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

Quinzième partie XV

Méta-Grammaires

Principe

Introduite par Candito

- Description de grammaires comme des ensembles (classes) de contraintes à satisfaire
- Ajout de la notion d'héritage et de croisement de classe pour factoriser l'information
  $\Rightarrow$ modularité
- Satisfaction partielle des contraintes hors-ligne pour construire les structures d'une grammaire TAG (ou LFG, ...)
  $\Rightarrow$ éviter les risques d'explosion combinatoire liés aux contraintes

Les Méta-Grammaires par l'exemple

Definition (Méta-Grammaire)

Description modulaire par classes regroupant des contraintes, avec héritage

Les Méta-Grammaires par l'exemple

Definition (Méta-Grammaire)

Description modulaire par classes regroupant des contraintes, avec héritage

```
class collect_real_subject_canonical {
 <: collect_real_subject;
 $arg.extracted = value(~cleft);
 S >> VSubj; V >> psubj;
 VSubj < V; VMod < psubj;
 node psubj: [cat:N2, id:subject,
              top:[wh:-,sat:+]];
 - psubj::agreement; psubj = psubj::N;
 psubj =>
  node(Infl).bot.inv = value(+),
  $arg.extracted = value(-),
  $arg.real = value(N2),
  desc.extraction = value(~-),
  node(V).top.mode= value(~inf|imp|...);
 ~psubj=> node(Infl).bot.inv = value(~+);
}
```

- Héritage (<:)
- Contraintes
  - dominance (>> et >>+)
  - précédence (<)
  - égalité (=)
  - Décorations (FS)
    - ★ noeuds
    - ★ classe
  - Éq. entre chemins (.)
    - ★ noeuds (node psubj)
    - ★ classe (desc)
    - ★ variable ($arg)
- Ressources + / Besoins -
  - Espace de noms (::)
- Gardes (=>)

Plusieurs formats pour les MG : DyALog

Format DyALog (à la Prolog) : comme entrée pour MGCOMP

```
class('adj_on_noun').
super('adj_on_noun', 'adj_as_modifier').
equation('adj_on_noun', desc:ht:arg0:real, value('N2')).
node('adj_on_noun','Root').
nodefeature('adj_on_noun', 'Root', [cat: 'N']).
```

Example de garde :

```
guard('verb_extraction_relative',
      'prel_object',+,[
       and([node('XGroup') : extracted : real = value(prel),
            node('XGroup') : extracted : kind = value(obj),
            node('XGroup') : extracted : pcas = value(-)])
       ]).
```

Contraintes sur les noeuds

- Dominance immédiate : Father >> Son
- Dominance strict : Ancestor >>+ Descendant
- Précédence : Left < Right
- Aliasing : Alias1 = Alias2

Note : pas de dominance reflexive ni de précédence immédiate

Contraintes sur les structures de traits

- valeur atomique, multiple et négation : value(~ncadj|np)|
- Affectation sur un noeud : node N: [cat: nc, top: $top ^ [gender:masc]]
- Affectation sur la classe : desc([ht: [cat: nc]])
- Équations entre chemins :
  - ancré sur un noeud node(N).top.gender=value(masc)
  - ancré sur la classe desc.ht.cat=node(N).cat
  - ancré sur une variable $top.number = value(sing)
- Macros sur les valeurs et chemins

```
%% Macro for value
template @emptyarg_fs = [pcas: -,kind:-,real:-,extracted:-]
%% Macro for pathes
path @arg0 = .ht.arg0
node(N).@real0 = value(@emptyarg_fs);
```

Note : Pas de vérifications de cohérences sur les structures (valeurs admissibles pour un trait).
mais possibilté d'écrire : node(N).top.cat = value(@cat^nc|np)

Structure des noeuds

Attributs standards sur un noeud :

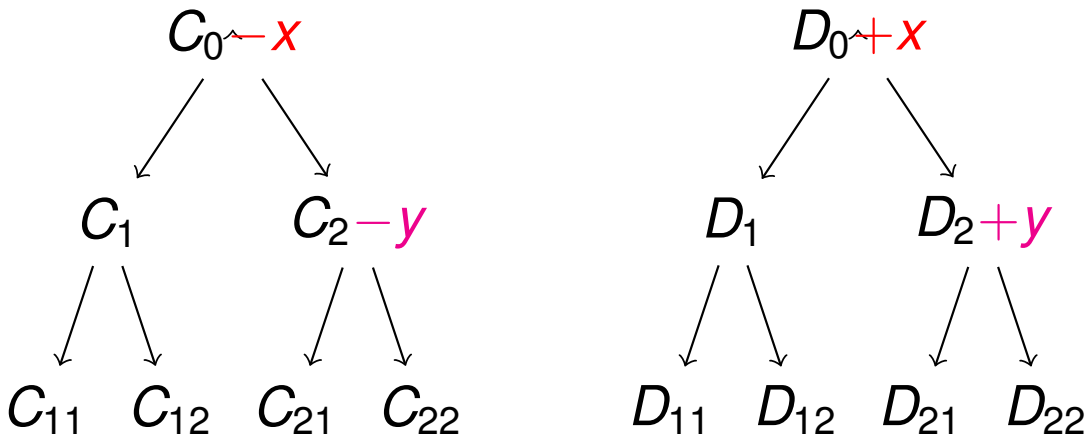
- type : dans std, subst, foot, anchor, coanchor, lex, sequence, alternative
- id : identification du noeud
- cat : catégorie syntaxique
- lex : forme (pour les noeuds de type lex)
- top et bot : arguments TAG
- adj : dans yes, no, strict
- optional : optionalité + -
- star : répétition * -

Noeud alternative :

```
node XGroup : [type: alternative];
XGroup >> pri_subj;
XGroup >> pri_obj;
pri_subj < pri_obj;
```

Accumulation de contraintes

- par héritage (multiple) $<$: super_class
- par croisement de classes sur ressource $+$ r et $-$ r



Gardes

- Possibilité de poser des gardes ou conditions sur la présence ou absence d'un noeud.

```
Subj => node(v).top.mode = value(~imperative);
~ Subj => node(v).top.mode = value(imperative);
```

- Les gardes sont accumulées par conjonction :
  N => G1 et N => G2 équivalent à N => G1,G2

- Les conditions sont des disjonctions de conjunctions de chemins

```
~ Subj =>
      node(v).top.mode = value(imperative)
    | node(S).top.inv = value(+),
      node(v).top.mode = value(imperative)
    ;
```

Outline

Compiler la méta-grammaire

Compilateur MGCOMP, développé avec DYALOG

Compiler la méta-grammaire

Compilateur MGCOMP, développé avec DYALOG

Étape 1 : Classes terminales

Héritage des contraintes par les classes terminales (+ vérif contraintes)

Compiler la méta-grammaire

Compilateur MGCOMP, développé avec DYALOG

Étape 1 : Classes terminales

Héritage des contraintes par les classes terminales (+ vérif contraintes)

Étape 2 : Classes neutres

- Croisement des classes terminales pour neutraliser ressources & besoins
  - $C_1[-R \cup \mathcal{K}_1] x C_2[+R \cup \mathcal{K}_2] = (C_1 x C_2)[=R \cup \mathcal{K}_1 \cup \mathcal{K}_2]$
  - (Espace de nom) $\Rightarrow$ import classe productrice avec renommage
    $C_1[-N::R \cup \mathcal{K}_1] x C_2[+R \cup \mathcal{K}_2] = (C_1 x N::C_2)[=N::R \cup \mathcal{K}_1 \cup N::\mathcal{K}_2]$
- Réduction des gardes (quand possible)
- Vérification des contraintes

Compiler la méta-grammaire

Compilateur MGCOMP, développé avec DYALOG

Étape 1 : Classes terminales

Héritage des contraintes par les classes terminales (+ vérif contraintes)

Étape 2 : Classes neutres

- Croisement des classes terminales pour neutraliser ressources & besoins
  - $C_1[-R \cup \mathcal{K}_1] x C_2[+R \cup \mathcal{K}_2] = (C_1 x C_2)[=R \cup \mathcal{K}_1 \cup \mathcal{K}_2]$
  - (Espace de nom) $\Rightarrow$ import classe productrice avec renommage
    $C_1[-N::R \cup \mathcal{K}_1] x C_2[+R \cup \mathcal{K}_2] = (C_1 x N::C_2)[=N::R \cup \mathcal{K}_1 \cup N::\mathcal{K}_2]$
- Réduction des gardes (quand possible)
- Vérification des contraintes

Étape 3 : Arbres TAG/TIG

Utilisation des contraintes des classes neutres pour construire les arbres

Croisements

Croisements guidés par les ressources, ordonnées par *profondeur croissante*

- r_1 dépend de r_2 si il existe $C[+r_1, -r_2]$
- $p(r) = 0$ si r est non dépendante
  $p(r) = 1 + \max p(r_i)$ si r dépend des r_i
- erreur si r dépend de elle-même (directement ou indirectement)

Croisements

Croisements guidés par les ressources, ordonnées par *profondeur croissante*

- r_1 dépend de r_2 si il existe $C[+r_1, -r_2]$
- $p(r) = 0$ si r est non dépendante
  $p(r) = 1 + \max p(r_i)$ si r dépend des r_i
- erreur si r dépend de elle-même (directement ou indirectement)

Accumulation des contraintes :

$$C^-[-N::R \cup \mathcal{K}^-] \oplus C^+[+R \cup \mathcal{K}^+] = (C^- \oplus N::C^+)[=N::R \cup \mathcal{K}^- \cup N::\mathcal{K}^+]$$

Croisements

Croisements guidés par les ressources, ordonnées par *profondeur croissante*

- r_1 dépend de r_2 si il existe $C[+r_1, -r_2]$
- $p(r) = 0$ si r est non dépendante
  $p(r) = 1 + \max p(r_i)$ si r dépend des r_i
- erreur si r dépend de elle-même (directement ou indirectement)

Accumulation des contraintes :

$$C^-[-N::R \cup \mathcal{K}^-] \oplus C^+[+R \cup \mathcal{K}^+] = (C^- \oplus N::C^+)[=N::R \cup \mathcal{K}^- \cup N::\mathcal{K}^+]$$

Croisement impossible si une même ressource (avec espace de noms et éventuellement neutralisée) présente dans les deux classes croisées :

$$\exists r : \mathrm{res}, \exists k_1, k_2 \in \{+, -, =\},$$
$$k_1 r \in C_1 \wedge k_2 r \in C_2 \wedge (k_1 = k_2 \vee k_1 = (=) \vee k_2 = (=))$$

Vérification des contraintes

Problème des méta-grammaires :

1. interpréter ce qui n'est pas dit : possible ou interdit ?
2. interpréter par rapport à une axiomatique :
   arbre (TAG), forêt (MC-TAG), graphe (shared MC-TAG)

- Liens entre les noeuds A et B dans une classe non neutre
  $\Rightarrow$ noeuds potentiellement aliasables

$$A = B \vee A < B \vee B < A \vee A \triangleright^+ B \vee B \triangleright^+ A$$

- Liens entre les noeuds A et B dans une classe neutre
  $\Rightarrow$ noeuds distincts

$$A < B \vee B < A \vee A \triangleright^+ B \vee B \triangleright^+ A$$

Vérification des contraintes

Problème des méta-grammaires :

1. interpréter ce qui n'est pas dit : possible ou interdit ?
2. interpréter par rapport à une axiomatique :
   arbre (TAG), forêt (MC-TAG), graphe (shared MC-TAG)

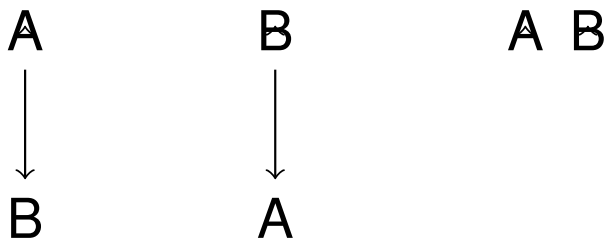
- Liens entre les noeuds A et B dans une classe non neutre
  $\Rightarrow$ noeuds potentiellement aliasables

$$A = B \vee A < B \vee B < A \vee A \triangleright^{+} B \vee B \triangleright^{+} A$$

- Liens entre les noeuds A et B dans une classe neutre
  $\Rightarrow$ noeuds distincts

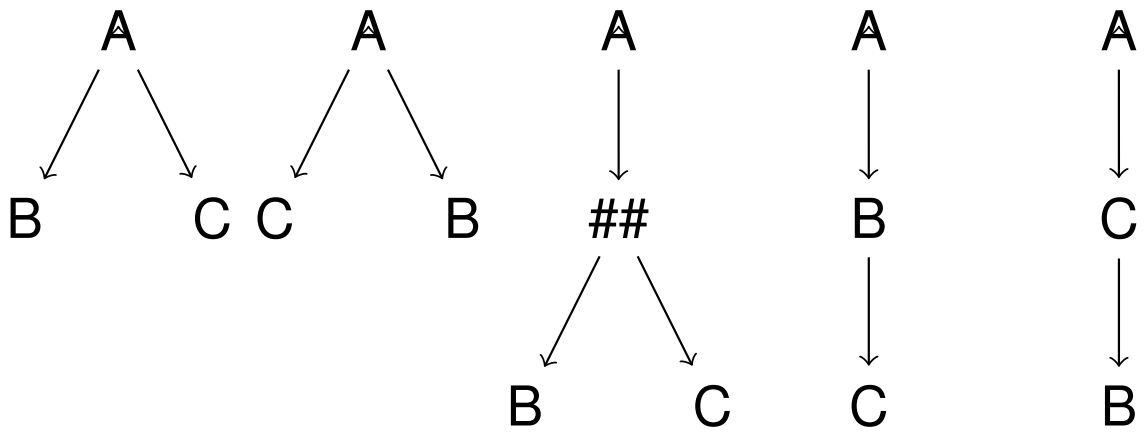
$$A < B \vee B < A \vee A \triangleright^{+} B \vee B \triangleright^{+} A$$

Modèle minimal pour 2 noeuds sans contraintes :



Contraintes et modèles minimaux

Soit A >>+ B et A >>+ C



Vérification des contraintes

Hiérachisation des types de contraintes :

1. Introduction des variables par analyse des équations
2. Aliasing entre noeuds
3. Gestion des noeuds anonymes
4. Dominance entre noeuds
5. Précédence entre noeuds (dépend de la dominance)
6. Décorations
7. Réduction des gardes

Fermetures transitives

La vérification des contraintes essentiellement effectuée par des calculs de fermeture transitive, exploitant les propriétés de tabulation (faible) de DyALog

```
:-light_tabular precedes_closure/3.
:-mode(precedes_closure/3,+(+,+,-)).

%% Precédence dans la classe
precedes_closure(Class,N1,N3) :-
      precedes(Class,N1,N2),
      (    N3=N2 ; precedes_closure(Class,N2,N3)).

%% Héritage des précédences des classes parentes
precedes_closure(Class,N1,N3) :-
      xsuper(Class,Super,NS),
      deep_module_unshift(N1,NS,_N1),
      %% Réutilisation de la fermeture sur Super
      '$answers'(precedes_closure(Super,_N1,_N2)),
      deep_module_shift(_N2,NS,N2),
      (    N3=N2 ; precedes_closure(Class,N2,N3) ).
```

Fermetures transitives

$$\mathcal{K} \cup (A = B) \vdash \mathcal{K}[A \mapsto B]$$

$$\mathcal{K} \vdash \mathcal{K} \cup (A \rhd^+ B) \qquad A \rhd B \in \mathcal{K}$$

$$\mathcal{K} \vdash \mathcal{K} \cup (A \rhd^+ C) \qquad \begin{cases} A \rhd^+ B \in \mathcal{K} \\ B \rhd^+ C \in \mathcal{K} \end{cases}$$

$$\mathcal{K} \vdash \mathcal{K} \cup (A < C) \qquad \begin{cases} A < B \in \mathcal{K} \\ B < C \in \mathcal{K} \end{cases}$$

$$\mathcal{K} \cup (A < B) \cup (B \rhd^+ C) \vdash \mathcal{K} \cup (A < B) \cup (B \rhd^+ C) \cup (A < C)$$

$$\mathcal{K} \cup (A < B) \cup (A \rhd^+ C) \vdash \mathcal{K} \cup (A < B) \cup (A \rhd^+ C) \cup (C < B)$$

$$\mathcal{K} \cup (A < A) \vdash \text{fail}$$

$$\mathcal{K} \cup (A \rhd^+ A) \vdash \text{fail}$$

$$\mathcal{K} \cup (A \mathcal{R}_1 B) \cup (A \mathcal{R}_2 B) \vdash \text{fail} \qquad \begin{cases} \mathcal{R}_1 \in \{<, >\} \\ \mathcal{R}_2 \in \{\rhd^+, \lhd^+\} \end{cases}$$

Exemple

```
precedes_closure(Class,N1,N3,K) :-
        precedes(Class,N1,N2),
        (    N3=N2, K=strict,
             true
        ; precedes_closure(Class,N2,N3,K),
             true
        ;'$answers'(dominates_closure(Class,N2,N3,_)), K= strict,
             true
        ; '$answers'(dominates_closure(Class,N3,N2,_)), K = dom,
             true
        ).
```

Réduction des gardes

- Éliminer les portions de gardes triviallement vraies ou fausses
- Appliquer les gardes (positives ou négatives) quand aucune alternative disponible
  - Un noeud gardé A devient nécessairement présent si sa garde négative G_A^- se réduit à fail.
    Application de G_A^+ si simple conjonction
  - un noeud gardé A devient nécessairement absent si sa garde positive G_A^+ se réduit à fail.
    Application de G_A^- si simple conjonction
- Répéter jusqu'à obtention d'un point fixe

Réduction des gardes

$$\frac{}{t = t \longrightarrow \text{true}}$$

$$\frac{}{t = r \longrightarrow x = r}\ \text{mgu}(t, r) \neq Id$$

$$\frac{}{t = r \longrightarrow \text{fail}}\ \neg\text{mgu}(t, r)$$

$$\frac{\alpha \longrightarrow \text{true} \quad \beta \longrightarrow \beta'}{\alpha \vee \beta \longrightarrow \text{true}}$$

$$\frac{\alpha \longrightarrow \text{fail} \quad \beta \longrightarrow \beta'}{\alpha \vee \beta \longrightarrow \beta'}$$

$$\frac{\alpha \longrightarrow \alpha' \quad \beta \longrightarrow \beta'}{\alpha \vee \beta \longrightarrow \alpha' \vee \beta'}\ \{\alpha', \beta'\} \cap \{\text{true}, \text{fail}\} = \emptyset$$

$$\frac{\alpha \longrightarrow \text{true} \quad \beta \longrightarrow \beta'}{\alpha \wedge \beta \longrightarrow \beta'}$$

$$\frac{\alpha \longrightarrow \text{fail} \quad \beta \longrightarrow \beta'}{\alpha \vee \beta \longrightarrow \text{fail}}$$

$$\frac{\alpha \longrightarrow \alpha' \quad \beta \longrightarrow \beta'}{\alpha \vee \beta \longrightarrow \alpha' \vee \beta'}\ \{\alpha', \beta'\} \cap \{\text{true}, \text{fail}\} = \emptyset$$

Emission des arbres minimaux

Utilisation de l'opérateur d'entrelacement en s'appuyant :

- notion de père potentiel :
  A pp B si B dominé strictement par un ancêtre de A et B sans père
- notion de précédence conditionnelle : $A < B$ quand A et B frères

Principe de l'algorithme :
Étant donné un noeud A et une liste L de fils potentiels de A non encore positionnés par ses ancètres :

- partitonner $L \cup \text{Descendants}(A)$ en $[C_1, ..., C_n]$
- chaque C_i de la forme $[B_{i0}, B_{i1}, \ldots, B_{in_i}]$ avec B_{ij} fils potentiels de B_{i0} $(j > 1)$
- $C_i < C_j \Rightarrow i < j$
- transformation de la partition en une formule avec les opérateurs séquence , et entrelacement $\#\#$ sur $\text{tree}(C_i)$.

$$C_i = \langle A_i, \mathcal{A}_i \rangle \Rightarrow \forall B \in \mathcal{A}_i,\ \neg(A_i < B) \wedge \neg(B < A_i) \wedge \neg(B \triangleright^+ A_i)$$

$$F_1 < F_2 \in F \Rightarrow \forall C_i = \langle A_i, \mathcal{A}_i \rangle \in F_1, \forall C_j = \langle A_j, \mathcal{A}_j \rangle \in F_2,\ \bigwedge \left\{ \begin{array}{l} A_i < A_j \\ \forall B_i \in C_i, \forall B_j \in C_j,\ \neg(B_j \prec \end{array} \right.$$

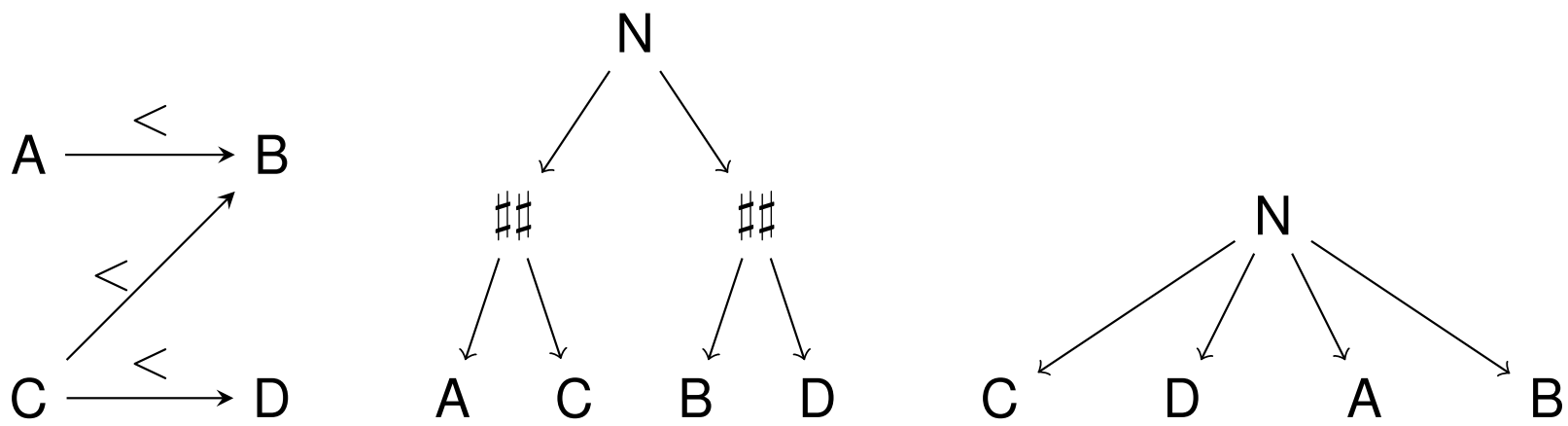
$$F_1 \sharp\sharp F_2 \in F \Rightarrow \forall C_i \in F_1, \forall C_j \in F_2,\ \forall B_i \in C_i, \forall B_j \in C_j,\ \bigwedge \{\ \neg(B_i \prec B_j) \neg(B_j \prec B_i)$$

Arbres factorisés

Quelques cas particuliers :

- Pas toujours possible de transformer un ordre partiel sur des fils en une seule formule séquence entrelacement.
  cas de : $A < B \wedge C < B \wedge C < D$ donne

$$A < D \Rightarrow (A \sharp\sharp C) < (B \sharp\sharp D)$$
$$D < A \Rightarrow C < D < A < B$$



Question : garder entrelacement ou utiliser directement l'ordre partiel

Arbres factorisés

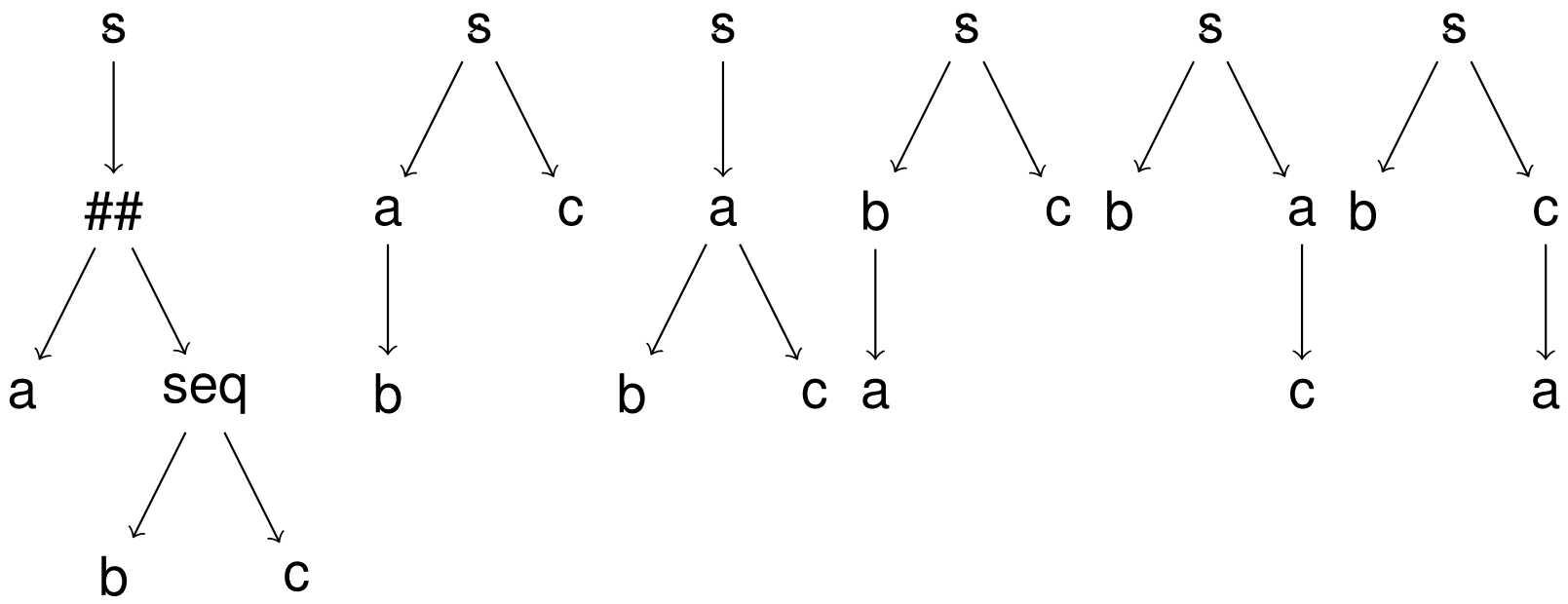
- Pas d'entrelacements au-dessus d'un pied,
  pour faciliter l'obtention d'arbre TIG.
- Prise en compte d'information de rang (premier ou dernier d'une fratrie)

Example

```
class bar {
  s >>+ a; s >>+ b; s >>+ c; b < c;
}
```

Example

```
class bar {
  s >>+ a; s >>+ b; s >>+ c; b < c;
}
```



Seizième partie XVI

Grammaires de Concaténation d'intervalles

RCG : short presentation

Range Concatenation Grammars (RCG) [Boullier] :
Constraints on intervals on the input string.
For language $a^n b^n c^n$

```
S(X @ Y @ Z) --> A(X,Y,Z).
A("a" @ X, "b" @ Y, "c" @ Z) --> a(X,Y,Z).
A("","","") --> .
```

aabbcc	$S(]0,6])$	$\longrightarrow$
aabbcc	$A(]0,2],]2,4],]4,6])$	$\longrightarrow$
aabbcc	$A(]1,2],]3,4],]5,6])$	$\longrightarrow$
aabbcc	$A(]2,2],]4,4],]6,6])$	$\longrightarrow$

RCG is an operational formalism used to encode linguistic formalisms such as TAG
where discontinuous constituents are used.

RCG allow modular grammar writing

concatenation `G(X @ Y) --> G1(X),G2(Y).`

union `G(X) --> G1(X) | G2(X).`

intersection `G(X) --> G1(X),G2(X).`

Prédicats RCG builtins

```
%% Bibliothèque de prédicats spéciaux
:-require 'rcg.pl'.

s (X@Y) --> s (X), { rcg_eqstrlen(X,Y) }.
s ("a") --> true.

?-recorded('N'(N)),rcg_phrase(s (0:N)).
```

- rcg_eqstr(R1,R2)
- rcg_length(R1,Length)
- rcg_eqstrlen(R1,R2)

RCG avec arguments

```
s(N) (X@Y@Z) --> a(N) (X,Y,Z).
a(M) ("a" @ X,"b" @ Y,"c" @ Z) --> a(N) (X,Y,Z), {M is N+1}.
a(0) ("","","") --> true.

axiom(s(N)).
```

RCG et intersection

Opération d'intersection immédiate : il suffit d'avoir plusieurs prédicats portant sur un même intervalle

```
s(X @Y @ Z) --> anbn(X @ Y), bncn(Y @ Z).
anbc(X @ Y) --> anbn2(X,Y).
anbn2("","") --> true.
anbn2("a" @ X,"b" @ Y) --> anbn2(X,Y).
```

Exemple d'utilisation pour des verbes à contrôle

Paul veut manger une pomme

```
control_verb(Subj @ V @ Sent) -->
    np(Subj),
    v(V),
    ctr_sentence(Subj,Sent).
```

Dix-septième partie XVII

Compléments sur DyALog

Lecture bidirectionnelle

```
s --> np,vp.
head(s,vp).   % ==> s --> np <+ vp
vp --> v(Type), v_args(Type).
head(vp,v).   % ==> vp --> v(Type) +> v_args(Type)
```

Lexicalisation

Lexicalization et DyALog

Non obligatoire dans les grammaires pour DyALog, mais permet une meilleure efficacité :
Principe : chargement conditionnel des clauses (ou arbres pour les TAGs) :

```
'$loader'( phrase([qui],_,_),
           ( np --> np, [qui], s_rel )
         ).
```

- Automatiquement induite pendant la compilation (option `-autoload`) pour TAG et TIG (noeuds lexicaux + ancres), DCG & BMG (lexicaux)
- Utiles pour tester des conditions plus spécifiques (pas nécessairement liées à la lexicalisation)

Robustesse

Robustesse et DyALog

- Facile de rendre des analyses partielles

```
% full parsing
?-recorded('N'(R)), L=0, phrase(S::s{},L,R).

% partial parsing
?-tag_phrase(s,L,R); phrase(np,L,R).
```

- possibilité d'utiliser des stratégies bidirectionnelles dirigées par les têtes(DCGs)
- utilisation des FSA en entrée : gestion des amabiguïtés, mots incnnus, ...
- prédicats d'introspection pour explorer le contenu de la table
  $\Rightarrow$ ouvre la voie vers des algorithmes de corrections

Modulation

dcg_mode(np/2,+(−,−),+,−)
Stratégie descendante : :−dcg_mode(_,+,+,−)
Stratégie ascendante : :−dcg_mode(_,−,−,−)

Définir son lecteur

```
:-skipper(skip_lexical).
:-std_prolog skip_lexical/3.
skip_lexical(Left,Token,Right) :-
    'C'(Left,
        lemma{ lex => Token, cat => epsilon },
        Right ).
```

Namespaces et modules

DyALog fournit

- la notion de namespaces pour éviter les conflits de noms : private !foo
- la notion demodules, fondée sur les namespaces et sur un mécanisme d'import

```
:-module(forest).              %% Forest module with namespace forest
:-import{ file=>'format.pl', preds => [format/2] }.
:-std_prolog display_forest/1.
....
:-end_require.                 %% public interface
```

```
:-import{ module=>forest, preds => display_forest/1 }. %% import
    forest module
```

Mais pas de vraie notion de prédicats privés (similaire à la notion de modules en Perl)

Modules

Directives `:-include`, `:-require`, `:-import` et `:-module/1` import et export.

```
:-import{ module=> sqlite,
          file => 'libdyalogsqlite.pl',
          preds => [
                     open/2,
                     close/1,
                     exec/4
                   ]
        }.

test :-
  argv([File]),
  open(File,DB),  %% équivalent à sqlite!open
  exec(DB,'create table toto (k TEXT,v INTEGER)',[],_R),
  exec(DB,'insert into toto(k,v) values (?,?)',[a,2],done),
  close(DB)
.
```

Interface vers C

```
socket_bind(Socket,Address) :-
  ( Address = 'UNIX'(Name) ->
      '$interface'( 'Socket_Bind_Unix'(Socket: int, Name: string)
    , [] )
  ;   Address = 'INET'(Host,Port),
      ( var(Port) -> In_Port = 0 ; In_Port = Port ),
        '$interface'( 'Socket_Bind_Inet'(Socket: int,
                                         Host: string,
                                         In_Port: int,
                                         Out_port: -int),
                      [] )
  ).
```