Supplément Différence de Listes

Table de Matières

- Comment intuitivement **concatener** des listes
- Un **détour mathématique** rend la concatenation plus simple et plus efficace
- La même méthode **applatit** les listes de mode plus efficace
- **DCGs** sont aussi une différence de listes
- On peut décrire les différences de listes en plusieurs **métaphores**
- **Gap-Threading** en est une des plus importantes pour la linguistique computationelle
- Un exemple très simple pour la **topicalisation**
- Gap-threading traite toutes les sortes de **dépendences à longue distance**
- Un exemple plus elaboré : une **grammaire GB pour questions**
- **Vos questions**

1 Concatenation des Listes

- Intuitivement : `append(L1,L2,L3) :- met L2 à la fin de L1`
- Avec pointeurs :

```
L1^.ele1^.next = L1^.ele2
L1^.ele2^.next = L1^.ele3
L1^.eleX^.next = NIL % X est dernier element
```

Si on a un pointeur au dernier élement :

```
L1^.last = L1.eleX
```

Pour concatener, il faut faire du premier élément de L2 l'élément suivant au dernier élément de L1 :

```
L1^.last^.next = L2.ele1
```

Et pour corriger le «last» pointeur :

```
L1^.last = L2^.last
```

- En Prolog, on n'a pas de pointeur à la fin → il faut trouver la fin commencant au début:

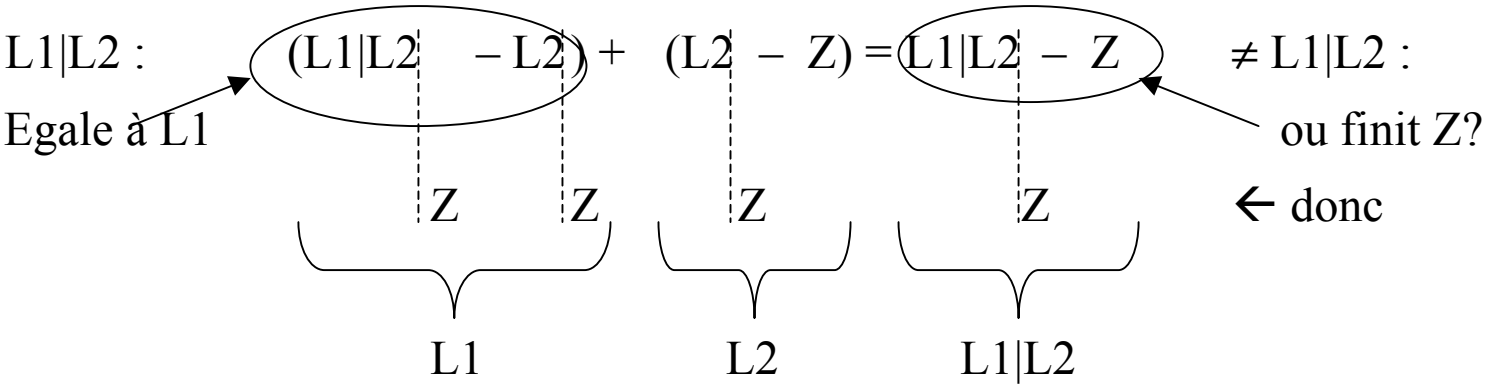
```
find_last([]).
find_last([F|R]) :- find_last(R).
```

Pour concatenation: Garder les éléments, puis les ajouter individuellement à la tête :

```
append([],L,L). % fin: copier arg2 à arg3
append([X|L1],L2,[X|L3]) :- % copier X de L1 à L3 indivuellement
    append(L1,L2,L3).
```

2 Un petit détour mathématique

- Concatenation : L1 + L2 = L1|L2 % (L1 suivi de L2)
- La mathématique dit : (X – Y) + (Y – Z) = X – Z
- Si on met X = L1|L2 : (L1|L2 – L2) + (L2 – Z) = L1|L2 – Z ≠ L1|L2 :

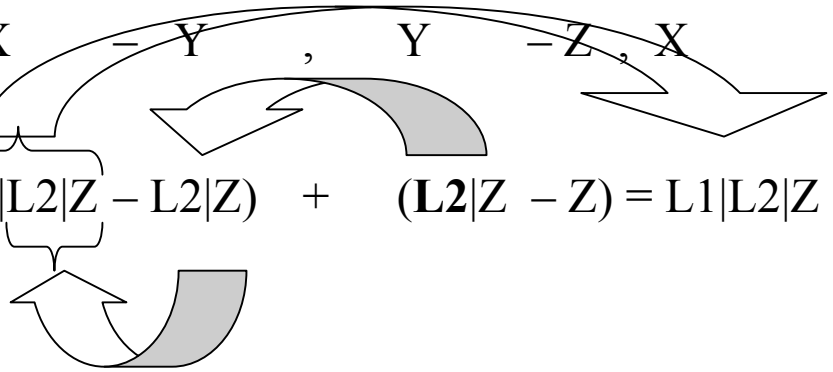


- On ne connaît pas Z (ou L2 au début) : «structures incompletes»

append_dl(X – Y , Y – Z , X – Z)

- Liage des Variables : (**L1**|L2|Z – L2|Z) + (**L2**|Z – Z) = L1|L2|Z – Z

(connus en **Gras**)



Trace p. 128 [1,2,3| Y] – Y , [4,5| Z] – Z) , R

3 Métaphores pour différences de listes

La DCG

```
S --> np, vp. % DCG
```

est équivalent à

```
S(X-Z) :- np(X-Y), vp(Y-Z). % internal representation
```

Notre conception locale (S from X to Z if NP from X to Y & VP from Y to Z) n'est que

☞ une méthaphore: Métaphore locale.

Nous connaissons déjà :

☞ la métaphore mathématique: (X – Y) + (Y – X) = Y – Z

Nous connaîtrons :

☞ la métaphore accumulateur

☞ la métaphore IN – OUT ou OUT – IN

4 Applatir avec différance de listes

- Version traditionelle :

```
flatten_r([H|T],L2) :-
  flatten_r(H,Temp1),
  flatten_r(T,Temp2),
  append(Temp1,Temp2,L2).

flatten_r(H,[H]) :-
  H \= [],
  H \= [_|_].

flatten_r([],[]).
%fin rec: liste vide
```

- Version _dl :

```
flatten_dl([H|T],X-Z) :-
  flatten_dl(H,X-Y),
  flatten_dl(T,Y-Z).
  %append unneeded

flatten_dl(H,[H|L]-L) :-
  H \= [],
  H \= [_|_].

flatten_dl([],L-L).
% fin rec: dliste vide
```

☞ Metaphore accumulateur :

```
flatten_a([H|T],Acc,L2) :-
  flatten_a(T,Acc,L2tmp),
  flatten_a(H,L2tmp,L2).

flatten_a(H,T,[H|T]) :-
  H \= [],
  H \= [_|_].

flatten_a([],L,L).
```

- Comparaison a _dl:

```
flatten_dl([H|T],X-Z) :-
  flatten_dl(H,X-Y),
  flatten_dl(T,Y-Z).
  %append unneeded

flatten_dl(H,[H|L]-L) :-
  H \= [],
  H \= [_|_].

flatten_dl([],L-L).
% fin rec: dliste vide
```

☞ Métaphore OUT – IN ou IN – OUT:

```
                 out  in
flatten_dl(H,[H|L]-L) :-
```

5 Gap-Threading

- dépendences à longue distance :

  = condition non-locale

  = condition qui s'éteint à plus d'une règle DCG → à plus d'une arbre locale

  = condition qui est plus expressif que CFG

  En Prolog, on se sert de _dl pour les CFGs, pour traîter les DCG

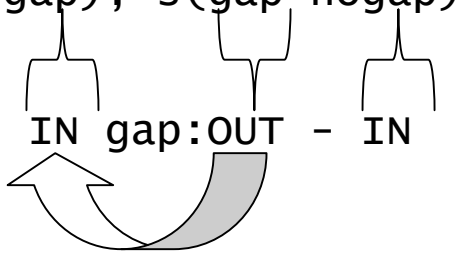
  Les dépendences à longue distance sont des «super-_dl» du même cadre

- Exemple le plus simple : Topicalisation

  «I like bananas» ←→ «bananas_1 I like t_1»

```
sbar(nogap-nogap) --> s(nogap-nogap). % forme canonique
sbar(nogap-nogap) --> np(nogap-gap), s(gap-nogap). % Topicalisé

%                     filler:OUT - IN gap:OUT - IN
```



```
% "Bananas_1, I like _1."
% "I like Bananas."
% Métaphore: OUT - IN
% ?- root([bananas,'I',like],[]).

root --> sbar(nogap-nogap). % OUT: nogap, IN: nogap

sbar(nogap-nogap) --> s(nogap-nogap). % forme canonique
sbar(nogap-nogap) --> np(nogap-gap), s(gap-nogap). % Topicalisé
%                     filler OUT-IN   gap OUT-IN

s(Out-In) --> np(nogap-nogap), vp(Out-In).  % Noeud sans trou

vp(Out-In) --> tv(nogap-nogap), np(Out-In). % Noeud sans trou

tv(G-G) --> tv.

np(nogap-nogap) --> n.     % cas normale
np(gap-nogap)   --> [].    % gap
np(nogap-gap)   --> n.     % filler
```

```
% grammaire simplistique pour questions, avec difference de listes
% Metaphore: OUT-IN
% verbes transitifs seulement. Avec raccoltage de la structure syntaxique

root(Structure) --> cp([]-[],Structure).
%wrapper ensures gaplist is empty

cp(F-F,cp(CBAR)) --> cbar(F-F,CBAR).
cp(F-F,cp(NP,CBAR)) --> np(F-[gap(NP)|F],NP), cbar([gap(NP)|F]-F,CBAR).
% np filler: np-gap-OUT from cbar, filled in this np.

cbar(F0-F,cbar(IP)) --> ip(F0-F,IP).
cbar(F0-F,cbar(V,IP)) --> v(F0-[gap(V)|F0],V), ip([gap(V)|F0]-F,IP).
% verb filler: v-gap-OUT from ip, filled in this v.

ip(F0-F,ip(NP,VP)) --> np(F0-F0,NP), vp(F0-F,VP).
% Node without gaps: Transmit gaplist unchanged

vp(F0-F,vp(V,NP)) --> v(F0-F1,V), np(F1-F,NP).
% Intermediate node

v(F-[gap(v(_))|F],v(voit)) --> ['voit'].       % explicit filler case
v(F-F,v(voit)) --> ['voit'].                   % normal case
v([gap(v(X))|F]-F,v(trace(v(X)))) --> [].      % gap case:
% add 1 item to gap list: OUT = IN + gapinfo

np(F-[gap(np(_))|F],np(qui)) --> ['qui'].      % explicit filler case
np(F-[gap(np(_))|F],np(tu)) --> ['tu'].        % explicit filler case
np(F-F,np(qui)) --> ['qui'].
np(F-F,np(tu)) --> ['tu'].
np([gap(np(X))|F]-F,np(trace(np(X)))) --> [].  % gap case:
% add 1 item to gap list: OUT = IN + gapinfo
```