Prolog
et
Traitement Automatique des Langues

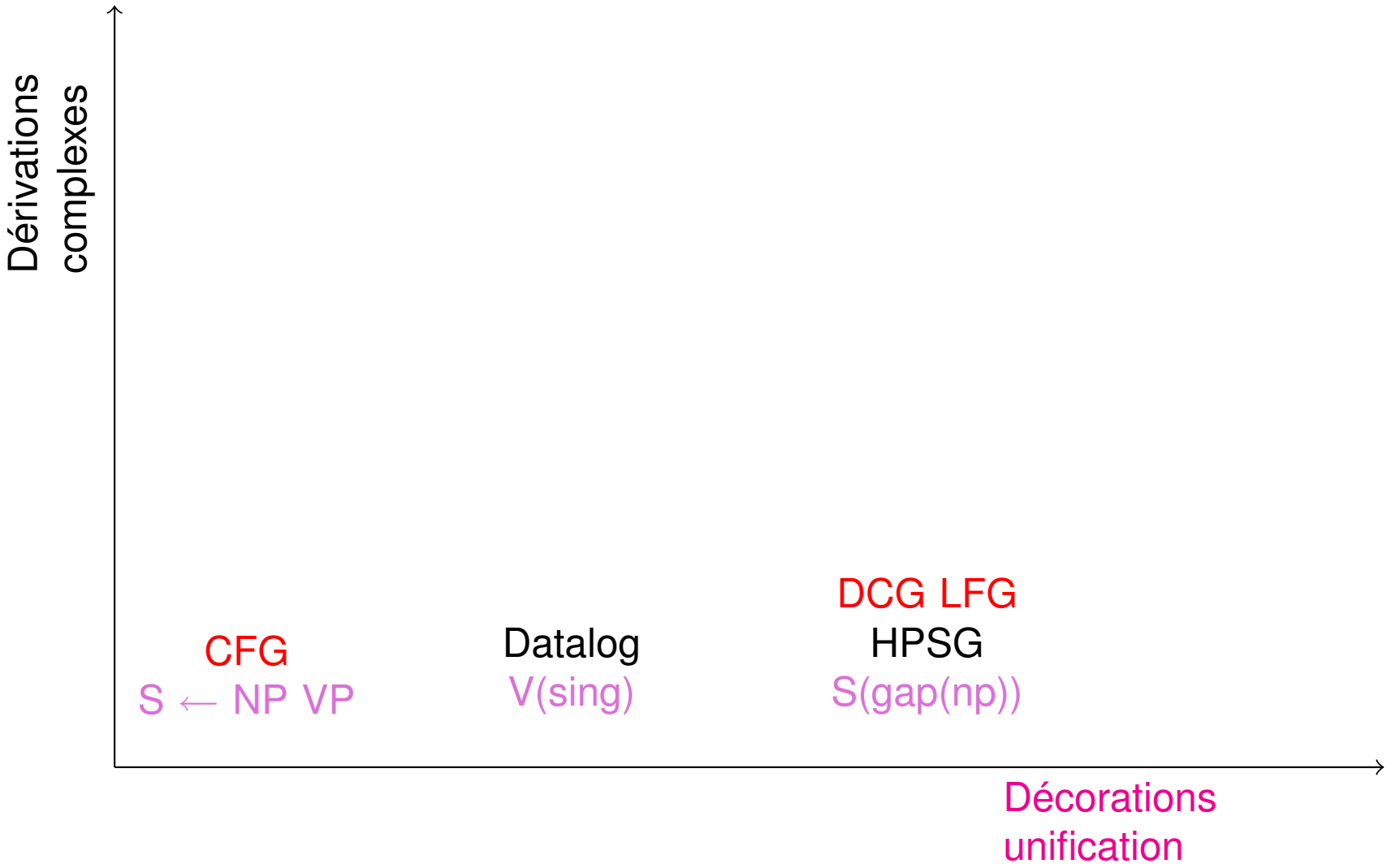
Éric de la Clergerie
Eric.De_La_Clergerie@inria.fr

ALPAGE – INRIA
http://alpage.inria.fr

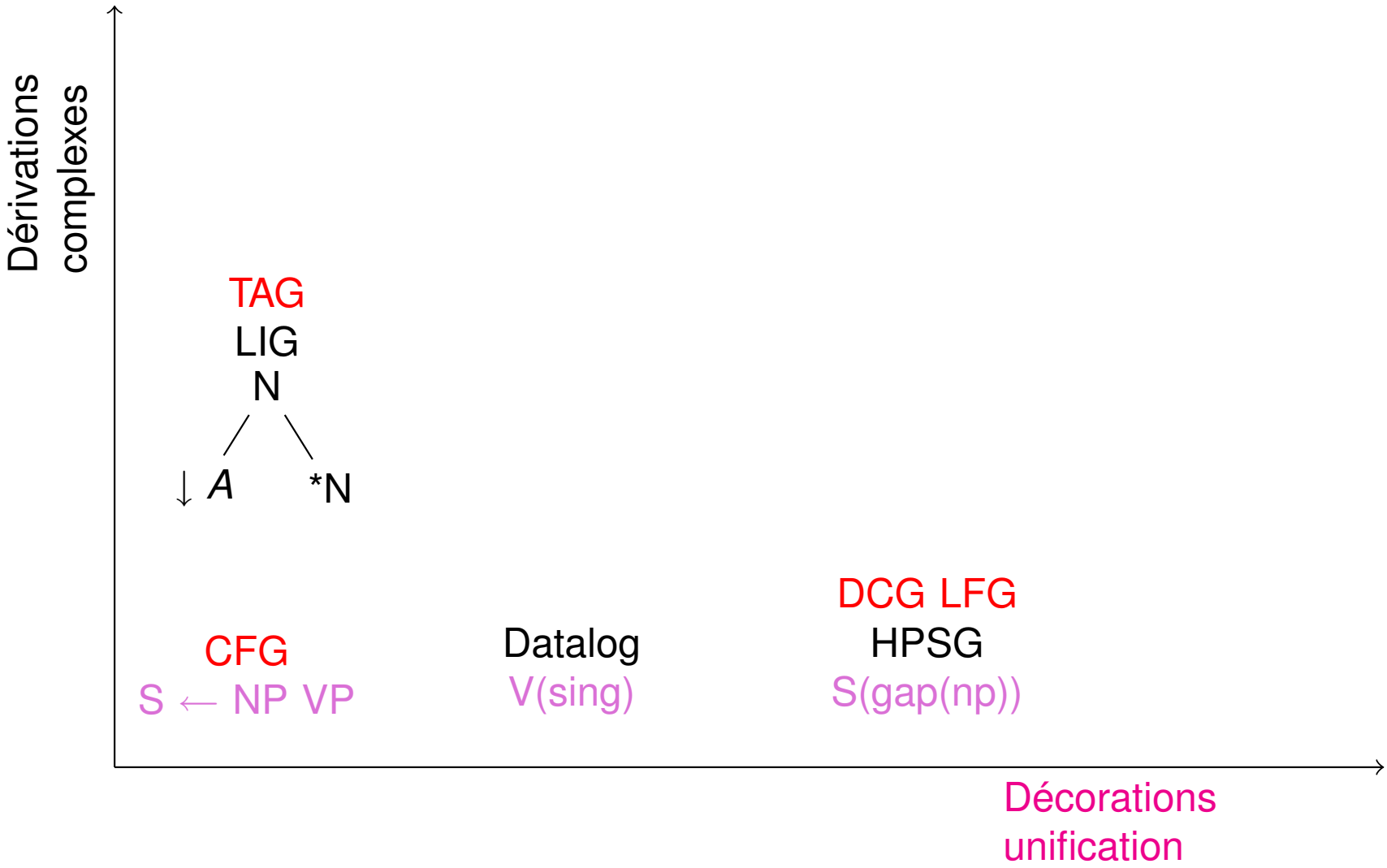
Cours M2 LI 2008

Aujourd'hui: Vers de nouveaux formalismes

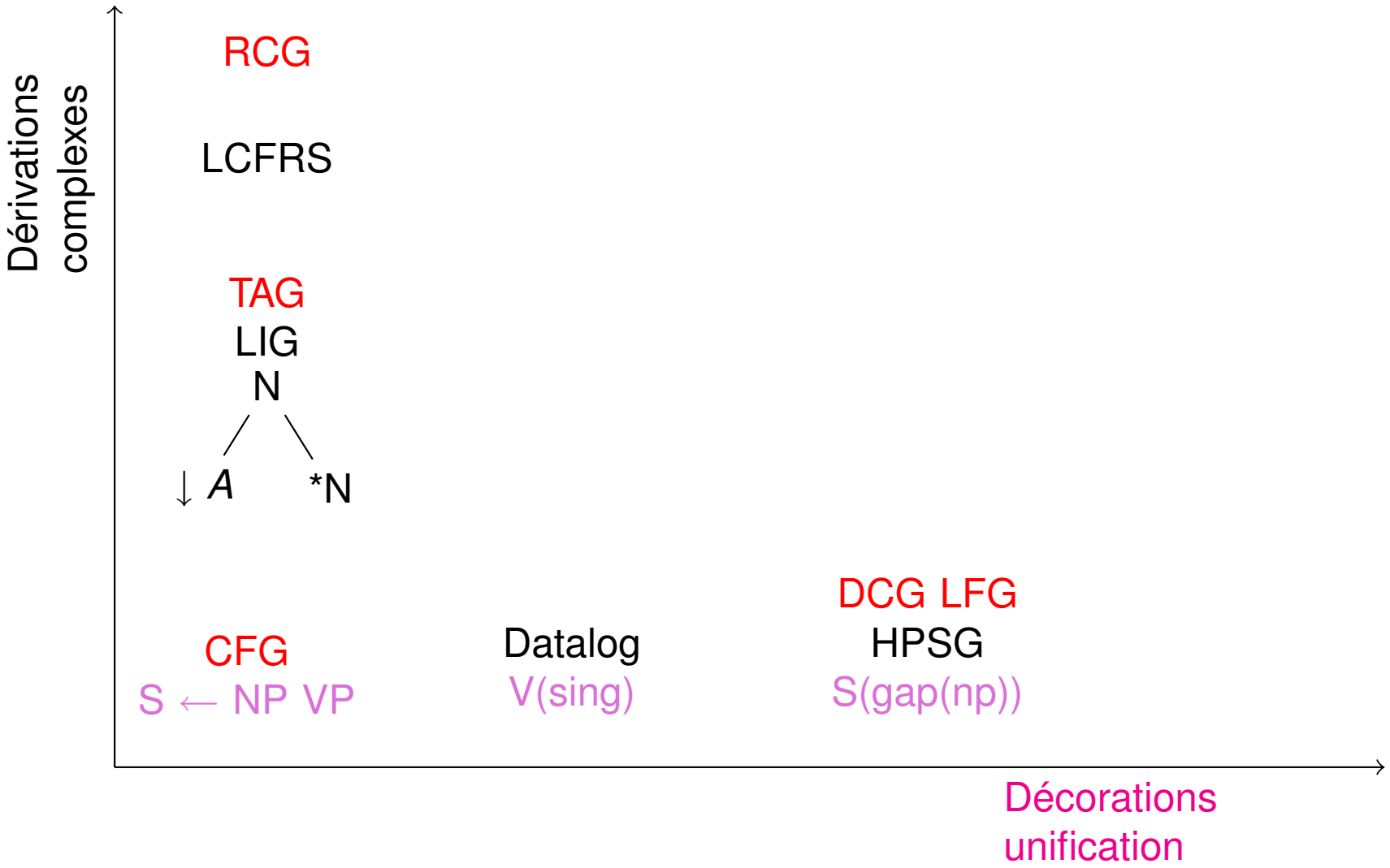
Un spectre de formalismes syntaxiques



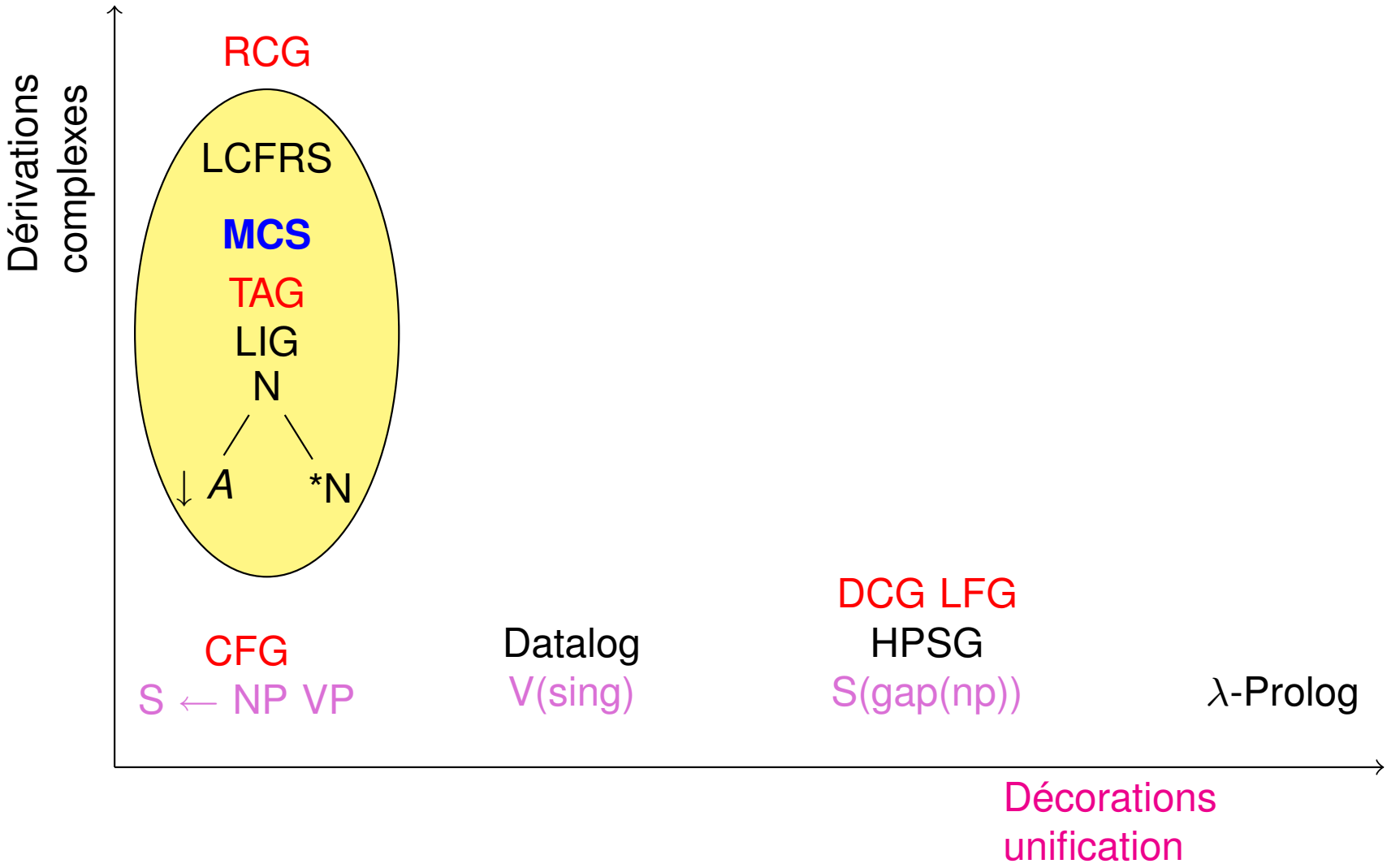
Un spectre de formalismes syntaxiques



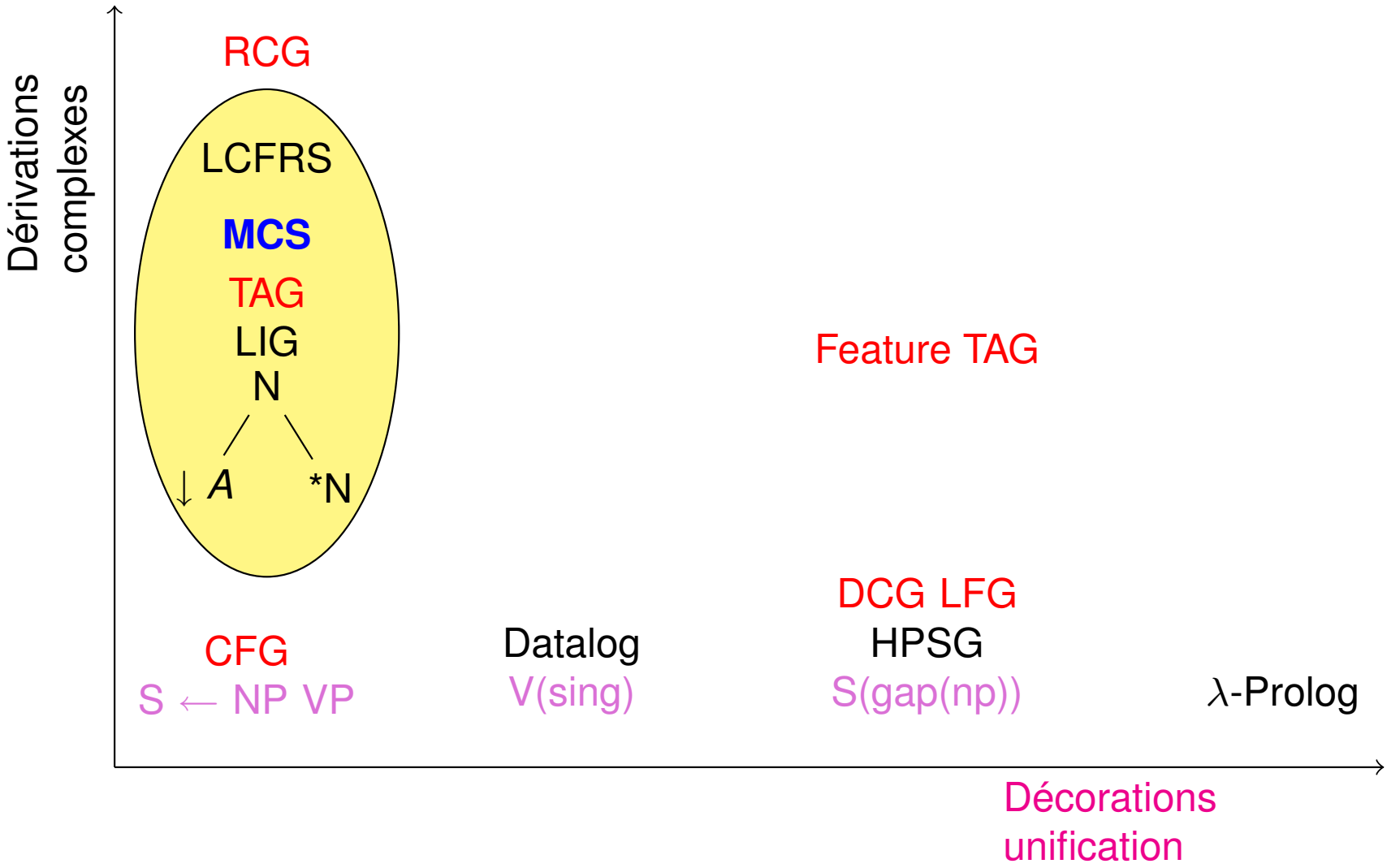
Un spectre de formalismes syntaxiques



Un spectre de formalismes syntaxiques



Un spectre de formalismes syntaxiques

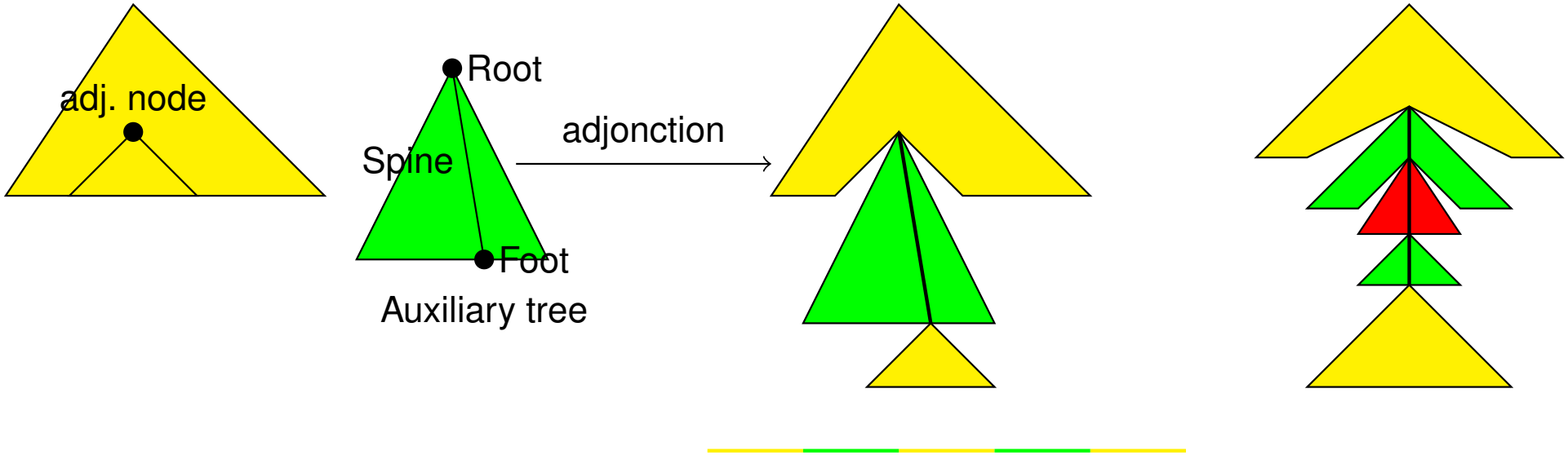


Treizième partie XIII

Grammaires d'adjonction d'arbres

Tree Adjoining Grammars [TAG]

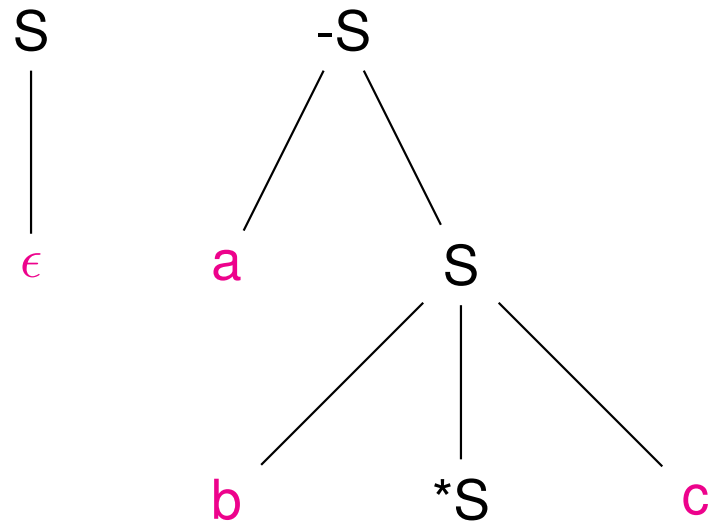
TAGs [Joshi] build derived trees from elementary initial and auxiliary trees by substitution and adjoining.



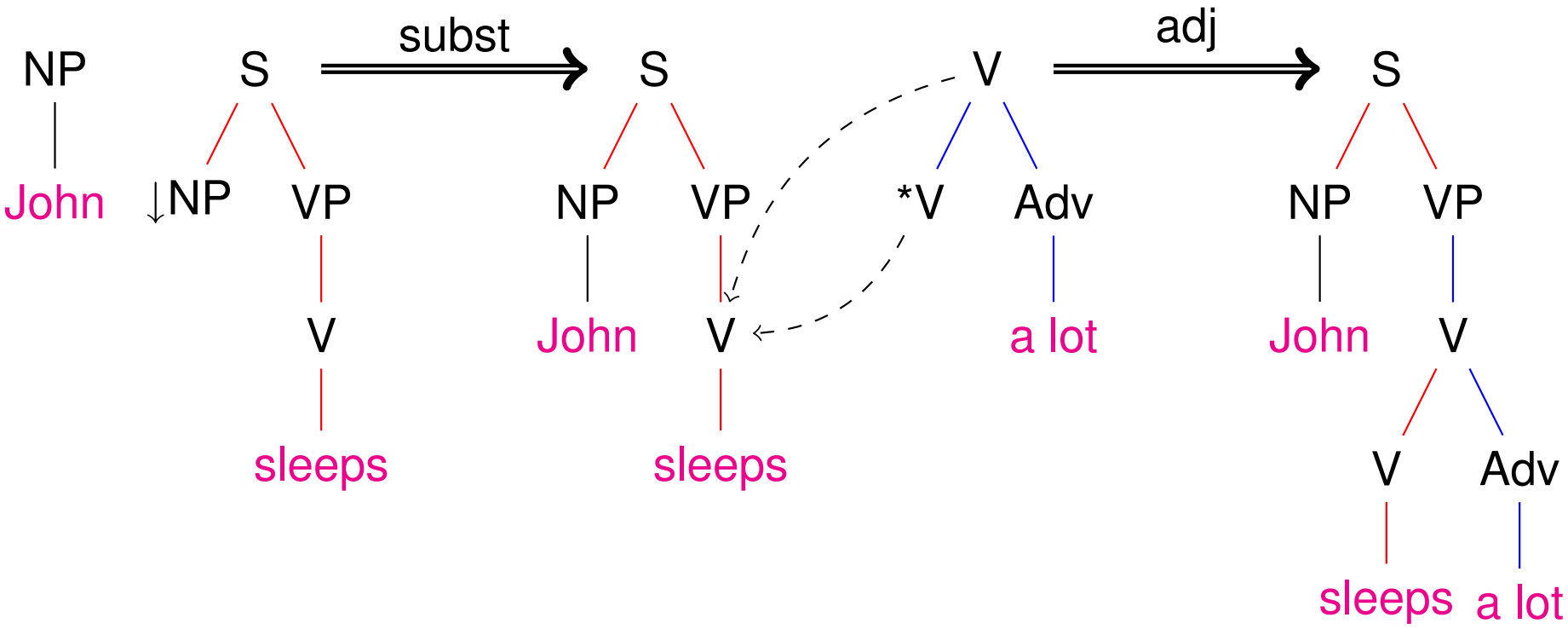
Tabular parsing in worst case time complexity $O(n^6)$ for pure TAGs.
Nodes may be decorated by pairs top and bot of logical attributes ([Feature TAG]).

Exemple jouet DyALog

```
%% a^nb^nc^n
tree s("").
auxtree -s("a",s("b",*s,"c")).
?-tag_phrase(s,0,N).
```



Plus linguistique




```
tree  np([John]).
tree  s(np,vp(v([sleeps]))).
auxtree   v(*v,adv(['a lot'])).
```

Modèle d'analyseur par continuation

Facile d'implémenter des méta-analyseurs tabulaires avec DyALog

```
:-mode([parse_node/5,parse_subtree/5], +(+,+,-,+,-) ).

parse_node( tag_node{ kind=>foot, cat=>Cat},
                    Left,Right, Adj_In, Left*Right
          ) :-
  '$answers'(register_continuation(Adj_In,Label,Old_Node,Old_Adj_In)),
  parse_subtree(Old_Node,Left,Right,Old_Adj_In,_).

parse_node( Node::tag_node{ kind=>std, cat=>Cat},
                    Left,Right, Adj_In,Adj_Out
          ) :-
  (   parse_subtree(Node,Left,Right,Adj_In,Adj_Out)
  ;
      register_continuation(Left,Cat,Node,Adj_In),
      parse_adj(Cat,Left,Right,Foot_Left*Foot_Right),
     '$answers'( parse_subtree(Node,Foot_Left,Foot_Right,Adj_In,Adj_Out)
    )
  ).

%% Register a continuation for the subtree below an adjunction node
register_continuation(Left,Cat,Node,Adj).
```

Construire des arbres

Structure d'un noeud assez complexe :

- un non-terminal ou un terminal
- un type de noeud indiqué par par une marque préfixe
  noeud pied $*$N ; ancre <>N, coancre <=>N
- des restrictions d'adjonction avec une marque préfixe
  (avant la marque de type) $-$N +N
- des décorations `top` et `bot`
- un label (pour la forêt)

```
tree s(
       top=np{}
       and bot=np{}
       and id=subject at - np(),
       vp(...)
     ).
```

Associer des décorations par défaut

```
:-tag_features(np,np{},np{}).
:-tag_features(vp,vp{ mode => M }, vp{ mode => M }).
```

```
tree s(
       id=subject at - np(),
       vp(...)
      ).
```

équivalent à

```
tree s(
       top=np{} and top=np{} and id=subject at - np(),
       top=vp{mode => M} and bot=vp{mode => M} at vp(...)
      ).
```

Exemples

```
tag_tree{ name=>n0v,                    % Verbe intransitif
          family=>n0v,
          tree => tree s(id=subject and top=np{num=>N} at np,
                         top=vp{num=>N} and bot=vp{mode=>M2,num=>
    N2}
                        at vp(
                              bot=v{mode=>M2,num=>N2} at <>v
                             )
                        )
        }.
tag_tree{ name=>n0vn1,                  % Verbe transitif
          family=>n0vn1,
          tree => tree s(id=subject and top=np{num=>N} at np,
                         top=vp{num=>N} and bot=vp{mode=>M2,num=>
    N2}
                        at vp(bot=v{mode=>M2,num=>N2} at <>v,
                              id=object at np))
        }.
```

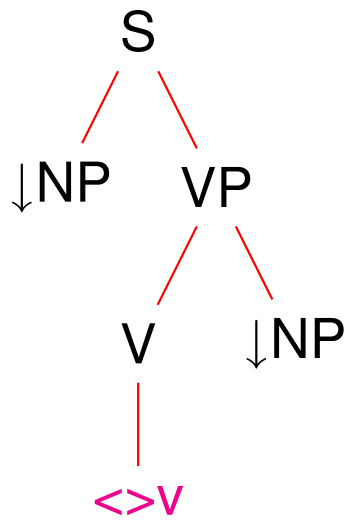
Exemples

```
tag_tree{ name => na,                    % Nom Adjectif
          family => adj,
          tree => auxtree bot=l at n(bot=l::n{num=>N,gen=>G} at *
    n,
                                          top=adj{num=>N,gen=>G} at <>
    adj)
        }.
```

Ancrage et familles

Idées (Architecture XTAG)

- Ancrer des schémas d'arbres récurrents par une catégorie et éventuellement d'autres information
- Regrouper sous un nom de famille un ensemble d'arbres associables à une même entrée lexicale




```
tag_tree{
   name => n0vn1_canonical,
   family => n0vn1,
   tree => tree s(id=subject at np,
                  vp(<>v,
                     id=object at np)
                 )
}.
tag_tree{
   name => n0vn1_reln1,
   family => n0vn1,
   tree => auxtree np(*np,
                      s(id=object at relpron,
                        s(id=subject at np,
                          vp(<>v))))
}
```

Ancrage

Possibilité de gérer finement le processus d'ancrage entre arbres et mots de la chaîne d'entrée

Par défaut, utilisation de la bibliothèque `tag_generic.pl`

```
anchor(tag_anchor{ name => Family,
                   coanchors => VarCoanchors,
                   equations => VarEquations
                 },
       Token,Label,Left,Right,Top
      ) :-
        phrase([Token],Left,Right),
        tag_lexicon(Token,Lemma,Label,Top),
        normalized_tag_lemma(Lemma,Label,Family,VarCoanchors,
    VarEquations)
        .

tag_lexicon(regarde, '*REGARDER*', v, v{ mode => mode[ind, subj],
     num => sing }).
tag_lemma('*REGARDER*',v,
          tag_anchor{ name=>'n0vn1',
                      equations=>[top = n{ restr=>plushum } at
```

Tree factorization

Idea : putting more in a single tree, because the trees share many common subparts

- defining more than one traversal path per tree (Harbush)
- using regular operators on trees :

  disjunctions $T[t_1;t_2] \equiv T[t_1] \cup T[t_2]$

  repetitions (Kleene Stars) $t@* \equiv \text{kleene}_t(\epsilon) \cup \text{kleene}_t(t, \text{kleene}_t)$

  interleaving (free ordering between node sequences)
  $(t_1, t_2)\#\#t_3 \equiv (t_1, t_2, t_3; t_1, t_3, t_2; t_3, t_1, t_2)$

  optionality (optional node) $t? \equiv (t; \epsilon)$

  guards (guarded nodes) $T[G_+, t; G_-] \equiv T[t].\sigma_+ \cup T[\epsilon].\sigma_-$
  guards : boolean formula over equation between feature structure paths

  These operators
  - do not modify expression power or complexity
  - may be removed by expansion
    bur resulting trees exponential wrt number of operators
  - more efficient to evaluate them without expansion
    $\Rightarrow$ more natural analysis
  - very generic operators (not specific to TAGs, TIGs, or DCGs)

Familles & Hypertag

On peut généraliser la notion de famille en utilisant de l'information partielle plutôt que des noms de familles explicites

⇒ Hypertag

```
tag_tree{
    name => 'v_active_canonique',
    family => ht{ cat => v,
                  diathesis => active,
                  arg0 => arg{ kind => subject },
                  arg1 => arg{ kind => object  }
                },
   tree => tree s(id=subject at np,
                  vp( <> v,
                      id=object at np
                    ))
}.
```

Assure :

- plus de flexibilité pour l'ancrage
- évite des problèmes de nommage des familles avec des noms complexes

Coupling lexicon & grammar : *hypertag*

Grammaire
hypertag #111

arg0	*arg0*	extracted	-
		kind	**subj**
		pcas	-
		real *real0*	- | **CS** | **N2** | **PP** | **S** | **cln** | **prel** | **pri**
arg1	*arg1*	extracted	-
		kind *kind1*	- | **acomp** | **obj** | **prepacomp** | **prepobj**
		pcas *pcas1*	**+** | - | **apres** | **à** | **avec** | **de** | **par** | ...
		real *real1*	- | **CS** | **N** | **N2** | **PP** | **S** | **V** | **adj** | **cla** | ...
arg2	*arg2*	extracted	-
		kind *kind2*	- | **prepacomp** | **prepobj** | **prepscomp** | **prepvcomp** | **scomp** | **vcomp** | **wh-comp**
		pcas *pcas2*	- | **+** | **apres** | **à** | ...
		real *real2*	- | **CS** | **N** | **N2** | **PP** | **S** | ...
cat	**v**		
diathesis	**active**		
refl	*refl*		

Coupling lexicon & grammar : *hypertag*

Grammaire
hypertag #111

arg0	*arg0*	extracted	-
		kind	**subj**
		pcas	-
		real	*real0* **-** | **CS** | **N2** | **PP** | **S** | **cln** | **prel** | **pri**
arg1	*arg1*	extracted	-
		kind	*kind1* **-** | **acomp** | **obj** | **prepacomp** | **prepobj**
		pcas	*pcas1* **+** | **-** | **apres** | **à** | **avec** | **de** | **par** | ...
		real	*real1* **-** | **CS** | **N** | **N2** | **PP** | **S** | **V** | **adj** | **cla** | ...
arg2	*arg2*	extracted	-
		kind	*kind2* **-** | **prepacomp** | **prepobj** | **prepscomp** | **prepvcomp** | **scomp** | **vcomp** | **wh-comp**
		pcas	*pcas2* **-** | **+** | **apres** | **à** | ...
		real	*real2* **-** | **CS** | **N** | **N2** | **PP** | **S** | ...
cat		**v**	
diathesis		**active**	
refl	*refl*		

Lexique
hypertag «promettre»

arg0	kind	**subj** | **-**
	pcas	**-**
arg1	kind	**obj** | **scomp** | **-**
	pcas	**-**
arg2	kind	**prepobj** | **-**
	pcas	**à** | **-**
refl	**-**	

Coupling lexicon & grammar : *hypertag*

Grammaire
hypertag #111

arg0 (*arg0*)
- extracted : -
- kind : **subj**
- pcas : -
- real (*real0*) : - | **CS** | **N2** | **PP** | **S** | **cln** | **prel** | **pri**

arg1 (*arg1*)
- extracted : -
- kind (*kind1*) : - | acomp | **obj** | prepacomp | prepobj
- pcas (*pcas1*) : + | - | apres | à | avec | de | par | ...
- real (*real1*) : - | **CS** | **N** | **N2** | **PP** | **S** | **V** | **adj** | **cla** | ...

arg2 (*arg2*)
- extracted : -
- kind (*kind2*) : - | prepacomp | **prepobj** | prepscomp | prepvcomp | scomp | vcomp | whcomp
- pcas (*pcas2*) : - | + | apres | **à** | ...
- real (*real2*) : - | **CS** | **N** | **N2** | **PP** | **S** | ...

cat : **v**

diathesis : **active**

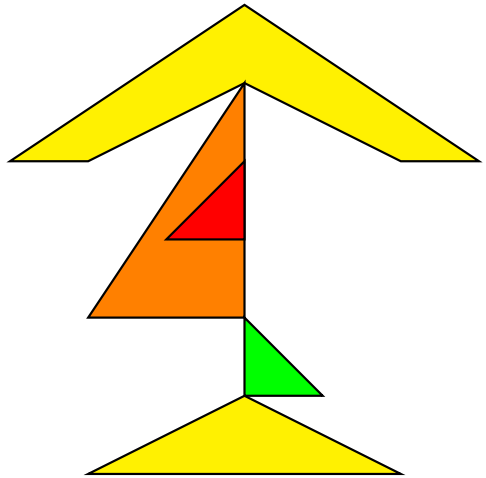
refl (*refl*) : -

Lexique
hypertag «promettre»

- arg0
  - kind : **subj** | -
  - pcas : -
- arg1
  - kind : **obj** | **scomp** | -
  - pcas : -
- arg2
  - kind : **prepobj** | -
  - pcas : **à** | -
- refl : -

Hybrid TIG/TAG parsing

TAGs are often too powerful $\Rightarrow$
TIG are a TAG variant (Schabes) where one adjoining step can only insert material on left or right side of the adjoining node.



- Tree Insertion Grammars [TIG] have equivalent to CFGs (with $O(n^3)$ time complexity)
- Real life TAGs are mostly TIG and possible to automatically detect TIG and TAG parts of a grammar
  $\Rightarrow$ pay higher complexity only for wrapping adjoining
- May switch to multiple adjoining on nodes
  getting more natural derivation forests

Analyse de grammaire

Une grammaire TAG peut être examinée pour détecter quels arbres sont TIG.

```
:-std_prolog right_tig/1.

right_tig(Root::tag_node{label=>NT}) :-
        potential_right_tig(Root),
        \+ not_right_tig(Root)
        .

:-light_tabular not_right_tig/1.
not_right_tig(Root) :-
        spine_node(Root,N::tag_node{label=>NT}),
        node_auxtree(NT,AuxTree),
        ( \+ potential_right_tig(AuxTree)
          xor not_right_tig(AuxTree)
        )
        .
```

Analyse de grammaire (suite)

```
:-std_prolog right_potential_tig/1.
right_potential_tig(Node) :-
        ( Node=tag_node{kind=>foot} ->
            true
        ;   Node=tag_node{spine=>yes,children=>[Child|_]},
            right_potential_tig(Child)
        )
        .

:-std_prolog spine_node/2.
spine_node(N,M) :-
  %% Every adjoinable spine node but the root
    ( N=tag_node{children=>Children},
      domain(K::tag_node{spine=>yes},Children),
      (   M=K
      ;
          spine_node(K,M)
      )
    ).
```

Analyse de grammaire (suite)

La détection des arbres TIG est disponible dans DyALog

```
> dyacc -analyze tag2tig frmg.tag -o tig_header.tag
```

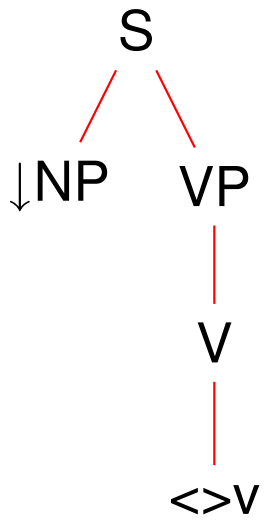
```
%% :-tig(TreeName,Kind).
:-tig('10_det_coord_shallow_auxiliary',right).
:-tig('102_adv_mod_on_coo_modifier_after_x_shallow_auxiliary',
    right).
:-tig('103_modifier_after_x_shallow_auxiliary_xpro_on_noun',right
    ).
...

%% :-adjkind(NonTerminal,Kind)
:-adjkind(np{},left).
:-adjkind(np{},right).
:-adjkind(v{},left).
:-adjkind(v{},right).
:-adjkind(adjP{},right).
:-adjkind(adjP{},wrap).
```

Quatorzième partie XIV

Grammaires et Contraintes

Arbres et contraintes

On peut voir un arbre TAG comme un ensemble de contraintes entre nœud de l'arbre d'analyse :



$$
\begin{aligned}
&S \text{ father\_of } NP\\
&S \text{ father\_of } VP\\
&VP \text{ father\_of } V\\
&V \text{ father\_of } v\\
&NP \preceq VP\\
&\ldots
\end{aligned}
$$

On peut se donner plus de liberté en spécifiant moins de contraintes.

Grammaires de Propriétés

Introduites par Blache

Description de constituants en terme de Contraintes

- Linéarité $N \preceq M$
- Requirement $N \Rightarrow M$
- Exclusion $N \not\mapsto M$
- Obligation $\triangle N$
- Unicité $N!$
- Dépendance $N \rightsquigarrow M$

Exemple1

NP (*Noun Phrase*)		
Features	Property Type : Properties	
[AVM]	obligation : $Obl(\mathrm{N} \vee \textsc{Pro})$	(P3.6)
	uniqueness : D!	(P3.7)
	: N!	(P3.8)
	: PP!	(P3.9)
	: Pro!	(P3.10)
	linearity : $\mathrm{D} \prec \mathrm{N}$	(P3.11)
	: $\mathrm{D} \prec \textsc{Pro}$	(P3.12)
	: $\mathrm{D} \prec \mathrm{AP}$	(P3.13)
	: $\mathrm{N} \prec \mathrm{PP}$	(P3.14)
	requirement : $\mathrm{N} \Rightarrow \mathrm{D}$	(P3.15)
	: $\mathrm{AP} \Rightarrow \mathrm{N}$	(P3.16)
	exclusion : $\mathrm{N} \Leftrightarrow \textsc{Pro}$	(P3.17)
	dependency : $\mathrm{N}\left[\begin{smallmatrix}\textsc{gend}\ \boxed{1}\\ \textsc{num}\ \boxed{2}\end{smallmatrix}\right] \rightsquigarrow \mathrm{D}\left[\begin{smallmatrix}\textsc{gend}\ \boxed{1}\\ \textsc{num}\ \boxed{2}\end{smallmatrix}\right]$	(P3.18)

Exemple2

VP (*Verb Phrase*)

Features	Property Type : Properties	
[AVM]	*obligation* : ΔV	(P3.19)
	uniqueness : $V_{[main\ past\ part.]}!$	(P3.20)
	: NP!	(P3.21)
	: PP!	(P3.22)
	linearity : $V \prec NP$	(P3.23)
	: $V \prec$ ADV	(P3.24)
	: $V \prec PP$	(P3.25)
	requirement : $V_{[past\ part.]} \Rightarrow V_{[aux.]}$	(P3.26)
	exclusion : $PRO_{[acc]} \Leftrightarrow NP$	(P3.27)
	: $PRO_{[dat]} \Leftrightarrow PRO_{[acc]}$	(P3.28)
	dependency : $V\begin{bmatrix}\text{PERS} & \boxed{1} \\ \text{NUM} & \boxed{2}\end{bmatrix} \rightsquigarrow \text{PRO}\begin{bmatrix}\text{TYPE} & pers \\ \text{CASE} & nom \\ \text{PERS} & \boxed{1} \\ \text{NUM} & \boxed{2}\end{bmatrix}$	(P3.29)

Évaluation avec CHR

- Adaptation des méthodes de programmation logique par contraintes
- Proposition d'utilisation des Constraint Handling Rules [CHR] Thom Frühwirth

CHR $\equiv$ Règles pour définir des contraintes par ajout, transformation, ou retrait du store

- propagation $H \Rightarrow Guard | B$
  ajout de B comme conséquence si $Guard$ et présence de H

```
transitivity @ X=<Y,Y=<Z ==> X=<Z.
```

- simplification $H \Longleftrightarrow Guard | B$
  consommation de H et ajout de B si $Guard$

```
antisymmetry @ X=<Y,Y=<X <=> X=Y.
reflexivity @ X=<Y <=> X=Y | true.
```

Consommation

```
reservation(Renter,Group,From,To),
available(car(Id,Group,...),From) <=>
          ...   |
          rentagreement(Renter,Id,From,To).
```

Après location, la voiture n'est plus disponible et la réservation disparaît.

Contrainte d'ordre lexical

```
[] lex [] <=> true.
[X|L1] lex [Y|L2] <=> X<Y | true.
[X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
[X|L1] lex [Y|L2] ==> X=<Y.
[X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
[X,U|L1] lex [Y,V|L2] <=> U>=V, L1=[_|_] |
                    [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

Exemple de Thom Frühwirth

```
terminal@ A->T, word(T+R) => parses(A,T+R,R).
non-term@ A->B*C, parses(B,I,J), parses(C,J,K) => parses(A,I,K)
substrng@ word(T+R) => word(R).
```

Représentation de la chaîne d'entrée par la contrainte initiale

```
word('Jean'+'aime'+'Marie').
```

Implantation

Par concurrence, avec `ask` et `tell`

$$H \Rightarrow Guard | B$$

- `ask` sur H et éventuellement $Guard$
- `tell` sur B
- + retrait du store sur H

CHR permet la construction d'algorithmes efficaces
(en spécifiant le bon jeu de règles !)

Utilisation de CHR pour les GP Blache & Dahl

```
%% Chargement des terminaux
[un] ::> det(sg).
[garçon] ::> nom(sg).
[rit] ::> v(sg).

%% Une règle CHR
det(NDet),nom(Nm),v(Nv) =>
        acceptable(accord,NDet,Nn,Nv,N,_) | phrase(N).

%% Le test d'une contrainte des PG
prop(accord,[NDet,Nn,Nv,N]) :- NdDet=Nn,NDet=Nv,!,N=Nv.
%% avec relâchement possible
prop(accord,[NDet,Nn,Nv,N],mismatch).
relax(accord).
```