Table of Contents

TP 8 - Programmation pour la préparation à l'agrégation maths option info

TP 8 : Programmation logique

- En Prolog. En fait, dans une version maison de Prolog.
- Les cellules de codes suivantes sont exécutées par un shell, ici Bash.

Introduction

Je vous ai envoyé un fichier `prolog.zip`, voyons comment l'extraire et construire l'interpréteur prolog.

In [10]:
```
ls -larth prolog.zip
zipinfo prolog.zip
```

```
-rw-r--r-- 1 lilian lilian 3,3K mars  16 17:13 prolog.zip
Archive:  prolog.zip
Zip file size: 3348 bytes, number of entries: 7
drwxr-xr-x  3.0 unx        0 bx stor 18-Mar-16 17:13 prolog/
-rw-r--r--  3.0 unx     2668 tx defX 17-Aug-28 16:34 prolog/lib.ml
-rw-r--r--  3.0 unx     2489 tx defX 17-Aug-28 16:34 prolog/resolution.ml
-rw-r--r--  3.0 unx      256 tx defX 17-Aug-28 16:34 prolog/prolog.ml
-rw-r--r--  3.0 unx       42 tx defX 17-Aug-28 16:34 prolog/pair.pl
-rw-r--r--  3.0 unx      310 tx defX 17-Aug-28 16:34 prolog/Makefile
-rw-r--r--  3.0 unx      223 tx defX 17-Aug-28 16:34 prolog/nat.pl
7 files, 5988 bytes uncompressed, 2234 bytes compressed:  62.7%
```

In [11]:
```
unzip prolog.zip
```

```
Archive:  prolog.zip
   creating: prolog/
  inflating: prolog/lib.ml
  inflating: prolog/resolution.ml
  inflating: prolog/prolog.ml
  inflating: prolog/pair.pl
  inflating: prolog/Makefile
  inflating: prolog/nat.pl
```

In [12]:
```
ls prolog/
```

```
lib.ml  Makefile  nat.pl  pair.pl  prolog.ml  resolution.ml
```

In [40]:
```
cd prolog/
```

```
Vous quittez le dossier '/home/lilian/agreg-2017/PROG/tp8'.
        /!\ Chemin relatif, car le vrai est '/home/lilian/teach/teachensren/AGREG/2
017_2018/PROG/tp8'.
        Direction ⟹ prolog/
```

Allons construire `prolog` :

In [41]:
```
/usr/bin/make clean
/usr/bin/make
rm -f *.cm[iox] *.annot
```

```
rm -f *.cm[iox] *~ prolog mytop
ocamlc -pp camlp4o -c lib.ml
ocamlc on -pp camlp4o -c lib.ml
ocamlc lib.cmo -c resolution.ml
ocamlc on lib.cmo -c resolution.ml
File "resolution.ml", line 97, characters 9-14:
Warning 52: Code should not depend on the actual values of
this constructor's arguments. They are only for information
and may change in future versions. (See manual section 8.5)
ocamlc -o prolog lib.cmo resolution.cmo prolog.ml
ocamlc on -o prolog lib.cmo resolution.cmo prolog.ml
'lib.cmi' supprimé
'lib.cmo' supprimé
'prolog.cmi' supprimé
'prolog.cmo' supprimé
'resolution.cmi' supprimé
'resolution.cmo' supprimé
'lib.annot' supprimé
'prolog.annot' supprimé
'resolution.annot' supprimé
```

Le binaire `prolog` ainsi construit est un exécutable OCaml. (pas natif, mais peu importe)

In [42]:
```
cd ..
ls prolog/prolog
file prolog/prolog
```

```
Vous quittez le dossier '/home/lilian/agreg-2017/PROG/tp8/prolog'.
        /!\ Chemin relatif, car le vrai est '/home/lilian/teach/teachensren/AGREG/2
017_2018/PROG/tp8/prolog'.
        Direction ⟹ ..
prolog/prolog*
prolog/prolog: a /home/lilian/.opam/4.04.2/bin/ocamlrun script executable (binary d
ata)
```

Je vous ai aussi envoyé `exemples.zip` :

In [23]:
```
ls -larth exemples.zip
zipinfo exemples.zip
```

```
-rw-rw-r-- 1 lilian lilian 1,5K mars  16 17:17 exemples.zip
Archive:  exemples.zip
Zip file size: 1525 bytes, number of entries: 5
drwxr-xr-x  3.0 unx        0 bx stor 17-Aug-28 16:39 exemples/
-rw-r--r--  3.0 unx      539 tx defX 17-Aug-28 16:34 exemples/domino.pl
-rw-r--r--  3.0 unx      358 tx defX 17-Aug-28 16:34 exemples/genealogie.pl
-rw-r--r--  3.0 unx      774 tx defX 17-Aug-28 16:34 exemples/famille.pl
-rw-r--r--  3.0 unx      250 tx defX 17-Aug-28 16:34 exemples/lapin.pl
5 files, 1921 bytes uncompressed, 693 bytes compressed:  63.9%
```

In [24]:
```
unzip exemples.zip
```

```
Archive:  exemples.zip
   creating: exemples/
  inflating: exemples/domino.pl
  inflating: exemples/genealogie.pl
  inflating: exemples/famille.pl
  inflating: exemples/lapin.pl
```

In [25]:
```
ls -larth exemples/*.pl
```

```
-rw-r--r-- 1 lilian lilian 250 août  28  2017 exemples/lapin.pl
-rw-r--r-- 1 lilian lilian 358 août  28  2017 exemples/genealogie.pl
-rw-r--r-- 1 lilian lilian 774 août  28  2017 exemples/famille.pl
-rw-r--r-- 1 lilian lilian 539 août  28  2017 exemples/domino.pl
```

Par exemple :

In [27]:
```
cat exemples/lapin.pl
```

```
blanc(jeannot).
grandesOreilles(jeannot).
blanc(Y) ⟵ enfant(jeannot,Y).
enfant(X,filsAuYeuxBleus(X)) ⟵ grandesOreilles(X),dentsNonCaries(X).
yeuxBleus(filsAuYeuxBleus(X)) ⟵ grandesOreilles(X),dentsNonCaries(X).
dentsNonCaries(X) ⟵ blanc(X).
```

Premier exemple

`pair.pl` définit les entiers pairs.

In [43]:
```
cd prolog
```

```
Vous quittez le dossier '/home/lilian/agreg-2017/PROG/tp8'.
        /!\ Chemin relatif, car le vrai est '/home/lilian/teach/teachensren/AGREG/2017_2018/PROG/tp8'.
        Direction ⟹ prolog
```

In [44]:
```
cat pair.pl
```

```
pair(o).
pair(s(s(X))) ⟵ pair(X).
```

J'ai modifié le programme `prolog` pour qu'il accepte une requête comme dernier argument après le fichier :

In [67]:
```
./prolog pair.pl "pair(o)."  # une valuation vide : c'est axiomatiquement vrai !
```

```
?- pair(o).
  { }
```

In [68]:
```
./prolog pair.pl "pair(s(o))."  # aucune valuation : c'est faux !
```

```
?- pair(s(o)).
```

In [71]:
```
./prolog pair.pl "pair(s(s(o)))."  # une valuation vide : c'est vrai !
```

```
?- pair(s(s(o))).
  { }
```

Vous pouvez expérimenter dans votre terminal, en faisant simplement `./prolog pair.pl` et en tapant les requêtes. Je recommande l'utilisation de rlwrap (https://github.com/hanslub42/rlwrap) ou ledit (https://opam.ocaml.org/packages/ledit/) pour faciliter l'édition (mais je peux pas montrer ça dans un notebook).

Second exemple

In [72]:
```
./prolog nat.pl "infEq(s(s(o)), s(s(s(o))))."  # 2 <= 3 ? oui
```

```
?- infEq(s(s(o)), s(s(s(o)))).
  { }
```

In [73]:
```
./prolog nat.pl "infEq(s(s(s(s(o)))), s(s(s(o))))."  # 4 <= 3 ? non
```

```
?- infEq(s(s(s(s(o)))), s(s(s(o)))).
```

In [81]:
```
./prolog nat.pl "plus(o,s(o),s(o))."  # 0+1 = 1 ? Oui
```

```
?- plus(o,s(o),s(o)).
  { }
```

In [79]:
```
./prolog nat.pl "plus(s(o),s(o),s(s(o)))."  # 1+1 = 2 ? Oui
```

```
?- plus(s(o),s(o),s(s(o))).
  { }
```

In [80]:
```
./prolog nat.pl "plus(s(o),o,s(s(o)))."  # 1+1 = 1 ? Non
```

```
?- plus(s(o),o,s(s(o))).
```

Vos premiers programmes logiques

Prédicat **`impair`**

In [82]:
```
cat pair.pl
```

```
pair(o).
pair(s(s(X))) ⟵ pair(X).
```

In [83]:
```
echo "impair(s(o))." > impair.pl
echo "impair(s(s(X))) <-- impair(X)." >> impair
```

In [84]:
```
./prolog impair.pl "impair(o)."  # faux
./prolog impair.pl "impair(s(o))."  # vrai
./prolog impair.pl "impair(s(s(o)))."  # faux
```

```
?- impair(o).
?- impair(s(o)).
  { }
?- impair(s(s(o))).
```

Famille

In [172]:
```
rm -vf famille.pl
```

```
'famille.pl' supprimé
```

In [173]:
```
echo "parent(cyrill, renaud)." >> famille.pl
echo "parent(cyrill, claire)." >> famille.pl
echo "parent(renaud, clovis)." >> famille.pl
echo "parent(valentin, olivier)." >> famille.pl
echo "parent(claire, olivier)." >> famille.pl
echo "parent(renaud, claudia)." >> famille.pl
echo "parent(claire, gaelle)." >> famille.pl
```

In [174]:
```
./prolog famille.pl "parent(cyrill, renaud)."  # vrai
./prolog famille.pl "parent(claire, renaud)."  # faux
```

```
?- parent(cyrill, renaud).
  { }
?- parent(claire, renaud).
```

In [175]:
```
./prolog famille.pl "parent(X, renaud)."  # cyrill
./prolog famille.pl "parent(X, gaelle)."  # claire
./prolog famille.pl "parent(X, olivier)."  # claire, valentin

./prolog famille.pl "parent(renaud, X)."  # clovis, claudia
./prolog famille.pl "parent(gaelle, X)."  # {}
./prolog famille.pl "parent(olivier, X)."  # {}
```

```
?- parent(X, renaud).
  { X = cyrill }
?- parent(X, gaelle).
  { X = claire }
?- parent(X, olivier).
  { X = valentin }
  { X = claire }
?- parent(renaud, X).
  { X = clovis }
  { X = claudia }
?- parent(gaelle, X).
?- parent(olivier, X).
```

On définit les frères et sœurs comme ayant un parent en commun, et les cousin-es comme ayant un grand-parent en commun :

In [176]:
```
echo "freresoeur(X,Y) <-- parent(Z,X), parent(Z,Y)." >> famille.pl
echo "grandparent(X,Y) <-- parent(X,Z), parent(Z,Y)." >> famille.pl
echo "cousin(X,Y) <-- grandparent(Z,X), grandparent(Z,Y)." >> famille.pl
```

In [177]:
```
./prolog famille.pl "freresoeur(cyrill, claire)."  # faux
./prolog famille.pl "freresoeur(renaud, claire)."  # vrai
./prolog famille.pl "freresoeur(claire, claire)."  # vrai

./prolog famille.pl "grandparent(X,olivier)."  # cyrill
./prolog famille.pl "grandparent(X,gaelle)."  # cyrill
```

```
?- freresoeur(cyrill, claire).
?- freresoeur(renaud, claire).
  { }
?- freresoeur(claire, claire).
  { }
?- grandparent(X,olivier).
  { X = cyrill }
?- grandparent(X,gaelle).
  { X = cyrill }
```

A vous de trouver une définition récursive de ce prédicat `ancetre` qui fonctionne comme on le souhaite.

In [178]:
```
#echo "ancetre(X,Y) <-- ancetre(X,Z), grandparent(Z,Y)." >> famille.pl
echo "ancetre(X,Y) <-- parent(X,Y)." >> famille.pl
echo "ancetre(X,Y) <-- grandparent(X,Y)." >> famille.pl
#echo "ancetre(X,X)." >> famille.pl
```

On peut vérifier tous les axiomes et règles qu'on a ajouté :

In [179]:
```
cat famille.pl
```

```
parent(cyrill, renaud).
parent(cyrill, claire).
parent(renaud, clovis).
parent(valentin, olivier).
parent(claire, olivier).
parent(renaud, claudia).
parent(claire, gaelle).
freresoeur(X,Y) ⟵ parent(Z,X), parent(Z,Y).
grandparent(X,Y) ⟵ parent(X,Z), parent(Z,Y).
cousin(X,Y) ⟵ grandparent(Z,X), grandparent(Z,Y).
ancetre(X,Y) ⟵ parent(X,Y).
ancetre(X,Y) ⟵ grandparent(X,Y).
```

Questions :

- Les ancêtres d'Olivier sont Valentin, Claire et Cyrill :

In [180]:
```
./prolog famille.pl "parent(X,olivier)."
./prolog famille.pl "grandparent(X,olivier)."
```

```
?- parent(X,olivier).
  { X = valentin }
  { X = claire }
?- grandparent(X,olivier).
  { X = cyrill }
```

In [181]:
```
./prolog famille.pl "ancetre(X,olivier)."
```

```
?- ancetre(X,olivier).
  { X = valentin }
  { X = claire }
  { X = cyrill }
```

- L'ancêtre commun d'Olivier et Renaud est Cyrill :

In [182]:
```
./prolog famille.pl "ancetre(olivier,X),ancetre(renaud,X)."
```

```
?- ancetre(olivier,X),ancetre(renaud,X).
```

In [183]:
```
./prolog famille.pl "ancetre(X,olivier),ancetre(X,renaud)."
```

```
?- ancetre(X,olivier),ancetre(X,renaud).
  { X = cyrill }
```

- Claudia et Gaëlle ne sont pas sœurs, mais elles sont cousines :

In [149]:
```
./prolog famille.pl "freresoeur(gaelle,claudia)."  # faux
./prolog famille.pl "cousin(gaelle,claudia)."  # vrai
```

```
?- freresoeur(gaelle,claudia).
?- cousin(gaelle,claudia).
  { }
```

- Claudia est la sœur de Clovis, et Olivier et Gaëlle sont ces cousins :

In [147]:
```
./prolog famille.pl "freresoeur(X,clovis)."
./prolog famille.pl "cousin(X,clovis)."
```

```
?- freresoeur(X,clovis).
  { X = clovis }
  { X = claudia }
?- cousin(X,clovis).
  { X = clovis }
  { X = claudia }
  { X = olivier }
  { X = gaelle }
```

Listes

Je donne la correction complète directement, ce petit exercice n'aurait pas dû poser de problème :

In [3]:
```
cd exemples
```

```
Vous quittez le dossier '/home/lilian/teach/teachensren/AGREG/2017_2018/PROG/tp8'.
      Direction ⟹ exemples
```

In [2]:
```
cat listes.pl
```

```
liste(nil).
liste(cons(X, Y)) ⟵ liste(Y).

dernier(X, cons(X, L)).
avant_dernier(Y, cons(X, cons(Y, L))).

taille(o, nil).
taille(s(K), cons(X, L)) ⟵ taille(K, L).

element_numero(X, o, cons(X, L)).
element_numero(X, s(K), cons(Y, L)) ⟵ element_numero(X, K, L).

dupliquer(nil, nil).
dupliquer(cons(X, cons(X, L2)), cons(X, L)) ⟵ dupliquer(L2, L).
```

On vérifie que l'on sait tester si des listes sont bien des listes :

In [4]:
```
../prolog/prolog listes.pl "liste(nil)."  # vrai
../prolog/prolog listes.pl "liste(cons(toto, nil))."  # vrai
../prolog/prolog listes.pl "liste(pasliste)."  # faux
../prolog/prolog listes.pl "liste(cons(zorro, pasliste))."  # faux
```

```
?- liste(nil).
  { }
?- liste(cons(toto, nil)).
  { }
?- liste(pasliste).
?- liste(cons(zorro, pasliste)).
```

Maintenant on vérifie qu'on peut accéder au dernier et avant-dernier élément d'une liste :

In [8]:
```
../prolog/prolog listes.pl "dernier(X, nil)."  # {} aucune solution !
../prolog/prolog listes.pl "dernier(X, cons(toto, nil))."  # X = toto
../prolog/prolog listes.pl "avant_dernier(X, cons(zorro, cons(toto, nil)))."  # X = toto
../prolog/prolog listes.pl "avant_dernier(X, cons(titeuf, cons(zorro, cons(toto, nil))))."  # X = zorro
```

```
?- dernier(X, nil).
?- dernier(X, cons(toto, nil)).
  { X = toto }
?- avant_dernier(X, cons(zorro, cons(toto, nil))).
  { X = toto }
?- avant_dernier(X, cons(titeuf, cons(zorro, cons(toto, nil)))).
  { X = zorro }
```

On peut calculer la taille d'une liste (en *unaire*) :

In [9]:
```
../prolog/prolog listes.pl "taille(K, nil)."  # K = o
../prolog/prolog listes.pl "taille(K, cons(toto, nil))."  # K = s(o)
../prolog/prolog listes.pl "taille(K, cons(zorro, cons(toto, nil)))."  # K = s(s(o))
../prolog/prolog listes.pl "taille(K, cons(titeuf, cons(zorro, cons(toto, nil))))."  # K = s(s(s(o)))
```

```
?- taille(K, nil).
  { K = o }
?- taille(K, cons(toto, nil)).
  { K = s(o) }
?- taille(K, cons(zorro, cons(toto, nil))).
  { K = s(s(o)) }
?- taille(K, cons(titeuf, cons(zorro, cons(toto, nil)))).
  { K = s(s(s(o))) }
```

On peut accéder au k-ième élément aussi :

In [10]:
```
../prolog/prolog listes.pl "element_numero(X, o, nil)."  # {} erreur
../prolog/prolog listes.pl "element_numero(X, o, cons(toto, nil))."  # X = toto
../prolog/prolog listes.pl "element_numero(X, s(o), cons(zorro, cons(toto, nil)))."  # X = toto
../prolog/prolog listes.pl "element_numero(X, o, cons(titeuf, cons(zorro, cons(toto, nil))))."  # X = titeuf
```

```
?- element_numero(X, o, nil).
?- element_numero(X, o, cons(toto, nil)).
  { X = toto }
?- element_numero(X, s(o), cons(zorro, cons(toto, nil))).
  { X = toto }
?- element_numero(X, o, cons(titeuf, cons(zorro, cons(toto, nil)))).
  { X = titeuf }
```

On peut accéder au troisième élément d'une liste (s'il existe). Ici la liste est $[1, 2, 3, 4]$ donc le troisième élément (d'indice $2 = s(s(0))$) est 3 :

In [11]:
```
../prolog/prolog listes.pl "element_numero(X, s(s(o)), cons(un, cons(deux, cons(trois, cons(quatre, nil)))))."  # X = titeuf
```

```
?- element_numero(X, s(s(o)), cons(un, cons(deux, cons(trois, cons(quatre, ni
l))))).
  { X = trois }
```

Enfin, on peut facilement dupliquer les éléments d'une liste :

In [12]:
```
../prolog/prolog listes.pl "dupliquer(X, cons(a, cons(b, cons(c, nil))))."  # X = [a, a, b, b, c, c]
```

```
?- dupliquer(X, cons(a, cons(b, cons(c, nil)))).
  { X = cons(a,cons(a,cons(b,cons(b,cons(c,cons(c,nil)))))) }
```

Problème : dominos

Cet exercice serait plus simple si on s'autoriser à utiliser la syntaxe de GNU Prolog pour les listes, mais malheureusement on ne l'a pas implémenté dans notre mini prolog.

On va devoir écrire les concaténations de listes avec un prédicat, un peu comme au dessus. On utiliser $p(a, b)$ pour les paires, et $c(a, u)$ pour la concaténation.

In [2]:
```
cd exemples
```

```
Vous quittez le dossier '/home/lilian/teach/teachensren/AGREG/2017_2018/PROG/tp8'.
        Direction ⟹ exemples
```

In [ ]:
```
[ -f dominos.pl ] && rm -vf dominos.pl
```

In [5]:
```
echo "est_liste(nil)." >> dominos.pl
echo "est_liste(c(X, L)) <-- est_liste(L)." >> dominos.pl

echo "est_paire(p(A, B))." >> dominos.pl
```

Pour l'enchaînement de dominos, ce n'est pas trop difficile : $[(a, b); (c, d); \dots]$ s'enchaîne bien si $b = c$ et si la suite s'enchaîne bien (définition récursive). Les cas de bases sont vrais pour la liste vide et un singleton.

In [ ]:
```
echo "enchainement(nil)." >> dominos.pl
echo "enchainement(c(p(A, B), nil))." >> dominos.pl
echo "enchainement(c(p(Z, X), c(p(X, Y), Q))) <-- enchainement(c(p(X, Y), Q))." >> dominos.pl
```

In [11]:
```
../prolog/prolog dominos.pl "enchainement(nil)."  # vrai
../prolog/prolog dominos.pl "enchainement(c(p(a, b), nil))."  # vrai
../prolog/prolog dominos.pl "enchainement(c(p(a, b), c(p(a, b), nil)))."  # faux
../prolog/prolog dominos.pl "enchainement(c(p(b, a), c(p(a, b), nil)))."  # vrai : b-a a-b s'enchaine bien
```

```
?- enchainement(nil).
  { }
?- enchainement(c(p(a, b), nil)).
  { }
?- enchainement(c(p(a, b), c(p(a, b), nil))).
?- enchainement(c(p(b, a), c(p(a, b), nil))).
  { }
```

Maintenant l'insertion, qui teste si `l2` peut s'obtenir par une insertion de `x` **quelque part** dans `l1` (et pas uniquement en tête ou en queue de `l1`).

In [12]:
```
echo "insere(X, L, c(X, L))." >> dominos.pl
echo "insere(X, c(T, Q1), c(T, Q2)) <-- insere(X, Q1, Q2)." >> dominos.pl
```

On a évidemment $n + 1$ possibilités pour insérer `a` si `l1` est de taille n :

In [23]:
```
../prolog/prolog dominos.pl "insere(a, c(b, c(d, nil)), L2)."  # 2+1 possibilités
```

```
?- insere(a, c(b, c(d, nil)), L2).
  { L2 = c(a,c(b,c(d,nil))) }
  { L2 = c(b,c(a,c(d,nil))) }
  { L2 = c(b,c(d,c(a,nil))) }
```

Pour les permutations, ce n'est pas tellement différent. On teste si L peut être obtenue par permutation depuis $T :: Q$ en testant si Q est obtenue par permutation d'une certaine liste L_2 et si T peut être inséré dans L_2 pour donner L.

In [37]:
```
echo "perm(nil, nil)." >> dominos.pl
echo "perm(L, c(T, Q)) <-- insere(T, L2, L), perm(L2, Q)." >> dominos.pl
```

In [39]:
```
../prolog/prolog dominos.pl "perm(c(a,c(b,nil)), X)."  # [a;b] et [b;a]
```

```
?- perm(c(a,c(b,nil)), X).
  { X = c(a,c(b,nil)) }
  { X = c(b,c(a,nil)) }
```

In [46]:
```
../prolog/prolog dominos.pl "perm(c(a,c(b,c(d,nil))), X)."  # 6 = 3! possibilités, toutes montrées
```

```
?- perm(c(a,c(b,c(d,nil))), X).
  { X = c(a,c(b,c(d,nil))) }
  { X = c(a,c(d,c(b,nil))) }
  { X = c(b,c(a,c(d,nil))) }
  { X = c(b,c(d,c(a,nil))) }
  { X = c(d,c(a,c(b,nil))) }
  { X = c(d,c(b,c(a,nil))) }
```

Pour les permutations de $[a; b; c; d]$, on devrait trouver $24 = 4!$ possibilités :

In [49]:
```
../prolog/prolog dominos.pl "perm(c(u,c(v,c(w,c(x,nil)))), X)."  # 24 = 4! possibilités, pas toutes montrées ?!
```

```
?- perm(c(u,c(v,c(w,c(x,nil)))), X).
  { X = c(u,c(v,c(w,c(x,nil)))) }
  { X = c(u,c(v,c(x,c(w,nil)))) }
```

Pour les arrangements, c'est similaire mais on considère aussi la possibilité d'inverser un domino (*i.e.,* $(a, b) \mapsto (b, a)$) :

In [57]:
```
echo "miroir(p(A, B), p(B, A))." >> dominos.pl
```

In [90]:
```
../prolog/prolog dominos.pl "miroir(p(u,v), X)."  # X = p(v,u)
```

```
?- miroir(p(u,v), X).
  { X = p(v,u) }
```

In [91]:
```
echo "arrangement(nil, nil)." >> dominos.pl
echo "arrangement(L, c(T,Q)) <-- insere(T, L2, L), arrangement(L2, Q)." >> dominos.pl
echo "arrangement(L, c(T,Q)) <-- miroir(T2, T), insere(T2, L2, L), arrangement(L2, Q)." >> dominos.pl
```

Les arrangements de petites listes ne donne pas grand chose :

In [92]:
```
../prolog/prolog dominos.pl "arrangement(nil, L)."  # L = nil
../prolog/prolog dominos.pl "arrangement(c(a,nil)), L)."  # rien
```

```
?- arrangement(nil, L).
  { L = nil }
?- arrangement(c(a,nil)), L).
```

In [93]:
```
../prolog/prolog dominos.pl "arrangement(c(a,c(b,nil)), X)."  # X = [a;b] ou [b;a]
```

```
?- arrangement(c(a,c(b,nil)), X).
  { X = c(a,c(b,nil)) }
  { X = c(b,c(a,nil)) }
```

Mais avec trois éléments ou plus :

In [94]:
```
../prolog/prolog dominos.pl "arrangement(c(a,c(b,c(d,nil))), X)."  # 6 réponses
```

```
?- arrangement(c(a,c(b,c(d,nil))), X).
  { X = c(a,c(b,c(d,nil))) }
  { X = c(a,c(d,c(b,nil))) }
  { X = c(b,c(a,c(d,nil))) }
  { X = c(b,c(d,c(a,nil))) }
  { X = c(d,c(a,c(b,nil))) }
  { X = c(d,c(b,c(a,nil))) }
```

In [96]:
```
# X = [(u,v);(w,u)] ou [(w,u);(u,v)] avec 0 miroir
# ou X = [(v,u);(w,u)] ou [(w,u);(v,u)] avec 1 miroir sur (u,v)
# ou X = [(u,v);(u,w)] ou [(u,w);(u,v)] avec 1 miroir sur (w,u)
# ou X = [(v,u);(u,w)] ou [(u,w);(v,u)] avec 2 miroirs
../prolog/prolog dominos.pl "arrangement(c(p(u,v),c(p(w,u),nil)), X)."
```

```
?- arrangement(c(p(u,v),c(p(w,u),nil)), X).
  { X = c(p(u,v),c(p(w,u),nil)) }
  { X = c(p(u,v),c(p(u,w),nil)) }
  { X = c(p(w,u),c(p(u,v),nil)) }
  { X = c(p(w,u),c(p(v,u),nil)) }
  { X = c(p(v,u),c(p(w,u),nil)) }
  { X = c(p(v,u),c(p(u,w),nil)) }
  { X = c(p(u,w),c(p(u,v),nil)) }
  { X = c(p(u,w),c(p(v,u),nil)) }
```

Maintenant on peut résoudre le problème, avec $u, v, w, x = 1, 2, 3, 4$

In [67]:
```
echo "quasisolution(L1, L2) <-- perm(L1, L2), enchainement(L2)." >> dominos.pl
echo "solution(L1, L2) <-- arrangement(L1, L2), enchainement(L2)." >> dominos.pl
```

On peut ordonner $[(1,2);(3,1);(2,4)]$ en $[(3,1);(1,2);(2,4)]$ si on ignore les miroirs :

In [102]:
```
../prolog/prolog dominos.pl "quasisolution(c(p(un,deux),c(p(trois,un),c(p(deux,quatre),nil))), L)."
```

```
?- quasisolution(c(p(un,deux),c(p(trois,un),c(p(deux,quatre),nil))), L).
  { L = c(p(trois,un),c(p(un,deux),c(p(deux,quatre),nil))) }
```

On peut ordonner $[(1,2);(3,1);(2,4)]$ en $[(3,1);(1,2);(2,4)]$ mais aussi en $[(4,2);(2,1);(1,3)]$ car on a le droit de tourner les dominos !

In [101]:
```
../prolog/prolog dominos.pl "solution(c(p(un,deux),c(p(trois,un),c(p(deux,quatre),nil))), L)."
```

```
?- solution(c(p(un,deux),c(p(trois,un),c(p(deux,quatre),nil))), L).
  { L = c(p(trois,un),c(p(un,deux),c(p(deux,quatre),nil))) }
  { L = c(p(quatre,deux),c(p(deux,un),c(p(un,trois),nil))) }
```

Comparaison avec GNU Prolog

La solution avec GNU Prolog serait :

In [3]:
```
cat domino.pl
```

```
enchainement([]).
enchainement([_]).
enchainement([ [_, X] | [[X, Y]| Q] ]) :- enchainement([[X, Y]|Q]).

insere(X, L, [X|L]).
insere(X, [T|Q1], [T|Q2]) :- insere(X, Q1, Q2).

miroir([X, Y], [Y, X]).

perm([], []).
perm(L, [T|Q]) :- perm(L2, Q), insere(T, L2, L).

assemblage([], []).
assemblage(L, [T|Q]) :- assemblage(L2, Q), insere(T, L2, L).
assemblage(L, [T|Q]) :- assemblage(L2, Q), miroir(T2, T), insere(T2, L2, L).

quasisolution(L1, L2) :- perm(L1, L2), enchainement(L1).
solution(L1, L2) :- assemblage(L1, L2), enchainement(L1).
```

Il faut la comparer à notre solution, un peu plus verbeuse à cause de l'absence de syntaxe spécifique aux listes et aux paires, mais qui encode la même logique.

In [103]:
```
cat dominos.pl
```

```
est_liste(nil).
est_liste(c(X, L)) ⟵ est_liste(L).
est_paire(p(A, B)).

enchainement(nil).
enchainement(c(p(A, B), nil)).
enchainement(c(p(Z, X), c(p(X, Y), Q))) ⟵ enchainement(c(p(X, Y), Q)).

insere(X, L, c(X, L)).
insere(X, c(T, Q1), c(T, Q2)) ⟵ insere(X, Q1, Q2).

perm(nil, nil).
perm(L, c(T, Q)) ⟵ insere(T, L2, L), perm(L2, Q).

miroir(p(A, B), p(B, A)).

arrangement(nil, nil).
arrangement(L, c(T,Q)) ⟵ insere(T, L2, L), arrangement(L2, Q).
arrangement(L, c(T,Q)) ⟵ miroir(T2, T), insere(T2, L2, L), arrangement(L2, Q).

quasisolution(L1, L2) ⟵ perm(L1, L2), enchainement(L2).

solution(L1, L2) ⟵ arrangement(L1, L2), enchainement(L2).
```

Conclusion

Fin. C'était le dernier TP.

- Essayez de travailler un peu toutes les notions vues dans les 8 TPs cette année pour vous entraîner aux oraux.
- Pour le plus grand nombre possible de texte d'annales de modélisations, essayez d'implémenter parfaitement la question de programmation obligatoire, **avec des tests et des exemples**, et essayez de faire un ou deux bonus pour chaque texte.