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: Grammaires DCG : présentation et utilisation

Troisième partie III

DCG : Présentation

Outline

Grammaires Hors-Contextes

Grammaire Hors-Contexte [CFG – *Context-Free Grammar*], formalisme de base en analyse syntaxique.

Une CFG est définie par

- un ensemble fini $\mathcal{T}$ de terminaux
- un ensemble fini $\mathcal{N}$ de non-terminaux
- un non-terminal axiome S
- un ensemble fini de productions $A_0 \leftarrow \alpha$, $\alpha \in (\mathcal{T} \cup \mathcal{N})^\star$

Grammaires Hors-Contextes

Grammaire Hors-Contexte [CFG – *Context-Free Grammar*], formalisme de base en analyse syntaxique.

Une CFG est définie par

- un ensemble fini $\mathcal{T}$ de terminaux
- un ensemble fini $\mathcal{N}$ de non-terminaux
- un non-terminal axiome S
- un ensemble fini de productions $A_0 \leftarrow \alpha$, $\alpha \in (\mathcal{T} \cup \mathcal{N})^\star$

Exemple : $(\{a, b\}, \{S\}, S, \mathcal{P})$ avec les productions $S \leftarrow \epsilon$ et $S \leftarrow a, S, b$.

- reconnaît le langage $a^n b^n$
- ce langage ne peut être reconnu par une expression régulière

Grammaires Hors-Contextes

Grammaire Hors-Contexte [CFG – *Context-Free Grammar*], formalisme de base en analyse syntaxique.

Une CFG est définie par

- un ensemble fini $\mathcal{T}$ de terminaux
- un ensemble fini $\mathcal{N}$ de non-terminaux
- un non-terminal axiome S
- un ensemble fini de productions $A_0 \leftarrow \alpha$, $\alpha \in (\mathcal{T} \cup \mathcal{N})^\star$

Exemple : $(\{a, b\}, \{S\}, S, \mathcal{P})$ avec les productions $S \leftarrow \epsilon$ et $S \leftarrow a, S, b$.

- reconnaît le langage $a^n b^n$
- ce langage ne peut être reconnu par une expression régulière

Par contre, $a^n b^n c^n$ ou $a^n b^m c^n d^m$ ne sont pas reconnaissables avec une CFG.

Lecture logique des CFG

Lecture immédiate d'une production CFG $A_0 \leftarrow A_1 \ldots A_n$ en terme de logique :

$$\frac{A_1 \ldots A_n}{A_0}$$

ou encore A_0 dérivable ssi $A_0 \ldots A_n$ dérivables

Lecture logique des CFG

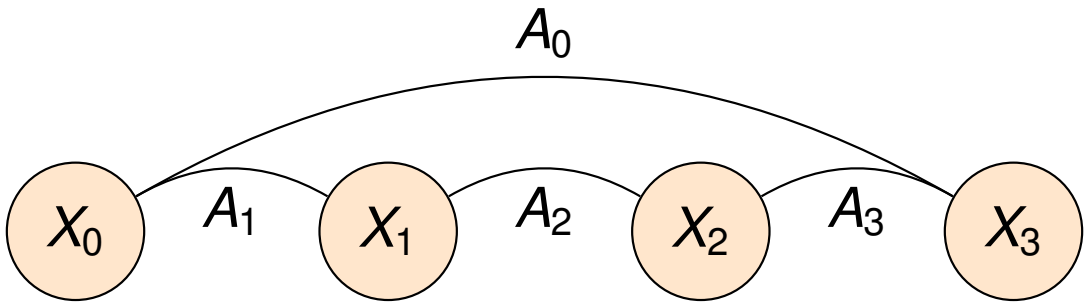
Lecture immédiate d'une production CFG $A_0 \leftarrow A_1 \ldots A_n$ en terme de logique :

$$\frac{A_1 \ldots A_n}{A_0}$$

ou encore A_0 dérivable ssi $A_0 \ldots A_n$ dérivables

Mais lecture logique vraie modulo des paires de positions

$$\frac{A_1(X_0, X_1) \ldots A_n(X_{n-1}, X_n)}{A_0(X_0, X_n)}$$



Digression : Plus proche de la logique linéaire (logique de ressources)

Grammaires de Clauses Définies

Lecture logique des CFG à la base des Grammaire de Clauses Définies [DCG - *Definite Clause Grammars*], Pereira & Warren – 1980

```
s --> [].
s --> [a],S,[b].
?-phrase(s,[a,a,a,b,b,b],[]).
```

Les DCGs sont disponibles dans la plupart des systèmes Prolog (dont DyALog).

Moteur DCG

Comme pour Prolog et les regexp ou FSA (TD1),
facile d'implanter un moteur DCG en Prolog :

```
dcg_solve(NT,L,R) :- dcg_clause((NT --> Body),L,R).
dcg_solve((Body1,Body2),L,R) :-
      dcg_solve(Body1,L,M),
      dcg_solve(Body2,M,R).
dcg_solve([],L,L).
dcg_solve([Token|TokenList],L,R) :-
      'C'(L,Token,M),
       dcg_solve(TokenList,M,R).
phrase(NT,L,R) :- dcg_solve(NT,L,R).
```

Expansion de termes

En pratique, immédiat de faire une expansion des clauses DCG en clause Prolog, en ajoutant des paires de positions :

A −−> B,C,D. $\rightsquigarrow$ A(X0,X3) :− B(X0,X1),C(X1,X2),D(X2,X3).

Expansion de termes

En pratique, immédiat de faire une expansion des clauses DCG en clause Prolog, en ajoutant des paires de positions :
A −−> B,C,D. ⇝ A(X0,X3) :− B(X0,X1),C(X1,X2),D(X2,X3).

```
s --> [].
s --> [a],S,[b].
?-phrase(s,[a,a,a,b,b,b],[]).
```

donne (lecteur abstrait 'C'/3)

```
s(X0,X0).
s(X0,X3) :- 'C'(X0,a,X1),s(X1,X2),'C'(X2,b,X3).
?-s([a,a,a,b,b,b],[]).
```

Expansion de termes

En pratique, immédiat de faire une expansion des clauses DCG en clause Prolog, en ajoutant des paires de positions :
A −−> B,C,D. ⇝ A(X0,X3) :− B(X0,X1),C(X1,X2),D(X2,X3).

```
s --> [].
s --> [a],S,[b].
?-phrase(s,[a,a,a,b,b,b],[]).
```

donne (lecteur abstrait 'C'/3)

```
s(X0,X0).
s(X0,X3) :- 'C'(X0,a,X1),s(X1,X2),'C'(X2,b,X3).
?-s([a,a,a,b,b,b],[]).
```

ou encore (spécialisation lecture liste)

```
s(X0,X0).
s([a|X1],X3) :- s(X1,[b|X3]).
?-s([a,a,a,b,b,b],[]).
```

Ajouter des arguments

La puissance des DCG vient de la possibilité d'ajouter des arguments aux terminaux et non-terminaux.

Example :

```
s(N) --> a(N),b(N),c(N).
a(0) --> [].
a(s(N)) --> [a],a(N).
b(0) --> [].
b(s(N)) --> [b],b(N).
c(0) --> [].
c(s(N)) --> [c],c(N).
```

Ajouter des arguments

La puissance des DCG vient de la possibilité d'ajouter des arguments aux terminaux et non-terminaux.

Example :

```
s(N) --> a(N),b(N),c(N).
a(0) --> [].
a(s(N)) --> [a],a(N).
b(0) --> [].
b(s(N)) --> [b],b(N).
c(0) --> [].
c(s(N)) --> [c],c(N).
```

Reconnaît le langage $a^n b^n c^n$ (non CFG)

Échappement Prolog

DCG étant une partie de Prolog, immédiat d'avoir des échappements vers Prolog dans les DCG

Reformulation DCG $a^n b^n c^n$, avec échappements arithmétiques

```
s(N) --> a(N),b(N),c(N).
a(0) --> [].
a(N) --> [a],a(M), {N is M+1}.
b(0) --> [].
b(N) --> [b],b(M), {N is M+1}.
c(0) --> [].
c(N) --> [c],c(M), {N is M+1}.
```

Empiler des terminaux

Possibilité (peu connue) d'empiler de nouveaux terminaux sur la liste d'entrée (*pushback*).

Gestion des agglutinés :

```
prep(à), [le] --> [au].
prep(à), [les] --> [aux].
```

La lecture du terminal aux

1. reconnait le non-terminal prep
2. empile le nouveau terminal les.

Implanté dans DyALog pour la lecture sur liste et (plus délicat) la lecture sur treillis de mots.

Alternatives

Possible d'avoir des alternatives dans les corps de productions

```
nc --> det, (adj ; []), nc.
```

équivalent à

```
nc --> det, adj , nc.
nc --> det, nc.
```

Intersection

Facile d'implanter un opérateur d'intersection A & B, expansable en A(X1,X2),B(X1,X2).

Intersection

Facile d'implanter un opérateur d'intersection A & B, expansable en A(X1,X2),B(X1,X2).

```
s --> (anbn,cs) & (as,bncn).
anbn --> ([] ; [a],anbn,[b]).
cs --> ([] ; [c],cs).
bncn --> ([] ; [b],bncn,[c]).
as --> ([] ; [a],as).
```

Intersection

Facile d'implanter un opérateur d'intersection A & B, expansable en A(X1,X2),B(X1,X2).

```
s --> (anbn,cs) & (as,bncn).
anbn --> ([] ; [a],anbn,[b]).
cs --> ([] ; [c],cs).
bncn --> ([] ; [b],bncn,[c]).
as --> ([] ; [a],as).
```

Reconnaît $a^n b^n c^n$ (hors CFG)

Disponible des DyALog

Étoile de Kleene

DyALog fournit l'opérateur de répétition @∗ ou étoile de Kleene
A −−> B,(C @∗),D équivalent à

```
A --> B,CRec,D.
CRec --> [] ; (C, CRec).
```

Gestion (très simplifiée) des coordonnées

```
gncoord -->
    ((gn,[',']) @*),
    gn, [et] gn.
```

un épluche légume, un ouvre-boite, un tire bouchon, une spatule et une cuillère

Étoile de Kleene avec agrégation et bornes

L'étoile de Kleene admet une forme complète permettant :

- de borner le nombre d'itération (`from` et `to`)
- d'agréger des résultats le long de boucle, en partant d'une valeur initiale `collect_first` jusqu'à une valeur finale `collect_last`

```
s(N) -->
    @*{ goal => a,
        from => 2,
        to => 10,
        collect_first => [0],
        collect_last  => [N],
        collect_loop  => [_N],  %% valeur en début de cycle
        collect_next  => [s(_N)] %% valeur en fin de cycle
      }.
```

Entrelacement

DyALog fournit l'opérateur d'entrelacement ##
permettant d'entrelacer de toute les manières possibles deux séquences.

$$\text{(a,b)\#\#(c,d)} \equiv \begin{cases} a, b, c, d \\ a, c, b, d \\ a, c, d, b \\ c, d, a, b \end{cases}$$

exemple1 : Ordre libre des arguments optionels d'un verbe ditransitif :

```
gv --> v, ((gn ; []) ## (gp ; [])).
```

exemple2 : Adjectifs et noms :

```
gn --> det, ((adj @*) ## nc).
```

Opérateurs réguliers

Les opérateurs d'alternative, de Kleene et d'entrelacement

- ne changent pas la complexité des CFG sous-jacentes aux DCG
- peuvent être expansés
- mais permettent une notation bien plus compacte
  (expansion $\Rightarrow$ #productions exponentiel en #opérateurs)
- et une exécution plus efficace

Quatrième partie IV

Jouer avec les DCG

Utilisations des arguments en DCG

Divers usages possibles :

- gestion d'accords
- blocage/activation de constructions syntaxiques
- gestion de déplacement de constituants
- construction de structures en sortie d'analyse

Outline

Gérer des accords en CFG

Problème : exprimer l'accord en nombre et genre avec une CFG

```
gn —> det,nc.
```

donne

```
gn_sg_masc —> det_sg_masc,nc_sg_masc.
gn_sg_fem —> det_sg_fem,nc_sg_fem.
gn_pl_masc —> det_pl_masc,nc_pl_masc.
gn_pl_fem —> det_pl_fem,nc_pl_fem.
```

Augmentation rapide (polynomiale) du nombre de clauses en fonction du nombre de valeurs possibles :

- ici $\{\mathrm{sg}, \mathrm{pl}\} \times \{\mathrm{masc}, \mathrm{fem}\} = 4$
- plus généralement $|V_1| \times \cdots \times |V_n|$

Gérer des accords en DCG

Accord en nombre et genre sur le groupe nominal en 1 clause DCG :

```
gn(Number,Gender) -->
      det(Number,Gender),
      nc(Number,Gender).
```

Accord avec le groupe verbal

```
s -->
      ( gn(Number,Gender), {Person=3}
      ; cln(Person)),
      gv(Number,Gender,Person).

gv(Number,Gender,Person) -->
      v(Number,Person),
      gn(_,_,_).
```

Bloquer/activer des constructions

Par exemple, bloquer la présence d'un sujet pour le mode impératif ou infinitif.

```
s --->
    (  {\+ domain(Mood,[imperative,infinitive])},
       gn(Number,Gender), {Person=3}
    ;  {\+ domain(Mood,[imperative,infinitive])},
       cln(Person)
    ;  {domain(Mood,[imperative,infinitive])}
    ),
    gv(Number,Gender,Person,Mood).
```

Un brin de sous-catégorisation

Bloquer/autoriser la présence d'un objet
en fonction de la sous-catégorisation d'un verbe

```
gv --> v(object),gn.
gv --> v(-), gn.
 v(object) --> [aime].
 v(-) --> [dort].
```

Plus complet

```
 gv --> v(Subcat),
        (({ domain(object:(+),SubCat) }, gn
         ; { domain(object:(-),SubCat) })
         ## ( { domain(iobject:Prep,SubCat) },gp(Prep) ;
              { domain(iobject:(-),Subcat) }
            )
        ).
v([object:(+),object:(-),iobject:à,iobject:(-)]) --> [donne].
gp(Prep) --> [Prep],gn.
```

Outline

Approches lexicalistes

Le lexique est un élément extrêmement important :

- fournit les informations morpho-syntaxiques (nombre, genre, temps, mode, . . .)
- mais peut aussi fournir les informations syntaxiques : sous-catégorisation, diathèse, contrôle, . . .
- $\Rightarrow$ tendance forte vers des grammaires lexicalisées

Question : comment représenter le lexique dans un monde Prolog ?

Lexique naif

Coder le lexique extensionnellement en tant que DCG :

```
nc(sg,fem) --> [pomme].
nc(pl,fem) --> [pommes].
adj(sg,masc) --> [petit].
adj(pl,masc) --> [petits]
adj(sg,fem) --> [petite].
adj(pl,fem) --> [petites].
v(sg,1,pres) --> [aime].
v(sg,2,pres) --> [aimes].
v(sg,3,pres) --> [aime].
...
```

Dissocier grammaire et lexique

Sortir le lexique de la grammaire et le voir comme une base de données externe :

Coté grammaire

```
nc(Num,Gen) --> [Form], {lexicon(nc,Form,[Num,Gen])}.
adj(Num,Gen) --> [Form], {lexicon(adj,Form,[Num,Gen])}.
v(Num,Pers,Tense) --> [Form], {lexicon(v,Form,[Num,Pers,Tense])}.
```

Coté lexique, sous forme extensionnelle

```
lexicon(nc,pomme,[sg,fem]).
lexicon(nc,pommes,[pl,fem]).
lexicon(adj,petit,[sg,masc]).
lexicon(adj,petits,[pl,masc]).
lexicon(adj,petite,[sg,fem]).
lexicon(adj,petites,[pl,fem]).
```

Factoriser le lexique

Pour gérer le lexique, on peut exploiter la puissance de Prolog pour calculer le lexique en fonction :

- lemme ou racine
- information morphologique de flexion

```
lexicon(nc,Form,[Num,fem]) :-
        domain(Suff:Num,['':sg,'s':pl]),
        name_builder('~w~w',[pomme,Suff],Form).
lexicon(adj,Form,[Num,Gen]) :-
        domain(Suff:Num:Gen,['':sg:masc,
                             's':pl:masc,
                             'e':sg:fem,
                             'es':pl:fem]),
        name_builder('~w~w',[petit,Suff],Form).
```

avec name_builder/3 utilisé pour construire de nouveaux symboles à partir d'un motif (~ `sprintf` en C)

Factoriser le lexique (suite)

On peut aussi factoriser en fonction de paradigmes morphologiques :

```
lexicon(nc,Form,[Num,Gen]) :-
        lemma(nc,Lemma,Gen,Paradigm),
        paradigm(Paradigm,Lemma,Form,Num).

lemma(nc,pomme,fem,nc_std).
lemma(nc,ami,masc,nc_std).

paradigm(nc_std,Lemma,Form,Num) :-
        domain(Suff:Num,['':sg,'s':pl]),
        name_builder('~w~w',[Lemma,Suff],Form).
```

Factoriser le lexique (suite)

```
lexicon(v,Form,Info) :-
        lemma(v,Lemma,Stem,Paradigm),
        paradigm(Paradigm,Stem,Form,Info).

lemma(v,aimer,aim,v1).
lemma(v,parler,parl,v1).

paradigm(v1,Stem,Form,Info) -->
        domain(Suff:Info,['e':sg:1:pres,'es':sg:2:pres,...]),
        name_builder('~w~w',[Stem,Suff],Form).
```

Externaliser le lexique

Il est possible d'aller loin dans la factorisation de l'information lexicale
Néanmoins :

- l'approche précédente engendre les formes à partir des lemmes
  $\Rightarrow$ plus efficace d'analyser la forme pour trouver son lemme (transducteur)
- lexiques très larges : plusieurs centaines de milliers de formes

Une meilleure approche consiste à totalement externaliser le lexique au sein d'un lexer.
Dans la grammaire :

```
nc(Num,Gen) --> [token(nc,Form,Lemma,[Num,Gen])].
adj(Num,Gen) --> [token(adj,Form,Lemma,[Num,Gen])].
...
```

Et lexer externe tel que `echo "petite pomme" | ./lexer` donne

```
'C'(0,token(adj,petite,petit,[sg,fem]),1).
'C'(1,token(nc,pomme,pomme,[sg,fem]),2).
```

Le lexer peut utiliser des techniques sophistiquées de représentation de grandes listes de mots (arbres à lettres, automates, ...).

Lexer et treillis de mots

L'approche lexer se prête à la construction de treillis de mots, permettant de représenter les ambiguïtés lexicales :

la belle ferme la voile

```
...
'C(1,token(adj,belle,beau,[sg,fem]),2).
'C(1,token(nc,belle,belle,[sg,fem]),2).
'C(2,token(nc,ferme,ferme,[sg,fem]),3).
'C(2,token(v,ferme,fermer,[sg,1,pres]),3).
'C(2,token(v,ferme,fermer,[sg,3,pres]),3).
...
```

Lexer et treillis de mots permettent aussi la gestion d'ambiguïtés de segmentation, des :

```
'C'(0,token(det,des,un,[pl,_,undef]),2).
'C'(0,token(prep,de,de,[]),1).
'C'(1,token(det,les,le,[pl,_,def]),2).
```

Outline

Déplacer des constituants

Les arguments DCG utilisable pour propager de l'information au sujet de constituants "déplacés" :

- relatives : Paul mange (la pomme)$_i$ que (Marie apporte ϵ_i)
- interrogatives : (quel livre)$_i$ ([S] Paul veut que ([S] Marie lise ϵ_i))

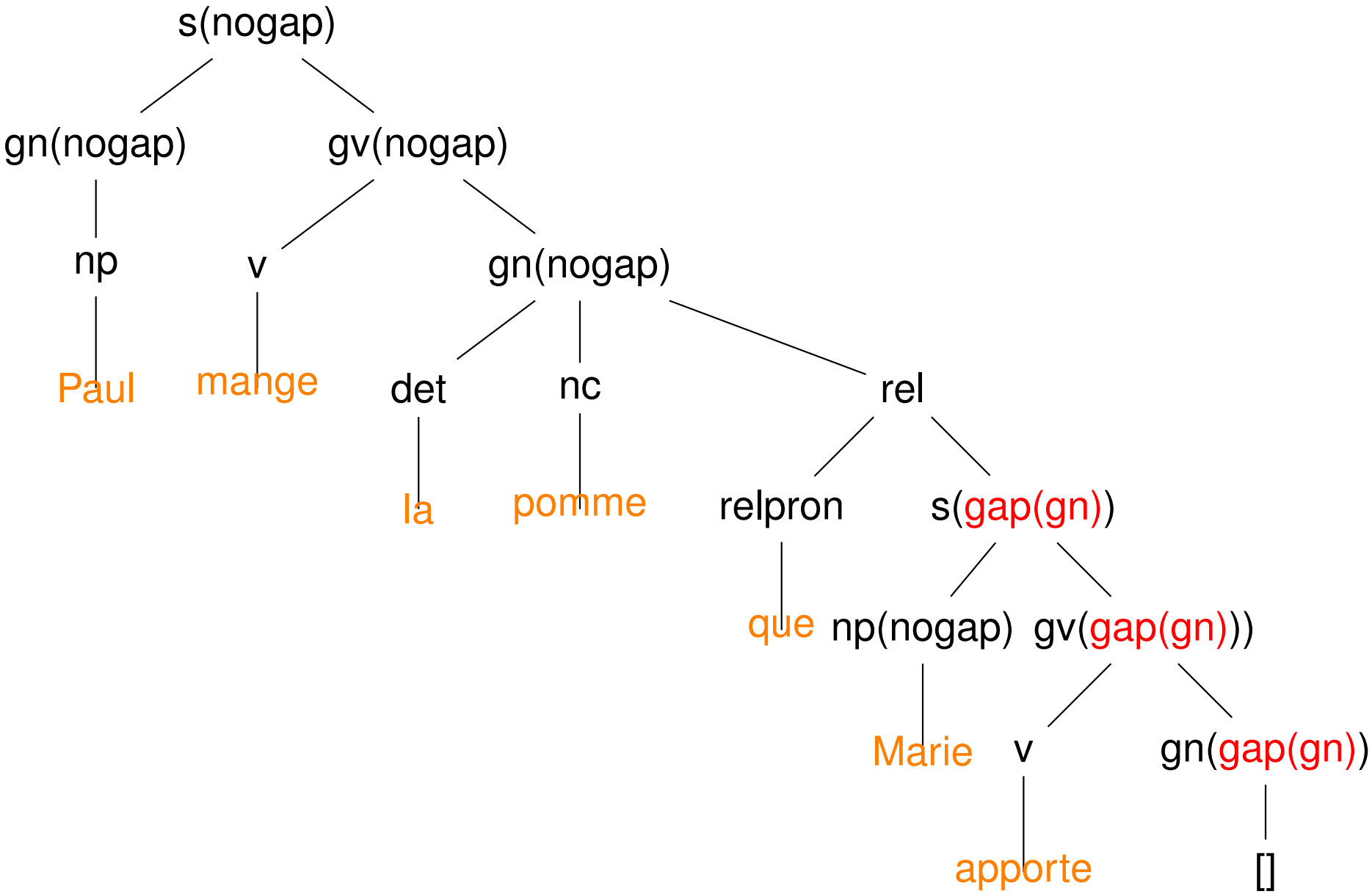
Le constituant extrait et sa trace peuvent être séparés par un nombre arbitrairement grand de constituants intermédiaires (GV, S, . . .) :

- $\Rightarrow$ très difficile d'exprimer cela avec des CFG

Représenter des relatives avec des gap

```
gn(nogap) --> det,nc, (rel ; []).
gn(gap(np)) --> [].
rel --> relpron, s(gap(gn)).
s(Gap) --> gn(Gap),gv(nogap).
s(Gap) --> gn(nogap),gv(Gap).
gv(Gap) --> v,gn(Gap).
gv(nogap) --> v.
gv(Gap) --> v, [que],s(Gap).
...
?-phrase(s(nogap),L,[]).
```

Représenter des relatives : exemple



Déclarer des piles de déplacements avec les BMG

La gestion des déplacements capturée avec un mécanisme de pile :

- empile une information sur le constituant à déplacer
- dépile cette information pour remplir une trace

Cette approche systématisée avec les Bound Movement Grammars [BMG] Pereira Lopes

- issues des grammaires d'extraposition
- étendent les DCG
- sont disponibles dans DyALog

Empiler et dépiler des constituants

- Déclarer des piles et ce qu'on peut y empiler

```
:-bmg_stacks([rel,quest]).
%% define stacks:  rel quest
%% define pushers: rel quest
%% define islands: isl (all stacks) isl_rel (rel)  isl_quest
    (quest)

:-bmg_pushable( gn,[quest,rel]).
%% gn pushable on stacks quest and rel
```

- Un non-terminal *A* est maintenant implicitement décorés des arguments (RelStackIn,RelStackOut) et (QuestStackIn,QuestStackOut)
- La grammaire :

```
gn --> det,nc, (rels ; []).
rels rel gn --> relpron, s.
s --> gn, gv.
gv --> v, gn.
gv --> v.
gv --> v, [que],s.
...
?-phrase(s,L,[]).
```

Bloquer des dépilements : ilots

La formulation précédente est trop laxiste avec la règle

```
gn --> gn,gp.
gp --> prep,gn.
```

Elle autorise

- Paul veut la pomme que la fille de ϵ mange un fruit.
- Paul veut la pomme que ϵ de Marie mange un fruit.

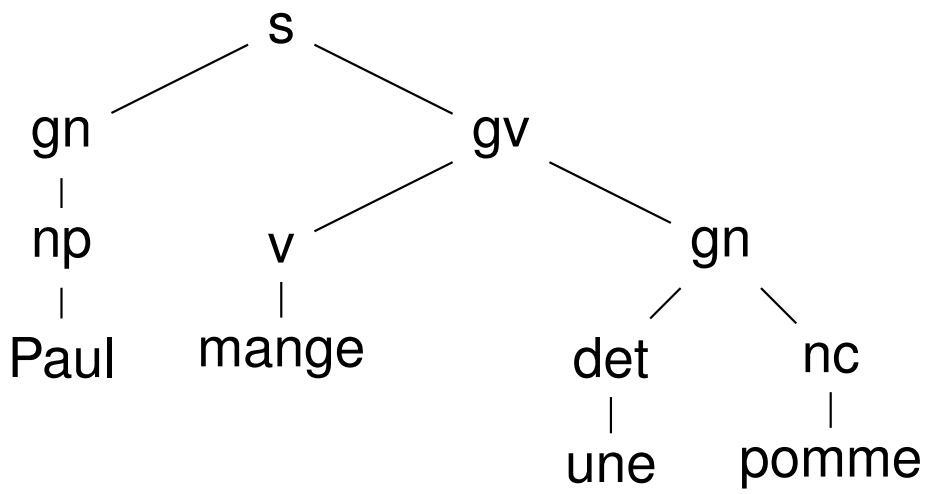
Remède : poser des barrières, empêchant le déchargement à l'intérieur de certains constituants :

```
s --> isl_rel gn, gv.
gv --> v, isl_rel gn.
```

Outline

Arbres d'analyse

```
s --> gn,v,gn.
gn --> det,nc.
gn --> np.
s --> s, gp.
gn --> gn,gp.
gp --> prep,gn.
```



Représentation sous forme de terme Prolog :
s(gn(np('Paul')),gv(v(mange),gn(det(une),nc(pomme))))

Construction compositionnelle

L'arbre d'analyse peut être construit de manière compositionelle, au fur et à mesure de l'application des productions

Utilisation d'un argument supplémentaire :

```
np(np(Form)) --> [Form], {lexicon(np,Form)}.
gn(gn(Det,Nc)) --> det(Det),nc(Nc).
det(det(Form)) --> [Form], {lexicon(det,Form)}.
nc(nc(Form)) --> [Form], {lexicon(nc,Form)}.
gv(gv(V,Obj)) --> v(V),gn(Obj).
v(v(Form)) --> [Form], {lexicon(v,Form)}.
```

Quand construire les arbres d'analyse

L'approche précédente utilise un argument pour construire les arbres pendant l'analyse :

- 1 argument à ajouter à tout les non-terminaux
- les arbres sont calculés même pour des analyses qui échouent
- l'argument supplémentaire nuit au partage de calculs
  (pour des analyseurs tabulaires – cours à venir)
- l'argument supplémentaire ne sert pas à diriger les analyses

Approche tabulaire DyALog :

- les arbres d'analyses sont extraits à partir des traces tabulées des calculs réussis
- l'ensemble des arbres d'analyse en fait représenté sous forme d'une forêt partagée d'analyse, exprimée comme une grammaire.
- forêt visible avec l'option –forest

Outline

Sémantique plate

Sémantique plate :

- des entités : paul, marie
- des variables d'entités : x, y
- des prédicats : $\text{aime}(\text{paul}, \text{marie})$, $\text{dormir}(x)$, $\text{manger}(x, y)$, $\text{donner}(x, y, z)$, $\text{pomme}(x)$, $\text{rouge}(x)$
- des formules construites par conjonction de prédicats : $\text{manger}(\textit{paul}, y) \wedge \text{pomme}(y) \wedge \text{rouge}(y)$
- la valeur logique true

Constructions "compositionnelles" à partir des sémantiques des constituants
Pour X −−> Head,Arg.

$$S_X = \text{comp}(S_{\text{Head}}, S_{\text{Arg}})$$

La nature de la composition change avec le type de la tête et le type de l'argument

Les sémantiques intermédiaires sont en général des λ-expressions comme $\lambda y.\lambda x.\text{manger}(x, y)$

Codage Prolog

- $\text{manger}(\textit{paul}, x) \rightsquigarrow$ manger(paul,X)
- $\text{manger}(\textit{paul}, x) \wedge \text{pomme}(x) \rightsquigarrow$ manger(paul,X) & pomme(X)
- $\lambda y.\lambda x.\textit{manger}(x, y) \rightsquigarrow$ Y^X^manger(X,Y)

Ajout dans les DCG

Ajout d'un argument supplémentaire dans les non-terminaux

```
gn(Sem) --> pn(Sem).
pn(paul^true) --> ['Paul'].
%% Paul => paul^true
```

```
gn(X^(SemHead & SemArg)) --> det(X^SemArg),nc(X^SemHead).
det(X^def(X)) --> [la].
det(X^undef(X)) --> [une].
nc(X^pomme(X)) --> [pomme].
%% la pomme => X^(pomme(X) & def(X))
```

Ajout dans les DCG (suite)

```
gv(X^(SemHead & SemArg)) --> v(Y^X^SemHead),gn(Y^SemArg).
v(Y^X^manger(X,Y)) --> [mange].
%% mange (la pomme)
%% => (Y^X^manger(X,Y)) (Z^(def(Z) & pomme(Z)))
%% => X^(manger(X,Z) & def(Z) & pomme(Z))
```

```
s(SemHead & SemArg) --> gn(X^SemArg),gv(X^SemHead).
%% Paul (mange la pomme)
%% => (paul^true) (X^(manger(X,Z) & def(Z) & pomme(Z)))
%% => paul^(manger(X,Z) & def(Z) & pomme(Z) & true)
```

```
s(Sem) -->
    gn(X^SemArg),gv(X^SemHead), {sem_simplify(SemHead & SemArg,
  Sem)}.
%% Paul (mange la pomme)
%% => paul^(mange(X,Z) & def(Z) & pomme(Z))
```

Réification des évènements

Problème : comment représenter la sémantique de paul vient demain
$\Rightarrow$ solution : réifier les évènements

- en introduisant des variables d'évènements $e_1, e_2, \ldots$ et
- en étiquetant les évènements par ces variables
- en autorisant des prédicats sur les variables d'évèments

paul vient demain $\rightsquigarrow e_1 : \text{venir}(\text{paul}) \wedge \text{demain}(e_1)$

```
gv(X^E^(E:SemHead)) --> v(X^SemHead).
v(X^venir(X)) --> [vient].
gv(X^(SemHead & SemMod)) --> gv(X^E^SemHead), adv(E^SemMod).
adv(E^demain(E)) --> [demain].
%% vient demain => (X^E^(E:venir(X))) (E^demain(E))
%% => X^(E:venir(X) & demain(E))
```

Déterminants et Quantifieurs

Problème : comment représenter la sémantique de chaque homme est mortel ?

Actuellement : $\text{chaque}(x) \wedge \text{homme}(x) \wedge \text{mortel}(x)$

Mais une formule quantifiée avec une portée est préférable :

$$\forall(x),\ \text{homme}(x) \wedge \text{mortel}(x)$$

ou encore

$$\forall(x),\ \text{homme}(x) \wedge e_1 : \text{mortel}(x)$$

Notation Hilog

DyALog offre la notation Hilog pour une pseudo-notation d'ordre supérieur, à savoir des variables de prédicats :

- P(X,Y) ou P (X,Y), équivalent à apply(P,X,Y)
- mange (X,Y), équivalent à apply(mange,X,Y)
  et différent de mange(X,Y)
- avec la directive DyALog :−hilog mange/2,
  mange(X,Y) équivalent à apply(mange,X,Y)

```
gv(X^(P(X,Y) & SemObj)) --> v(P),gn(Y^SemObj).
v(manger) --> [mange].
```

Limitations

Les systèmes Prolog (dont DyALog) n'offrent pas une véritable implémentation du λ-calcul :

- il manque une vraie β-réduction $(\lambda x.t)a \rightarrow_\beta t[x/a]$
- on simule (avec l'unification) une β-reduction, mais pas de gestion de renommage de variables et applications successives :

$$
\begin{aligned}
(\lambda f \lambda x.f(f(x)))(\lambda y.y * y) &\rightarrow_\beta & \lambda x.(\lambda y_1.y_1 * y_1))((\lambda y_2.y_2 * y_2)x) \\
&\rightarrow_\beta & \lambda x.(\lambda y_1.y_1 * y_1)(x * x) \\
&\rightarrow_\beta & \lambda x.(x * x) * (x * x)
\end{aligned}
$$

Pour des systèmes Prolog avec λ-calcul, voir λ-Prolog Dale Miller et Gopalan Nadathur, tutoriel Olivier Ridoux

Pour aller plus loin

Plus d'information sur les sémantiques plates et leurs mises en œuvre :

- Minimal Recursion Semantic [MRS]
  Copestake, Pollard, Sag et Flickinger

- SEMCONST (LORIA) `http://trac.loria.fr/~semconst/`
  travaux de Claire Gardent et Yannick Parmentier

Quand construire les formes sémantiques

Comme pour les arbres d'analyse :

- 1 argument à ajouter à chaque non-terminal
- en général, la forme sémantique ne guide pas l'analyse
- construction effectuée même pour les analyses qui vont échouer
- nuit au partage de calculs (en analyse tabulaire)

$\Rightarrow$ envisager une construction post-analyse à partir des traces tabulées des analyses réussies.