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

Première partie I

Lignes directrices

Introduction

Le Traitement Automatique des Langues (TAL) présent dès les origines de la Programmation en logique

- Recherche d'un formalisme syntaxique puissant, fondé sur une base logique solide
- Q-systèmes de Colmeraurer – 1970
- Grammaires de Clauses Définies (DCG - Definite Clause Grammars) Pereira et Warren – 1980
- Parsing as Deduction Pereira, Warren, Shabes, Shieber
- Prolog et contraintes, Dahl

Paradigme déclaratif

La force de la programmation en logique est de pouvoir

- exprimer de manière déclarative des informations complexes
- sans se préoccuper de la manière dans elles seront utilisées opérationnellement
  confiance dans le principe de déduction logique sous jacent

  algorithme = logique + contrôle [Kowalski]
- évolution nette par rapport à des approches antérieures mêlant données et opérations (RTN – *Recursive Transition Network*)

Non-déterminisme et ambiguité

Prolog conçu pour la gestion du non-déterminisme

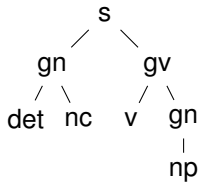
- exploration de l'espace de recherche par retour arrière (backtrack)
- important pour gérer les ambiguïtés importantes du langage, en particulier en analyse syntaxique

- mais gestion du non-déterminisme pas encore assez efficace
  ⇒ extensions tabulaires (partage de calculs)

Notation par constructeur

Flexibilité de construction (récursif) de termes avec les constructeurs, constantes et variables

- représentation d'arbres s(gn(det,nc),gv(v,gn(np)))



- représentation d'expressions : $a \wedge (b \vee c) \rightsquigarrow$ (a,(b;c))
  mais aussi

```
:-op(yfx,[or],500).  %% associatif droit
:-op(yfx,[and],400). %% associatif droit
?- F= a and (b or c).
```

- représentation de λ-expressions : $\lambda x \lambda y, \text{aime}(x, y) \rightsquigarrow$ X^Y^aime(X,Y)

Information partielle et sous-spécification

Prolog s'appuie sur l'unification :

- permet des notations compactes en ne précisant que la partie de l'information nécessaire pour appliquer une clause
  ⇒ sous-spécification
- permet de propager l'information

```
?-det(NumDet,GenDet),
  NumDet=plur,                    % les
  nom(NumNom,GenNom),
  NumDet=NumNom,GenDet=GenNom,    % enfants
  v(NumV,GenV),
  NumV=NumNom,GenV=GenNom,GenV=fem    % sont venues
```

- utile pour gérer des objets linguistiques complexes, comme les mots (beaucoup de propriétés), partageant des info (accord)
- Extensions vers Structures de traits typées [TFS - *Typed Feature Structure*] pour encore plus de compacité.

Méta-interprétation et compilation

Facile en Prolog d'écrire un méta-interprète pour Prolog
autrement dit : écrire Prolog en Prolog (similaire en Lisp, Scheme)

```
solve(Goal) :- clause((Goal :- Body)), solve(Body).
solve((Formula1,Formula2)) :- solve(Formula1), solve(Formula2).
solve((Formula1;Formula2)) :- solve(Formula1).
solve((Formula1;Formula2)) :- solve(Formula2).
```

Permet d'écrire en Prolog des extensions de Prolog :

- pour des formalismes syntaxiques
- pour de nouvelles stratégies d'analyse
- très pratique pour du prototypage rapide

Cependant, la méta-interprétation a un coût en temps
⇒ à terme, préférable d'évoluer vers une compilation

$$\text{compilation} = \text{méta-interprétation} + \text{évaluation partielle}$$

En s'inspirant d'un méta-interprète, obtention d'un compilateur en Prolog.

DyALog

Un Prolog maison, adapté au TAL, permettant :

- la gestion de divers formalismes syntaxiques (DCG, BMG=DCG++, TAG, RCG)
- une meilleure gestion du non-déterminisme (approche tabulaire)
- plus de choix sur les stratégies d'analyses
- l'utilisation de structures de traits (typées ou non)
- ...

`http://alpage.inria.fr/dyalog.fr.html`

DyALog : mode toplevel

DyALog peut être utilisé en mode interactif (–toplevel),
mais moins de fonctionnalités que d'autres Prolog

```
%> dyalog -toplevel
Entering DyALog toplevel
DyALog> ancetre(X,Y) :- parent(X,Y).
DyALog> ancetre(X,Z) :- parent(X,Y),ancetre(Y,Z).
DyALog> parent(paul,marie).
DyALog> parent(marie,jean).
DyALog> ?-ancetre(paul,Y).
        Y = jean
        Y = marie
DyALog> quit.
Leaving toplevel. Good bye!
```


DyALog : mode compilateur

DyALog surtout conçu pour **compiler** des programmes et des analyseurs syntaxiques.

- le programme : `ancetre.pl`

```
%% Directive: fait Prolog de type base de données
:-extensional parent/2.

%% Clauses (recursive)
ancetre(X,Y) :- parent(X,Y).
ancetre(X,Z) :- parent(X,Y),ancetre(Y,Z).

%% Requete - argv = accès aux args après '--'
?- argv([X]), ancetre(X,Y).
```

- la base de données généalogique : `famille.db`

```
parent(paul,marie).
parent(marie,jean).
```

Note : Séparation claire entre programme et données

DyALog : mode compilateur (suite)

- compilation : dyacc ancetre.pl –o ancetre
- la requête (avec argv([paul]))

```
%> ./ancetre famille.db -- paul
Answer:
    X = paul
    Y = marie
Answer:
    X = paul
    Y = jean
```

- variantes pour la requête :
  - ./ancetre –db famille.db -- paul
  - cat famille.db | ./ancetre - -- paul

DyALog : aide

dyalog –help

```
Usage for command dyalog
  –h –help          — this help
  –v <level>        — trace for <level> in dyam, index, share,
   or all
  –verbose <level> — same as –v
  –db <filename>    — load database filename
  –I <path>         — add <path> to DyALog search path
  –server           — enter server mode
  –loop             — enter loop mode
  –forest           — display the shared forest
  –fcount           — count number of derivations per answer
  –slex <string>    — parse from <string>
  –flex <filename>  — parse from <filename>
  –a <args>         — all remaining arguments given to DyALog
  — <args>          — all remaining arguments given to DyALog
```


DyALog : aide (suite)

- dyacc –help
- Documentation incomplète à
  `http://alpage.inria.fr/docs/dyalog.pdf`
  et dans la distribution de DyALog

Deuxième partie II

Quelques gammes : Expressions régulières, Automates à états finis et Transducteurs

Expressions régulières

Une valeur sûre de l'informatique et du TAL
- analyse lexicale (lexer)
- entités nommées
- morphologie

Constructeurs de base :
- constantes : alphabet fini (caractères, ou ce qu'on veut)
- Concaténation
- Alternation
- Étoiles de Kleene (répétitions)

Constructeurs additionnels (sucre syntaxique)
- intervalles de constantes `[...]`,
- optionel `E?` ,
- au moins une fois `E+`,
- entre *n* et *m* fois `E{n,m}`,
- différence `E1-E2`,
- intersection `E1 & E2`,
- ...

⇒ Langage d'expresions riche

Représentation Prolog

Nombre : (\+|−)?[0−9]∗(\.[0−9]+)? ⇝

```
:−op(yf,['@?'],500).
:−op(yf,['@+'],500).
:−op(yf,['@∗'],500).

rx(numbers,
   ( (c(+) ; c(−)) @?,
     range("0123456789") @∗,
        ( c(0'.),␣range("0123456789")␣@+␣)␣@?
␣␣␣)).
```

Notes :

- "0123456789" équivalent à la liste [0'0,0' 1,..,0 '9]
- D'autres représentations sont possibles !

Un interpréteur en Prolog

Utilisation d'un prédicat regexp(RegExp,Left,Right)

- pour reconnaître l'expression régulière RegExp
- entre la "*position*" Left
- et la "*position*" Right

Note : Notion de position définie plus loin mais

- utilisation récurrente en Prolog/TAL de paires de positions
- lien avec notion de différence de liste Left−Right ≡ Left=[c1 ,.., cN|Right]
- et instance de la notion d'accumulateur

```
reverse(X, Rev) :− reverse(X, [], Rev).
reverse([], Rev, Rev). % Rien de plus à renverser
reverse([H|T], Prev, Rev) :− reverse(T, [H|Prev], Rev).
```


Déconstruire

Déconstruire les divers constructeurs d'expressions régulières :

```
regexp((E1,E2),L,R) :−
        regexp(E1,L,M),
        regexp(E2,M,R).

regexp((E1;E2),L,R) :−
        ( regexp(E1,L,R)
        ; regexp(E2,L,R)
        ).

regexp(E @∗,L,R) :−
        ( L=R ;
          regexp(E,L,M),
          regexp(E @∗,M,R)
        ).
```


Déconstruire (suite)

Quelques constructeurs supplémentaires

```
regexp(true,L,L).

regexp(range(Range),[C|R],R) :−
  domain(C,Range). %% ou member(C,Range) en Prolog

regexp(E @?,L,R) :−
  regexp((true;E),L,R).

regexp(E @+,L,R) :−
regexp(E,L,M),
regexp(E @∗,M,R).
```

Lire les symboles

```
regexp(c(C),L,R) :- 'C'(L,C,R).
regexp(range(Range),L,R) :- 'C'(L,C,R), domain(C,Range).
```

Abstraction du lecteur de symbole avec le prédicat 'C'/3 :

```
:-extensional 'C'/3. %% directive DyALog
'C'([C|R],C,R).
```

L'abstraction permet de changer facilement de source de lecture :

- source = liste [il ,mange,une,pomme] (formulation ci-dessus)
  immédiat et réversible : lecture liste / génération liste
- source = treillis de mots

```
'C'(0,il,1).
'C'(1,mange,2).
'C'(2,une,3).
'C'(3,pomme,4).
```

avantage : plus efficace et extensible

La requête

```
:-extensional rx/2.
?-argv([Name,Symb]),
  rx(Name,RX),
  atom_chars(Symb,CharString),
  regexp(RX,CharString,[])
  .
```

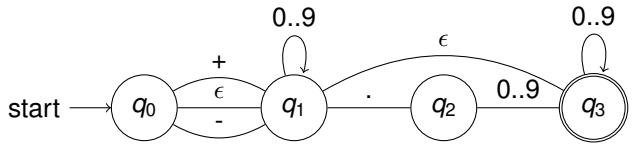
Compilation : dyacc regexp.pl -o regexp

Exécution : ./regexp numbers.rx -- numbers +123.98

Automates à états finis

- Équivalence entre expressions régulières et Automates à États Finis
  [FSA – *Finite State Automata*]
- FSA ≡ langage opérationnel bas niveau
  (vs RegExp ≡ langage déclaratif haut niveau)
- simplicité formelle ⇒ moteur Prolog simple

'numbers' en FSA




```
init_state(numbers,0).
final_state(numbers,3).
trans(numbers,0,1,c(0'+)).
trans(numbers,0,1,c(0'-)).
trans(numbers,0,1,true).
trans(numbers,1,1,range("0123456789")).
trans(numbers,1,2,c(0'.)).
trans(numbers,2,3,range("0123456789")).
trans(numbers,3,3,range("0123456789")).
trans(numbers,1,3,true).
```

Moteur FSA en DyALog

```
:-extensional trans/4.
:-extensional init_state/2.
:-extensional final_state/2.

fsa(Name,L,R) :-
        init_state(Name,Init_State),
        fsa(Name,Init_State,L,R).

fsa(Name,S,L,L) :- final_state(Name,S).
fsa(Name,S,L,R) :-
        trans(Name,S,T,A),
        fsa_action(A,L,M),
        fsa(Name,T,M,R)
        .
```


Moteur FSA : actions

```
fsa_action(true,L,L).
fsa_action(c(X),L,R) :- 'C'(L,X,R).
fsa_action(range(D),L,R) :- 'C'(L,X,R),domain(X,D).
```

Note : facile d'ajouter de nouvelles actions

Moteur FSA : requête

```
?-argv([Name,Symb]),
  rx(Name,RX),
  atom_chars(Symb,CharString), % prédicat builtin
  fsa(RX,CharString,[])
  .
```

Compilation : dyacc fsa.pl -o fsa

Exécution : ./fsa numbers.fsa -- numbers +123.98

Lire et émettre

Facile d'étendre en un moteur pour des transducteurs à états finis

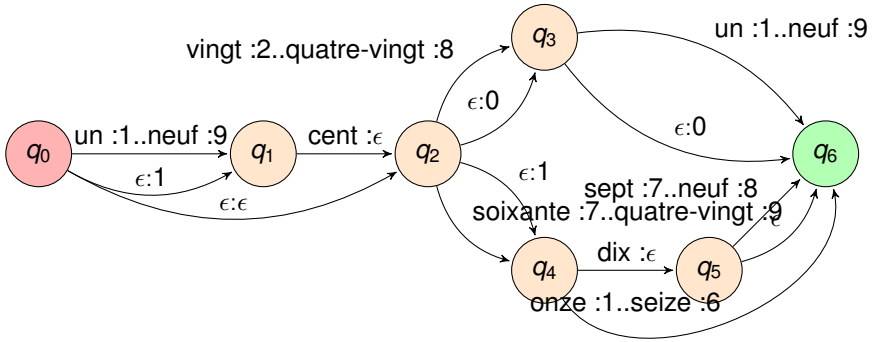
- permet de lire sur une bande et écrire de l'autre
- les rôles peuvent être théoriquement être inversés

Exemple : normaliser des nombres entre 0 et 999 écrits en toute lettre
ex : lire deux cent seize et retourner [2,1,6]
ex2 : réciproquement, lire [2,1,6] et retourner deux cent seize.

Les transitions du transducteur sont maintenant décorées d'une paire d'action :

```
fst_trans(numbers2,0,1,true : c(1)).
fst_trans(numbers2,0,1,c(un) : c(1)).
fst_trans(numbers2,0,1,c(deux) : c(2)).
..
```

Forme du transducteur (simplifié)



Au moins deux faiblesses dans ce transducteur !

Moteur FST

Similaire au moteur FSA, mais :

- utilise 2 paires de positions L1,R1 et L2,R2
- utilise 1 paire d'actions A1 et A2

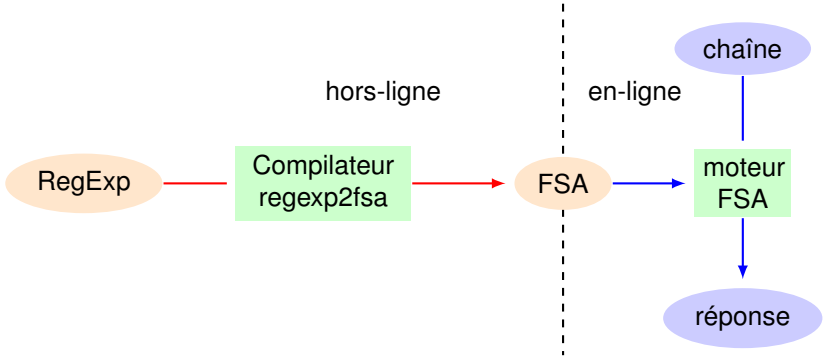
```
fst(Name,S,L1,R1,L2,R2) :-
    fst_trans(Name,S,T,(A1:A2)),
    fst_action1(A1,L1,M1),
    fst_action2(A2,L2,M2),
    fst(Name,T,M1,R1,M2,R2).
```

Notes :

- Jeux d'actions identiques ou différents pour chaque niveau (1 ou 2).
- Immédiat d'étendre à 3 (ou plus) bandes
  par exemple pour lire les nombres en français, les normaliser et les générer en anglais.

Principe

- Utiliser la notation haut-niveau des expressions régulières
- Obtenir l'efficacité des moteurs FSA ou FST
  (ou directement le moteur Prolog)



Regexp2fsa

Modèle générique de compilateur

1. lire le source (une regexp)
2. transformer (similaire au moteur regexp)
   déconstruire les expressions en introduisant de nouveaux états
   regexp2fsa(Name,RegExp,State1,State2)
3. émettre la cible (FSA)
   émettre les transitions avec emit_trans(Name,State1,State2,Action)

Déconstruire les expressions

```
regexp2fsa(Name,(E1,E2),S1,S2) :-
        regexp2fsa(Name,E1,S1,S3),
        regexp2fsa(Name,E2,S3,S2)
        .
regexp2fsa(Name,(E1;E2),S1,S2) :-
        regexp2fsa(Name,E1,S1,S2),
        regexp2fsa(Name,E2,S1,S2)
        .
regexp2fsa(Name,E @*,S1,S2) :-
        regexp2fsa(Name,E,S1,S1),
        regexp2fsa(Name,true,S1,S2)
        .
```

Note : similaire pour les autres constructions

Construire les transitions

```
regexp2fsa(Name,A,S1,S2) :-
    domain(A,[c(X),range(D),true]),
    (var(S2) -> mutable_new_state(Name,S2) ; true),
    emit_trans(Name,S1,S2,A).
```


Emettre les transitions

```
%% bibliotheque analogue a C printf
:-require 'format.pl'.

%% directive DyALog: prédicat à la Prolog
:-rec_prolog emit_trans/4.

emit_trans(Name,S1,S2,A) :-
     format('trans(~w,~w,~w,~w).\n',[Name,S1,S2,A]).
```


Chapeau de compilation

```
:-extensional rx/2.
:-rec_prolog regexp2fsa/4.

?-rx(Name,RX),
  mutable_init_state(Name,Init),
  regexp2fsa(Name,RX,Init,Final),
  format('init_state(~w,~w).\n',[Name,Init]),
  format('final_state(~w,~w).\n',[Name,Final]),
  %% la compilation se contente d'émettre
  %% => pas de réponse avec fail
  fail.
```

Engendrer des états

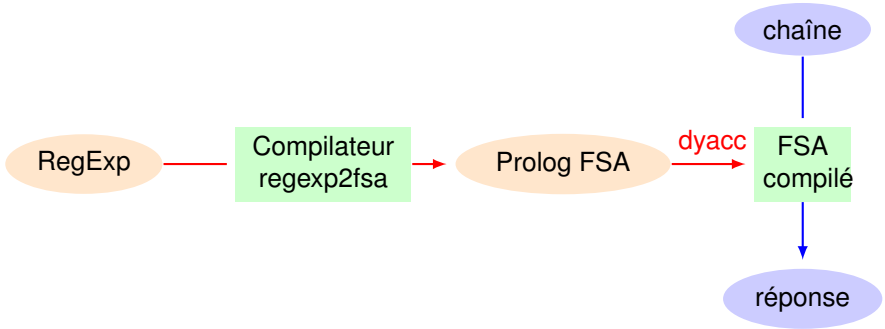
- Utilisation de prédicats *builtin* non logiques de DyALog : mutable/1 et mutable_inc/1
- Remplacement possible par des assert/retract
- Gestion possible sans prédicats non logiques (en propageant le max des états déjà générés)

```
:-std_prolog mutable_init_state/2, mutable_new_state/2.

mutable_init_state(Name,0) :-
    mutable(MX,1),
    record(state(Name,MX)).

mutable_new_state(Name,S) :-
    recorded(state(Name,MX)),
    mutable_inc(MX,S).
```


Compiler vers Prolog/DyALog

Une compilation plus poussée des expressions régulières peut prendre Prolog comme cible (plutôt que des transitions FSA),

- en émettant des clauses Prolog encapsulant le comportement du moteur FSA pour chaque transition de l'automate.
- (alternative) composer deux compilateurs : regexp $\rightsquigarrow$ fsa, puis fsa $\rightsquigarrow$ Prolog
  approche fréquente de compilation multi-passe avec des représentations intermédiaires.



Pour aller plus loin

Voir le package FSA de Gert van Noord

- disponible sur `http://www.let.rug.nl/~vannoord/Fsa/`
- fonctionne avec YAP, SICStus et SWI.
- Beaucoup plus d'opérateurs d'expression régulières
- Optimisations des FSA/FST (minimisation et déterminisation)
- FSA et FST pondérés

Troisième partie III

DCG : Présentation

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

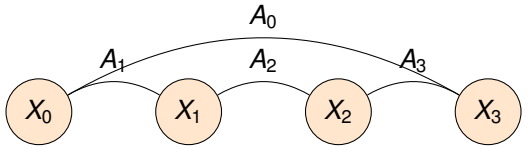
Lecture immédiate d'une production CFG $A_0 \leftarrow A_1 \ldots A_n$ en terme de logique :

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

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

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. ⇝ 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).
```

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).

```
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

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 $\{\text{sg}, \text{pl}\} \times \{\text{masc}, \text{fem}\} = 4$
- 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.
```

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 ($\sim$ `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).
```

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))

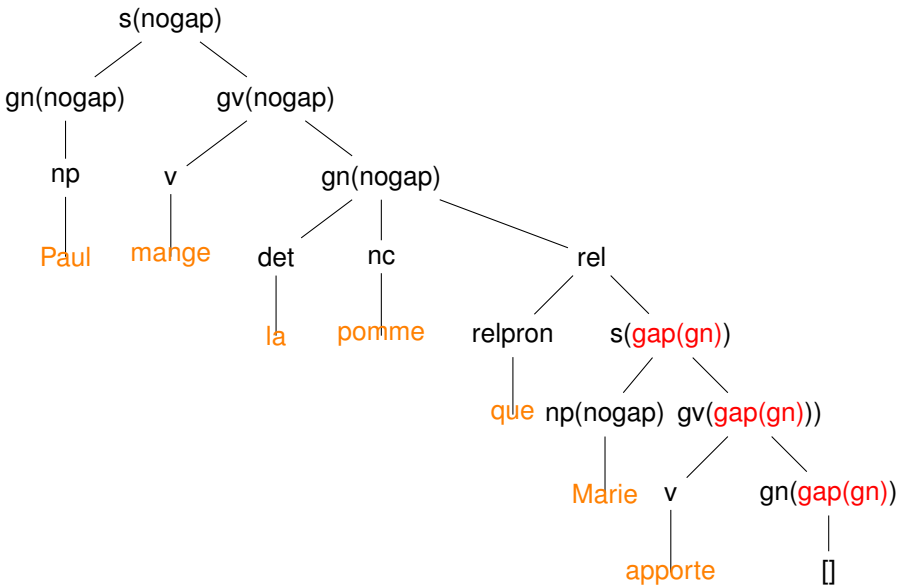
Le constituant extrait et sa trace peuvent être séparés par un nombre arbitrairement grand de constituants intermédiaires (GV, S, ...) :

- $\Rightarrow$ 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

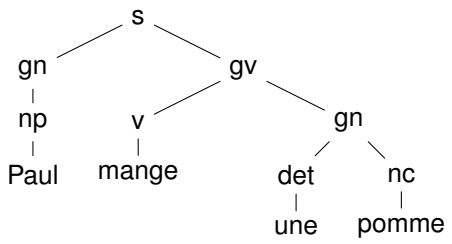
- Paul veut la pomme que la fille de ϵ mange un fruit.
- Paul veut la pomme que ϵ de Marie mange un fruit.

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

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


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

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
⇒ 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.

Cinquième partie V

Représenter l'information

Motivations

La notation par termes Prolog n'est pas toujours adaptée à l'écriture compacte
- de termes riches en information
- mais en général partiellement spécifiés dans les productions.

Exemple d'un verbe avec ses diverses informations morpho-syntactiques :
v(Number,Gender,Person,Tense,Mood,Aux,Voice)

Facile de se tromper :
- sur l'arité : v(Number,Gender,Person,Mood,Aux,Voice)
- sur l'ordre des arguments : v(Number,Gender,Person,Mood,Tense,Aux,Voice)
- sur les valeurs des propriétés :
  v(pl ,Gender,Person,pres,ind,Aux,Voice)
  v(plural ,Gender,Person,present,ind,Aux,Voice)

Notation vite lourde car tout les champs doivent être explicitement remplis, même si c'est de manière anonyme : v(_,_,3,_,_,_,_)

Mise à jour compliquée si on change l'arité d'un terme :
v(Number,Gender,Person,Tense,Mood,Aux,Voice,SupportVerb)

Motivations (suite)

On recherche des notations :
- évitant les problèmes d'arité et d'ordre sur les propriétés
- permettant de ne fournir que les informations utiles (sous-spécification)
- permettant l'ajout de nouvelles propriétés
- évitant au maximum les erreurs en spécifiant des contraintes de bonne formation
  similaire au typage en ML (CAML) ou aux DTD/schémas XML en XML
- exploitant ces contraintes pour encore plus de compacité
- exploitant ces contraintes pour plus d'efficacité pendant l'exécution

Quelques pistes

Abstraitement, un objet linguistique
- d'un certain type
- se caractérise par un ensemble de propriétés ou traits
- prenant leurs valeurs dans des domaines de valeurs, souvent finis

Les verbes de type v, avec les propriétés et valeurs

propriétés	valeurs
number	sg, pl
gender	fem, masc
person	1,2,3
tense	present, future, imperfect, past,perfect, ...
mood	indicative, subjonctive, conditional, infinitive, ...
aux	être, avoir
voice	active, passive

Proposition :
- domaines finis de valeurs
- structures de traits, typés ou non

Notation Hilog

Déjà vue

Préliminaire : unification immédiate

En DyALog, l'opérateur :: permet l'unification de termes pendant la phase de lecture des programmes (plutôt que pendant la compilation ou l'exécution).

```
X::gn(Number,Gender) --> X, rel(gap(X)).
```

(presque) équivalent à :

```
gn(Number,Gender) --> gn(Number,Gender), rel(gap(Number,Gender)).
```

Motivation principale pour :: : partager des occurrences de termes complexes

Unification immédiate (suite)

Attention : Unification immédiate délicate d'utilisation :

```
p(X) :- q(X::f(a)).
p(X) :- r(X::g(b)).
```

⇒ OK

```
p(X) :- q(X::f(a)) ; r(X::g(b)).
```

⇒ Erreur ! Échec de l'unification immédiate

Important : Portée de l'unification immédiate = portée des variables
= "clause" Prolog (clause, fait, production, ...)

Principe

Possibilité de

1. nommer un ensemble fini de valeurs (atomiques)
2. utiliser des variables prenant leur valeur dans des sous-ensembles

```
%% Declaration
:-finite_set(mood,[cond,ger,imp,ind,inf,part,subj]).

%% Utilisation
s(Mood) --> gv(Mood::mood[inf,imp,part,ger]).
s(Mood) --> gn, gv(Mood::mood[cond,ind,subj]).
```

Spécifique DyALog, mais proche :

- énumérations à la C `enum` (ou en CAML)
- programmation logique avec contraintes sur les domaines finis cc(FD), Van Hentenryck, V. Saraswat, Y. Deville
  mais mise en oeuvre bien plus limitée dans DyALog (pas de contraintes)

Quelques extensions

- Toutes les valeurs : person[] ≡ person[1,2,3]
- Définition par différence :

```
%% Utilisation
s(Mood) --> gv(Mood::mood[~ [cond,ind,subj]]).
s(Mood) --> gn, gv(Mood::mood[cond,ind,subj]).
```

- Définition de sous-ensembles :

```
%% Déclaration
:-subset(finite,mood[cond,ind,subj]).
:-subset(non_finite,mood[~ [cond,ind,subj]]).
%% Utilisation
s(Mood) --> gv(Mood::non_finite[]).
s(Mood) --> gn, gv(Mood::finite[]).
```

Note : Les sous-ensembles sont du *sucre syntaxique* :
finite [ind,subj] ⇝ mood[ind,subj]

Aspects formels

Un domaine fini de type T est associé à un domaine fini de valeurs $\mathcal{D}_T$.

- L'interprétation de FD-terme t[a1 ,..., an] est $(T, \{a_1, \ldots, a_n\})$, avec $a_i \in \mathcal{D}_T$
  (cas spécial) $\mathcal{I}(\mathrm{t}[]) = (T, \mathcal{D}_T)$
- L'interprétation de t[~[a1 ,..., an]] est $(T, \mathcal{D}_T / \{a_1, \ldots, a_n\})$
- L'unification de deux FD-termes $(T, \mathcal{D}_1)$ et $(T, \mathcal{D}_2)$ donne $(T, \mathcal{D}_1 \cap \mathcal{D}_2)$
  avec échec de l'unification si $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$
- le terme $(T, \mathcal{D}_1)$ généralise (ou subsume) le terme $(T, \mathcal{D}_2)$ ssi $\mathcal{D}_2 \subset \mathcal{D}_1$

Unification : exemple

```
?–X=mood[cond, ger, imp, ind, inf],
  Y = mood[ger, imp, ind, inf, part],
  X=Y.
%% Answer:
  X = A__12::mood[ger, imp, ind, inf]
  Y = A__12::mood[ger, imp, ind, inf]
```

Note : les deux variables X et Y ont été redirigées vers une nouvelle variable A__12, liée au nouveau domaine de valeurs.

```
?– finite[] = non_finite[].
%% => échec !
```


Singleton

DyALog propose un traitement spécial des ensembles singleton
le terme singleton $(T, \{a\})$ est considéré comme équivalent à la constante a.

```
?–X=mood[ind].
   X = ind
```

Impact sur l'unification :

- l'unification de 2 FD-termes peut être une constante

```
?–X=mood[ind,subj], X=mood[ind,cond].
        X = ind
```

- Un FD-terme peut être unifié avec une constante

```
?–X=ind,X=mood[ind,cond].
        X = ind
```


Singletons (suite)

Attention : L'interprétation des singletons est pratique
mais incorrecte (formellement) !
En effet, l'unification n'est plus associative : $(t_1 \sqcup (t_2 \sqcap t_3)) \neq ((t_1 \sqcup t_2) \sqcup t_3)$

```
:–finite_set(foo,[a,b,c]).
:–finite_set(bar,[c,d,e]).

?–X=foo[],X=bar[],X=c.
%% => Échec
?–X=foo[],X=c,X=bar[].
        X = c
```

Représentation et efficacité

Les FD-termes sont utiles d'un point de vue descriptif :

```
s(Mood) --> gn, gv(Mood::finite[]).
```

équivalent à :

```
s(Mood) --> gn, gv(Mood), { domain(Mood,[cond,ind,subj]) }.
```

Mais FD plus efficace en pratique car évitant une exploration en retour arrière des différentes valeurs.

Efficacité

L'efficacité provient d'une implantation par vecteurs de bits :

```
%%          bit pos:0 1 2 3 4
:-finite_set(foo,[a,b,c,d,e]).
?-X=foo[a,c,d],
  X =.. T.    %% destructure X en T
%% Answer
   X = B__5::foo[a, c, d]
   T = [$SET$,B__5,foo(a, b, c, d, e),13]  %% 13 = 1+4+8 = 10110
```

ie DVar::foo[a, c, d] représenté en interne par 'SET'(DVar,foo(a,b,c,d,e),13)

Unification $\equiv$ **et**-booléen && : $(T, \mathrm{bv}_1) \sqcup (T, \mathrm{bv}_2) = (T, \mathrm{bv}_1 \,\&\&\, \mathrm{bv}_2)$

- pas de retours arrière
- unification en temps (quasi) constant

Idem pour le test de subsumption $t_1 \sqsubseteq t_2$ ssi $\mathrm{bv}_1 \,\&\&\, \mathrm{bv}_2 == \mathrm{bv}_2$

Termes déréférençables

Les FD-termes sont un cas particulier de termes déréférençables (DyALog) :

- en Prolog, une variable X peut être liée à un terme t
  t est la valeur de X par déréférencement
- en DyALog, un deref term t_1 peut être lié à un terme t_2
  t_2 est la valeur de t_1 par déréférencement

Les deref-termes possède une variable en 1er argument
'SET'(DVar,foo(a,b,c,d,e),13)

Un deref-term est lié quand cette variable est liée.

Un dérérencement peut suivre une chaîne de liaison
X::foo[a,b,c,d,e] $\rightsquigarrow$ (liaison X) Y::foo[a,b,c] $\rightsquigarrow$ (liaison Y) b

Quelques exemples

Très nombreux usages des domaines finis en TAL,

- morpho-syntaxe
  valeurs de propriétés, comme number, gender, mood, tense,
- syntaxe (valeurs de propriétés)
- morphologie
  valeurs de propriétés et classes de lettres
- phonétique, phonologie
  valeurs de propriétés et classes de sons (fricative, labiale, ...)

Exemple : regexp 2 bandes

Reconnaissance d'un lemme à partir d'une forme et de règles morphologiques exprimées avec des transducteurs

```
:-finite_set(letters,[a,b,d,...,z]).
:-subset(voyelles,letters[a,e,i,o,u,y]).
:-subset(consonnes,letters[b,c,d,f,g,h,...,z]).
:-subset(doubles,consonnes[l,n,m]).

suffix(v1,
       ( c(X::doubles[]) : c(X),
         c(X) : true,              %% même lettre doublée
         c(Y::voyelles[e]) : c(Y), %% c(e) : c(Y) aussi OK
         true : c(consonnes[r])
       ),
       [ person: person[1,3],
         tense:  tense[present],
         mood:   mood[ind,subj]
       ]).

%% appe+lle => appe+ler    1|3.pres.ind|subj
```


Exemple (variante)

Exploitation plus agressive de l'unification immédiate !
partage des actions plutôt que partage des lettres

```
suffix(v1,
       ( X::c(doubles[]) : X,
         X : true,
         Y::c(voyelles[e]) : Y,
         true : c(consonnes[r])
       ),
       [ person: person[1,3],
         tense:  tense[present],
         mood:   mood[ind,subj]
       ]).

%% appe+lle => appe+ler    1|3.pres.ind|subj
```


Représenter des propriétés

Besoin naturel de représenter des ensembles de propriétés.

Plus élégant de nommer explicitement les propriétés, ce qui rend superflu :

- leur ordre d'apparition
- leur présence, quand non informative (information partielle)

⇒ même soucis dans la plupart des langages de programmation :

- `struct` en C
- `hash` en Perl (Python, Ruby, ...)
- listes de propriétés en Lisp

Notion de structures de traits en TAL

Notation avm

Notation matricielle des structures de traits, sous forme de Matrice Attributs Valeurs [AVM- *Attribute Value Matrix*]

$$\begin{bmatrix} \text{number} & \text{sg} \\ \text{person} & 2 \\ \text{tense} & \text{present} \\ \text{mood} & \text{indicative} \end{bmatrix} \equiv \begin{bmatrix} \text{person} & 2 \\ \text{mood} & \text{indicative} \\ \text{tense} & \text{present} \\ \text{number} & \text{sg} \end{bmatrix}$$

Récursion

La valeur d'une propriété peut être, récursivement, une structure de traits (AVM)

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance

Plusieurs propriétés peuvent partager une même valeur :

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\equiv \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance (suite)

Quelques subtilités derrière la notion de réentrance :

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\not\equiv \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance & instanciation

La réentrance doit être préservée par instanciation :

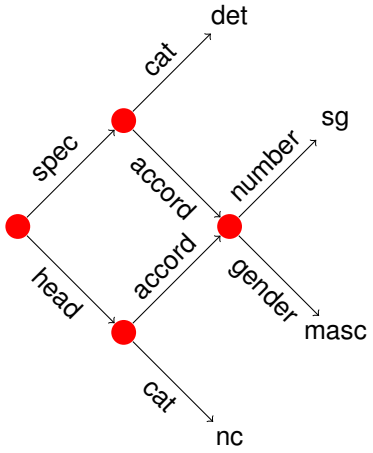
$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \rightsquigarrow \text{(inst.)} \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{fem} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \not\rightsquigarrow \text{(inst.)} \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{fem} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Notation graphe

Jusqu'à un certain point, les structures de traits sont représentables sous formes de graphes

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$



Note : intuitif de parler de chemin pour une séquence de traits dans le graphe et dans l'AVM
ex : chemin spec.accord.gender

Unification

- Une structure de traits précise de l'information sur un objet, au travers des valeurs de traits
- L'unification de deux termes t_1 et t_2 accumule l'information $\rightsquigarrow t_1 \cup t_2$ (information plus spécifiée) – opération commutative et associative
- $\Rightarrow$ algorithme :
  - tout chemin p présent dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$
  - toute valeur pour un chemin p dans t_1 ou t_2 est présente dans $t_1 \sqcup t_2$
  - tout point de partage (réentrance) dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$

$$\begin{bmatrix} \text{Det} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Num} & \text{sing} \end{bmatrix} \end{bmatrix} \\ \text{Nom} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Num} & \text{sing} \end{bmatrix} \\ \text{Cat} & \text{N} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{Nom} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Genre} & \text{Masc} \end{bmatrix} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{Det} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Num} & \text{sing} \\ \text{Genre} & \text{masc} \end{bmatrix} \end{bmatrix} \\ \text{Nom} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Num} & \text{sing} \end{bmatrix} \\ \text{Cat} & \text{N} \end{bmatrix} \end{bmatrix}$$

Unification (suite)

Cas avec réentrance :

$$\begin{bmatrix} \text{Det} & \begin{bmatrix} \text{Accord} & \boxed{1}\begin{bmatrix} \text{Num} & \text{sing} \end{bmatrix} \end{bmatrix} \\ \text{Nom} & \begin{bmatrix} \text{Accord} & \boxed{1} \\ \text{Cat} & \text{N} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{Det} & \begin{bmatrix} \text{Accord} & \begin{bmatrix} \text{Genre} & \text{Masc} \end{bmatrix} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{Det} & \begin{bmatrix} \text{Accord} & \boxed{1}\begin{bmatrix} \text{Num} & \text{sing} \\ \text{Genre} & \text{masc} \end{bmatrix} \end{bmatrix} \\ \text{Nom} & \begin{bmatrix} \text{Accord} & \boxed{1} \\ \text{Cat} & \text{N} \end{bmatrix} \end{bmatrix}$$

Subsomption

- une FS t_1 généralise (ou subsume) une FS t_2, noté $t_1 \sqsubseteq t_2$, ssi t_1 précise moins d'information que t_2
- (rappel) $\sqsubseteq$ est un pré-ordre partiel : $t \sqsubseteq t$; $t_1 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_3 \Rightarrow t_1 \sqsubseteq t_3$
- $\Rightarrow$ algorithme :
  - tout chemin dans t_1 est présent dans t_2
  - toute valeur pour un chemin p dans t_1 est présente pour p dans t_2
  - tout point de partage associé à un chemin p dans t_1 est présent dans t_2

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \sqsubseteq \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Structures de traits ouvertes en Prolog

Les structures sont directement représentables comme des listes Prolog

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

```
?- FS = [spec: [accord: X::[number: sg],
                cat: det],
         head: [accord: X,
                cat: nc]
        ].
```


Efficacité

Divers problèmes d'efficacité avec la représentation ouverte en liste des FS :

- l'accès à la valeur d'un trait oblige à parcourir la list

```
fs_value(FS,F,Val) :- domain(F:Val,FS).
```

- l'unification de base Prolog n'est pas suffisante : il faut implanter une unification spécifique non efficace :

```
fs_unif(t1,t2) retourne FS
  t3 := []
  pour chaque f:v1 dans t1,
     si f:v2 dans t2,
     alors ajouter f:fs_unif(v1,v2) à t3
     sinon ajouter f:v1 à t3
  pour chaque  f:v2 dans t2,
     si f:v1 n'est pas dans t1,
     alors ajouter f:v2 à t3
  retourner t3
```

- $\Rightarrow$ nombreux parcours de listes :
  complexité en $O(n^2)$, n taille des structures
- Plus efficace si utilisation de listes triées de traits (ou arbres balancés)
- Sinon, codage en interne (en C) appelé à partir de Prolog

Manque de contrôle

Les FS ouvertes n'offrent pas de contrôle sur les traits introduits
$\Rightarrow$ facile de faire une erreur :

- sur le nom d'un trait : num vs number
- sur l'utilisation d'un trait dans un contexte inapproprié
  ex : par unification, ajout d'un trait tense dans la FS associée à un nom

Structures de traits fermées

Principe :

1. typer les structures de traits
2. associer à chaque type la liste (finie) des traits admissibles

Cela permet :

1. de contrôler les occurrences des traits (noms et contextes)
2. d'obtenir des représentations efficaces

Notation DyALog

Déclaration des types et traits admissibles :

```
:-features([nc,gn],[number,gender]).
:-features(v,[number,gender,person,tense,mood,voice,aux]).
:-features(gv,[number,gender,person, mood]).
```

Représentation :

```
s --> gn{ number => Num },
      gv{ number => Num, person => 3, mood => finite[]}.
gn{} --> GN::gn{}, rel(GN).
```


Représentation interne

La déclaration des traits permet :

- d'associer un rang à chaque trait pour un type donnée
- de représenter une FS fermée par un terme Prolog standard
- dans DyALog, la transformation est effectuée au moment de la lecture des FS, avec contrôle des traits :

```
DyALog> :-features(nc,[number,gender]).
DyALog> ?-X=nc{ num => sg }.
Line 2 of user_input:
        Syntax Error : feature num not allowed within feature
    structure nc{}
```

```
DyALog> ?-X=nc{ number => sg }, X =.. T.
        X = nc{number=> sg, gender=> B__5}
        T = [nc!$ft,sg,B__5]
```

La FS X=nc{ number => sg } représentée par nc!ft(sg,_)

- les valeurs non spécifiées sont remplis par des variables
- nc!ft est spécifique DyALog : foncteur ft dans le domaine de noms nc

Unification

Directement fournie par l'unification standard Prolog !

```
DyALog> ?-X=nc{ number => sg }, X = nc{ gender => masc }.
%% equiv: nc!ft(sg,_) = nc!ft(_,masc) = nc!ft(sg,masc)
        X = nc{number=> sg, gender=> masc}
```


pro et cons

Les structures de traits fermées sont

- \- moins flexibles que les FS ouvertes
- \+ plus sûres d'un point de vue contrôle
- \+ efficaces en utilisation,
- \+ pas de surcouche dans Prolog, (sauf dans le lecteur de termes)

Néanmoins, le contrôle n'est pas parfait !

- on peut se tromper dans la valeur d'un trait

```
?-X = v{ tense => indicative, number => plural }
%% au lieu de: v{ mood => ind, number => pl }
```

⇒ on souhaite pouvoir préciser le domaine de valeurs d'un trait, dans le contexte d'un type.

Hiérarchie de types

Les structures de traits typées [TFS – *Typed Feature Structures*] s'appuient sur une hiérarchie de types :

- un type τ peut avoir des sous-types $\tau_1, \ldots, \tau_n$
- un type τ peut être sous-type de plusieurs types parents (héritage multiple)
- un type τ peut introduire un trait f, en spécifiant le type $\rho_{f,\tau}$ le plus général approprié pour f (sous τ)
- un trait f est approprié pour un type τ si il est introduit par τ ou par un type ancêtre de τ

Formalisation développée par Bob Carpenter

- Implantation dans ALE en surcouche Prolog, Gerald Penn
  `http://www.cs.toronto.edu/~gpenn/ale.html`
- autre implantation dans LKB, Ann Copestake
- TFS surtout utilisées pour les HSPG
- Implantation dans DYALOG

Exemple

Fragment d'une hiérarchie pour une grammaire de génération (*Semantic-Head-Driven Generation*, Shieber et al, dans ALE)

```
bot sub [pred, list, sem, form, agr, sign].
 form sub [finite, nonfinite].
    finite sub [].
    nonfinite sub [].
 sign sub [sentence, verbal, np, adv, p]
       intro [sem:sem].
    sentence sub [].
    verbal sub [s, vp] intro [form:form].
      s sub [].
      vp sub [] intro [subcat:subcat_list].
    np sub [det, n]
       intro [agr:agr, arg:sem].
      det sub [] intro [np_sem:sem].
      n sub [].
    adv sub [] intro [varg:sem].
    p sub [].
 ...
```


Notation AVM pour les TFS

```
bot sub [pred, list, sem, form, agr, sign].
 sign sub [sentence, verbal, np, adv, p]
       intro [sem:sem].
    sentence sub [].
    verbal sub [s, vp] intro [form:form].
      s sub [].
      vp sub [] intro [subcat:subcat_list].
    np sub [det, n]
       intro [agr:agr, arg:sem].
      det sub [] intro [np_sem:sem].
      n sub [].
    adv sub [] intro [varg:sem].
    p sub [].
 ...
```

$$\text{np}\begin{bmatrix}\text{agr} & \text{sg1} \\ \text{arg} & \text{sem}\begin{bmatrix}\ldots\end{bmatrix} \\ \text{sem} & \text{sem}\begin{bmatrix}\ldots\end{bmatrix}\end{bmatrix}$$

Constantes et types

Version puriste des TFS :
les constantes sont aussi des types
maximaux (sans sous-types) et sans traits

```
bot sub [pred, list, sem, form, agr, sign].
  agr sub [sg1, sg2, sg3, pl1, pl2, pl3].
    sg1 sub []. sg2 sub []. sg3 sub [].
    pl1 sub []. pl2 sub []. pl3 sub [].
  pred sub [decl, imp, love, call_up, leave, see, ...].
    decl sub []. imp sub [].
    leave sub []. love sub []. call_up sub []. see sub [].
    ...
  ...
```

- déclarations assez lourdes, pour des ensembles larges de constantes
- quid des ensembles ouverts, comme les entiers ou les chaînes ?

Échappements

Dans DyALog, version non puriste des constantes
les constantes gérables par échappements pour revenir vers les types *builtins* Prolog :

```
bot sub [humain,name,age].
   humain intro [name: name, age: age].
   name escape symbol.
   age escape integer.
```

Types d'échappements : char, integer, symbol, et compound (tout terme)

Ne permet pas (encore) d'associer à un type un domaine fini de valeur

```
%% Dans un monde idéal
bot sub [v,mood].
   v intro [mood: mood, ...]
   mood escape finite_set(mood)
```


Termes squelettes

À chaque type τ peut être associé la TFS la plus générale possible compatible avec les contraintes de la hiérarchie.
$\Rightarrow$ notion de terme squelette $\mathrm{skel}(\tau)$

Plus précisément, si $f_1, \ldots f_n$ sont appropriés pour τ
avec les types $\rho_{f_1,\tau}, \ldots, \rho_{f_n,\tau}$, alors

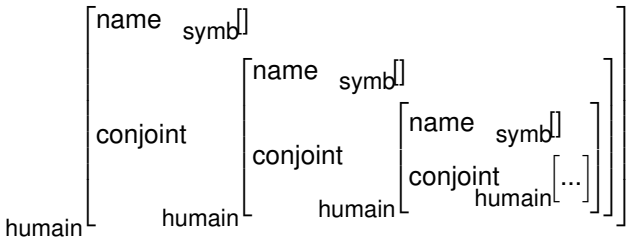
$$\mathrm{skel}(\tau) = {}_{\tau}\begin{bmatrix} f_1 & \mathrm{skel}(\rho_{f_1,\tau}) \\ \ldots & \ldots \\ f_n & \mathrm{skel}(\rho_{f_n,\tau}) \end{bmatrix}$$

Type récursifs

Un type ne peut être directement récursif au travers d'un de ses traits,
ie $\rho_{f,\tau} \neq \tau$

```
bot sub [humain, symbol]].
  humain intro [name: symbol, conjoint: humain].
  symbol escape symbol.
```

Motivation : conduit à un terme squelette infini

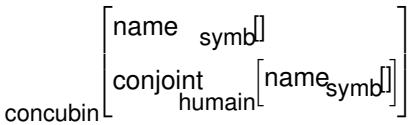


Dé-récursifier les types

Principe : introduire des types intermédiaires sous-spécifiés

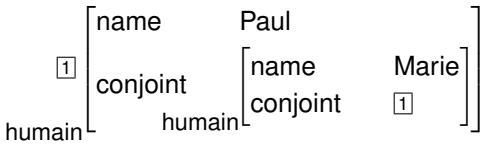
```
bot sub [humain, symbol]].
  humain sub concubin intro [name:symbol].
    concubin sub [] intro [conjoint: humain].
  symbol escape symbol.
```

Terme squelette pour concubin



Termes récursifs

Une TFS peut être récursive et même cyclique sans que les types soient récursifs

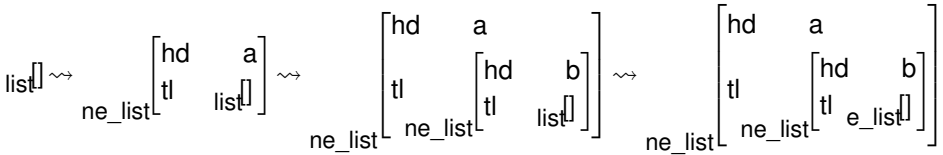


Note : la plupart des systèmes TFS refusent les termes cycliques, ils sont acceptés par DyALog

Listes

On peut définir la notion de liste avec une hiérarchie de type, en utilisant un type dé-récursifié list

```
bot sub [list,symbol,...].
  list sub [e_list, ne_list].
    e_list sub [].
    ne_list sub []
            intro [hd:bot, tl:list].
  ...
```



Listes spécialisées

Souvent, on souhaite pouvoir spécifier une liste d'éléments d'un certain type :

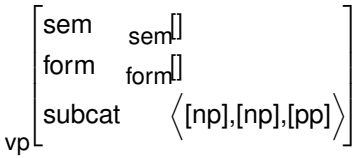
- non directement possible d'utiliser des listes paramétriques X list
- mais possible en sous-typant

```
list sub [e_list, ne_list, arg_list, subcat_list].
  e_list sub [].
  ne_list sub [arg_ne_list, subcat_ne_list]
          intro [hd:bot, tl:list].
  arg_list sub [e_list, arg_ne_list].
    arg_ne_list sub [] intro [hd:sem, tl:arg_list].
  subcat_list sub [e_list, subcat_ne_list].
    subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```


Listes et AVM

Les listes sont une notion suffisamment importante pour bénéficier d'une notation spéciale dans AVM :

```
verbal sub [s, vp] intro [form:form].
  s sub [].
  vp sub [] intro [subcat:subcat_list].
```



Contraintes sur les hiérarchies (suite 1)

- un type τ ne doit pas avoir de trait approprié f tel que $\rho_{f,\tau} = \tau$
  (pas de type récursif)

Contraintes sur les hiérarchies (suite 2)

- un type τ_1 peut ré-introduire un trait f avec un type ρ_{f,τ_1}
  (déjà introduit par un ancêtre avec le type $\rho_{f,\tau}$)
  mais alors ρ_{f,τ_1} doit être un sous-type de $\rho_{f,\tau}$
  (instanciation des types appropriés)

```
ne_list sub [arg_ne_list, subcat_ne_list]
       intro [hd:bot, tl:list].
subcat_list sub [e_list, subcat_ne_list].
  subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```


Contraintes sur les hiérarchies (suite 2)

- si τ_1 et τ_2 introduisent le trait f,
  alors un type ancêtre τ de τ_1 et τ_2 doit introduire f
  $\Rightarrow$ existence d'un type plus général σ_f introducteur de f Cas de hd et tl

```
list sub [e_list, ne_list, arg_list, subcat_list].
  e_list sub [].
  ne_list sub [arg_ne_list, subcat_ne_list]
         intro [hd:bot, tl:list].
  arg_list sub [e_list, arg_ne_list].
    arg_ne_list sub [] intro [hd:sem, tl:arg_list].
  subcat_list sub [e_list, subcat_ne_list].
    subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```


Unification

L'unification entre TFS est :

- similaire à l'unification entre FS fermée
- mais doit prendre en compte l'instantiation vers des sous-types
- $\Rightarrow$ étend strictement l'unification Prolog

Algorithme :

```
tfs_unif(t1 type τ1, t2 type τ2) retourne TFS
   τ3 := mgst(τ1,τ2) ou fail
   t3 := [{τ3}]
   pour tout f dans t1 avec v1 et t2 avec v2,
        t3 += f: v1 ⊓ v2 ⊓ skel(ρf,τ3)
   pour tout f dans t1/t2 avec v1
        t3 += f: v1 ⊓ skel(ρf,τ3)
   pour tout f dans t2/t1 avec v2
        t3 += f: v2 ⊓ skel(ρf,τ3)
   pour tout f approprié pour τ3, non dans t1 ou t2
        t3 += f: skel(ρf,τ3)
   retourne t3
```

Unification : implantation

L'unification entre TFS étend strictement l'unification entre termes Prolog

On peut implanter l'algorithme d'unification de manière générique en surchouche de Prolog

Mais l'information de la hiérarchie de type permet en fait de définir hors-ligne une opération d'unification spécialisée par paire de type τ_1 et τ_2.

Compilation : tfs2lib

DYALOG compile les hiérarchies de types en bibiothèque C dynamique en utilisant TFS2LIB

Hiér. .def $\xrightarrow{\text{tfs}}$ Desc .def $\xrightarrow{\text{Perl}}$ .c $\xrightarrow{\text{gcc}}$ .so

Exploitation :

```
dyacc -tfs list tfs_append.pl -o tfs_append
```


Description expansée

TFS produit une description expansée d'une hiérarchie de types

```
TYPES = p adv n det np vp s verbal ...  bot
SUBTYPES np = det n
SUBTYPES verbal = s vp
SUBTYPES sign = s vp det n sentence verbal np adv p
SUBTYPES agr = sg1 sg2 sg3 pl1 pl2 pl3
SUBTYPES form = finite nonfinite
SUBTYPES list = arg_ne_list subcat_ne_list e_list ne_list
    arg_list subcat_list
...
FEATURES = varg np_sem arg agr subcat form sem args pred tl hd
INTRO varg = adv
INTRO np_sem = det
INTRO tl = ne_list
INTRO hd = ne_list
..
```


Description expansée (skel)

```
SKEL p = sem:sem
SKEL adv = varg:sem sem:sem
SKEL n = arg:sem agr:agr sem:sem
SKEL det = np_sem:sem arg:sem agr:agr sem:sem
SKEL np = arg:sem agr:agr sem:sem
SKEL vp = subcat:subcat_list form:form sem:sem
SKEL sem = args:arg_list pred:pred
SKEL subcat_ne_list = tl:subcat_list hd:sign
SKEL subcat_list =
SKEL arg_ne_list = tl:arg_list hd:sem
SKEL arg_list =
SKEL ne_list = tl:list hd:bot
...
```

Description expansée (unif)

```
UNIF bot list = list
UNIF bot e_list = e_list
UNIF sign vp = vp
+       UNIF sem LEFT 1 RIGHT 3 DIRECT 3
+       INHERIT form RIGHT 2 DIRECT 2
+       INHERIT subcat RIGHT 1 DIRECT 1
...
```

Note :

- les fonctions exploitent le fait qu'un rang peut être associé à chaque trait pour un type donné.
- des fonctions de subsumption sont aussi engendrées

Subsomption

Similaire à la subsumption pour les FS fermées mais en

Représentation interne (DyALog)

- Même représentation que pour les FS fermées, pour les types maximaux
- Passage en deref-termes avec un argument supplémentaire pour les termes non maximaux
  permet l'instantiation de types

Notation en chemin

Les informations de la hiérarchie permettent des inférences de termes à partir de chemins de traits.

```
append(e_list{}, Y, Y).

append( A::tl=>X,Y::list{}, B::tl=>Z) :-
        A .> hd .= B.> hd,
        append(X,Y,Z).
```

Extensions

Valeurs ensemblistes, multiset ; négation

Sixième partie VI

Prolog comme analyseur

Quelques problèmes avec Prolog

Divers problèmes bien connus, dus à la gestion en profondeur d'abord avec retour-arrière de Prolog :

- terminaison sur des productions récursives
- non-complétude des réponses
- duplication importante de calculs

Terminaison

```
s --> gn, v, gn.
gn --> np.
gn --> gn, gp.
```

```
s L=[Paul,aime,Marie]
gn,v,gn L=[Paul,aime,Marie]
   gn L=[Paul,aime,Marie]
      np L=[Paul,aime,Marie]
         [Paul] L=[Paul,aime,Marie]
         v,gn L=[aime,Marie]
            v L=[aime,Marie]
               [aime] L=[aime,Marie]
                  gn L=[Marie]
                     np L=[Marie]
                        [Paul] L=[Marie]     %fail
                        [Marie] L=[Marie]
                        [] L=[]
Answer:
    L = [Paul,aime,Marie]
```

Terminaison (suite)

```
gn,gp L=[Marie]
   gn L=[Marie]
      np L=[Marie]
         [Paul] L=[Marie]      % fail
         [Marie] L=[Marie]
            gp L=[]            % fail
      gn,gp L=[Marie]
         gn L=[Marie]
            np L=[Marie]
               [Paul] L=[Marie]  % fail
               [Marie] L=[Marie]
               gp L=[]           % fail
             gn,gp L=[Marie]
             ...
```

⇒ boucle de prédiction sur gn −−> gn,gp.

Complétude

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

```
s L=[Paul,aime,Marie]
  gn,v,gn L=[Paul,aime,Marie]
     gn L=[Paul,aime,Marie]
        gn,gp L=[Paul,aime,Marie]
           gn L=[Paul,aime,Marie]
              gn,gp L=[Paul,aime,Marie]
                 gn L=[Paul,aime,Marie]
                    gn,gp L=[Paul,aime,Marie]
                       ...
```

⇒ boucle, sans aucune réponse

Importance de l'ordre des productions :
Ennuyeux pour un formalisme déclaratif !
(cause : ordre non équitable de sélection des productions)

Boucles

Boucles et récursions en présence de récursions// en particulier sur les récursions gauches

$\text{NP} \leftarrow \text{NP Prep NP}$ (a) directe

$A \leftarrow B\ c \quad B \leftarrow A$ (b) indirecte

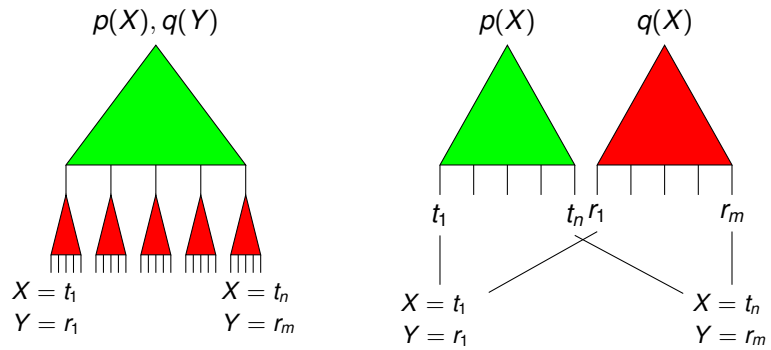
$A \leftarrow B\ A\ c \quad B \leftarrow \epsilon | b$ (c) cachée

Remèdes :

1. Transformation de programmes,
   - pas toujours possible ou facile
   - éventuellement incompatible avec des constructions sémantiques
   - pertube les arbres d'analyse attendus
2. Compare l'appel courant avec les appels en cours
   ⇒ mémo-fonction et tabulation
3. changer de stratégie d'analyse et de contrôle

Duplication de calculs

La gestion de non-déterminisme à la Prolog (retour-arrière) ⇒ beaucoup de re-calculs :



Sans partage, les GN sont recalculés 5 fois

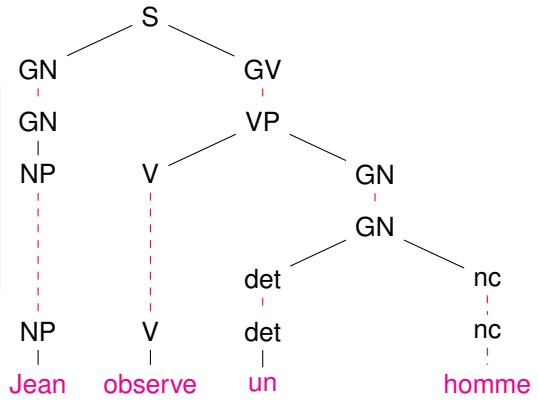
Jean observe un homme sur sur la colline avec un télescope.

Le nombre de re-calculs croit exponentiellement avec le nombre de points d'ambiguïtés.

Construire une analyse

Par collage d'arbres (partiels) d'analyse :

```
s  —> gn gv
gn —> np
gn —> det nc
gn —> gn gp
gv —> v np
gv —> gv gp
gp —> prep gn
```



Stratégies d'analyse

Une stratégie d'analyse décrit quels pas de calculs sont autorisés pendant l'analyse :

- stratégies descendantes (*top-down*), guidées par les buts, en partant de l'axiome
- stratégies ascendantes (*bottom-up*)
  guidées par les réponses, en partant des terminaux
- stratégies hybrides (comme la stratégie Earley)
- stratégies dirigées par des tables (Left Corner, Head Corner, LR, . . .)

Stratégies de contrôle

Une stratégie de contrôle spécifie la gestion du non-déterminisme, en particulier en terme d'ordonnancement :

Ambiguïtés Que faire des ambiguïtés

- désambiguïser (probabilités, heuristiques, regard en avant)
  désambiguïsation totale ou partielle
- exploration par retour arrière,
- exploration par tabulation, . . .

Ordonnancement Dans quel ordre examiner les alternatives ?

- en profondeur d'abord (dernière alternative examinée en premier)
- en largeur d'abord
- en fonction de probabilités ou heuristiques
- synchronisation lecture (gauche-droite) de la chaîne
- parallèle, concurrent, . . .

Septième partie VII

Implanter des stratégies d'analyse

Analyseur descendant

DCG Prolog $\equiv$ { analyse : descendante gauche-droite
contrôle : profondeur d'abord avec retour-arrière

```
phrase(A,L,R) :- td_parse(A,L,R).
td_parse(A,L,R) :-
        recorded((A --> G)),
        td_parse(G,L,R).
td_parse([],L,L).
td_parse([T|Rest],L,R) :-
        'C'(L,T,M),
         td_parse(Rest,M,R).
```

Facile à mettre en oeuvre ($\equiv$ descente récursive gauche [LL])
mais boucles, incomplétude et duplication de calculs.

En profondeur incrémentale

Même stratégie descendante, mais contrôle = profondeur d'abord mais bornée
ie :

- en profondeur d'abord, jusqu'à une certaine profondeur k
- en relançant avec une profondeur $k+1$

```
tdi_parse_iterate(A,L,R,Depth) :-
( tdi_parse(A,L,R,Depth)
; NewDepth is Depth+1,
  tdi_parse_iterate(A,L,R,NewDepth)
).
tdi_parse(A,L,R,Depth) :-
    Depth > 0, NewDepth is Depth-1,
    recorded((A --> G)),
    tdi_parse(G,L,R,NewDepth).
```

Permet d'éviter les incomplétudes dans les réponses, mais

- toujours des problèmes de terminaison
- encore plus de duplication de calculs

Analyseur ascendant : moteur

L'analyse part des terminaux pour remonter vers l'axiome
mise en oeuvre : réduction progressive de corps de production dans la chaîne

```
phrase(A,L,R) :-
     bu_parse(L,[A|R]).
bu_parse(L,L).
bu_parse(L,R) :-
       %% L = Front.Body.Tail
        split(L,Front,Body,Tail),
       %% trouve une prod Head --> Body
        dcg_match(Body,Head),
        split(M,Front,[Head],Tail),
       %% M = Front.[Head].Tail
        bu_parse(M,R).
```

```
:-std_prolog split/4.
split(L,A,B,C) :-
    append(A,BC,L),
    append(B,C,BC).
```


Analyseur ascendant : trace

```
bu [Paul,aime,Marie]
bu [np,aime,Marie]
bu [gn,aime,Marie]
bu [gn,v,Marie]
bu [gn,v,np]
bu [gn,v,gn]
bu [s]
Answer:
    L = [Paul,aime,Marie]
...
```

L'analyse termine, car aucune boucle de prédiction, mais ...

Analyseur ascendant : efficacité

Extrêmement inefficace !

- les constituants sont reconstruits dans tous les ordres possibles
- des constituants intermédiaires inutiles sont créés

```
Answer:
    L = [Paul,aime,Marie]
bu [gn,aime,np]
bu [gn,v,np]
bu [gn,v,gn]
bu [s]
Answer:
    L = [Paul,aime,Marie]
bu [gn,aime,gn]
bu [gn,v,gn]
bu [s]
Answer:
    L = [Paul,aime,Marie]
...
```


Analyseur shift-reduce : principe

Approche ascendante légèrement plus efficace :

- les terminaux sont lus de gauche à droite et empilés (shift) sur une pile
- réduction (reduce) quand le corps d'une production est présent sur la pile la tête de la production est empilée à la place

Peut être rendue plus efficace en pré-calculant des tables des `shift` et `reduce` autorisés dans un certain état :
⇒ stratégies LR et (non déterministe tabulaire) GLR (Tomita)

Analyseur shift-reduce : moteur

```
%% sr_parse(StackIn,StackOut,StringIn,StringOut)
sr_parse(SI,SI,L,L).

sr_parse(SI,SO,[A|M],R) :-
    sr_parse([A|SI],SO,M,R).

sr_parse(SI,SO,L,R) :-
    %% match en reverse le haut de la pile SI=[RevBody|SRed]
    %% avec le corps d'une clause Head --> Body
    sr_dcg_match(SI,Head,SRed),
    sr_parse([Head|SRed],SO,L,R).
```


Shift-reduce : Trace

```
shift Paul => [Paul]
shift aime => [aime,Paul]
shift Marie => [Marie,aime,Paul]
reduce to np => [np,aime,Paul]
reduce to gn => [gn,aime,Paul]%f
reduce to v => [v,Paul]
shift Marie => [Marie,v,Paul]
reduce to np => [np,v,Paul]
reduce to gn => [gn,v,Paul]%f
reduce to np => [np]
shift aime => [aime,np]
shift Marie => [Marie,aime,np]
reduce to np => [np,aime,np]
reduce to gn => [gn,aime,np]%f
reduce to v => [v,np]
shift Marie => [Marie,v,np]
reduce to np => [np,v,np]
reduce to gn => [gn,v,np]%
reduce to gn => [gn]
shift aime => [aime,gn]
shift Marie => [Marie,aime,gn]
reduce to np => [np,aime,gn]
reduce to gn => [gn,aime,gn]%f
reduce to v => [v,gn]
shift Marie => [Marie,v,gn]
reduce to np => [np,v,gn]
reduce to gn => [gn,v,gn]
reduce to s => [s]
Answer:
    L = [Paul,aime,Marie]
```

Plus efficace et terminaison, mais

- gestion explicite d'une pile
- toujours des re-calculs

Relation coin-gauche

Analyseur coin-gauche

Huitième partie VIII

Analyseurs tabulaires

Tabuler ?

Principe : garder des traces de calculs dans une table, pour

- détecter des boucles, afin d'améliorer la terminaison
- partager des sous-calculs
- déployer des stratégies d'ordonnancement plus flexibles
- extraire des traces des sous-calculs réussis (preuves, arbres d'analyse)
- agréger une valeur sur un ensemble de résultats (moyenne, somme, plus court chemin, . . .)

Tabulation et CFG

Une longue histoire avec de nombreux algorithmes :

- CKY [Cocke-Kasami-Younger]
- Algorithme d'Earley – Analyseurs à Charte [Kay]
- Generalized LR [Tomita]
- Automates à piles / programmation dynamique [Lang]

Mais essentiel de clairement distinguer

- Stratégie d'analyse
- Stratégie de contrôle (niveau tabulation)

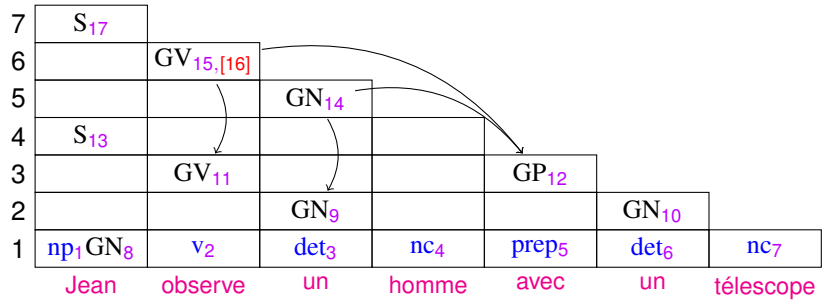
Tabulation passive

- La table ne sert qu'à enregistrer des calculs terminés.
- La table ne sert pas à diriger les calculs

Cocke-Kasami-Younger [CKY]

Algorithme en Programmation Dynamique (1965)
Analyse ascendante avec tabulation des constituants

> *Si il existe une production $A_0 \leftarrow A_1 \dots A_n$ avec, pour tout $i > 0$, A_i présent dans (x_i, l_i) et $x_{i+1} = x_i + l_i$, alors tabuler le non terminal A_0 dans l'entrée $(x_1, \Sigma_i l_i)$ (sauf si déjà tabulé).*



Les constituants généralement construits par longueur croissante,
de la gauche vers la droite
mais en fait non obligatoire !

Complexité

```
table_initialize
for all positions x and lengths l
  for all productions A0 ← A1 ... Av
    for all lengths l1, ..., lv−1 with Σk=1..v−1 lk < l
          lv = l − Σk=1..v−1 lk
          xj = x + l1 + ··· + lj−1
          if Aj ∈ T[xj, lj] for all j > 1
          then add A0 in T[x, l]   (unless present)
```

Complexité temps pire des cas fournie par les itérations enchâssées sur x, l et l_j $(1 \leq j < v)$ bornées par la longueur de la chaîne n.
$\Rightarrow O(n^{v+1})$ où v est la longueur de la plus longue production

Pour un reconnaisseur, complexité espace pire des cas fournie par le # de cellules dans la table and le # de constituants par cellule
$\Rightarrow O(n^2)$

Forme normale de Chomsky (binarisation)

Complexité en $O(n^{v+1})$ réduite à $O(n^3)$ en mettant sous forme normale de Chomsky (**binarization**).

Règle ternaire VP −−> V, NP, NP donne une complexité $O(n^4)$ mais peut être remplacée par les règles binaires

```
VP —> V, VP_ARGS.
VP_ARGS —> NP, NP.
```

Mais implique une transformation de la grammaire
plus élégant de manipuler des règles pointées

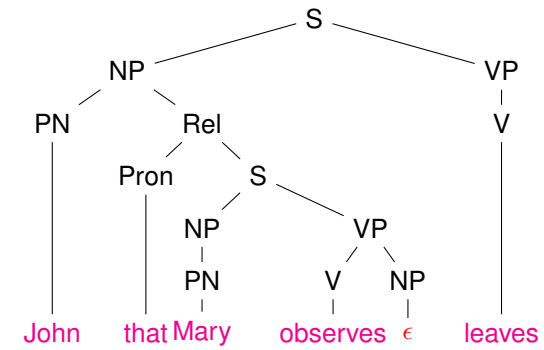
Pire des cas temps en $O(n^3)$ et espace en $O(n^2)$ (quasi) optimal pour les CFG mais CKY non efficace (défauts des stratégies purement ascendantes)

Limitations de CKY

Constituants inutiles

John who looks [$_S$ Marie leaves]

Traces



Analyses avec Charte

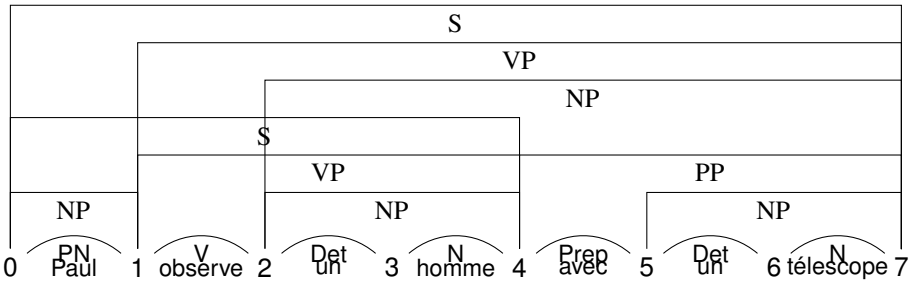
Historiquement, approches motivées par le souhait :

- d'utiliser de la tabulation (pour le partage de calculs)
- de préserver la complexité optimale $O(n^2)$ pour les CFGs
- d'introduire de la prédiction (descendante)

⇝ développement de techniques génériques fondées sur des chartes
une charte *equiv* ensemble d'arcs (items) entre 2 positions de la chaîne, avec divers types de labels

CKY reformulé en charte passive

Les entrées de la table CKY visuellement représentés par des arcs et stockés comme items $\langle i, j, Cat \rangle$.



Complexité temps en $O(n^{v+1})$

Interpréteur Prolog

Règles pointées

Une charte active stocke non seulement des constituants incomplets mais aussi des constituants partiels, pouvant servir à guider l'analyse.

Utilisation de

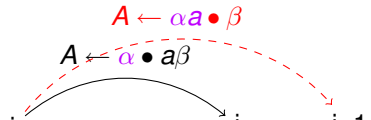
- règles pointées [*dotted rules*] $A_0 \leftarrow A_1 \ldots A_i \bullet A_{i+1} \ldots A_n$
- arcs étiquetés par des règles pointées (items $\equiv \langle i, j, A \leftarrow \alpha \bullet \beta \rangle$)
- un système déductif spécifie comment dériver les items

Un système déductif pour CKY

$$\frac{}{\langle i, i, A \leftarrow \bullet \alpha \rangle} \quad \exists A \leftarrow \alpha$$

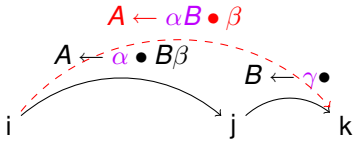
(Seed)

$$\frac{\langle i, j, A \leftarrow \alpha \bullet a\beta \rangle}{\langle i, j+1, A \leftarrow \alpha a \bullet \beta \rangle} \quad a = a_{j+1}$$



(Scan)

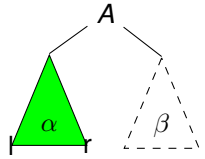
$$\frac{\langle i, j, A \leftarrow \alpha \bullet B\beta \rangle \quad \langle j, k, B \leftarrow \gamma \bullet \rangle}{\langle i, k, A \leftarrow \alpha B \bullet \beta \rangle}$$



(Complete)

Invariant

Chaque item $\langle l, r, A \leftarrow \alpha \bullet \beta \rangle$ vérifie l'invariant : $\alpha \rightarrow^\star a_{l+1} \ldots a_r$

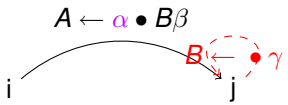


L'utilisation des règles pointées induit une binarisation implicite
$\Rightarrow$ complexité temps $O(n^3)$

Algorithme d'Earley

Possibilité d'ajouter une règle de prédiction (descendante)
⇒ Algorithme d'Earley [1970]

$$\frac{\langle i, j, A \leftarrow \alpha \bullet B\beta \rangle}{\langle j, j, B \leftarrow \bullet \gamma \rangle} \quad \exists B \leftarrow \gamma$$



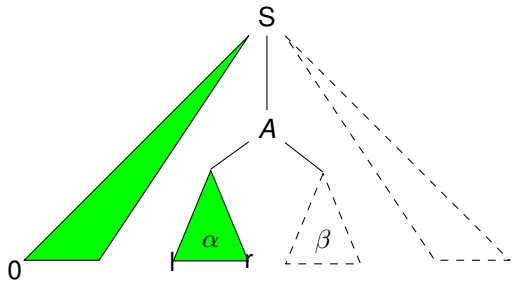
(Pred)

+ rules (Scan) and (Complete)

Invariant et complexité

Chaque item $\langle l, r, A \leftarrow \alpha \bullet \beta \rangle$ satisfait 2 invariants :

1. Reconnaissance de α entre l et r (comme pour CKY)
2. validité des préfixes : $\exists \gamma \in (T \cup N)^*,\ S \rightarrow^* a_1 \ldots a_l A \gamma$



La complexité pire des cas en temps demeure $O(n^3)$
Mais, en pratique, la prédiction réduit l'espace de recherche et réduit la complexité

Charte : mise en oeuvre

Un algorithme à charte repose sur :

- une table (i.e. charte) où sont stockés les items, sans duplication.
- un agenda où sont stockés les items à traiter

Un cycle de l'algorithme implique

1. Sélectionner un item *I* dans l'agenda
2. Si *I* n'est pas déjà tabulé, l'ajouter ; sinon retour étape 1
3. Construire de nouveaux items en combinant *I* avec les items déjà tabulés
4. Insérer les nouveaux items dans l'agenda

Variante : Les items sont d'abord tabulés avant insertion dans l'agenda
Ordre de sélection (Earley) : $\langle i, j, A \rangle$ choisi avant $\langle k, l, B \rangle$ si $j < l$:
⇒ synchronisation gauche-droite de la lecture

Pour les CFG, l'ordre de sélection importe peu (univers fini) :
⇒ l'algorithme termine et est complet

Formulation Prolog

Un item (A −−> Body)(I,J), en notation Hilog
Variante de Earley : règle pointée $A \leftarrow \alpha \bullet \beta$ représentée par $A \leftarrow \beta$

```
earley_parse(A,L,R) :-
  predict(A(L,L),[],Agenda),
  process(Agenda),
  stored((A --> true)(L,R)).
```

```
process([]).
process([Item|OldAgenda]) :-
  process_item(Item,OldAgenda,Agenda),
  process(Agenda).
```

- process traite le 1er item de l'agenda
- stored test si l'item est dans la table

Formulation Prolog (process)

```
process_item((B-->true)(J,K),OldAgenda,Agenda) :-
    resolve_passive(B(J,K),OldAgenda,Agenda).

process_item(Item::(A-->B,Beta)(I,J),OldAgenda,Agenda) :-
    predict(B(J,J),OldAgenda,MiddleAgenda),
    resolve_active(Item,MiddleAgenda,Agenda).
```


Formulation Prolog (predict)

Prédit de nouvelles production à réduire.

```
predict(G::A(I,I),Agenda1,Agenda2) :-
  all_solutions(NewItem,
                G^prediction(G,NewItem),
                Agenda1,
                Agenda2).
prediction(A(I,I),NewItem) :-
   recorded(Prod::(A-->Body)),
   store(NewItem::Prod(I,I)).
```

- all_solutions : collecte tout les items résultant de la prédiction et les ajoute à l'agenda
  (Prolog : findall , bagof ou setof, DyALog : iterate , . . .)
- store ajoute l'item à la table ; échoue si déjà présent

Formulation Prolog (passive)

Utilise un fait pour réduire des règles pointées

```
resolve_passive(Fact::B(J,K),Agenda1,Agenda2) :-
  all_solutions(NewItem,
                Fact^passive(Fact,NewItem),
                Agenda1,
                Agenda2).

passive(B(J,K),NewItem) :-
    stored((A -->B,Beta)(I,J)
    store(NewItem::(A->Beta)(I,K)).
```


Formulation Prolog (active)

À partir d'une règle pointée, cherche un fait tabulé pour avancer

```
resolve_active(Item::(A-->B,Beta)(I,J),Agenda1,Agenda2) :-
  all_solutions(NewItem,
                Item^active(Item,NewItem),
                Agenda1,
                Agenda2).

active((A-->B,Beta)(I,J),NewItem) :-
    stored((B -> true)(J,K))
    store(Item::(A->Beta)(I,K)).
```

findall (Prolog)

```
habite(jean, belfort).
habite(lucie, paris).
habite(christian, toulouse).
habite(adeline, paris).
habite(nicolas, paris).

?- findall(X, habite(X,paris), R).
R = [lucie,adeline,nicolas] ?
```


Schéma d'analyse

Description of parsing strategies in terms (of classes) of partial parse trees

> *[Sikkel]* *"These intermediate results are not necessarily partial trees, but they must be objects that denote relevant properties of those partial parses."*

A schema indicates

- the domain of items (and their form)
- the item invariants

Very close from chart algorithms

Stratégie par coin-gauche

Stratégie dirigée par les têtes

Construire un arbre d'analyse

Earley revisité

Test de redondance

Boucles : variance vs sumsomption

Subsomption faible et forte

Terminaison et spirales

Grammaires finiment ambiguës

Boucles de prédiction

Couper les boucles de prédiction

Restrictions

Implantation des restrictions en Prolog

Restrictions en DyALog

Ordonnancement et complétude

Neuvième partie IX

Efficient tabular parsing strategies

Efficient strategies

Tabulation and parsing strategies are two largely orthogonal issues.

But some strategies are more efficient than others when using tabulation

These strategies may differ from those used for deterministic evaluations.

Useless argument

Building and propagating parse trees reduce computation sharing

$$\mathrm{VP} \leftarrow \mathrm{VP}\ \mathrm{PP} \rightsquigarrow \mathrm{VP}(vp(VP, PP)) \leftarrow \mathrm{VP}(VP)\ \mathrm{PP}(PP)$$

$$\langle i, j, \mathrm{VP}(t)\rangle + \begin{cases} \langle j, k, \mathrm{PP}(t_1)\rangle & \rightarrow \quad \langle i, k, \mathrm{VP}(vp(t, t_1))\rangle \\ \langle j, k, \mathrm{PP}(t_2)\rangle & \rightarrow \quad \langle i, k, \mathrm{VP}(vp(t, t_2))\rangle \end{cases}$$

$\Rightarrow$ duplication of computations and no search space pruning

More judicious to extract a tree (or forest) from tabulated traces

Similar problem for semantic forms

Example of modulation : hanoi

Idée : oublier pendant les prédiction les arguments non-prédictifs

```
% hanoi with accumulator list
hanoi([],_,_,_,M,M).
hanoi([X|Y],L,C,R,M1,M2) :-
        hanoi(Y,C,L,R,M3,M2),
        hanoi(Y,L,R,C,M1,[m(L,R)|
    M3]).
```

Pour partager sur les plots & accumulateur :

	Call	Return
hanoi(stack, l, c, r, moves, acc)	call_hanoi_6(stack)	ret(*l*, *c*, *r*, moves, acc)

$$\left.\begin{array}{l}\text{hanoi}(Y, C, L, R, M3, M2)\\ \text{hanoi}(Y, L, R, C, M1, [m(L, R)|M3])\end{array}\right\} \rightsquigarrow \text{call\_hanoi\_6}(Y) \quad \Rightarrow \quad \text{Sharing}$$

En DyALog : :−mode(hanoi/6,+(+,−,−,−,−,−).

Dixième partie X

Forêts partagées d'analyse

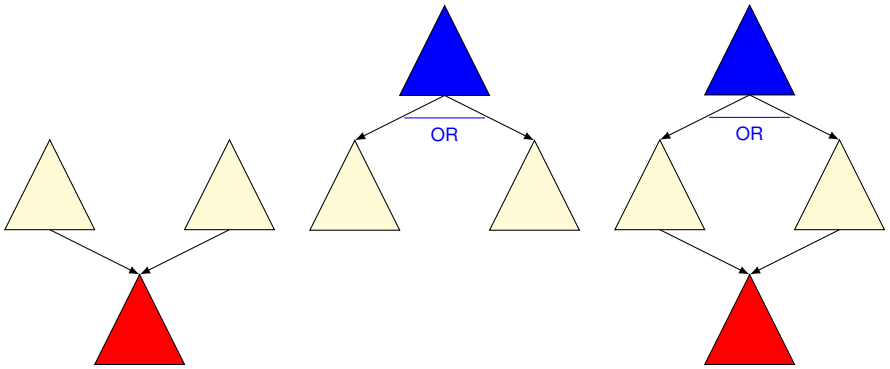
Forêts partagées

Ambiguïtés du langage ⇒
Plusieurs analyses possibles par phrases !

Forêt ≡ ensemble d'arbres d'analyse

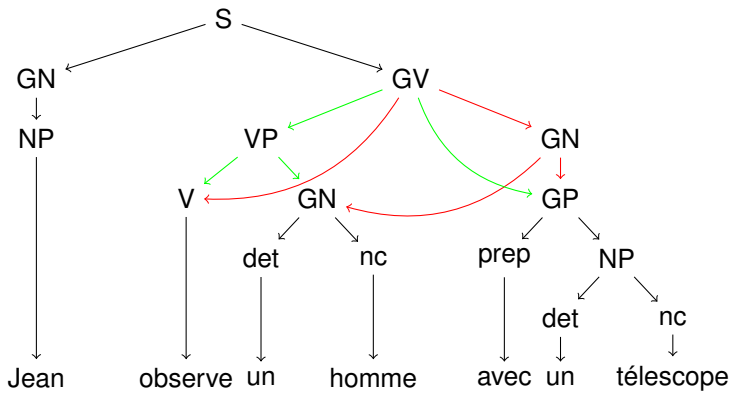
Forêts partagées (ou *packed*) ≡ Représentation compacte représentant des sous-arbres identiques ou similaires.

Graphes ET-OU



(d) Partage de sous-arbres (e) Partage de contexte (f) Partage complet

Forêts partagées et arbres



Forêts partagées et grammaires

Une forêt est une grammaire G' instance de G [Lang].

0 Jean 1 observe 2 un 3 homme 4 avec 5 un 6 télescope 7

```
s  --> np vp
np --> pn
np --> det n
np --> np pp
vp --> v np
vp --> vp pp
pp --> prep np
```

```
s07  --> np01 vp17
np01 --> pn01
vp17 --> v12 np27
vp17 --> vp14 pp47
np27 --> np24 pp47
n37  --> n34 pp47
np24 --> det23 n34
pp47 --> prep45 np57
np57 --> det56 n67
vp14 --> v12 np24
```

```
pn01 --> Jean
 v12 --> observe
det23 --> un
 n34 --> homme
prep45 --> avec
det56 --> un
 n67 --> télescope
```

Certains non-terminaux (vp17) définis plusieurs fois (ambiguïtés).
Certains non-terminaux (v12,np24,pp47) utilisés plusieurs fois (partage).

Forêts partagées et grammaires (suite)

En fait, une forêt partagée représente l'intersection d'une grammaire avec un langage régulier (engendré par un automate à états finis [FSA]).

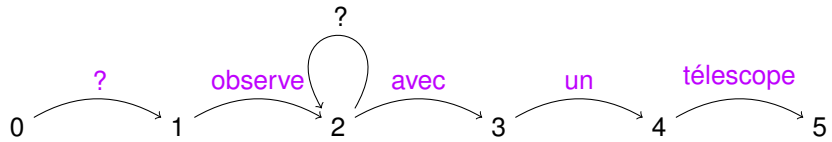
$$L(G') = L(G) \cap \{\text{“Jean observe un homme avec un télescope”}\}$$

Chaîne vs FSA

Une chaîne peut être remplacée par un FSA en entrée d'analyse :

$$L(G') = L(G) \cap L(FSA)$$

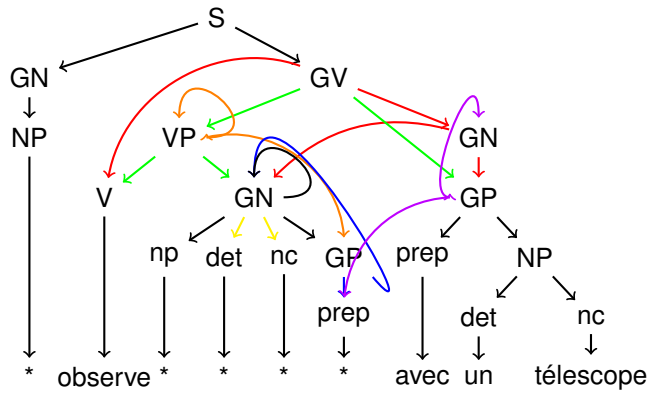
“[mot illisible] observe [mots illisibles] avec un télescope”



Représentation par une base de faits (pour les DCGs)

```
'C'(0,_,1).  'C'(1,observe,2).  'C'(2,_,2).  'C'(2,avec,3).
'C'(3,un,4).  'C'(4,télescope,5).
```

Forêt d'analyse pour une phrase incomplète



Grammaire pour une phrase incomplète

```
 s05 --> np01   vp15          pn01 --> *
np01 --> pn01                 v12 --> observe
vp15 --> v12    np25          pn22 --> *
vp15 --> vp12   pp25         det22 --> *
np25 --> np22   pp25           n22 --> *
vp12 --> vp12   pp22       prep22 --> *
vp12 --> v12    np22       prep23 --> avec
pp25 --> prep22 np25        det34 --> un
pp25 --> prep23 np35          n45 --> télescope
np22 --> np22   pp22
np22 --> det22  n22
np22 --> pn22
pp22 --> prep22 np22
np35 --> det34  n45
```

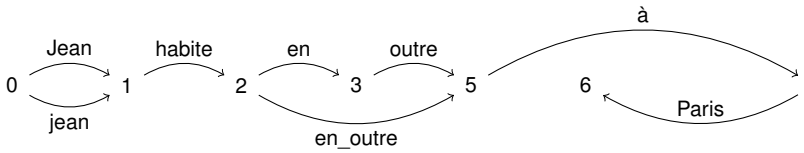

FSA

Les analyseurs tabulaires immédiatement adaptables pour prendre en entrée un FSA (ou un treillis de mots).

Analyser un FSA peut se faire en complexité temps $O(n^3)$ pour les CFGs où n est le nombre d'états du FSA.

FSAs (ou treillis de mots) utiles pour

- des phrases bruitées ou incomplètes (données orales)
- les ambiguïtés lexicales
- les ambiguïtés de segementation



FSA décorables avec des probablités ou des poids (FSA pondérés)

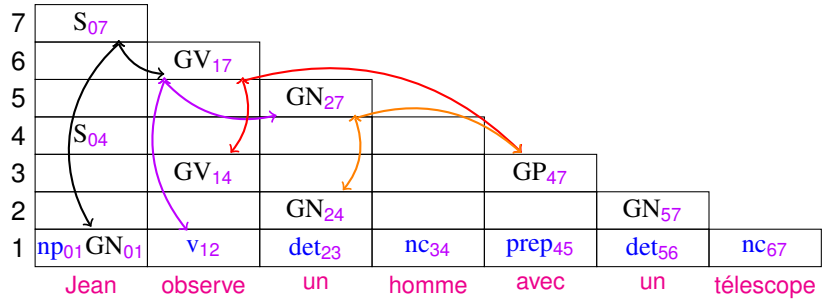
Les mêmes résultats se généralisent pour beaucoup de formalismes

Extraction des forêts

Les forêts constructibles ou extractibles après l'analyse

L'extraction utilise des pointeurs arrières (*backpointers*) allant des objets tabulés à leurs parents

> *Partant d'une réponde (`S07`), les backpointers sont suivis pour retrouver des instanciations des productions et identifier les non-terminaux.*

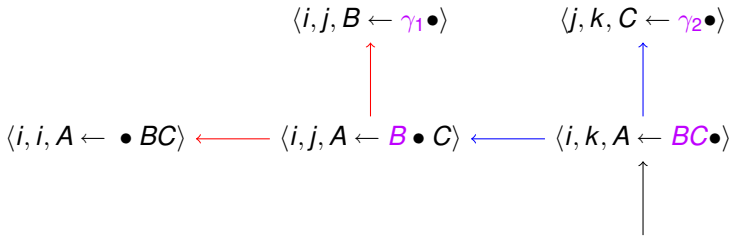


Note : La complexité en place augmente de $O(n^2)$ à $O(n^3)$ pour des grammaires binarisées (forme normale de Chomsky).

Extraction de forêts : filtrage

Pour des stratégies d'analyse autre que CKY :

- suivre les binarisations des règles pointées
- collecter les constituants (passifs) complets (et re-démarrer l'extraction de ceux-ci)
- éliminer les étapes de prédictions



Retourne la production $A_{ik} \leftarrow B_{ij} C_{jk}$ et redémarre de C_{jk} et B_{ij} (sauf si déjà extraits).

Exploiter les forêts

Une forêt partagée d'analyse permet :

- stockage d'un nombre exponentiel (ou même infini) d'arbres au sein d'une structure en complexité polynomiale en place ($O(n^3)$ pour les CFG)
  $\Rightarrow$ format d'entrée pour des traitements post-analyse
- focus sur les points d'ambiguïté
  $\Rightarrow$ aide pour la désambiguïsation post-analyse (éventuellement avec une supervision humaine).

Grammaires d'unification sans interaction

Forêt d'analyse pour les grammaires d'unification sont "sans-interaction" [Dymetman]

> *Un grammaire sans interaction peut être utilisée pour énumérer les réponses sans échec de l'unification*

```
p(X,Y) :- q(X,Y),r(X,Y).
q(Z,f(Z)).      r(a,T).      r(_,g(_)).
?- p(X,K).      %% X=a  K=f(a)
```

Une grammaire possible sans interaction :

```
p(X,Y) :- q(X,Y),r(X,Y).
q(Z,f(Z)).          r(a,f(a)).
```

Une autre :

```
p(a,f(a)) :- q(a,f(a)),r(a,f(a)).
q(a,f(a)).          r(a,f(a)).
```


Forêts partagées et DyALog

Les analyseurs DyALog acceptent l'option `-forest` pour extraire la forêt partagée

```
%% Yves loves Sabine
s{inv=> -, mood=> mood[ind,subj]}(0,3) 1 <-- [subj]2 [<>]3 [obj]4
np{gen=> masc, num=> sing }(0,1)        2 <-- [<>]5
tag_anchor(loves,1,2,tn1)               3 <--
np{gen=> fem, num=> sing }(2,3)         4 <-- [<>]6
tag_anchor(Yves,0,1,np)                 5 <--
tag_anchor(Sabine,2,3,np)               6 <--
```

Ambiguïté $\equiv$ disjonctions de productions

```
%% Yves searches the flowers on the table
s{}(0,7) 1 <-- ([subj]2 [<>]3 [obj]4 [vp]5 | [subj]2 [<>]3 [obj]
    6)
```

Labels dans les forêts

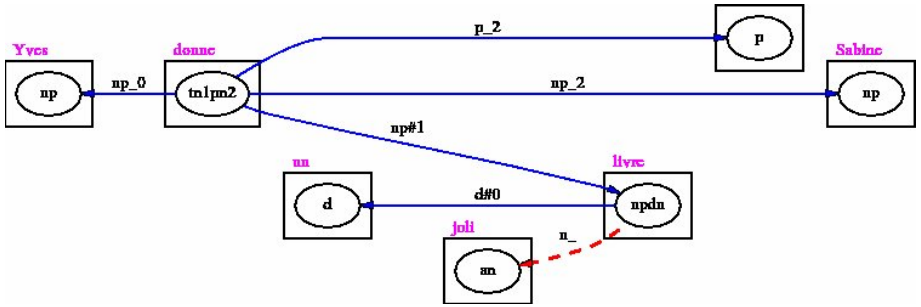
Possibilité d'ajouter des labels sur les non-terminaux, pour une meilleur lecture des forêts

```
:-tagop(':').
s --> subject:np, verb:v, object:np.
```


Forêts et DyALog (suite

Les forêts extraites avec DyALog peuvent être converties en des représentations XML et des vues graphiques

- vue graphe de dérivations
- vue graphe de dépendance



Test en-ligne : serveur d'analyseurs `http://alpage.inria.fr/demos`

Travailler sur les forêts

DyALog fournit des prédicats d'introspection permettant de travailler sur des pointeurs arrière des objets

```
%% forest(ObjAddr,Type,Parent1_Addr,Parent2_Addr,Label,Ind)

backptr(Answer) :-
    item_term(Item,Answer),
    recorded(Item,Item_Addr),
    forest(Item_Addr,Type,Parent1_Addr,Parent2_Addr,_).
```

```
forest_type(0,init).   % for initial objects                bptr =
    init
forest_type(1,call).   % for objects resulting from call    bptr =
    call
forest_type(2,and).    % for std objects                    bptr =
    and(trans,item)
forest_type(3,or).     % for alternatives                   bptr =
    or(bptr1,bptr2)
```


Travailler sur les forêts (suite)

Existence d'une bibliothèque de plus haut niveau pour afficher les forêts

```
:-require 'forest.pl'
show_forest(Answer) :- wrapped_forest(Answer).
```

Onzième partie XI

Tabulation et suspension

Formulation fonctionnelle de Fibonacci

Fibonacci récursivement définie par : $\begin{cases} \text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n) \\ \text{fib}(2) = \text{fib}(1) = 1 \end{cases}$

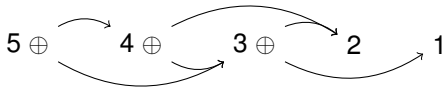
- formule directe en fonctionel
- évaluation avec pile ⇒faible consommation mémoire

```
int fib( int n ){
  if (n>2) return fib(n-1) + fib(n-2);
  else     return 1;
}
```

Mais programme non efficace :

$\text{fib}(5)$

n	5	4	3	2	1
appels	1	1	2	3	5

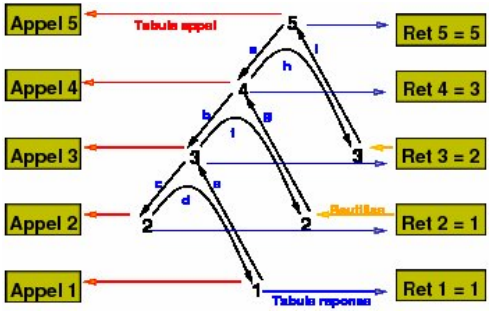


Remède : mémoriser appels et réponses à $\text{fib}(n)$.

Memo-functions or cache

Mémorisation ajoutée à l'évaluation par pile en gardant des traces des appels et retours dans une table :

```
int fib( int n ){
  int r;
  if (tab_call_fib[n])       /**/
    return tab_ret_fib[n]; /**/
  tab_call_fib[n]=1;         /**/
  if (n>2)
    r = fib(n-1)+fib(n-2);
  else { r = 1; }
  tab_ret_fib[n]=r;          /**/
  return r;
}
```



Version simplifiée

Note : Pour Fibonacci, la table des appels est non nécessaire, car pas de cycle.

```
int fib( int n ){
  int r;
  if (tab_ret_fib[n])       /**/
    return tab_ret_fib[n]; /**/
  if (n>2)
    r = fib(n-1)+fib(n-2);
  else { r = 1; }
  tab_ret_fib[n]=r;         /**/
  return r;
}
```

Que faire des boucles

Que faire des boucles dans les appels :

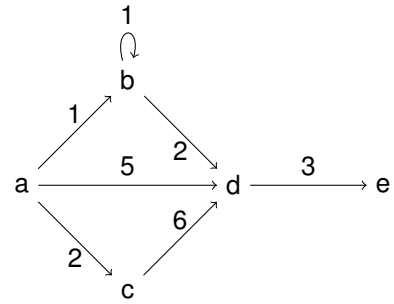
- souvent correspondent à des branches inutiles, qu'on peut ignorer
- mais pas toujours !
  Dépend de type d'agrégation de valeur qu'on souhaite appliquer.

Boucles

Exemples :

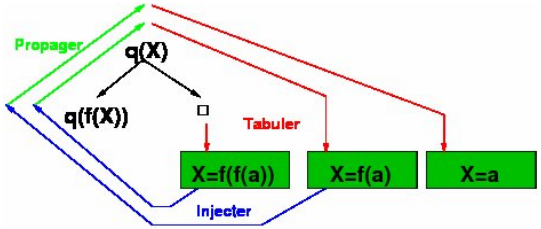
- OK sur trouver un chemin ($\exists$) ou le plus court chemin dans un graphe ($\min$)
- Pb sur trouver tout les chemins ($\cup$), le chemin le plus long ($\max$), la moyenne des chemins (avg), ...

$$\text{sp}(x,f) = \min_{arc(x,z)} d(x,z) + \text{sp}(z,f)$$



Memoization (Prolog)

Memoization (ou tabling) étend les mémo-fonctions pour la programmation en logique.
Extension : Gestion de la propagation des réponses au travers des boucles



Notion de

noeud producteur avec une liste associée de réponses

noeud consommateur des réponses issues d'un noeud producteur

Memoization (2)

Approche suivie dans le système XSB [D.S. Warren]

La memoization permet

- d'entrelacer des calculs avec et sans tabulation
- de savoir quand un noeud producteur a produit toutes ses réponses
- d'implanter plus facilement et complètement la négation

D'un autre coté, algorithme complexe :

- gestion des boucles enchâssées
- gestion de suspensions et reprises de continuations (avec soit sauvegarde de la pile de contrôle ou re-calcul)

Memoization & NLP

- [Johnson] memoization pour l'unification et les grammaires d'unification grammars.
- Formulation fonctionnelle multi-valuée de l'analyse [Leermakers] :

$$[A \leftarrow \alpha \bullet \beta](i) = (\beta \rightarrow^\star x_{i+1} \dots x_j) \triangleright j = \{j | \exists \beta \rightarrow^\star x_{i+1} \dots x_j\}$$

$$[A \leftarrow \alpha \bullet \beta](i) =$$
$$\beta = x\gamma \wedge x = x_{i+1} \triangleright [A \leftarrow \alpha x \bullet \gamma](i+1)$$
$$|\beta = B\gamma \wedge B \leftarrow \delta \triangleright [A \leftarrow \alpha B \bullet \gamma]([B \leftarrow \bullet\delta](i))$$
$$|\beta = \epsilon \triangleright i$$

Prédicats tabulés

Type par défaut pour les prédicats DyALog
Ces prédicats sont tabulés et suspendables (pour gérer des boucles)

Nécessaires pour les non-terminaux récursifs

```
np --> np,gp.
```

OK mais trop puissant pour fibonacci :

```
fib(0,1).
fib(1,1).
fib(N,M) :- N > 1,
    N1 is N-1, fib(N1,M1),
    N2 is N-2, fib(N2,M2),
    M is M1+M2.
```

Ces prédicats laissent une trace dans la forêt

Divers types de prédicats

Il existe plusieurs types de prédicats en DyALog organisé autour des notions de

- tabulation
- suspension

Classification repose aussi sur la notion de descendant
un prédicat `q` est un descendant de `p` ssi :

- `q` apparaît dans le corps d'une clause définissant `p`
- ou `q` est le descendant d'un descendant de `p`

Note : un prédicat récursif `p` est son propre descendant

Prédicats faiblement tabulés

Prédicat `p` faiblement tabulé :

- tabulé
- `p` et ses descendants non suspendables
  pas de boucles, ou boucles ignorées
- laisse une trace dans la forêt

```
:-light_tabulate fib/2, foo{}.
:-light_tabulate dcg(np{}).
:-light_tabulate dcg(_).
```

Le plus efficace pour fibonnaci

```
:-light_tabulate fib/2.
fib(0,1).
fib(1,1).
fib(N,M) :- N > 1,
    N1 is N-1, fib(N1,M1),
    N2 is N-2, fib(N2,M2),
    M is M1+M2.
```

Prédicats faiblement tabulés

Fonctionne pour “trouver un chemin” sur des graphes cycliques.

```
:-light_tabulate exists_path/2.
exists_path(N,N).
exists_path(N,M) :- arc(N,P),exists_path(P,M).
```


Prédicats prolog suspendables

Un prédicat p est de type prolog si

- non tabulé
- peut avoir des descendants suspendables
  ⇒ entraîne une suspension indirecte de p
  ⇒ oblige à tabuler le fragment de pile d’appel entre p et le point de suspension sur q
- ne laisse pas de trace dans la forêt

```
:-prolog foo/3.
```


Prédicats prolog non suspendables

Un prédicat p est de type prolog récursif si

- non tabulé
- tous les descendants (p inclus) sont non suspendables
- très proches du mode Prolog standard (profondeur d’abord, retours arrière) même problèmes que pour Prolog !
- ne laisse pas de trace dans la forêt

```
:-rec_prolog verbose/2.
verbose(Fmt,Args) :- option('-verbose'), format(Fmt,Args).
verbose(Fmt,Args) :- option('-noverbose').
```

Version peu efficace de fibonnaci :

```
:-rec_prolog fib/2.
fib(0,1).
fib(1,1).
fib(N,M) :- N > 1,
    N1 is N-1, fib(N1,M1),
    N2 is N-2, fib(N2,M2),
    M is M1+M2.
```


Boucle sur parcours de graphes cycliques

Prédicat prolog non suspendables

Cas particulier de prédicat récursivement prolog pour ceux définis par une seule clause.

- même propriétés que pour rec_prolog
- mais traitement plus efficace

```
:-std_prolog fib/2.
fib(N,M) :-
   ( N = 0 -> M = 1
   ; N = 1 -> M = 1
   ; N > 1,
    N1 is N-1, fib(N1,M1),
    N2 is N-2, fib(N2,M2),
    M is M1+M2
   ).
```

Prédicats extensionels

Prédicats uniquement définis

- en extension par une liste de faits
- et par aucune clause

Ils sont tabulés, non suspendables et ne laisse pas de trace dans la forêt.

```
:-extensional 'C'/3.
'C'(0,une,1).
'C'(1,pomme,2).

%% pour une trace dans la forêt
:-light_tabulate token/3.
token(L,Token,R) :- 'C'(L,Token,R).
```


Gardes

Utilisées pour contruire des conditionnelles IF-THEN-ELSE

```
( Guard -> TrueExp ; FalseExp )
```

Une garde est une expression non suspendable, donc construite à partir

- de prédicats non suspendables
  builtins, light_tabular, rec_prolog, std_prolog, extensional
- des opérateurs de corps de clauses
  conjonction, disjonction, conditionnelles ->

Note : une garde peut apparaître dans des prédicats suspendables

Parallélisme

Proposition de parallélisme pour Prolog Gupta, Santos Costa, Warren, Karlsson

- Parallélisme OR : les alternatives sont explorées en parallèle
- Parallèlisme AND : les littéraux d'un corps de clause sont explorés en parallèle
  ⇒ nécessite communication pour les instanciations de variables
- concurrence : parallélisme AND avec points de synchronization
  ⇒ gestion possible par suspension

Programmation logique par contraintes

Exemple

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],      % Create variables
    Digits :: [0..9],                % Associate domains to
     variables
    S #\= 0,                         % Constraint: S must be
     different from 0
    M #\= 0,
    alldifferent(Digits),            % all the elements must take
     different values
                  1000*S + 100*E + 10*N + D      % Other constraints
                + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling(Digits).                % Start the search
```


Programmation concurrente par contraintes

Formalisée par Saraswat

- existence d'un magasin (*store*) de contraintes
- tell (C) pour ajouter une contrainte *C* dans le magasin
  échec si cette contrainte rend le magasin incohérent
- ask(C) pour être averti dès que la contrainte *C* devient vraie grâce au magasin
- un propagateur de contraintes peut s'écrire en combinant `ask` et `tell`

```
solve :-
    set_constraints,
    propagate,
    display.
propagate :- ask(X<Y),ask(Y<Z),tell(X<Z).
...
```

Note : le store est nettoyé au moment des retours-arrière
i.e. élimination des contraintes ajoutées après le point de choix

Oz/Mozart

Langage multi-paradigmes :

- fonctionnel
- orienté objets
- programmation logiques
- contraintes
- concurrence
- distribué et mobilité

`http://www.mozart-oz.org/`

Mozart : exemple

```
proc {Ints N Xs}
   or N = 0 Xs = nil
   [] Xr in
      N > 0 = true Xs = N|Xr
      {Ints N-1 Xr}
   end
end
local
   proc {Sum3 Xs N R}
      or Xs = nil R = N
      [] X|Xr = Xs in
         {Sum3 Xr X+N R}
      end
   end
in proc {Sum Xs R} {Sum3 Xs 0 R} end
end
local N S R in
   thread {Ints N S} end
   thread {Sum S {Browse}} end
   N = 1000
end
```

DyALog et concurrence

La tabulation dans DyALog offre une forme réduite de concurrence :

- le store est donné par la table.
  Un `tell` correspond à ajouter un objet dans la table

```
tell(X) :- check_coherence(X).
```

- l'opération `ask` est donnée par le prédicat '$answers'(G) qui attend les réponses à `G` dans la table

```
ask(X) :- '$answers'(tell(X)).
```

Forme approchée de concurrence car :

- la table est une structure globale
- mais de backtrack sur la table

Exemple :

guidage DCG avec une approximation en langage régulier

```
?-recorded('N'(N)),phrase(approx_s,0,N),fail.
?-recorded('N'(N)),phrase('$answers'(approx_s),0,N),tag_phrase(s
    {},0,N).
%% Similar check for all DCG non-terminals
```


Exemple : coordination

Gestion de la coordination :
Idée : suivre des dérivations parallèles similaires à droite et gauche du coordonnant

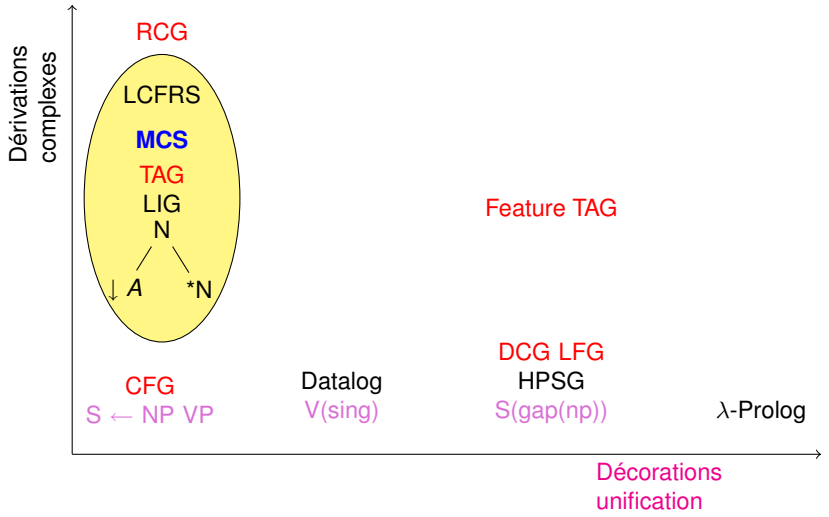
```
%% one main parsing ''agent''
?-recorded('N'(N)), tag_phrase(s,0,N).
%% auxiliary ''agents'' for coords
?-coord_handler,fail.

coord_handler :-
        'C'(L, et, R),   % locate coordination places
        %% wait for derivations reaching coordination
        tag_phrase('$answers'(NT),K,L),
        tag_phrase(NT,gen_pos(Right,Left1),gen_pos(Right2,Left))
        .

%% Try to reuse a coordination handled by watch_coord
my_subst_handler(P,Bot,Left,Right,N) :-
        tag_phrase('$answers'(id=N at P),Left,Right1),
        tag_phrase(id=coord at <=> coo,Right1,Left2),
        tag_phrase('$answers'(id=N at P),
                   gen_pos(Left2,Left),gen_pos(Right,Right1)).
```


Exemple : méta-analyseur

Facile d'implémenter des méta-analyseurs tabulaires avec DyALog
mais plus difficile pour d'extraire la forêt partagée (séparation grammaires et méta-niveau)

```
:-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).
```

Un spectre de formalismes syntaxiques

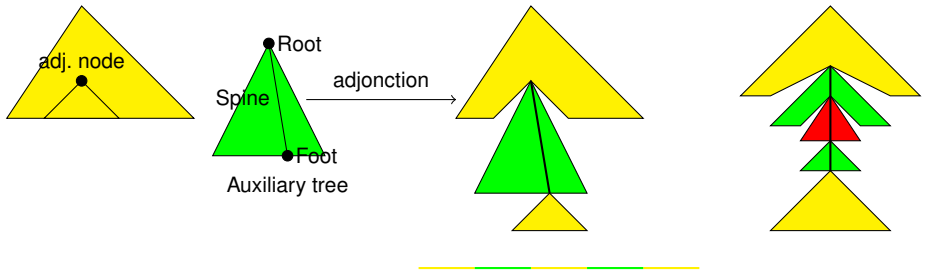


Douzième partie XII

Grammaires d'adjonction d'arbres

Tree Adjoining Grammars [TAG]

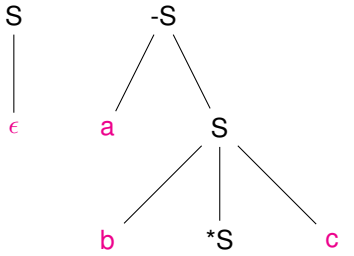
TAGs [Joshi] build derived trees from elementary initial and auxiliary trees by substitution and adjoining.



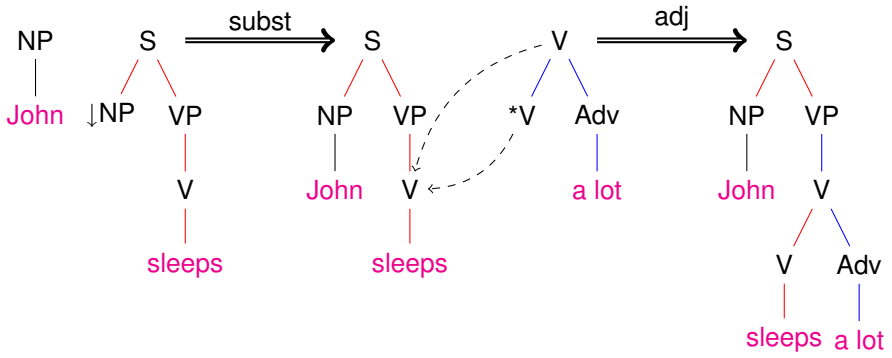
Tabular parsing in worst case time complexity $O(n^6)$ for pure TAGs.
Nodes may be decorated by pairs top and bot of logical attributes ([Feature TAG]).

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))
        }.
```

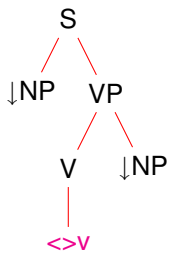

Exemples

```
tag_tree{ name => na,             % Nom Adjectif
          family => adj,
          tree => auxtree bot=I at n(bot=I::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
    ⇒ 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 -
]
```

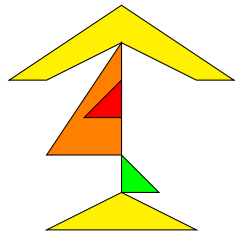
Lexique
hypertag «promettre»

```
[ arg0 [ kind subj | - ]
       [ pcas -      ]
  arg1 [ kind obj | scomp | - ]
       [ pcas -      ]
  arg2 [ kind prepobj | - ]
       [ pcas à | -  ]
  refl -
]
```

The variables propagate the hypertag values to the nodes and guards.

Hybrid TIG/TAG parsing

TAGs are often too powerful ⇒
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
  ⇒ 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
```

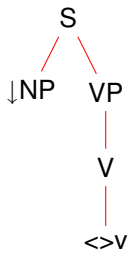
```
%% :-tig(TreeName,Kind).
:-tig('10 det_coord shallow_auxiliary',right).
:-tig('102 adv_mod_on_coo modifier_after_x shallow_auxiliary',
    right).
:-tig('103 modifier_after_x shallow_auxiliary xpro_on_noun',right
    ).
...

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


Treizième partie XIII

Grammaires et Contraintes

Arbres et contraintes

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



S father_of *NP*
S father_of *VP*
VP father_of *V*
V father_of *v*
NP $\preceq$ *VP*
...

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 $N \not\Leftrightarrow M$
- 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 : D $\prec$ N	(P3.11)
	: D $\prec$ Pro	(P3.12)
	: D $\prec$ AP	(P3.13)
	: N $\prec$ PP	(P3.14)
	requirement : N $\Rightarrow$ D	(P3.15)
	: AP $\Rightarrow$ N	(P3.16)
	exclusion : N $\not\Leftrightarrow$ Pro	(P3.17)
	dependency : N[GEND 1, NUM 2] $\rightsquigarrow$ D[GEND 1, NUM 2]	(P3.18)

Exemple2

VP (*Verb Phrase*)

Features	Property Type : Properties	
[AVM]	*obligation* : $\triangle$V	(P3.19)
	uniqueness : $\mathrm{V}_{[main\ past\ part.]}$!	(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 : $\mathrm{V}_{[past\ part.]} \Rightarrow \mathrm{V}_{[aux.]}$	(P3.26)
	exclusion : $\textsc{Pro}_{[acc]} \not\Leftrightarrow$ NP	(P3.27)
	: $\textsc{Pro}_{[dat]} \not\Leftrightarrow \textsc{Pro}_{[acc]}$	(P3.28)
	dependency : V[PERS 1, NUM 2] $\rightsquigarrow$ Pro[TYPE pers, CASE nom, PERS 1, NUM 2]	(P3.29)

Évaluation avec CHR

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

CHR ≡ 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].
```


CKY

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 !)

PG et CHR

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).
```


Quatorzième partie XIV

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
  ⇒ modularité
- Satisfaction partielle des contraintes hors-ligne pour construire les structures d'une grammaire TAG (ou LFG, . . .)
  ⇒ 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

```
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 :

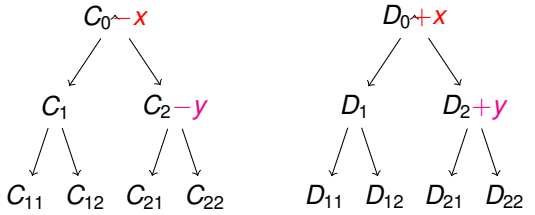
- type : dans std, subst, foot, anchor, coanchor, lex, sequence, alternative
- id : identification du noeud
- cat : catégorie syntaxique
- lex : forme (pour les noeuds de type lex)
- top et bot : arguments TAG
- adj : dans yes, no, strict
- optional : optionalité + -
- star : répétition * -

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



Espaces de noms

Les conflits de noms (lors de l'import multiple d'une même classe) sont évités grace aux espaces de noms sur

- les ressources − foo::r
- les noeuds foo::N
- les variables $foo::bar

Permet une écriture proche d'appels de fonction avec liaisons de noms

```
- foo::node_agreement; A=foo::N1; B=foo::N2;
%% close of: foo::node_agreement( N1 => A, N2 => B );
```

Mais les liaisons peuvent être faites ultérieurement

Proche de XMG sauf que les noms sont globaux par défaut dans FRMG

Identification des noeuds

Dans FRMG :

- identification explicite sur le nom des noeuds
- identification explicite par aliasing A=B
- identification implicitement sur les pères father(N)

```
class agreement { %% Generic class to add agreement equations
    + agreement;
    father(N).bot.number = node(N).top.number;
    father(N).bot.gender = node(N).top.gender;
    father(N).bot.person = node(N).top.person;
}
class superlative_as_adj_mod {
    <: superlative_as_mod;
    node(Foot).cat = value(adj);
    - det::agreement; det = det::N;
    - adj::agreement; Foot = adj::agreement;
}
```

Pas d'identification implicite via des couleurs comme dans XMG
Reflexions sur l'expression de condition d'extensionalité

```
extensional [cat: v];
```

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)
  ;
```


Gardes (suite)

Les gardes sont très puissantes mais fastidieuses
⇒ notation plus pratique

- Exclusion entre les noeuds *A* et *B*

```
A => desc.dummy.A_or_B = value(a);
~ A => desc.dummy.A_or_B = value(~ a);
B => desc.dummy.A_or_B = value(b);
~ B => desc.dummy.A_or_B = value(~ b);
```

- Unicité (un seul arg verbal extrait)

```
XGroup =>
   $a0.extr = value(+), $a1.extr=value(−), $a2.extr=value(−)
 | $a0.extr = value(−), $a1.extr=value(+), $a2.extr=value(−)
 | $a0.extr = value(−), $a1.extr=value(−), $a2.extr=value(+)
   ;
~ XGroup =>
   $a0.extr = value(−), $a1.extr=value(−), $a2.extr=value(−)
   ;
```


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) ⇒ 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)

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 :

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

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)

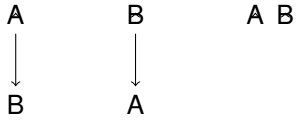
- Liens entre les noeuds A et B dans une classe non neutre
  $\Rightarrow$ noeuds potentiellement aliasables

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

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

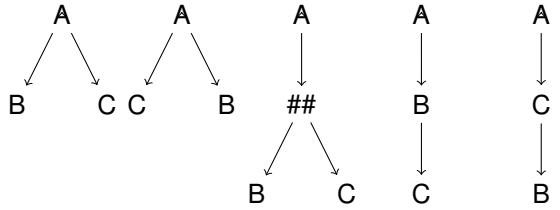
$$A < B \vee B < A \vee A \triangleright^+ B \vee B \triangleright^+ A$$

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

$$\mathcal{K} \cup (A = B) \mid\!- \mathcal{K}[A \mapsto B]$$

$$\frac{}{\mathcal{K} \mid\!- \mathcal{K} \cup (A \rhd^+ B)} \quad A \rhd B \in \mathcal{K}$$

$$\frac{}{\mathcal{K} \mid\!- \mathcal{K} \cup (A \rhd^+ C)} \quad \begin{cases} A \rhd^+ B \in \mathcal{K} \\ B \rhd^+ C \in \mathcal{K} \end{cases}$$

$$\frac{}{\mathcal{K} \mid\!- \mathcal{K} \cup (A < C)} \quad \begin{cases} A < B \in \mathcal{K} \\ B < C \in \mathcal{K} \end{cases}$$

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

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

$$\mathcal{K} \cup (A < A) \mid\!- \text{fail}$$

$$\mathcal{K} \cup (A \rhd^+ A) \mid\!- \text{fail}$$

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

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

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

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

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

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

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

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

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

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

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

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 $\text{tree}(C_i)$.

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

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

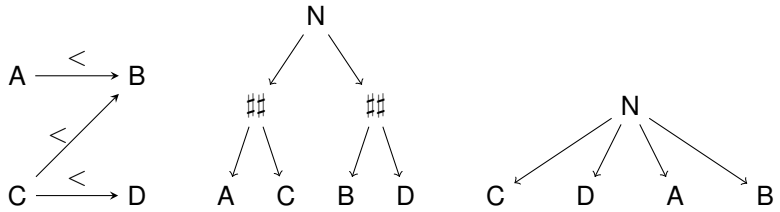
$$F_1 \sharp\sharp F_2 \in F \Rightarrow \forall C_i \in F_1, \forall C_j \in F_2,\ \forall B_i \in C_i, \forall B_j \in C_j,\ \bigwedge \{\ \neg(B_i \prec B_j) \neg(B_j \prec B_i)$$

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

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



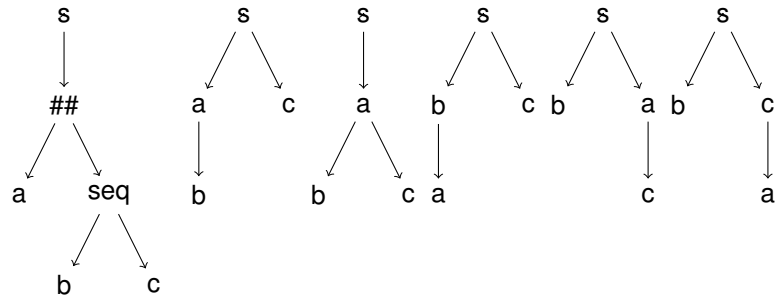
Question : garder entrelacement ou utiliser directement l'ordre partiel

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;
}
```



Quinzième partie XV

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

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).
```


Seizième partie XVI

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),
                      [] )
  ).
```