

# 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

## Forêts partagées

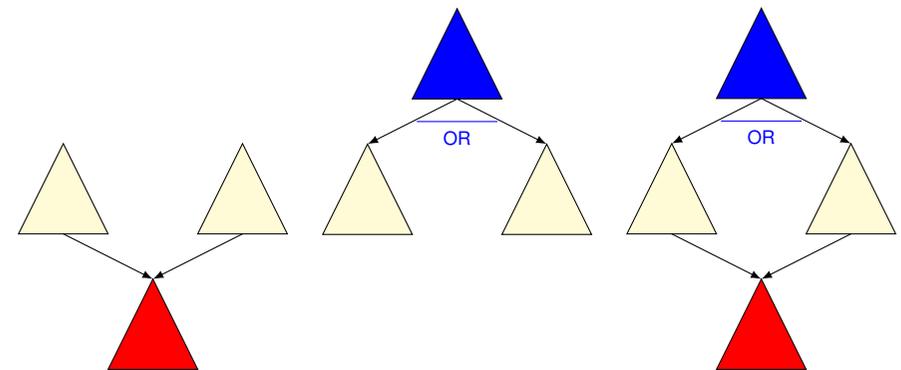
Ambiguïtés du langage  $\Rightarrow$   
Plusieurs analyses possibles par phrases !

Forêt  $\equiv$  ensemble d'arbres d'analyse

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

## Dixième partie X Forêts partagées d'analyse

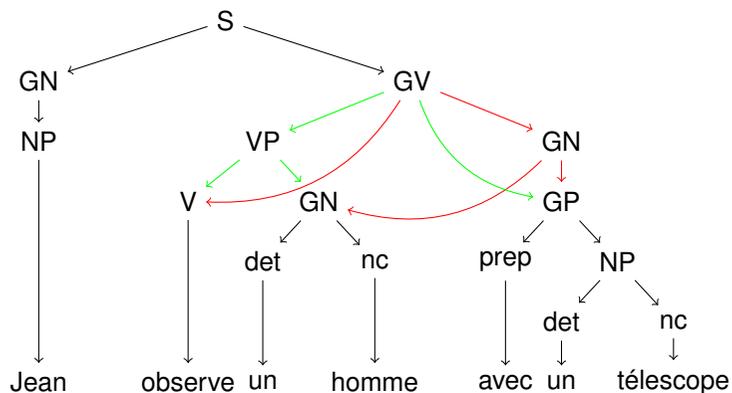
## Graphes ET-OU



(g) Partage de sous-arbres

(h) Partage de contexte

(i) Partage complet



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	pn01	-->	Jean
np	-->	pn	v12	-->	observe
np	-->	det n	det23	-->	un
np	-->	np pp	n34	-->	homme
vp	-->	v np	prep45	-->	avec
vp	-->	vp pp	det56	-->	un
pp	-->	prep np	n67	-->	télescope
s07	-->	np01 vp17	pn01	-->	Jean
np01	-->	pn01	v12	-->	observe
vp17	-->	v12 np27	det23	-->	un
vp17	-->	vp14 pp47	n34	-->	homme
np27	-->	np24 pp47	prep45	-->	avec
n37	-->	n34 pp47	det56	-->	un
np24	-->	det23 n34	n67	-->	télescope
pp47	-->	prep45 np57			
np57	-->	det56 n67			
vp14	-->	v12 np24			

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

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

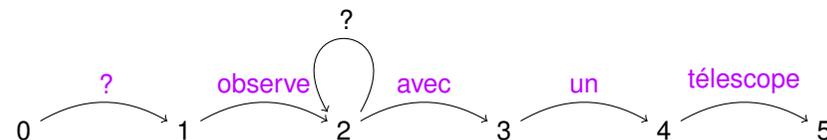
$$L(G') = L(G) \cap \{ \text{"Jean observe un homme avec un télescope"} \}$$



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

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

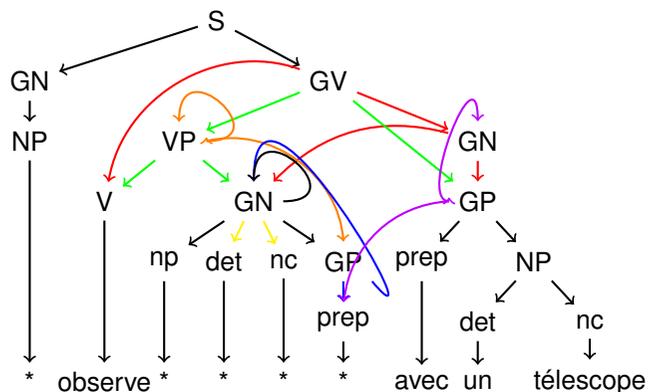
"[mot illisible] observe [mots illisibles] avec un télescope"



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	

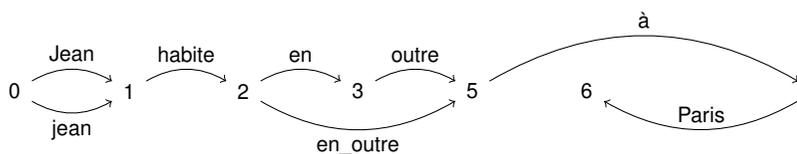
## FSA

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.

FSA (ou treillis de mots) utiles pour

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



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

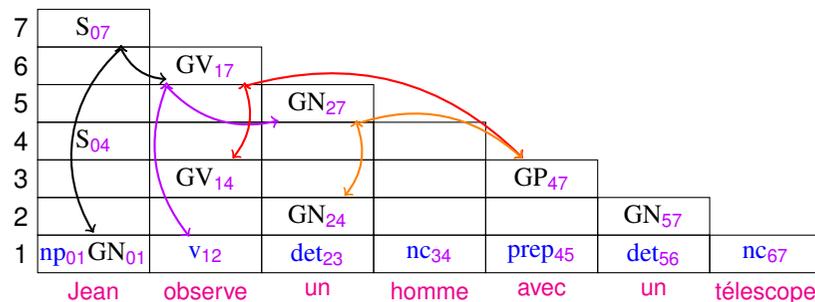
Les mêmes résultats se généralisent pour beaucoup de formalismes

## 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 réponse (s07), les backpointers sont suivis pour retrouver des instanciations des productions et identifier les non-terminaux.*

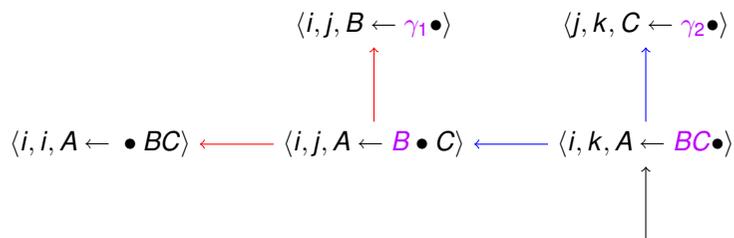


**Note :** La complexité en place augmente de  $O(n^2)$  à  $O(n^3)$  pour des grammaires binarisées (forme normale de Chomsky).

## 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)  
⇒ format d'entrée pour des traitements post-analyse
- focus sur les points d'ambiguïté  
⇒ 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 ≡ 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)
```

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

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

## 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)
```

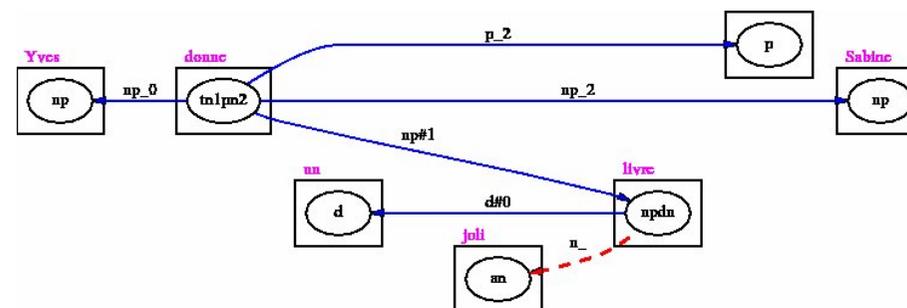
```
backptr (Answer) :-
    item_term (Item, Answer),
    recorded (Item, Item_Addr),
    forest (Item_Addr, Type, Parent1_Addr, Parent2_Addr, _, _).
```

```
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)
```

## Forêts et DyALog (suite)

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

- vue graphe de dérivation
- vue graphe de dépendance



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

## 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

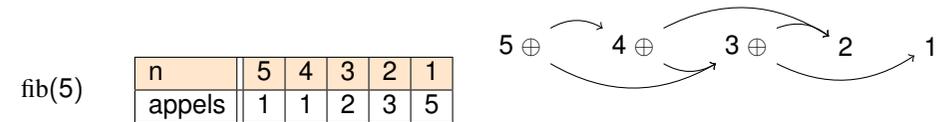
Fibonacci récursivement définie par :

$$\begin{cases} \text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n) \\ \text{fib}(2) = \text{fib}(1) = 1 \end{cases}$$

- formule directe en fonctionnel
- évaluation avec pile  $\Rightarrow$  faible consommation mémoire

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

Mais programme non efficace :

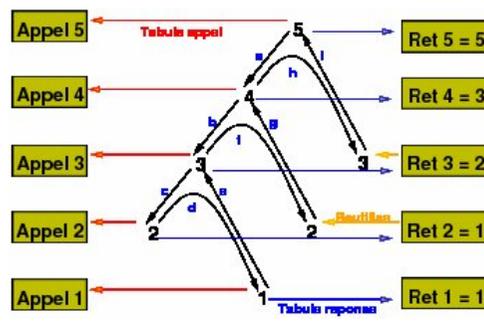


Remède : mémoriser appels et réponses à fib(n).

# Memo-fonctions 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 :

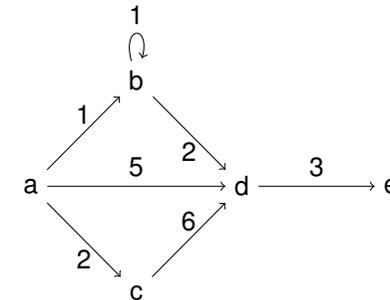
- souvent correspondent à des branches inutiles, qu'on peut ignorer
- mais pas toujours !  
Dépend de type d'agrégation de valeur qu'on souhaite appliquer.

## 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), ...

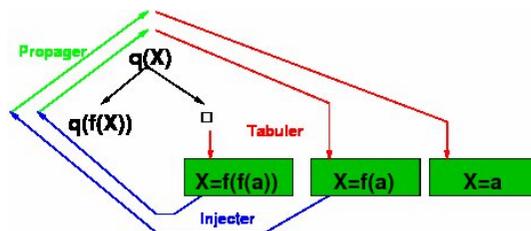
$$sp(x, f) = \min_{arc(x,z)} d(x, z) + sp(z, f)$$



## 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-calcule)

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

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

$$[A \leftarrow \alpha \bullet \beta](i) =$$

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

$$|\beta = B\gamma \wedge B \leftarrow \delta \triangleright [A \leftarrow \alpha B \bullet \gamma]([B \leftarrow \bullet \delta](i))$$

$$|\beta = \epsilon \triangleright i$$

## 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  $\rightarrow$  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

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 fibonacci

```
:-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 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.
```

## 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.
```

## 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 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).
```

Proposition de parallélisme pour Prolog [Gupta, Santos Costa, Warren, Karlsson](#)

- Parallélisme OR : les alternatives sont explorées en parallèle
- Parallélisme 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

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

## Exemple

```
sendmore(Digits) :-
  Digits = [S,E,N,D,M,O,R,Y],      % Create variables
  Digits :: [0..9],                % Associate domains to
  variables                          % 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
```

## Oz/Mozart

Langage multi-paradigmes :

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

<http://www.mozart-oz.org/>

## 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

## 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
```

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

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).
```

## Douzième partie XII

### Compléments sur DyALog

```
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)

- 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 ambigüités, mots incnnus, ...
- prédicats d'introspection pour explorer le contenu de la table  
⇒ 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 lfoo`
- la notion de **modules**, 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),
                  [] )
    ).
```