Language Technology
http://cs.lth.se/edan20/
Chapter 13: Dependency Parsing

Pierre Nugues

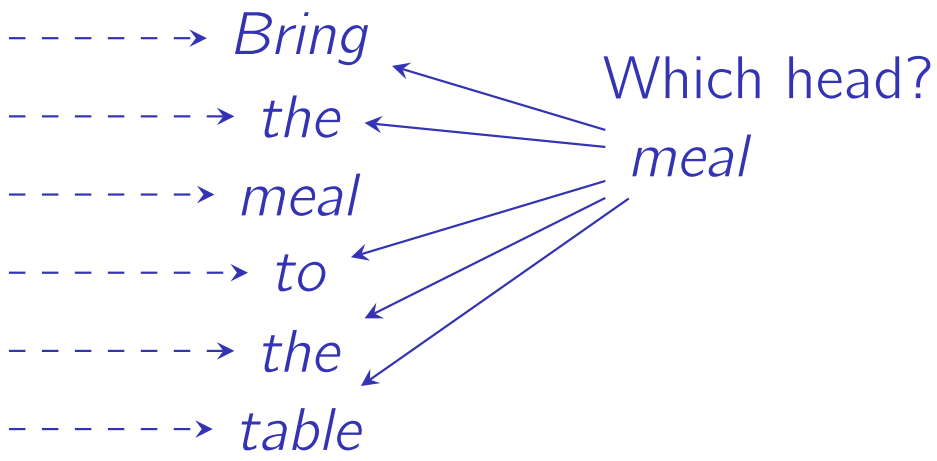
Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

October 1st, 2020

Parsing Dependencies

Generate all the pairs:

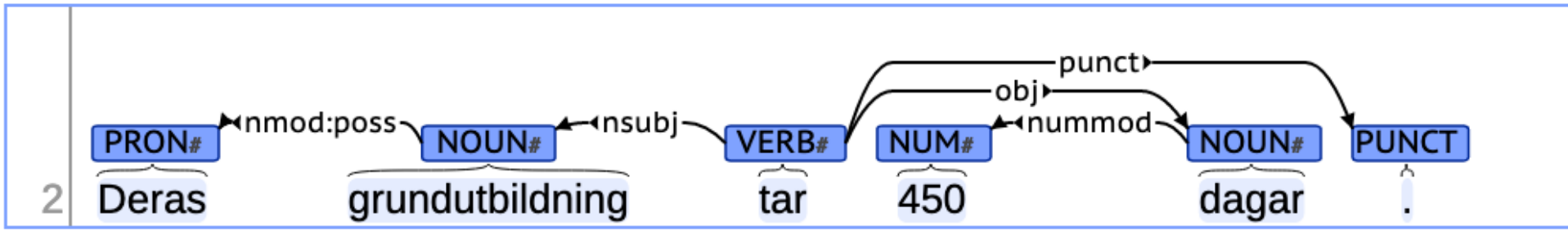
Which sentence root?



Talbanken: An Annotated Corpus of Swedish

1	Deras	de	PRON	2	nmod:poss
2	grundutbildning	grundutbildning	NOUN	3	nsubj
3	tar	ta	VERB	0	root
4	450	450	NUM	5	nummod
5	dagar	dag	NOUN	3	obj
6	.	.	PUNCT	3	punct

Visualizing the Graph



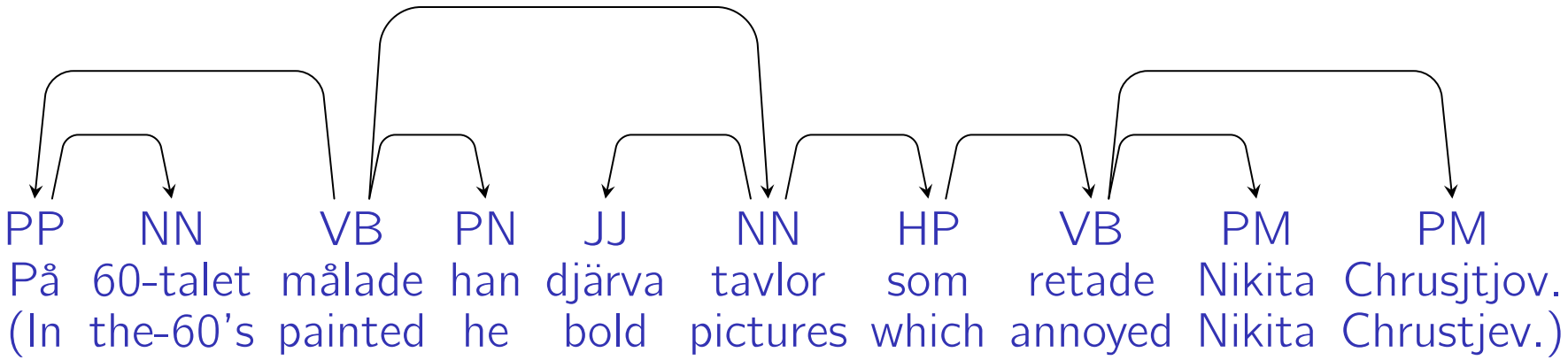
Parser Input

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

1	Deras	de	PRON
2	grundutbildning	grundutbildning	NOUN
3	tar	ta	VERB
4	450	450	NUM
5	dagar	dag	NOUN
6	.	.	PUNCT

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 (Arc-Eager)

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:

1. **Shift**, pushes the input token onto the stack
2. **Right arc**, adds an arc from the token on top of the stack to the next input token and pushes the input token onto the stack.
3. **Reduce**, pops the token on the top of the stack
4. **Left arc**, adds an arc from the next input token to the token on the top of the stack and pops it.

Transitions' Definition

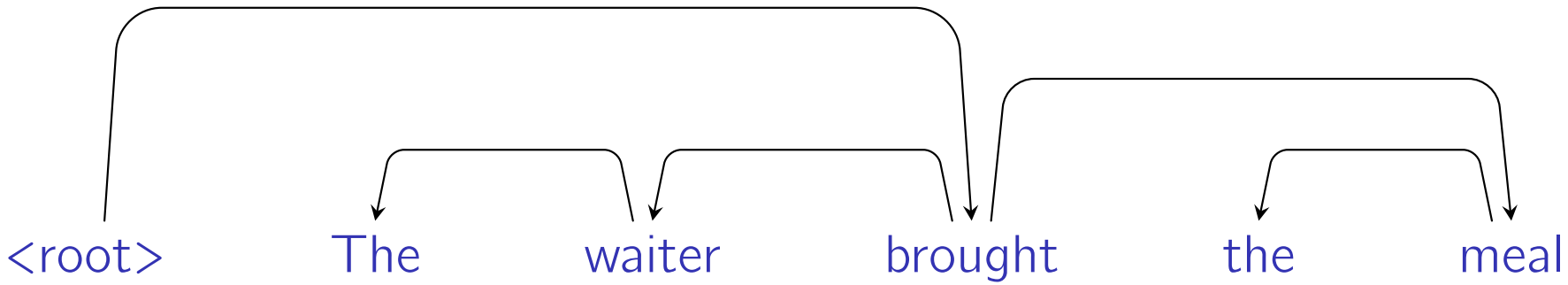
We use a triple: $\langle S, I, A \rangle$, where S is the stack, I, the input list, and A, the set of arcs in the graph.

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

1. Left-arc is an augmented reduce, and right-arc, an augmented shift.
2. The first condition $\exists n'(n', n) \in A$, where n' is the head and n, the dependent, is to ensure that the graph is connected.
3. The second condition $\nexists n''(n'', n) \in A$, where n'' is the head and n, the dependent, is to enforce a unique head.

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

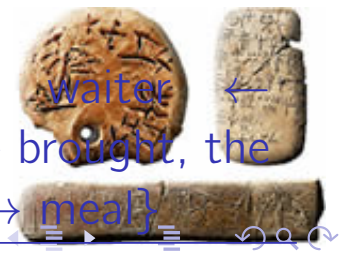
[sh, sh, la, sh, la, ra, sh, la, ra]

Trans.	Stack	Queue	Graph
start	$\emptyset$	[ROOT, the, waiter, brought, the, meal]	{}
sh			
	[ROOT]	[the, waiter, brought, the, meal]	{}
sh			
	$\begin{bmatrix}\text{the}\\ \text{ROOT}\end{bmatrix}$	[waiter, brought, the, meal]	{}
la			
	[ROOT]	[waiter, brought, the, meal]	{the ← waiter}
sh			
	$\begin{bmatrix}\text{waiter}\\ \text{ROOT}\end{bmatrix}$	[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			
	$\begin{bmatrix}\text{brought}\\ \text{ROOT}\end{bmatrix}$	[the, meal]	{the ← waiter, waiter ← brought, ROOT → brought}
sh			
	$\begin{bmatrix}\text{the}\\ \text{brought}\\ \text{ROOT}\end{bmatrix}$	[meal]	{the ← waiter, waiter ← brought, ROOT → brought}
la			
	$\begin{bmatrix}\text{brought}\\ \text{ROOT}\end{bmatrix}$	[meal]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal}
ra			
end	$\begin{bmatrix}\text{meal}\\ \text{brought}\\ \text{ROOT}\end{bmatrix}$	[]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal, brought → meal}



Nivre's Parser in Python: Shift and Reduce

We use a `stack`, a `queue`, and a partial `graph` that contains all the arcs.

```
def shift(stack, queue, graph):
    stack = [queue[0]] + stack
    queue = queue[1:]
    return stack, queue, graph
```

```
def reduce(stack, queue, graph):
    return stack[1:], queue, graph
```

Nivre's Parser in Python: Left-Arc

The partial graph is a dictionary of dictionaries with the heads and the functions (deprels): `graph['heads']` and `graph['deprels']`
The `deprel` argument is is either to assign a function or to read it from the manually-annotated corpus.

```
def left_arc(stack, queue, graph, deprel=False):
    graph['heads'][stack[0]['id']] = queue[0]['id']
    if deprel:
        graph['deprels'][stack[0]['id']] = deprel
    else:
        graph['deprels'][stack[0]['id']] = stack[0]['deprel']
    return reduce(stack, queue, graph)
```

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`.

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

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).
```

Tracing Nivre's Parser in Python

1	Deras	de	PRON	2	nmod:poss
2	grundutbildning	grundutbildning	NOUN	3	nsubj
3	tar	ta	VERB	0	root
4	450	450	NUM	5	nummod
5	dagar	dag	NOUN	3	obj
6	.	.	PUNCT	3	punct

Transitions:

```
['sh', 'sh', 'la.nmod:poss', 'sh', 'la.nsubj', 'ra.root',
'sh', 'la.nummod', 'ra.obj', 're', 'ra.punct']
```

Parser state:

```
{'heads': {'0': '0', '1': '2', '2': '3', '3': '0',
'4': '5', '5': '3', '6': '3'},
'deprels': {'0': 'ROOT', '1': 'nmod:poss', '2': 'nsubj', '3':
'4': 'nummod', '5': 'obj', '6': 'punct'}}
```

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. Parsing *The waiter brought the meal*

Stack	**Queue**	**Stack**		**Queue**		**Trans.**
POS(T_0)	POS(Q_0)	POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	
nil	ROOT	nil	nil	ROOT	DET	sh
ROOT	DET	ROOT	nil	DET	NOUN	sh
DET	NOUN	DET	ROOT	NOUN	VERB	la
ROOT	NOUN	ROOT	nil	NOUN	VERB	sh
NOUN	VERB	NOUN	ROOT	VERB	DET	la
ROOT	VERB	ROOT	nil	VERB	DET	ra
VERB	DET	VERB	ROOT	DET	NOUN	sh
DET	NOUN	DET	VERB	NOUN	nil	la
VERB	NOUN	VERB	ROOT	NOUN	nil	ra

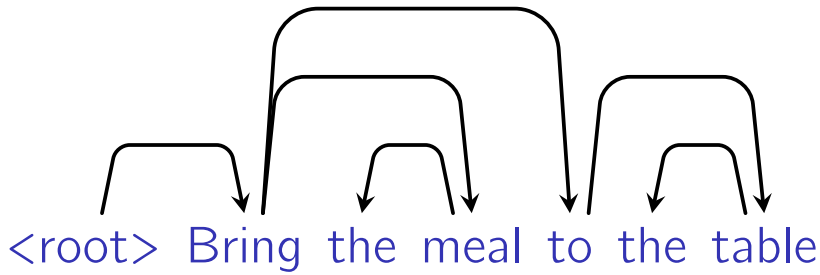
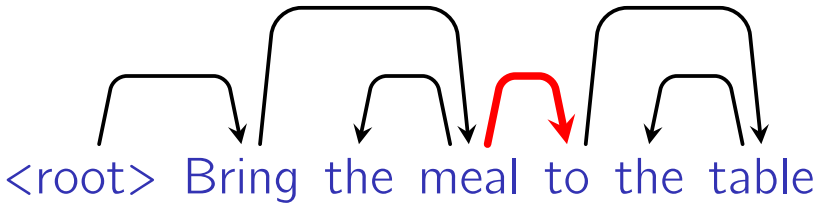
Using CoNLL data, you can train decision trees and implement a parser.

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.

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**
	

Parser Variant: Arc-Standard

Nivre's parser has two variants in addition to arc-eager: arc-standard (Yamada and Matsumoto) and swap
The first step is a POS tagging
The transitions are:

1. **Shift**, pushes the input token onto the stack
2. **Right arc**, adds an arc from the second token in the stack to the top of the stack and pops it.
3. **Left arc**, adds an arc from the top of the stack to the second in the stack and removes the second in the stack.

Transitions' Definition of Arc-Standard

Actions	**Parser states**	**Conditions**
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle [ROOT], [], A \rangle$	
Shift	$\langle S, [n\|I], A \rangle \rightarrow \langle [S\|n], I, A \rangle$	
Left-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\} \rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\} \rangle$	

Arc Standard Parser in Action (I)

Trans.	Stack	Queue	Graph
start	[ROOT]	[I, booked, a, ticket, to, Google]	{}
sh			
	$\begin{bmatrix} \text{I} \\ \text{ROOT} \end{bmatrix}$	[booked, a, ticket, to, Google]	{}
sh			
	$\begin{bmatrix} \text{booked} \\ \text{I} \\ \text{ROOT} \end{bmatrix}$	[a, ticket, to, Google]	{}
la			
	$\begin{bmatrix} \text{booked} \\ \text{ROOT} \end{bmatrix}$	[a, ticket, to, Google]	{I ← booked}
sh			
	$\begin{bmatrix} \text{a} \\ \text{booked} \\ \text{ROOT} \end{bmatrix}$	[ticket, to, Google]	{I ← booked}

Arc Standard Parser in Action (II)

Trans.	Stack	Queue	Graph
sh			
	$\begin{bmatrix}\text{ticket}\\ \text{a}\\ \text{booked}\\ \text{ROOT}\end{bmatrix}$	[to, Google]	{I ← booked}
la			
	$\begin{bmatrix}\text{ticket}\\ \text{booked}\\ \text{ROOT}\end{bmatrix}$	[to, Google]	{I ← booked, a ← ticket}
sh			
	$\begin{bmatrix}\text{to}\\ \text{ticket}\\ \text{booked}\\ \text{ROOT}\end{bmatrix}$	[Google]	{I ← booked, a ← ticket}

Arc Standard Parser in Action (III)

Trans.	Stack	Queue	Graph
sh			
	[Google, to, ticket, booked, ROOT]	[]	{I ← booked, a ← ticket}
ra			
	[to, ticket, booked, ROOT]	[]	{I ← booked, a ← ticket, to → Google}
ra			
	[ticket, booked, ROOT]	[]	{I ← booked, a ← ticket, to → Google, ticket → to}

Arc Standard Parser in Action (IV)

Trans.	Stack	Queue	Graph
ra			
	$\begin{bmatrix}\text{booked}\\ \text{ROOT}\end{bmatrix}$	[]	{I ← booked, a ← ticket, to → Google, ticket → to, booked → ticket}

Transitions' Definition of Swap

The Swap variant enables the parser to parse nonprojective sentences

Actions	**Parser states**	**Conditions**
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle [ROOT], [], A \rangle$	
Shift	$\langle S, [n\|I], A \rangle \rightarrow \langle [S\|n], I, A \rangle$	
Left-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\} \rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\} \rangle$	
Swap	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], [n\|I], A \rangle$	$n \neq ROOT$ and $inx(n) < inx(n')$

Application: Dependency Parsing for Knowledge Extraction

IBM Watson, the question-answering system, uses dependency parsing to

- Analyze questions and
- Extract knowledge from text.

Given the question:

> *POETS & POETRY: He was a bank clerk in the Yukon before* **he published "Songs of a Sourdough"** *in 1907*

Watson extracts:

```
authorOf(he, 'Songs of a Sourdough')
```

A predicate–argument structure representation would be:

```
published(he, 'Songs of a Sourdough')
```

IBM Watson: Parsing the Question

Watson parses the question in the form of dependencies.

- In the CoNLL format:

Inx	Form	Lemma	POS	Head	Funct.
1	he	he	pronoun	2	subject
2	published	publish	verb	0	root
3	Songs of a Sourdough	Songs of a Sourdough	noun	2	object

- In the Watson format:

```
lemma(1, "he").                    partOfSpeech(1,pronoun).
lemma(2, "publish").               partOfSpeech(2,verb).
lemma(3, "Songs of a Sourdough"). partOfSpeech(3,noun).

subject(2,1).
object(2,3).
```

IBM Watson: Inferences

Watson uses Prolog rules to detect author/composition relationships:

```
authorOf(Author,Composition) :-
    createVerb(Verb),
    subject(Verb,Author),
    author(Author),
    object(Verb,Composition),
    composition(Composition).

createVerb(Verb) :-
    partOfSpeech(Verb,verb),
    lemma(Verb,VerbLemma),
    member(VerbLemma, ["write", "publish",...]).
```

Eventually, the question is reduced to:

```
authorOf(he, 'Songs of a Sourdough')
```

IBM Watson: Evidence from Text

Watson parses large volumes of text, for instance Wikipedia and the *New York Times*.

From the excerpt:

> *Songs of a Sourdough by Robert W. Service*

Watson extracts:

```
authorOf('Robert W. Service', 'Songs of a Sourdough')
```

The classical predicate–argument structure representation of the same phrase is:

```
by('Songs of a Sourdough', 'Robert W. Service')
```

IBM Watson: Matching Evidences

The relation is extracted from text using the rule:

```
authorOf(Author,Composition) :-
    composition(Composition),
    argument(Composition,Preposition),
    lemma(Preposition, "by"),
    objectOfPreposition(Preposition,Author),
    author(Author).
```

Leading to:

```
authorOf('Robert W. Service', 'Songs of a Sourdough')
```

This predicate–argument structure can be compared to:

```
authorOf(he, 'Songs of a Sourdough')
```

Or unified with:

```
authorOf(X, 'Songs of a Sourdough')
```