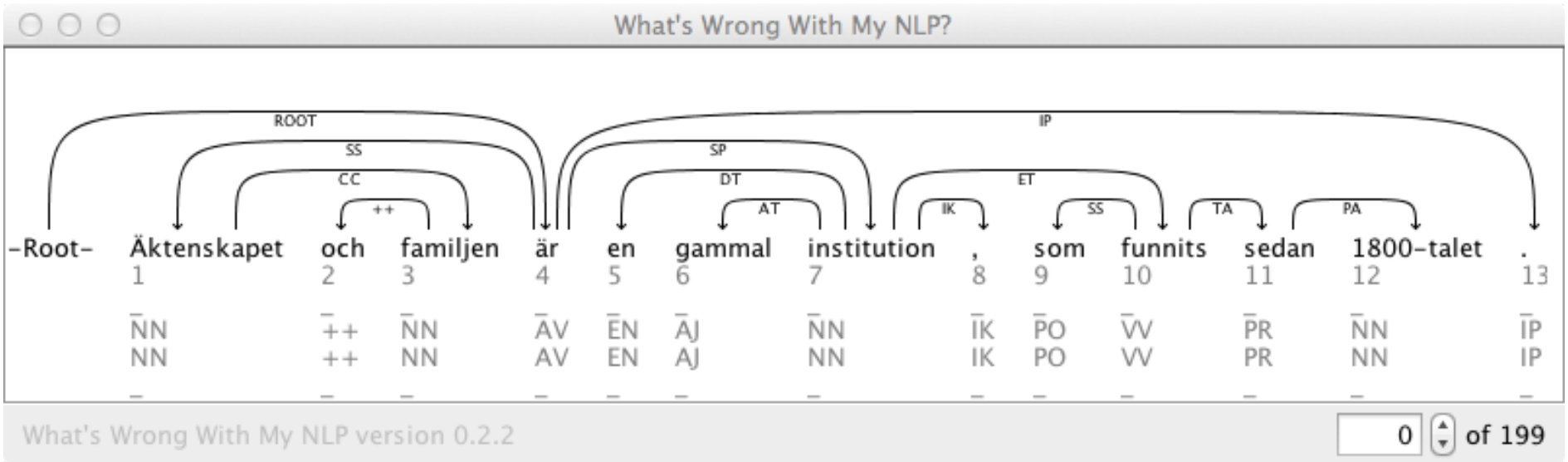
Talbanken: An Annotated Corpus of Swedish

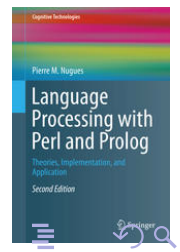
1	Äktenskapet	_	NN	NN	_	4	SS
2	och	_	++	++	_	3	++
3	familjen	_	NN	NN	_	1	CC
4	är	_	AV	AV	_	0	ROOT
5	en	_	EN	EN	_	7	DT
6	gammal	_	AJ	AJ	_	7	AT
7	institution	_	NN	NN	_	4	SP
8	,	_	IK	IK	_	7	IK
9	som	_	PO	PO	_	10	SS
10	funnits	_	VV	VV	_	7	ET
11	sedan	_	PR	PR	_	10	TA
12	1800-talet	_	NN	NN	_	11	PA
13	.	_	IP	IP	_	4	IP



Visualizing the Graph

Using *What's Wrong With My NLP* (`https://code.google.com/p/whatswrong/`):

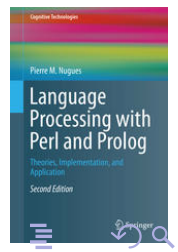




Parser Input

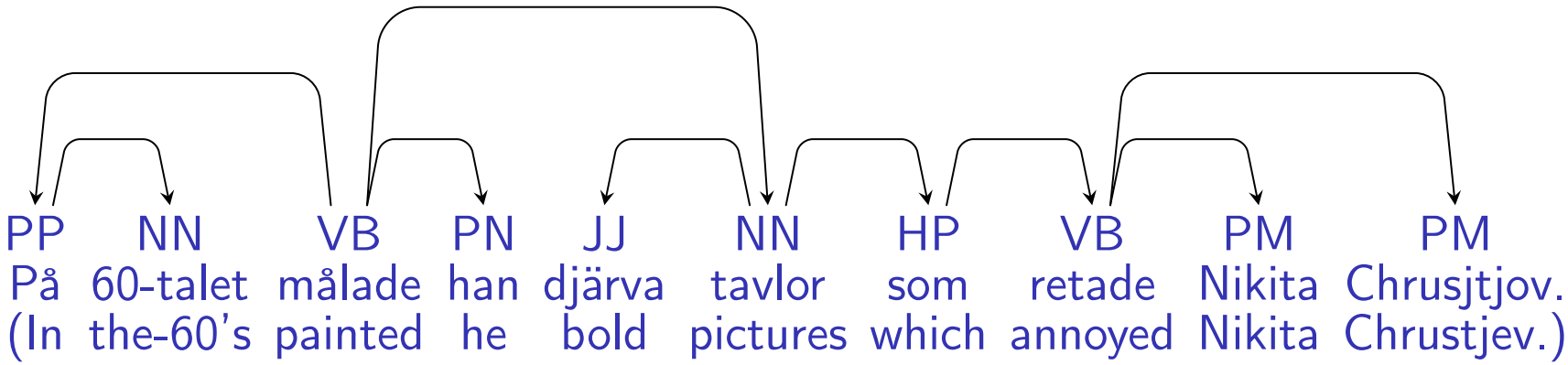
The words and their parts of speech obtained from an earlier step.

1	Äktenskapet	_	NN	NN	_
2	och	_	++	++	_
3	familjen	_	NN	NN	_
4	är	_	AV	AV	_
5	en	_	EN	EN	_
6	gammal	_	AJ	AJ	_
7	institution	_	NN	NN	_
8	,	_	IK	IK	_
9	som	_	PO	PO	_
10	funnits	_	VV	VV	_
11	sedan	_	PR	PR	_
12	1800-talet	_	NN	NN	_
13	.	_	IP	IP	_

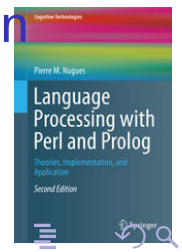


Nivre's Parser

Joakim Nivre designed an efficient dependency parser extending the shift-reduce algorithm.
He started with Swedish and has reported the best results for this language and many others.



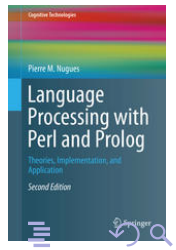
His team obtained the best results in the CoNLL 2007 shared task on dependency parsing.



The Parser

The first step is a POS tagging
The parser applies a variation/extension of the shift-reduce algorithm since dependency grammars have no nonterminal symbols
The transitions are:

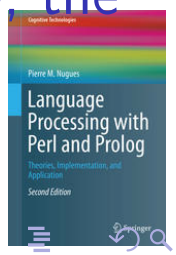
- Shift, pushes the input token to the stack
- Reduce, reduces the token on the top of the stack
- Left arc, adds an arc from the next input token to the token on the top of the stack and reduces it.
- Right arc, adds an arc from the token on top of the stack to the next input token and pushes the input token on the top of the stack.



Transitions' Definition

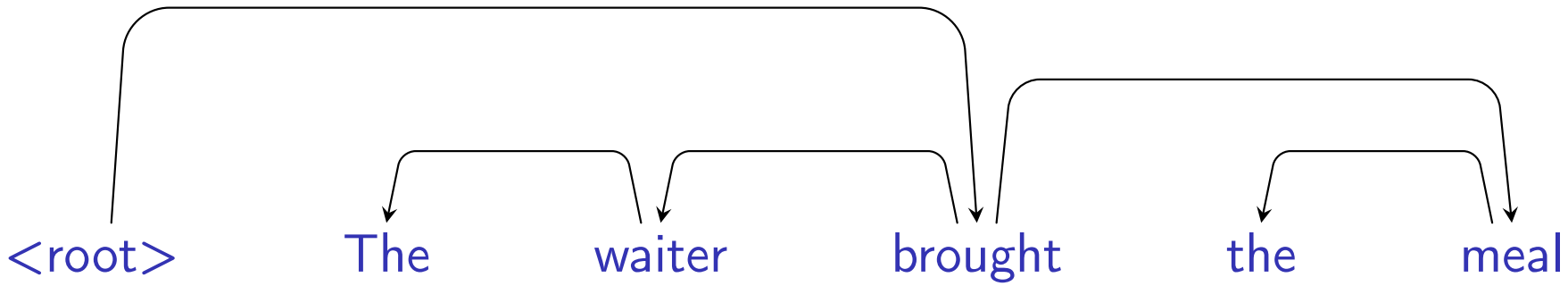
Actions	Parser states	Conditions
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle S, nil, A \rangle$	
Left-arc	$\langle n\|S, n'\|I, A \rangle \rightarrow \langle S, n'\|I, A \cup \{(n', n)\} \rangle$	$\nexists n''(n'', n) \in A$
Right-arc	$\langle n\|S, n'\|I, A \rangle \rightarrow \langle n'\|n\|S, I, A \cup \{(n, n')\} \rangle$	
Reduce	$\langle n\|S, I, A \rangle \rightarrow \langle S, I, A \rangle$	$\exists n'(n', n) \in A$
Shift	$\langle S, n\|I, A \rangle \rightarrow \langle n\|S, I, A \rangle$	

1. The first condition $\nexists n''(n'', n) \in A$, where n'' is the head and n, the dependent, is to enforce a unique head.
2. The second condition $\exists n'(n', n) \in A$, where n' is the head and n, the dependent, is to ensure that the graph is connected.



Nivre's Parser in Action

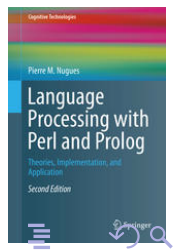
Input W = The waiter brought the meal.
The graph is:



$$\{\text{the} \leftarrow \text{waiter}, \text{waiter} \leftarrow \text{brought}, \text{ROOT} \rightarrow \text{brought}, \text{the} \leftarrow \text{meal},$$
$$\text{brought} \rightarrow \text{meal}\},$$

Let us apply the sequence:

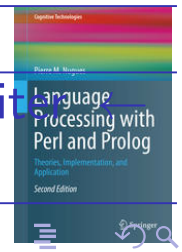
`[sh, sh, la, sh, la, ra, sh, la, ra]`



Nivre's Parser in Action

[sh, sh, la, sh, la, ra, sh, la, ra]

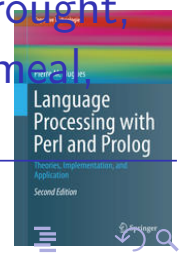
Trans.	Stack	Queue	Graph
start	∅	ROOT the waiter brought the meal	{}
sh			
	ROOT	the waiter brought the meal	{}
sh			
	the ROOT	waiter brought the meal	{}
la			
	ROOT	waiter brought the meal	{the ← waiter}
sh			
	waiter ROOT	brought the meal	{the ← waiter}
la			
	ROOT	brought the meal	{the ← waiter, waiter ← brought}



Nivre's Parser in Action (II)

[~~sh, sh, la, sh, la~~, ra, sh, la, ra]

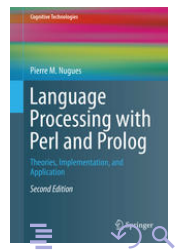
Trans.	Stack	Queue	Graph
ra			
	brought ROOT	the meal	{the ← waiter, waiter ← brought, ROOT → brought}
sh			
	the brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought}
la			
	brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal}
ra			
end	meal brought ROOT	[]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal, brought → meal}



Nivre's Parser in Prolog: Left-Arc

```
% shift_reduce(+Sentence, -Graph)
shift_reduce(Sentence, Graph) :-
  shift_reduce(Sentence, [], [], Graph).

% shift_reduce(+Words, +Stack, +CurGraph, -FinGraph)
shift_reduce([], _, Graph, Graph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  left_arc(Words, Stack, NewStack, Graph, NewGraph),
  write('left arc'), nl,
  shift_reduce(Words, NewStack, NewGraph, FinalGraph).
```



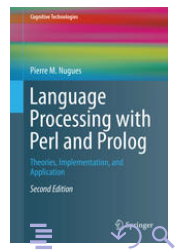
Gold Standard Parsing

Nivre's parser uses a sequence of actions taken in the set `{la, ra, re, sh}`.
We have:

- A sequence of actions creates a dependency graph
- Given a projective dependency graph, we can find an action sequence creating this graph. This is gold standard parsing.

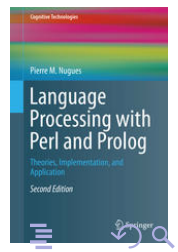
Let *TOP* be the top of the stack and *FIRST*, the first token of the input list, and *A* the dependency graph.

1. **if** $arc(TOP, FIRST) \in A$, **then** right-arc;
2. **else if** $arc(FIRST, TOP) \in A$, **then** left-arc;
3. **else if** $\exists k \in Stack, arc(FIRST, k) \in A$ or $arc(k, FIRST) \in A$, **then** reduce;
4. **else** shift.



Parsing a Sentence

When parsing an unknown sentence, we do not know the dependencies yet
The parser will use a "guide" to tell which transition to apply in the set `{la, ra, re, sh}`.
The parser will extract a context from its current state, for instance the part of speech of the top of the stack and the first in the queue, and will ask the guide.
D-rules are a simply way to implement this

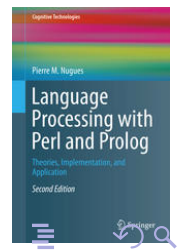


Dependency Rules

D-rules are possible relations between a head and a dependent.
They involve part-of-speech, mostly, and words

1. `determiner` ← `noun`.
2. `adjective` ← `noun`.
3. `preposition` ← `noun`.
4. `noun` ← `verb`.
5. `preposition` ← `verb`.
6. `verb` ← `root`.

$$\begin{bmatrix} category : noun \\ number : N \\ person : P \\ case : nominative \end{bmatrix} \leftarrow \begin{bmatrix} category : verb \\ number : N \\ person : P \end{bmatrix}$$



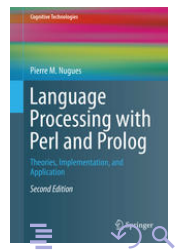
Parsing Dependency Rules in Prolog

```
%drule(Head, Dependent, Function).

drule(noun, determiner, determinative).
drule(noun, adjective, attribute).
drule(verb, noun, subject).
drule(verb, noun, object).
```

D-Rules may also include a direction, for instance a determiner is always to the left

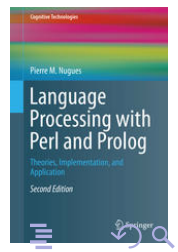
```
%drule(Head, Dependent, Function, Direction).
```



Nivre's Parser in Prolog: Left-Arc (II)

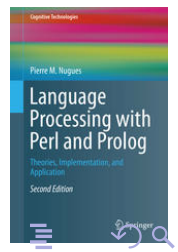
```
% left_arc(+WordList, +Stack, -NewStack, +Graph, -NewGraph)

left_arc([w(First, PosF) | _], [w(Top, PosT) | Stack],
    Stack, Graph, [d(w(First, PosF),
    w(Top, PosT), Function) | Graph]) :-
  word(First, FirstPOS),
  word(Top, TopPOS),
  drule(FirstPOS, TopPOS, Function, left),
  \+ member(d(_, w(Top, PosT), _), Graph).
```



Tracing Nivre's Parser

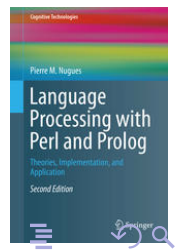
```
shift_reduce([w(the, 1), w(waiter, 2), w(brought, 3),
w(the, 4), w(meal, 5)], G).
shift
left arc
shift
left arc
shift
shift
left arc
right arc
G = [d(w(brought, 3), w(meal, 5), object),
d(w(meal, 5), w(the, 4), determinative),
d(w(brought, 3), w(waiter, 2), subject),
d(w(waiter, 2), w(the, 1), determinative)]
```



Using Features

D-rules consider a limited context: the part of speech of the top of the stack and the first in the queue
We can extend the context:

- Extracts more features (attributes), for instance two words in the stack, three words in the queue
- Use them as input to a four-class classifier and determine the next action

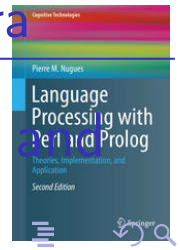


Training a Classifier

Gold standard parsing of a manually annotated corpus produces training data

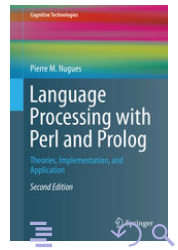
Stack	Queue	Stack		Queue		Trans.
$\mathrm{POS}(T_0)$	$\mathrm{POS}(Q_0)$	$\mathrm{POS}(T_0)$	$\mathrm{POS}(T_{-1})$	$\mathrm{POS}(Q_0)$	$\mathrm{POS}(Q_{+1})$	
nil	ROOT	nil	nil	ROOT	DT	sh
ROOT	DT	ROOT	nil	DT	NN	sh
DT	NN	DT	ROOT	NN	VBD	la
ROOT	NN	ROOT	nil	NN	VBD	sh
NN	VBD	NN	ROOT	VBD	DT	la
ROOT	VBD	ROOT	nil	VBD	DT	ra
VBD	DT	VBD	ROOT	DT	NN	sh
DT	NN	DT	VBD	NN	nil	la
VBD	NN	VBD	ROOT	NN	nil	ra

Using Talbanken and CoNLL 2006 data, you can train decision trees and implement a parser.



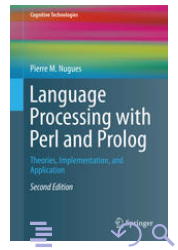
Feature Vectors

You extract one feature (attribute) vector for each parsing action.
The most elementary feature vector consists of two parameters: `POS_TOP`, `POS_FIRST`
Nivre et al. (2006) used from 16 to 30 parameters and support vector machines.
As machine-learning algorithm, you can use decision trees, perceptron, logistic regression, or support vector machines.



Finding Dependencies using Constraints

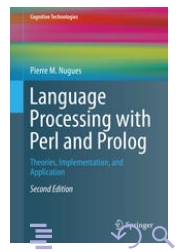
Parts of speech	Possible governors	Possible functions
Determiner	noun	det
Noun	verb	object, iobject
Noun	prep	pcomp
Verb	root	root
Prep	verb, noun	mod, loc



Tagging

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	nil, root	3, det 6, det	4, pcomp 1, object 1, iobject	3, mod 1, loc	3, det 6, det	4, pcomp 1, object 1, iobject

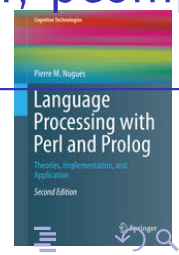
A second step applies and propagates constraint rules.
Rules for English describe: projectivity – links must not cross –, function uniqueness – there is only one subject, one object, one indirect object –, topology



Constraints

- A determiner has its head to its right-hand side
- A subject has its head to its right-hand side when the verb is at the active form
- An object and an indirect object have their head to their left-hand side (active form)
- A prepositional complement has its head to its left-hand side

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	nil, root	3, det	1, iobject 1, object	3, mod 1, loc	6, det	4, pcomp



Evaluation of Dependency Parsing

Dependency parsing: The error count is the number of words that are assigned a wrong head, here 1/6.

Reference	Output