Telecom ParisTech

Syntactic analysis
Parsing
SD213

Jean-Louis Dessalles

Friday, 11 May 2018

Links

- Natural Language Processing Techniques in Prolog
  Patrick Blackburn and Kristina Striegnitz
  http://cs.union.edu/~striegnk/courses/nlp-with-prolog/html/
- . . .

Symbolic NLP

- **Using grammars, rules and structures is important**
  - when accuracy is needed (*e.g.* for safety reasons),
  - to build context dependent semantic interpretations,
    - e.g. *to distinguish "the neighbour of my sister" from "the sister of my neighbour"*
    - *or to understand that "the sister of my neighbour" designates "Marlene"*
    - *or to capture the ambiguity of "Marlene will write the letter in then minutes"* *(= in 10 min. from now* vs. *it will take her 10 min.),*
  - to generate reasoning and argumentation on top of natural language understanding (*e.g.* "that will be too late").

Symbolic NLP

- **Symbolic NLP builds on linguistics**
  - Linguistics = science
  - Mechanisms
    - *Government & binding*
    - *Movement*
    - *Case*
    - *Aspect*
    - . . .
  - Implementable

Examples

- “She thinks that John’s sister is deaf”
  “The fact that she might be sick bothers Lisa”

The pronoun and the referential expression may designate the same person in the second sentence, but not in the first one.

« She thinks that Lisa’s sister is deaf »

[present] → think → that → is → deaf
↑ She
↑ John’s sister

« The fact that she might be sick bothers Lisa »

[present] → bother → Lisa
SD ↑ the → SN fact → SC that ⇒ SI might be → SA sick
↑ she

Coreference is blocked when the node right above the pronoun dominates the referential expression. (≈)

Examples

- “She thinks that John’s sister is deaf”
  “The fact that she might be sick bothers Lisa”
- “The boy she is talking about _.”
  « La fille qu’il a suivi _**e**. »

Linguistics postulates phrase movement.
Movement leaves a trace.
Trace are sometimes visible in French.

- “She looked at him **during** one minute”
  * “She looked at him **in** one minute”

  “She ate the whole cake **in** one minute”
  * “She ate the whole cake **during** one minute”

Aspect is processed algorithmically.

- “There are 24 lamps in the room.”

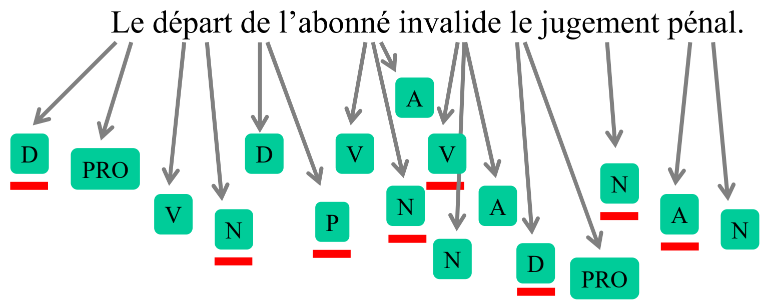
Relevance result from computations as well.

Symbolic NLP remains challenging

- Apparent complexity of language
- Ambiguity
- Robustness
- Interface with context
- Interface with reasoning

Efficient Algorithms

Ambiguity



288 combinations

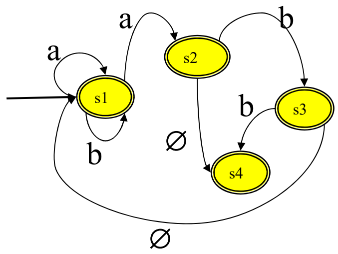
Combinatorial explosion.

- **Finite State automata**
- **Context-free grammars**
- **Syntactic parsing with CF grammars**
  - Top-down parsing
  - Bottom-up parsing
  - Chart parsing
- **Feature structures**

Rules

Gram

Finite State Automatons




```
final(s3).
trans(s1,a,s1).
trans(s1,a,s2).
trans(s1,b,s1).
trans(s2,b,s3).
trans(s3,b,s4).
silent(s2,s4).
silent(s3,s1).
```

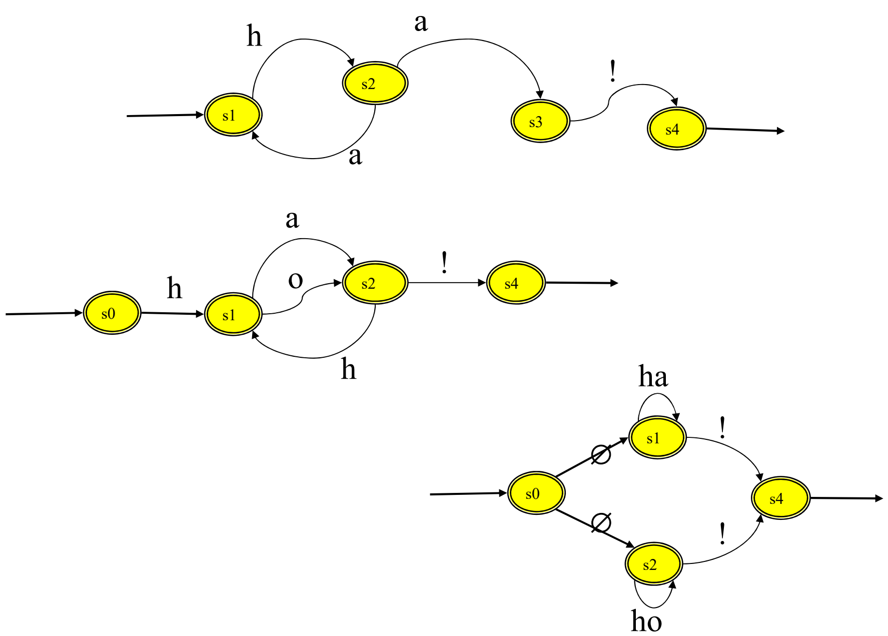
This automaton accepts *ab* or *aabaab*, but not *abb* nor *abba*.

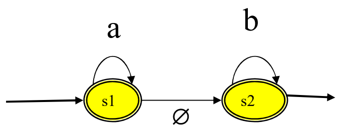
```
?- Str = [_,_,_], correct(s1, Str).
Str = [a,a,b];
Str = [b,a,b];
no
```

```
correct(Etat,[]) :-
        final(State).
correct(Etat,[X|Rest]) :-
        trans(State,X,State1),
        correct(State1,Rest).
correct(State, L) :-
        silent(State,State1),
        correct(State1,L).
```

npp autom.pl

autom.pl





This automaton accepts $a^n b^m$
but does not recognize $a^n b^n$
exclusively.

No FSA is able to do this.

- **Finite state automata**
  - can recognize regular languages
  - are useful in phonology and morphology
  - are insufficient to parse NL sentences

- **Finite State automata**
- **Context-free grammars**
- **Syntactic parsing with CF grammars**
  - Top-down parsing
  - Bottom-up parsing
  - Chart parsing
- **Feature structures**

Rules

Gram

Phrase Structure Grammars

Non-terminal symbols: abstract phrase constituent names, such as “sentence”, “noun”, “verb” (in blue)

Terminal symbols: words of the language, such as “Bob”, “eats”, “drinks”

Given two disjoint sets of symbols, N and T, a (context-free) grammar **is a relation between N and strings over $N \cup T$: $G \subset N \times (N \cup T)^*$**

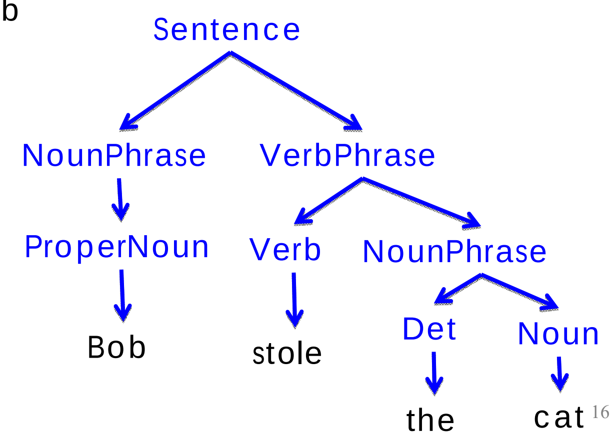
N = {Sentence, Noun, Verb}
T = {Bob, eats}

Sentence -> Noun Verb
Noun -> Bob
Verb -> eats

Production rules

A More Complex Example

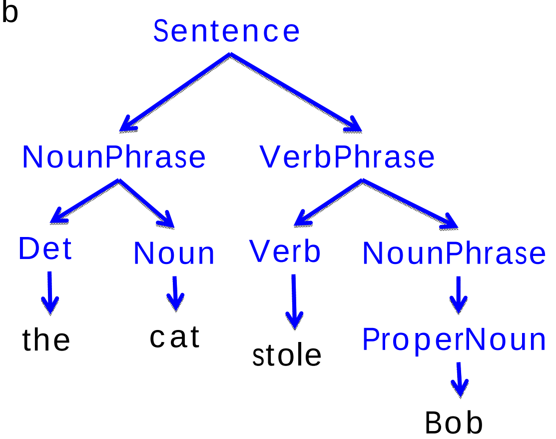
1. Sentence -> NounPhrase VerbPhrase
2. NounPhrase -> ProperNoun
3. VerbPhrase -> Verb NounPhrase
4. NounPhrase -> Det Noun
5. ProperNoun -> Bob
6. Verb -> stole
7. Noun -> cat
8. Det -> the



Fabian Suchanek

A More Complex Example

1. Sentence -> NounPhrase VerbPhrase
2. NounPhrase -> ProperNoun
3. VerbPhrase -> Verb NounPhrase
4. NounPhrase -> Det Noun
5. ProperNoun -> Bob
6. Verb -> stole
7. Noun -> cat
8. Det -> the




```
% --- Productions
sentence --> nounphrase,
verbphrase.
nounphrase --> propernoun.

nounphrase --> det, noun.

verbphrase --> verb,
nounphrase.

% Lexicon
propernoun --> [bob].

verb --> [stole].
noun --> [cat].
det -->  [the].
```

```
?- sentence(S, []).
S = [bob, stole, bob] ;
S = [bob, stole, the, cat] ;
S = [the, cat, stole, bob] ;
S = [the, cat, stole, the, cat].
```

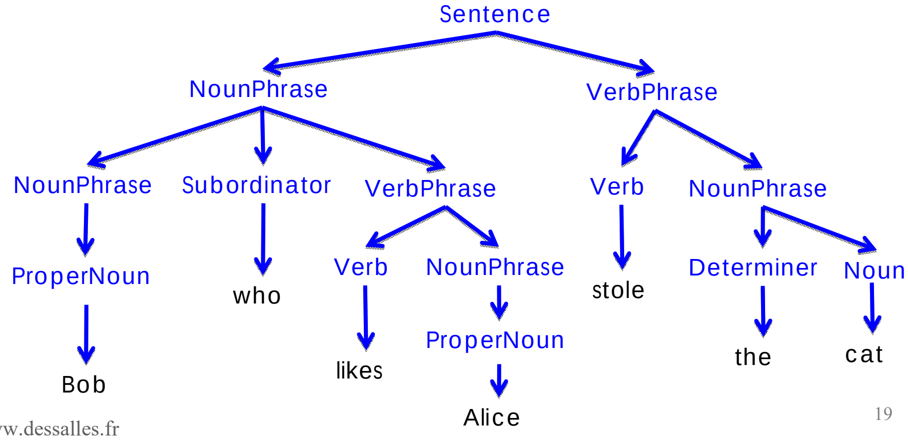
BobCat.pl

pl BobCat

Recursive Structures

1. Sentence -> NounPhrase VerbPhrase
2. NounPhrase -> ProperNoun
3. NounPhrase -> Determiner Noun
4. NounPhrase -> NounPhrase Subordinator VerbPhrase
5. VerbPhrase -> Verb NounPhrase

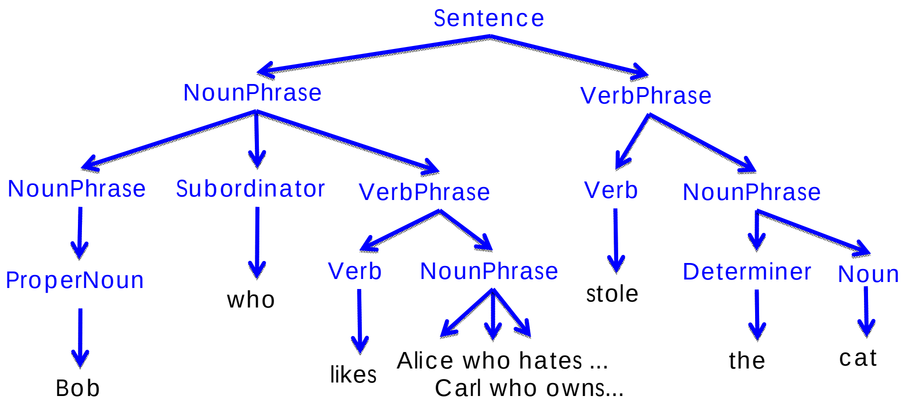
Recursive rules: allow a circle in the derivation



Fabian Suchanek

Recursive Structures

1. Sentence -> NounPhrase VerbPhrase
2. NounPhrase -> ProperNoun
3. NounPhrase -> Determiner Noun
4. NounPhrase -> NounPhrase Subordinator VerbPhrase
5. VerbPhrase -> Verb NounPhrase



Language

The **language of a grammar** is the set of all sentences that can be derived from the start symbol by rule applications.

Bob stole the cat
Bob stole Alice
Alice stole Bob who likes the cat
The cat likes Alice who stole Bob
Bob likes Alice who likes Alice who...
...

The grammar is a finite description of an infinite set of sentences

~~The Bob stole likes.~~
~~Stole stole stole.~~
~~Bob cat Alice likes.~~

- **Finite State automata**
- **Context-free grammars**
- **Syntactic parsing with CF grammars** ←
  - Top-down parsing
  - Bottom-up parsing
  - Chart parsing
- **Feature structures**

Rules

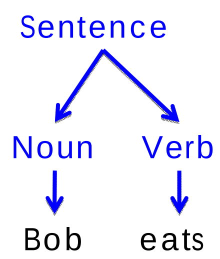
Gram

Parsing

Parsing is the process of, given a grammar and a sentence, finding the phrase structure tree.

Sentence -> Noun Verb
Noun -> Bob
Verb -> eats

N = {Sentence, Noun, Verb}
T = {Bob, eats}

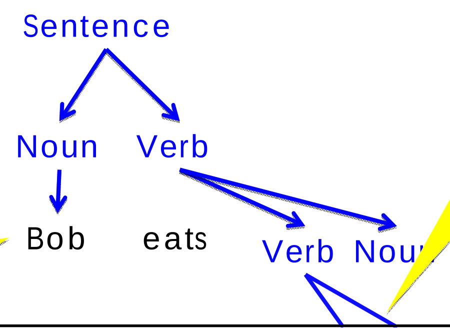


Parsing

A naïve parser would try all rules systematically from the top to arrive at the sentence.

Sentence -> Noun Verb
Noun -> Bob
Verb -> eats
Verb -> Verb Noun

N = {Sentence, Noun, Verb}
T = {Bob, eats}



This can go very wrong with recursive rules

Going bottom up is not much smarter

Top-down recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).
```

```
% --- Grammar
s --> np, vp.
np --> det, n.
np --> det, n, pp.
np --> [kirk].
vp --> v.
vp --> v, np.
vp --> v, pp.
vp --> v, np, pp.
vp --> v, pp, pp.
pp --> p, np.
```

```
% -- Lexicon
det --> [the].
det --> [my].
det --> [her].
det --> [his].
det --> [a].
det --> [some].
n --> [dog].
n --> [daughter].
n --> [son].
n --> [sister].
n --> [aunt].
n --> [neighbour].
n --> [cousin].
```

```
v --> [grumbles].
v --> [likes].
v --> [gives].
v --> [talks].
v --> [annoys].
v --> [hates].
v --> [cries].
p --> [of].
p --> [to].
p --> [about].
```


Top-down recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).
```

```
s --> [np,vp]
np --> [det,n]
det --> [the]
        ****  recognized: the
n --> [dog]
n --> [daughter]
n --> [son]
n --> [sister]
        ****  recognized: sister
vp --> [v]
v --> [grumbles]
v --> [likes]
v --> [gives]
v --> [talks]
        ****  recognized: talks
v --> [annoys]
v --> [hates]
v --> [cries]
vp --> [v,np]
v --> [grumbles]
v --> [likes]
v --> [gives]
v --> [talks]
        ****  recognized: talks
```

```
np --> [det,n]
det --> [the]
det --> [my]
det --> [her]
det --> [his]
det --> [a]
det --> [some]
np --> [det,n,pp]
det --> [the]
det --> [my]
det --> [her]
det --> [his]
det --> [a]
det --> [some]
np --> [kirk]
v --> [annoys]
v --> [hates]
v --> [cries]
vp --> [v,pp]
v --> [grumbles]
v --> [likes]
v --> [gives]
v --> [talks]
        ****  recognized: talks
```

```
pp --> [p,np]
p --> [of]
p --> [to]
p --> [about]
        ****  recognized: about
np --> [det,n]
det --> [the]
det --> [my]
det --> [her]
        ****  recognized: her
n --> [dog]
n --> [daughter]
n --> [son]
n --> [sister]
n --> [aunt]
n --> [neighbour]
n --> [cousin]
        ****  recognized: cousin
true .
```

Top-down recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).

tdr(Proto, Words) :- % top-down recognition - Proto = list of non-terminals or words
    match(Proto, Words, [ ], [ ]).  % Final success. This means that Proto = Words
tdr([X|Proto], Words) :-  % top-down recognition.
    rule(X, RHS),  % retrieving a candidate rule that matches X
    append(RHS, Proto, NewProto),  % replacing X by RHS (= right-hand side)
    % see if beginning of NewProto matches beginning of Words
    match(NewProto, Words, NewProto1, NewWords),
tdr(NewProto1, NewWords).  % lateral recursive call

% match() eliminates common elements at the front of two lists
match([X|L1], [X|L2], R1, R2) :-
    !,
    write('\t**** recognized: '), write(X),
    match(L1, L2, R1, R2).
match(L1, L2, L1, L2).
```


Top-down recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).
```

```
% --- Grammar
s --> np, vp.
np --> det, n.
~~np --> det, n, pp.~~
np --> np, pp.
np --> [kirk].
vp --> v.
vp --> v, np.
vp --> v, pp.
vp --> v, np, pp.
vp --> v, pp, pp.
pp --> p, np.
```

```
% -- Lexicon
det --> [the].
det --> [my].
det --> [her].
det --> [his].
det --> [a].
det --> [some].
n --> [dog].
n --> [daughter].
n --> [son].
n --> [sister].
n --> [aunt].
n --> [neighbour].
n --> [cousin].
```

```
v --> [grumbles].
v --> [likes].
v --> [gives].
v --> [talks].
v --> [annoys].
v --> [hates].
v --> [cries].
p --> [of].
p --> [to].
p --> [about].
```

Introducing a left-recursive rule leads naïve top-down parsing to loop.

Top-down recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).
```

```
s --> [np,vp]
np --> [det,n]
det --> [the]       ****  recognized: the
n --> [dog]
n --> [daughter]
n --> [son]
n --> [sister]      ****  recognized: sister
vp --> [v]
v --> [grumbles]
v --> [likes]
v --> [gives]
v --> [talks]       ****  recognized: talks
v --> [annoys]
v --> [hates]
v --> [cries]
vp --> [v,np]
v --> [grumbles]
v --> [likes]
v --> [gives]
v --> [talks]       ****  recognized: talks
```

```
np --> [det,n]
det --> [the]
det --> [my]
det --> [her]
det --> [his]
det --> [a]
det --> [some]
np --> [np,pp]
np --> [det,n]
det --> [the]
det --> [my]
det --> [her]
det --> [his]
det --> [a]
det --> [some]
np --> [np,pp]
np --> [det,n]
det --> [the]
det --> [my]
det --> [her]
det --> [his]
det --> [a]
det --> [some]
np --> [np,pp]
```

www.dessalles.fr

29

Bottom-up recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).
```

```
[the,sister,talks,about,her,cousin]
[det,sister,talks,about,her,cousin]
[det,n,talks,about,her,cousin]
[np,talks,about,her,cousin]
[np,v,about,her,cousin]
[np,vp,about,her,cousin]
[s,about,her,cousin]
[s,p,her,cousin]
[s,p,det,cousin]
[s,p,det,n]
[s,p,np]
[s,pp]
[s,p,her,n]
[s,p,det,n]
[s,p,np]
[s,pp]
[s,about,det,cousin]
[s,p,det,cousin]
[s,p,det,n]
[s,p,np]
[s,pp]
[s,about,det,n]
[s,p,det,n]
[s,p,np]
[s,pp]
```

```
...
[np,vp,p,np]
[s,p,np]
[s,pp]
[np,vp,pp]
[s,pp]
[np,v,p,det,n]
[np,vp,p,det,n]
[s,p,det,n]
[s,p,np]
[s,pp]
[np,vp,p,np]
[s,p,np]
[s,pp]
[np,vp,pp]
[s,pp]
[np,v,p,np]
[np,vp,p,np]
[s,p,np]
[s,pp]
[np,vp,pp]
[s,pp]
[np,v,pp]
[np,vp,pp]
[s,pp]
[np,vp]
[s]
```

Naïve bottom-up parsing is not disturbed by left-recursive rules. However, it is still amnesic (reanalyzing the same phrases again and again).

www.dessalles.fr

30

Bottom-up recognition

```
test :-    tdr([s], [the, sister, talks, about, her, cousin]).

bup([s]).  % success when one gets s after a sequence of transformations
bup(P):-
    write(P), nl, get0(_),
    append(Pref, Rest, P),  % P is split into three pieces
    append(RHS, Suff, Rest),  % P = Pref + RHS + Suff
    rule(X, RHS),       % bottom up use of rule
    append(Pref, [X|Suff], NEWP),  % RHS is replaced by X in P: NEWP = Pref + X + Suff
    bup(NEWP).  % lateral recursive call

go :-
    bup([the, sister, talks, about, her, cousin]).
```

tdr([s], [blahblah, sister, talks, about, her, cousin]).

loops !

Naïve bottom-up parsing is not immune to looping either.

- **These method are amnesic**
  - Solution: remember parsed phrases

    → chart parsing
    - *Earley's algorithm* (top-down)
    - *CYK algorithm* (bottom-up)

- **Finite State automata**
- **Context-free grammars**
- **Syntactic parsing with CF grammars**
  - Top-down parsing
  - Bottom-up parsing
  - Chart parsing
- **Feature structures**

Rules

Gram

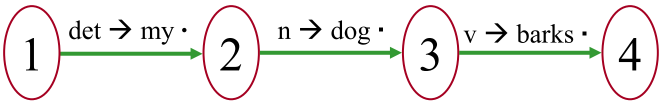
CYK parser

- **Bottom-up chart parsing**
- **Simplified implementation**
  - edge(StartPos, EndPos, Cat, Done, Rest)
    - *StartPos: Position in input of the first word of the candidate phrase*
    - *EndPos: Position in input of the last recognized word for the candidate phrase*
    - *Cat: left hand side of the candidate rule*
    - *Done: list of categories in the right-hand side of the candidate rule that have been recognized*
    - *Rest: list of categories in the right-hand side of the candidate rule that are not yet recognized*

'edge' stores a table of partially analyzed phrases.
It can be seen as a graph, where nodes are separations between words and edges are partially analyzed phrases.

Sentence to parse: my dog barks

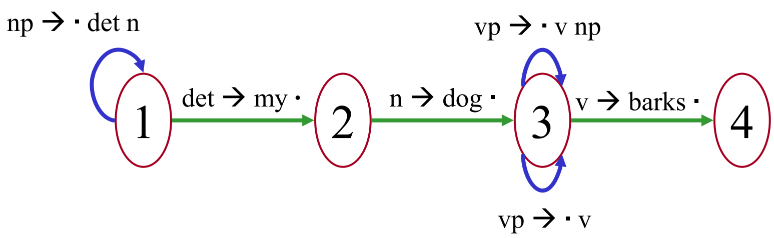
```
s
 |__np
 | |__det : my
 | |__n : dog
 |__vp
   |__v : barks
```



Lexical edges

Sentence to parse: my dog barks

```
s
 |__np
 | |__det : my
 | |__n : dog
 |__vp
   |__v : barks
```

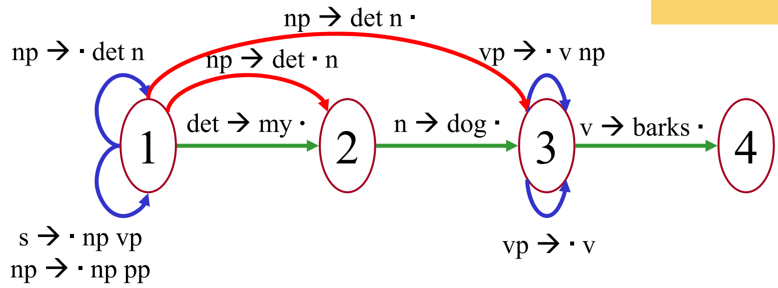


Generating new edges

Each time an edge becomes inactive (the dot reaches the end, which means that the phrase has been fully analyzed), new candidate edges (those that start with the recognized phrase) are generated with a dot at the front.

Sentence to parse: my dog barks

```
s
 |__np
 | |__det : my
 | |__n : dog
 |__vp
   |__v : barks
```



Extending edges

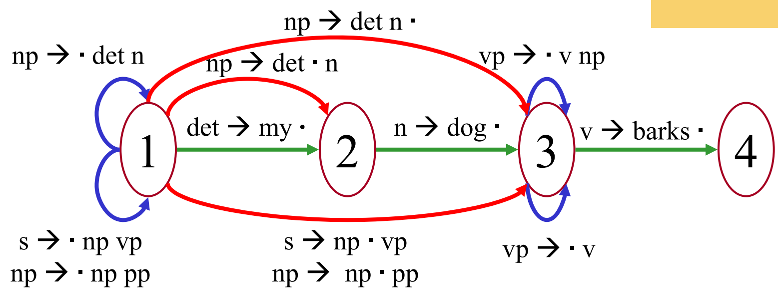
When an edge becomes inactive, edge waiting for the corresponding phrase generate new edges by extension (their dot is moved on step to the right).

www.dessalles.fr

37

Sentence to parse: my dog barks

```
s
 |__np
 | |__det : my
 | |__n : dog
 |__vp
   |__v : barks
```

Extending edges

When an edge becomes inactive, edge waiting for the corresponding phrase generate new edges by extension (their dot is moved on step to the right).

www.dessalles.fr

38

Sentence to parse: my dog barks

Lexical edge :	1->2	det --> my ·	det(my)
Generating edge:	1->1	np --> · det n	np
Extending edge :	1->2	np --> det · n	np(det(my))
Lexical edge :	2->3	n --> dog ·	n(dog)
Extending edge :	1->3	np --> det n ·	np(det(my),n(dog))
Generating edge:	1->1	s --> · np vp	s
Generating edge:	1->1	np --> · np pp	np
Extending edge :	1->3	s --> np · vp	s(np(det(my),n(dog)))
Extending edge :	1->3	np --> np · pp	np(np(det(my),n(dog)))
Lexical edge :	3->4	v --> barks ·	v(barks)
Generating edge:	3->3	vp --> · v	vp
Generating edge:	3->3	vp --> · v np	vp
Generating edge:	3->3	vp --> · v pp	vp
Generating edge:	3->3	vp --> · v np pp	vp
Generating edge:	3->3	vp --> · v pp pp	vp
Extending edge :	3->4	vp --> v ·	vp(v(barks))
Extending edge :	1->4	**s --> np vp ·**	**s(np(det(my),n(dog)),vp(v(barks)))**
Extending edge :	3->4	vp --> v · np	vp(v(barks))
Extending edge :	3->4	vp --> v · pp	vp(v(barks))
Extending edge :	3->4	vp --> v · np pp	vp(v(barks))
Extending edge :	3->4	vp --> v · pp pp	vp(v(barks))

```
s
 |__np
 |  |__det : my
 |  |__n : dog
 |__vp
    |__v : barks
```


- **Finite State automata**
- **Context-free grammars**
- **Syntactic parsing with CF grammars**
  - Top-down parsing
  - Bottom-up parsing
  - Chart parsing
- **Feature structures**

Rules

Gram

What we cannot (yet) do

What is difficult to do with context-free grammars:

- agreement between words

  Bob kicks the dog.
  I kicks the dog. ✗

- sub-categorization frames

  Bob sleeps.
  Bob sleeps you. ✗

- meaningfulness

  Bob switches the computer off.
  Bob switches the cat off. ✗

We could differentiate VERB3rdPERSON and VERB1stPERSON, but this would multiply the non-terminal symbols exponentially.

Agreement features

np(Number) --> det(Number), n(Number).

det(singular) --> [a].
det(plural) --> [many].
det(_) --> [the].
n(singular) --> [dog].
n(plural) --> [dogs].

vp --> v(none).
vp --> v(transitive), np.
vp --> v(intransitive), pp.

v(none) --> [sleeps].
v(transitive) --> [likes].
v(intransitive) --> [talks].

?- np(N, [a, dog], [ ]).
N = singular.
?- np(N, [many, dogs], [ ]).
N = plural.
?- np(N, [many, dog], [ ]).
false

Semantic features

```
s --> np(Sem), vp(Sem).
np(Sem) --> det, n(Sem).
vp(Sem) --> v(Sem, _).
vp(Sem1) --> v(Sem1, Sem2), np(Sem2).

n(sentient) --> [daughter]; [sister]; [aunt]; [sister].
n(nonEdible) --> [door].
n(edible) --> [apple].
v(sentient, _) --> [sleeps].
v(sentient, _) --> [likes].
v(sentient, edible) --> [eats].
v(sentient, _) --> [talks].
v(sentient, _) --> [give].
```

```
?- s([the, daughter, eats, the, door], [ ]).
false.

?- s([the, daughter, eats, the, apple], [ ]).
true .
```


Fabian Suchanek

Feature Structures

A **feature structure** is a mapping from attributes to values.
Each **value** is an atomic value or a feature structure.

A sample feature structure:
Category = Noun
Agreement = { Number = Singular
Person = Third }

Attribute = Value

Represented differently:

$$\begin{bmatrix} \text{Category} & \text{Noun} \\ \text{Agreement} & \begin{bmatrix} \text{Number} & \text{Singular} \\ \text{Person} & \text{Third} \end{bmatrix} \end{bmatrix}$$

Feature Structure Grammars

A **feature structure grammar** combines traditional grammar with feature structures in order to model agreement.

Sentence -> Noun Verb

$$\begin{bmatrix}\text{Cat.} & \text{Sentence}\end{bmatrix} \rightarrow \begin{bmatrix}\text{Cat.} & \text{Noun}\\ \text{Number} & [1]\end{bmatrix} \begin{bmatrix}\text{Cat.} & \text{Verb}\\ \text{Number} & [1]\end{bmatrix}$$

The grammatical rule contains feature structures instead of non-terminal symbols

A feature structure can cross-refer to a value in another structure

Feature Structure Grammars

A **feature structure grammar** combines traditional grammar with feature structures in order to model agreement.

$$\begin{bmatrix}\text{Cat.} & \text{Sentence}\end{bmatrix} \rightarrow \begin{bmatrix}\text{Cat.} & \text{Noun}\\ \text{Number} & [1]\end{bmatrix} \begin{bmatrix}\text{Cat.} & \text{Verb}\\ \text{Number} & [1]\end{bmatrix}$$

$$\begin{bmatrix}\text{Cat.} & \text{Noun}\\ \text{Number} & \text{Singular}\\ \text{Gender} & \text{Male}\end{bmatrix} \rightarrow \text{Bob}$$

Rules with terminals have constant values in their feature structures

Rule Application

Grammar rules are applied as usual.

```
[Cat. Sentence]

    Apply rule    [Cat. Sentence] -> ....

[Cat.    Noun]  [Cat.    Verb]
[Number  [1] ]  [Number  [1] ]

                [Cat.    Noun    ]  ->  Bob
Apply rule      [Number  Singular]
                [Gender  Male    ]
```

Feature structures have to be **unified** before applying a rule: Additional attributes are added, references instantiated, and values matched (possibly recursively)

Fabian Suchanek

Unification

Grammar rules are applied as usual.

```
[Cat. Sentence]

    Apply rule    [Cat. Sentence] -> ....

[Cat.    Noun]  [Cat.    Verb]
[Number  [1] ]  [Number  [1] ]  -> Singular

                [Cat.    Noun    ]
                [Number  Singular]
                [Gender  Male    ]

                [Cat.    Noun    ]
Unification:    [Number  Singular]
                [Gender  Male    ]
```

Value matched: Noun=Noun

Reference instantiated: [1] = Singular

Attribute added: Gender=Male

www.dessalles.fr

Fabian Suchanek

Unification

Grammar rules are applied as usual.

[Cat. Sentence]

Apply rule [Cat. Sentence] -> ....

[Cat. Noun, Number [1]] [Cat. Verb, Number [1]] → Singular

Unify, then apply rule

[Cat. Noun, Number Singular, Gender Male] -> Bob

Unified feature structure is thrown away, its only effect was (1) compatibility check and (2) ref. instantiation

Bob [Cat. Verb, Number Singular]

Now we can make sure the verb is singular, too.

Implementing feature structures

np([number:sing, person:3, gender:feminine, sentience:true]) --> [mary].

v([subj:[number:sing, person:3, gender:_, sentience:true], event:false]) --> [thinks].

v([subj:[number:sing, person:3, gender:_, sentience:_], event:true]) --> [falls].

s --> gn(FS), v([subj:FS | _]).

Variable-length feature structures

```
?-  A = [number:sing, person:3, sentience:true, gender:feminine | _],
    B = [number:sing, person:3 | _],
  A = B.

  A = [number:sing, person:3, sentience:true, gender:feminine|_G1512],
  B = [number:sing, person:3, sentience:true, gender:feminine|_G1512].
```

Order can be made irrelevant.

Recursive feature structures

```
[ event:false, subj:[cat:np, number:sing, person:3, sentience:true|_],
          compl:[cat:clause|_] |_]

[subj:[person:3 |_], event:false |_]
```

Feature Structures Summary

Feature structures can represent additional information on grammar symbols and enforce agreement.

Various more sophisticated grammars use feature structures:
- head-driven phrase structure grammars (HPSG)
- Lexical-functional grammars (LFG)

- **Symbolic NLP remains challenging**
  - Apparent complexity of language
  - Ambiguity
  - Robustness
  - Interface with context
  - Interface with reasoning
- **Efficient Algorithms**