— Cours d'Informatique S9 —

Le langage Prolog

Travaux Dirigés

Jacques TISSEAU

Ecole Nationale d'Ingénieurs de Brest
Technopôle Brest-Iroise
CS 73862 – 29238 Brest cedex 3 – France
tisseau@enib.fr

Avec la participation de Pierre De Loor, Pierre-Alexandre Favier et Ahmed Naim.

Ce document regroupe l'ensemble des exercices du cours de programmation en logique du $9^{ème}$ semestre (S9) de l'Ecole Nationale d'Ingénieurs de Brest (ENIB : www.enib.fr). Il accompagne les notes de cours « Le langage Prolog ».

Tisseau J.,*Le langage prolog*, ENIB, cours d'Informatique S9, Brest, 1990-2010.

version du 15 mai 2010

Sommaire

Avant-propos

Ce document regroupe les exercices (dans la partie I) proposés dans le cadre du cours de « Programmation en Logique » de l'ENIB ainsi que les corrigés associés (dans la partie II). La version actuelle sera prochainement complétée par 2 nouveaux chapitres : jeux et interpréteurs.

Les solutions Prolog correspondant aux exercices de ce document ont été testées avec l'interpréteur du « domaine public » SWI-Prolog (`www.swi-prolog.org`) issu des travaux de l'Université d'Amsterdam.

D'autres implémentations, également « open source », sont disponibles et compatibles avec la norme ISO-Prolog (ISO/IEC 13211-1, [6]) :

Ciao Prolog	: `www.clip.dia.fi.upm.es/Software/Ciao`	(Université de Madrid)
GNU Prolog	: www.gprolog.org	(INRIA)
YAP Prolog	: `www.dcc.fc.up.pt/~vsc/Yap`	(Université de Porto)

Pour démarrer SWI-Prolog sous `Linux` avec un interpréteur de commandes (un *shell*), il suffit de lancer la commande `pl` pour voir le prompt (`?-`) de l'interpréteur s'afficher.

```
$ pl
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.64)
Copyright (c) 1990-2008 University of Amsterdam. SWI-Prolog comes
with ABSOLUTELY NO WARRANTY. This is free software, and you are
welcome to redistribute it under certain conditions. Please visit
http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Pour sortir de l'interpréteur SWI-Prolog, il suffit d'appeler le prédicat `halt/0`.

```
?- halt.
$
```

A partir des énoncés du TD 3, les arguments des prédicats Prolog à définir sont systématiquement précédés de leur mode d'instantiation, à savoir :

mode `+` : l'argument doit être instantié à l'appel (mode *input*),

mode `-` : l'argument doit être libre à l'appel (mode *output*) et

mode `?` : l'argument peut être libre ou instantié à l'appel (mode *input-output*).

Exemples :

– « Définir le prédicat `unique(?X,+L)` où `X` n'apparaît qu'une seule fois dans la liste `L`. »
Ainsi le prédicat `unique(?X,+L)` peut être appelé de 2 ($= 2^1$) manières différentes : `X` et `L` connus ou `X` inconnu et `L` connu.

```
?- unique(b,[a,b,c,a]).
true ;
false.
```

```
?- unique(X,[a,b,c,a]).
X = b ;
X = c ;
false.
```

– « Définir le prédicat `occurences(?X,?N,+L)` où `X` apparaît `N` fois dans la liste `L`. »
Ainsi le prédicat `occurences(?X,?N,+L)` peut être appelé de 4 ($= 2^2$) manières différentes : `X`, `N` et `L` connus, `X` connu, `N` inconnu et `L` connu, `X` inconnu, `N` et `L` connus, `X` et `N` inconnus et `L` connus.

```
?- occurences(a,3,[a,b,a,c,b,a]).
true ;
false.

?- occurences(a,X,[a,b,a,c,b,a]).
X = 3 ;
false.
```

```
?- occurences(X,3,[a,b,a,c,b,a]).
X = a ;
false.

?- occurences(X,Y,[a,b,a,c,b,a]).
X = a, Y = 3 ;
X = b, Y = 2 ;
X = c, Y = 1 ;
false.
```

On trouvera en annexe (partie III) la liste de tous les exercices (page 111), la liste des listings (page 114) ainsi qu'une liste de références bibliographiques (page 115).

Première partie

Enoncés

TD 1

De la logique à Prolog

TD 1.1 : Puzzle logique

Trois personnes, de nationalités différentes (anglaise, espagnole et française) et pratiquant des sports différents (football, natation et tennis), habitent dans trois maisons de couleurs distinctes (blanc, bleu, vert). Ces trois maisons sont situées dans la même rue ; une des maisons est située au début de la rue, une autre au milieu, et la troisième au bout de la rue. Chacune des 3 maisons est donc caractérisée par un quadruplet `(E,C,N,S)`, où `E` est l'emplacement de la maison dans la rue, `C` la couleur de la maison, `N` et `S` la nationalité et le sport pratiqué par son occupant. On dispose des 5 indices suivants :

- Dans la maison verte on pratique la natation.
- La maison verte est située avant la maison de l'espagnol.
- L'anglais habite la maison blanche.
- La maison blanche est située avant la maison où on pratique le football.
- Le tennisman habite au début de la rue.

Déterminer les caractéristiques de chacune des 3 maisons.

▽ 55

TD 1.2 : Formes clausales

Pour chacune des expressions logiques suivantes :

- $\forall x\, P(x)$
- $\forall x\, (P(x) \Rightarrow Q(x))$
- $\forall x\, (\exists y\, (P(x) \Rightarrow Q(x, f(y))))$
- $\forall x, y\, (P(x) \Rightarrow ((Q(x, y) \Rightarrow \neg(\exists u\, P(f(u)))) \vee (Q(x, y) \Rightarrow \neg R(y))))$
- $\forall x\, (\, P(x) \Rightarrow (\, (\neg\forall y\, (Q(x, y) \Rightarrow (\exists u\, P(f(u))))) \wedge (\forall y\, (Q(x, y) \Rightarrow P(x)))\,)\,)$

1. Mettre l'expression logique sous forme clausale en respectant les 6 étapes suivantes :
   (a) élimination des $\Rightarrow$
   (b) déplacement des $\neg$ vers l'intérieur
   (c) élimination des $\exists$ (“skolémisation”)
   (d) déplacement des $\forall$ vers l'extérieur
   (e) distribution de $\wedge$ sur $\vee$
   (f) mise en clauses
2. Ecrire la forme clausale obtenue sous forme de clauses Prolog.

▽ 55

TD 1.3 : Des relations aux prédicats

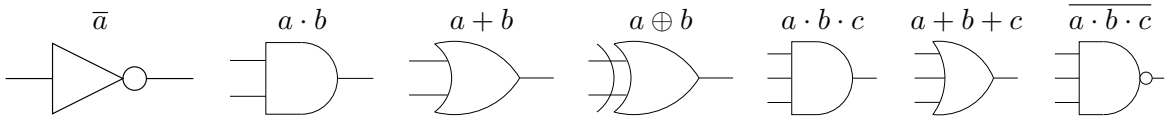
Traduire en faits Prolog les affirmations suivantes :

1. Thomson est une entreprise dynamique.
2. L'ENIB est une école d'ingénieurs.
3. La voisine aime les chats.
4. Le voisin aime les chiens.
5. Le champ magnétique est à flux conservatif.
6. Le facteur sonne toujours deux fois.
7. Mozart est l'auteur de « La flûte enchantée ».
8. Molière est l'auteur de « L'avare ».

▽ 57

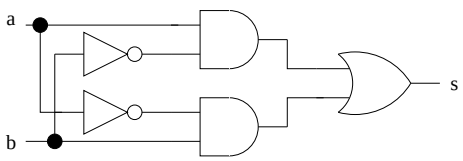
TD 1.4 : Algèbre de Boole

1. Traduire par des faits Prolog les tables de vérité des opérateurs logiques de base : la conjonction (`et/3` : $a\cdot$ b), la disjonction (`ou/3` : $a+b$) et la négation (`non/2` : $\overline{a}$).
2. Définir les opérateur `xor/3` ($a \oplus b$), `nor/3` ($\overline{a+b}$), `nand/3` ($\overline{a\cdot b}$), `imply/3` ($a \Rightarrow b$) et `equiv/3` ($a \Leftrightarrow b$), à partir des opérateur de base `et/3` ($a\cdot$ b), `ou/3` ($a+b$) et `non/2` ($\overline{a}$).
3. Vérifier les principales propriétés des opérateurs logiques :
   (a) `Commutativité` : $X\cdot Y = Y\cdot X$ et $X+Y = Y+X$
   (b) `Associativité` : $X\cdot(Y\cdot Z) = (X\cdot Y)\cdot Z$ et $X+(Y+Z) = (X+Y)+Z$
   (c) `Distributivité` : $X\cdot(Y+Z) = (X\cdot Y)+(X\cdot Z)$ et $X+(Y\cdot Z) = (X+Y)\cdot(X+Z)$
   (d) `Idempotence` : $X+X = X$ et $X\cdot X = X$
   (e) `Complémentarité` : $X+\overline{X} = 1$ et $X\cdot\overline{X} = 0$
   (f) `Théorèmes de De Morgan` : $\overline{X\cdot Y} = \overline{X}+\overline{Y}$ et $\overline{X+Y} = \overline{X}\cdot\overline{Y}$
4. On considère les conventions graphiques traditionnelles pour les opérateurs logiques :

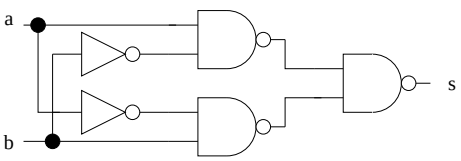


Définir les prédicats correspondant aux circuits logiques suivants.

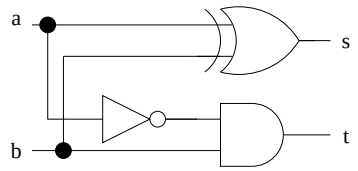
(a) a et b sont les entrées, s la sortie.



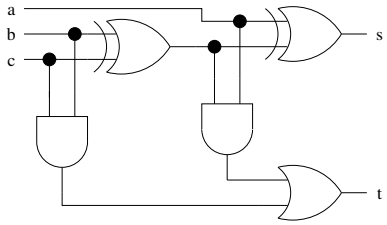
(a) a et b sont les entrées, s la sortie.



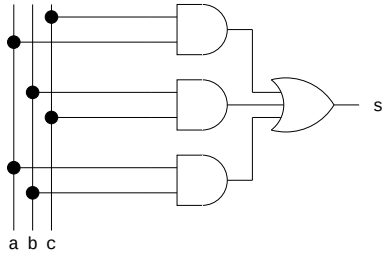
(a) a et b sont les entrées, s et t les sorties.



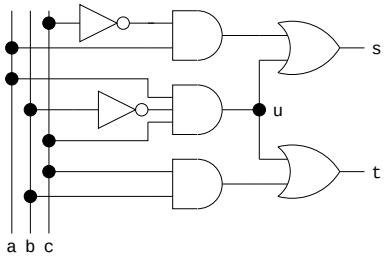
(b) a, b et c sont les entrées, s et t les sorties.



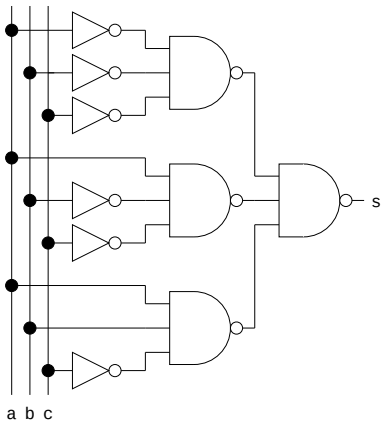
(c) a, b et c sont les entrées et s la sortie.



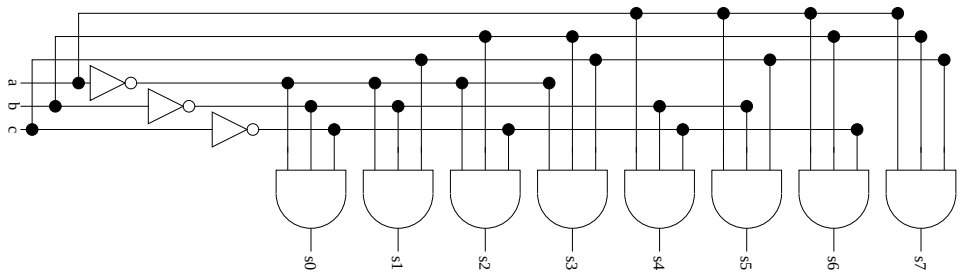
(a) a, b et c sont les entrées, s et t les sorties.



(b) a, b et c sont les entrées, s la sortie.



(a) a, b et c sont les entrées, s_0,s_1...s_7 les sorties.



▽ 57

TD 1.5 : Une base de données Air-Enib

On considère une base de faits décrivant des vols de la compagnie Air-Enib :

```
vol(numero du vol,
    ville de depart,
    ville d'arrivee,
    heure de depart,
    heure d'arrivee,
    nombre de passagers)
```

1. Décrire plusieurs vols selon le format précédent.
2. Quelles questions Prolog doit–on poser pour déterminer la liste des vols (identifiés par leur numéro)
   (a) au départ d'une ville donnée ?
   (b) qui arrivent dans une ville donnée ?
   (c) au départ d'une ville donnée et qui partent avant midi ?
   (d) arrivant dans une ville donnée à partir de 14 h ?
   (e) qui ont plus de 100 passagers et qui arrivent dans une ville donnée avant 17 h ?
   (f) qui partent à la même heure de deux villes différentes ?
   (g) qui durent plus de deux heures ?

▽ 59

TD 1.6 : Un petit réseau routier

On considère un réseau routier décrit par la base de faits suivante :

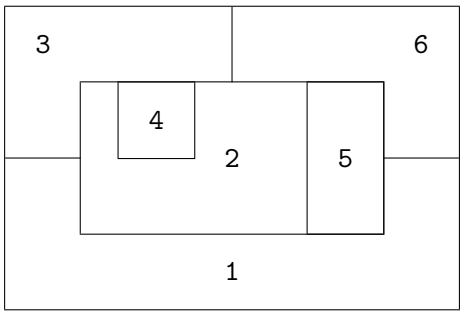
```
route(s,a).  % une route relie la ville s a la ville a
route(s,d).
route(a,b).
route(a,d).
route(b,c).
route(b,e).
route(d,e).
```

1. Représenter le réseau ainsi défini.
2. Quels types de questions peut-on poser lors de la consultation de cette base de faits ?
3. Définir le prédicat
   `voisines(X,Y)` *% Il existe une route directe entre* `X` *et* `Y`
4. Tracer l'arbre de résolution pour la question
   `?- voisines(d,X).`

▽ 59

TD 1.7 : Coloriage d'une carte

On dispose de 4 couleurs (rouge,jaune,vert,bleu) pour colorier la carte représentée ci-dessous.



Ecrire un programme Prolog qui permet d'associer une couleur (rouge, jaune, vert, bleu) à une région (1,2,3,4,5,6) de telle manière que deux régions adjacentes ne soient pas de la même couleur.

▽ 60

TD 1.8 : Arbre généalogique

On considère un arbre généalogique décrit par la base de faits suivante :

```
% homme/1
  homme(jean).                                          % jean est un homme
  homme(fabien). homme(jerome). homme(emile).
  homme(franck). homme(bruno). homme(marc).
% femme/1
  femme(evelyne).                                       % évelyne est une femme
  femme(louise). femme(julie). femme(anne). femme(aurelie).
  femme(sophie). femme(marie). femme(eve).
% pere/2
  pere(emile,jean).                                     % émile est le père de jean
  pere(jean,fabien). pere(fabien,eve). pere(jean,jerome).
  pere(bruno,evelyne). pere(bruno,franck). pere(franck,sophie).
  pere(franck,aurelie).
% mere/2
  mere(louise,jean).                                    % louise est la mère de jean
  mere(julie,evelyne). mere(julie,franck).
  mere(evelyne,fabien). mere(evelyne,jerome).
  mere(anne,sophie). mere(anne,aurelie).
  mere(marie,eve). mere(sophie,marc).
```

1. Représenter l'arbre généalogique ainsi décrit.
2. Définir les prédicats suivants :
   (a) `parent(X,Y)` % *X est un des parents de Y*
   (b) `fils(X,Y)` % *X est le fils de Y*
   (c) `fille(X,Y)` % *X est la fille de Y*
   (d) `femme(X,Y)` % *X est la femme de Y*
   (e) `mari(X,Y)` % *X est le mari de Y*
   (f) `gdPere(X,Y)` % *X est le grand-pere de Y*
   (g) `gdMere(X,Y)` % *X est la grand-mere de Y*
   (h) `gdParent(X,Y)` % *X est un des grand-parents de Y*
   (i) `aGdPere(X,Y)` % *X est l'arriere grand-pere de Y*
   (j) `aGdMere(X,Y)` % *X est l'arriere grand-mere de Y*
   (k) `frereSoeur(X,Y)` % *X est le frere ou la soeur de Y*
   (l) `frere(X,Y)` % *X est le frere de Y*
   (m) `soeur(X,Y)` % *X est la soeur de Y*
   (n) `beauFrere(X,Y)` % *X est le beau-frere de Y*
   (o) `belleSoeur(X,Y)` % *X est la belle-soeur de Y*
   (p) `ancetre(X,Y)` % *X est l'ancetre de Y*

▽ 60

TD 2

Termes

TD 2.1 : Etat-civil

1. Traduire par un fait Prolog une fiche d'état-civil du type :

état–civil	nom	NGAOUNDERE
	prénom	Richard
	date de naissance	15/02/1960
	nationalité	camerounaise
	sexe	masculin
adresse	rue	4 rue Leclerc
	ville	Brest
	département	Finistère (29)

2. On suppose que la base de faits contient un certain nombre de fiches d'état-civil.
   Traduire par des buts Prolog les questions suivantes :
   (a) Quels sont les individus (nom,prénom) de nationalité française ?
   (b) Quels sont les individus (nom,prénom,nationalité) de nationalité étrangère ?
   (c) Quels sont les individus (nom,prénom,nationalité) de nationalité étrangère habitant Brest dans le Finistère ?
   (d) Y a-t-il des individus habitant la même adresse ?

▽ 63

TD 2.2 : Entiers naturels

Traduire en Prolog les axiomes[1] définissant les nombres entiers non négatifs ($\mathbb{N}$) et les principales opérations les concernant.

Le vocabulaire initial de la théorie des entiers non négatifs comprend :
- une constante z qui représente l'entier 0 (zéro) ;
- une fonction unaire $s(X)$ qui traduit la notion de successeur d'un entier X ;
- un prédicat unaire `entier(X)` (X *est un entier*).

1. **Génération :** $z \in \mathbb{N}$ et $\forall x \in \mathbb{N}, s(x) \in \mathbb{N}$
2. **Addition :** $\forall x \in \mathbb{N}, x + z = x$ et $\forall x, y \in \mathbb{N}, x + s(y) = s(x + y)$
3. **Multiplication :** $\forall x \in \mathbb{N}, x \cdot z = z$ et $\forall x, y \in \mathbb{N}, x \cdot s(y) = x \cdot y + x$
4. **Exponentiation :** $\forall x \in \mathbb{N}, x^z = s(z)$ et $\forall x, y \in \mathbb{N}, x^{s(y)} = x^y \cdot x$

[1] voir par exemple : Manna Z., Waldinger R., (1985) *The logical basis for computer programming*, Addison-Weslay.

5. **Prédécesseur :** $\forall x \in \mathbb{N}, p(s(x)) = x$
6. **Soustraction :** $\forall x \in \mathbb{N}, x - z = x$ et $\forall x, y \in \mathbb{N}, s(x) - s(y) = x - y$
7. **Inférieur ou égal :** $\forall x \in \mathbb{N}, x \leq z \Leftrightarrow x = z$ et $\forall x, y \in \mathbb{N}, x \leq y \Leftrightarrow x = y$ ou $x \leq p(y)$
8. **Strictement inférieur :** $\forall x, y \in \mathbb{N}, x < y \Leftrightarrow x \leq y$ et $x \neq y$
9. **Quotient et reste :**

$$\forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, x < y \Rightarrow quot(x, y) = 0 \text{ et } quot(x + y, y) = s(quot(x, y))$$

$$\forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, x < y \Rightarrow rest(x, y) = x \text{ et } rest(x + y, y) = rest(x, y)$$

10. **Division :** $\forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, x \div y \Leftrightarrow \exists q \in \mathbb{N}^*$ tel que $x \cdot q = y$
11. **Plus grand commun diviseur :**

$$\forall x \in \mathbb{N}, pgcd(x, z) = x \text{ et } \forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, pgcd(x, y) = pgcd(y, rest(x, y))$$

12. **Factorielle :** $z! = s(z)$ et $\forall x \in \mathbb{N}, s(x)! = s(x) \cdot x!$

▽ 63

TD 2.3 : Polynômes

Définir le prédicat `polynome(+Expression,+Variable)` vrai si et seulement si l'`Expression` est un polynôme par rapport à la `Variable` (variable prise au sens mathématique du terme).
Exemples :

```
?- polynome(3*x^2 + 2*x,x).
true.
?- polynome(3*x^2 + 2*x,y).
false.
?- polynome(3*cos(x)^2 + 1,x).
false.
?- polynome(3*cos(x)^2 + 1,cos(x)).
true.
```

▽ 64

TD 2.4 : Arithmétique

1. Définir sous forme de prédicats Prolog les fonctions mathématiques suivantes :
   (a) `abs(X,Y)` % $Y = \mid X \mid$
   (b) `min(X,Y,Z)` % $Z = \min(X, Y)$
   (c) `pgcd(X,Y,Pgcd)` % $Pgcd = pgcd(X, Y)$
   (d) `ppcm(X,Y,Ppcm)` % $Ppcm = ppcm(X, Y)$
   (e) `ch(X,Y)` % $Y = ch(X)$
2. Définir les prédicats Prolog correspondant aux suites récurentes suivantes :
   (a) **Suite factorielle :** $u_0 = 1$ et $u_n = n \cdot u_{n-1}$ $\forall n \in \mathbb{N}^*$
   (b) **Suite de Fibonacci :** $u_0 = 1$, $u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$ $\forall n \in \mathbb{N}, n > 1$
3. Un nombre complexe sera représenté ici par un terme binaire `c(Reel,Imaginaire)`.
   Définir les opérations suivantes sur les nombres complexes :
   (a) `argC(Z,Arg)` % $Arg = \arg(Z)$
   (b) `modC(Z,Mod)` % $Mod = \mid Z \mid$

(c) `addC(Z1,Z2,Z)` % $Z = Z_1 + Z_2$

(d) `mulC(Z1,Z2,Z)` % $Z = Z_1 \cdot Z_2$

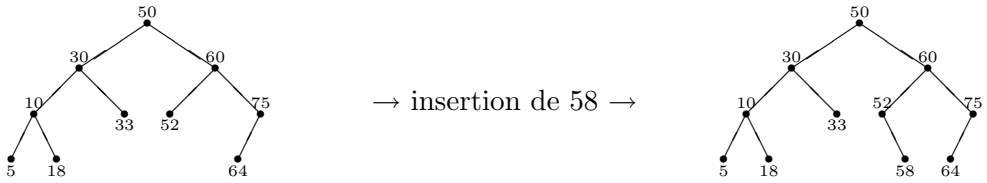
(e) `conj(Z,Z1)` % $Z_1 = \overline{Z}$

4. Définir le prédicat `dans(I,Min,Max)` vrai si et seulement si :
$\forall I, Min, Max \in \mathbb{N},\ Min \leq I \leq Max$

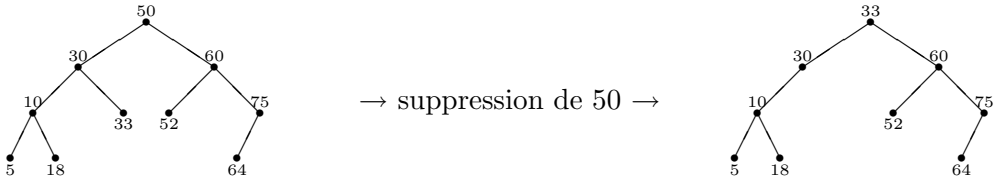
▽ 64

TD 2.5 : Arbres et dictionnaires binaires

1. **Arbres binaires :** structure soit vide, soit composée de trois éléments :
   – une racine `Val`,
   – un sous–arbre binaire `Gauche`,
   – un sous–arbre binaire `Droite`.
   Un arbre binaire vide sera représenté ici par le terme atomique `[]`, un arbre non vide par le terme composé `bt(Gauche,Val,Droite)` (`bt` comme *binary tree*).
   Exemple : `bt(bt(bt([],3,[]),5,bt([],7,[])),9,bt([],12,[]))`
   Définir les prédicats suivants concernant les arbres binaires :
   (a) `arbreBinaire(T)` % `T` *est un arbre binaire*
   (b) `dansArbre(X,A)` % `X` *est un élément de l'arbre binaire* `A`
   (c) `profondeur(A,N)` % `N` *est la profondeur de l'arbre binaire* `A`
2. **Dictionnaires binaires :** arbre binaire `bt(Gauche,Val,Droite)` ordonné tel que :
   – tous les nœuds de `Gauche` ont une valeur strictement inférieure à `Val` ;
   – tous les nœuds de `Droite` ont une valeur strictement supérieure à `Val` ;
   – `Gauche` et `Droite` sont eux-mêmes des dictionnaires binaires.
   Définir les prédicats suivants concernant les dictionnaires binaires :
   (a) `dicoBinaire(T)` % `T` *est un dictionnaire binaire*
   (b) `inserer(X,D1,D2)` % `X` *est inséré dans le dictionnaire* `D1` *pour donner* `D2`
   Exemple :



   (c) `supprimer(X,D1,D2)` % `X` *est supprimé du dictionnaire* `D1` *pour donner* `D2`
   Exemple :



▽ 66

TD 2.6 : Termes de base

1. Définir le prédicat `base(?Terme)` vrai si et seulement si `Terme` est un terme de base ; c'est–à–dire soit un terme atomique, soit un terme composé dont tous les arguments sont des termes de base.

2. Définir le prédicat `sousTerme(?SousTerme,+Terme)` vrai si et seulement si `SousTerme` est un sous-terme de `Terme`.
3. Définir le prédicat `substituer(?Ancien,?Nouveau,+Terme,?NouveauTerme)` vrai si et seulement si `NouveauTerme` correspond au `Terme` dans lequel toutes les occurences du sous–terme `Ancien` ont été remplacées par le sous-terme `Nouveau`.

▽ 66

TD 2.7 : Opérateurs

1. Représenter sous forme d'arbres les expressions suivantes :
   (a) `2 * a + b * c`
   (b) `2 * (a + b) * c`
   (c) `2 * (a + b * c)`
   (d) `((2 * a) + b) * c`
   (e) `2 + a + b + c`
2. Définir les opérateurs Prolog nécessaires pour que les phrases suivantes soient des termes Prolog autorisés :

```
diane est la secretaire de la mairie de brest.
pierre est le maire de brest.
jean est le gardien de but des verts.
```

3. Définir les opérateurs Prolog nécessaires pour que l'écriture de règles puissent se faire de la manière suivante :

```
regle r1 avant                    regle r2 arriere
       si a et b ou c                    si f ou g
       alors d et e.                     alors h.
```

▽ 67

TD 3

Listes

TD 3.1 : Représentations de listes

1. Représenter par un arbre la liste Prolog `[a,b,c]`.
2. Ecrire de toutes les manières possibles la liste `[a,b,c]`.

▽ 69

TD 3.2 : Informations sur les listes

Définir les prédicats suivants :

1. `liste(?L)` : `L` est une liste.

```
?- liste([a,b,c]).
true.
?- liste([a,b|c]).
false.
?- liste([a,b|[c]]).
true.
```

```
?- liste(L).
L = [] ;
L = [_G504] ;
L = [_G504, _G507] ;
L = [_G504, _G507, _G510] ;
...
```

2. `premier(?X,?L)` : `X` est le premier élément de la liste `L`.

```
?- premier(a,[a,b,c]).
true.
?- premier(a,L).
L = [a|_G507].
```

```
?- premier(X,[a,b,c]).
X = a.
?- premier(X,L).
L = [X|_G522].
```

3. `dernier(?X,?L)` : `X` est le dernier élément de la liste `L`.

```
?- dernier(c,[a,b,c]).
true ;
false.
?- dernier(c,L).
L = [c] ;
L = [_G506, c] ;
L = [_G506, _G509, c] ;
...
```

```
?- dernier(X,[a,b,c]).
X = c ;
false.
?- dernier(X,L).
L = [X] ;
L = [_G521, X] ;
L = [_G521, _G524, X] ;
...
```

4. `nieme(?N,?X,+L)` : `X` est le $N^{ème}$ élément de la liste `L`.

```
?- nieme(2,b,[a,b,c]).
true ;
false.
?- nieme(2,X,[a,b,c]).
X = b ;
false.
```

```
?- nieme(N,b,[a,b,c]).
N = 2 ;
false.
?- nieme(N,X,[a,b,c]).
N = 1, X = a ;
N = 2, X = b ;
N = 3, X = c ;
false.
```

5. `longueur(?N,+L)` : `N` est le nombre d'éléments de la liste `L`.

```
?- longueur(3,[a,b,c]).
true ;
false.
```

```
?- longueur(N,[a,b,c]).
N = 3 ;
false.
```

6. `dans(?X,?L)` : `X` appartient à la liste `L`.

```
?- dans(b,[a,b,c]).
true ;
false.
?- dans(b,L).
L = [b|_G275] ;
L = [_G274, b|_G278] ;
L = [_G274, _G277, b|_G281] ;
...
```

```
?- dans(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
false.
?- dans(X,L).
L = [X|_G290] ;
L = [_G289, X|_G293] ;
L = [_G289, _G292, X|_G296] ;
...
```

7. `hors(+X,+L)` : `X` n'appartient pas à la liste `L`.

```
?- hors(b,[a,b,c]).
false.
```

```
?- hors(d,[a,b,c]).
true ;
false.
```

8. `suivants(?X,?Y,?L)` : `X` et `Y` se suivent immédiatement dans la liste `L`.

```
?- suivants(a,b,[a,b,c]).
true ;
false.
?- suivants(a,b,L).
L = [a, b|_G280] ;
L = [_G276, a, b|_G283] ;
...
?- suivants(a,X,[a,b,c]).
X = b ;
false.
?- suivants(a,X,L).
L = [a, X|_G295] ;
L = [_G291, a, X|_G298] ;
L = [_G291, _G294, a, X|_G301] ;
...
```

```
?- suivants(X,b,[a,b,c]).
X = a ;
false.
?- suivants(X,b,L).
L = [X, b|_G295] ;
L = [_G291, X, b|_G298] ;
...
?- suivants(X,Y,[a,b,c]).
X = a, Y = b ;
X = b, Y = c ;
false.
?- suivants(X,Y,L).
L = [X, Y|_G310] ;
L = [_G306, X, Y|_G313] ;
...
```

9. `unique(?X,+L)` : `X` n'apparaît qu'une seule fois dans la liste `L`.

```
?- unique(b,[a,b,c,a]).
true ;
false.
```

```
?- unique(X,[a,b,c,a]).
X = b ;
X = c ;
false.
```

10. `occurences(?X,?N,+L)` : `X` apparaît `N` fois dans la liste `L`.

```
?- occurences(a,3,[a,b,a,c,b,a]).
true ;
false.
?- occurences(a,X,[a,b,a,c,b,a]).
X = 3 ;
false.
```

```
?- occurences(X,3,[a,b,a,c,b,a]).
X = a ;
false.
?- occurences(X,Y,[a,b,a,c,b,a]).
X = a, Y = 3 ;
X = b, Y = 2 ;
X = c, Y = 1 ;
false.
```

11. `prefixe(?P,?L)` : `P` est un préfixe de la liste `L`.

```
?- prefixe([a,b],[a,b,c]).
true.
?- prefixe([a,b],L).
L = [a, b|_G290].
```

```
?- prefixe(P,[a,b,c]).
P = [] ;
P = [a] ;
P = [a, b] ;
P = [a, b, c] ;
false.
?- prefixe(P,L).
P = [] ;
P = [_G289], L = [_G289|_G293] ;
P = [_G289, _G295],
        L = [_G289, _G295|_G299] ;
...
```

12. `suffixe(?S,?L)` : `S` est un suffixe de la liste `L`.

```
?- suffixe([b,c],[a,b,c]).
true ;
false.
?- suffixe([b,c],L).
L = [b, c] ;
L = [_G286, b, c] ;
L = [_G286, _G289, b, c] ;
...
```

```
?- suffixe(S,[a,b,c]).
S = [a, b, c] ;
S = [b, c] ;
S = [c] ;
S = [] ;
false.
?- suffixe(S,L).
S = L ;
L = [_G289|S] ;
L = [_G289, _G292|S] ;
...
```

13. `sousListe(?SL,?L)` : `SL` est une sous–liste de la liste `L`.

```
?- sousListe([b],[a,b,c]).
true ;
false.
?- sousListe([b],L).
L = [b|_G284] ;
L = [_G280, b|_G290] ;
L = [_G280, _G286, b|_G296] ;
...
```

```
?- sousListe(SL,[a,b,c]).
SL = [] ;
SL = [a] ;
SL = [a, b] ;
SL = [b] ;
SL = [a, b, c] ;
SL = [b, c] ;
SL = [c] ;
false.
?- sousListe(SL,L).
SL = [] ;
SL = [_G289], L = [_G289|_G296] ;
SL = [_G289, _G298],
       L = [_G289, _G298|_G302] ;
SL = [_G289],
       L = [_G292, _G289|_G302] ;
...
```

▽ 69

TD 3.3 : Manipulation de listes

Définir les prédicats suivants :

1. `conc(?L1,?L2,?L3)` : concaténation de listes (`L1 + L2 = L3`).

```
?- conc([a],[b,c],[a,b,c]).
true.
?- conc([a],[b,c],L3).
L3 = [a, b, c].
?- conc([a],L2,[a,b,c]).
L2 = [b, c].
?- conc([a],L2,L3).
L3 = [a|L2].
```

```
?- conc(L1,[b,c],[a,b,c]).
L1 = [a] ;
false.
?- conc(L1,[b,c],L3).
L1 = [], L3 = [b, c] ;
L1 = [_G303], L3 = [_G303, b, c] ;
L1 = [_G303, _G309],
       L3 = [_G303, _G309, b, c] ;
...
?- conc(L1,L2,[a,b,c]).
L1 = [], L2 = [a, b, c] ;
L1 = [a], L2 = [b, c] ;
L1 = [a, b], L2 = [c] ;
L1 = [a, b, c], L2 = [] ;
false.
?- conc(L1,L2,L3).
L1 = [], L2 = L3 ;
L1 = [_G306], L3 = [_G306|L2] ;
L1 = [_G306, _G312],
       L3 = [_G306, _G312|L2] ;
...
```

2. `inserer(?X,?L1,?L2)` : `X` est inséré en tête de `L1` pour donner `L2`.

```
?- inserer(a,[b,c],[a,b,c]).
true.
?- inserer(a,[b,c],L2).
L2 = [a, b, c].
?- inserer(a,L1,[a,b,c]).
L1 = [b, c].
?- inserer(a,L1,L2).
L2 = [a|L1].
```

```
?- inserer(X,[b,c],[a,b,c]).
X = a.
?- inserer(X,[b,c],L).
L = [X, b, c].
?- inserer(X,L1,[a,b,c]).
X = a,
L1 = [b, c].
?- inserer(X,L1,L2).
L2 = [X|L1].
```

3. `ajouter(?X,?L1,?L2)` : `X` est ajouté en fin de `L1` pour donner `L2`.

```
?- ajouter(c,[a,b],[a,b,c]).
true ;
false.
?- ajouter(c,[a,b],L).
L = [a, b, c] ;
false.
?- ajouter(c,L1,[a,b,c]).
L1 = [a, b] ;
false.
?- ajouter(c,L1,L2).
L1 = [], L2 = [c] ;
L1 = [_G291], L2 = [_G291, c] ;
L1 = [_G291, _G297],
       L2 = [_G291, _G297, c] ;
...
```

```
?- ajouter(X,[a,b],[a,b,c]).
X = c ;
false.
?- ajouter(X,[a,b],L2).
L2 = [a, b, X] ;
false.
?- ajouter(X,L1,[a,b,c]).
X = c, L1 = [a, b] ;
false.
?- ajouter(X,L1,L2).
L1 = [], L2 = [X] ;
L1 = [_G306], L2 = [_G306, X] ;
L1 = [_G306, _G312],
       L2 = [_G306, _G312, X] ;
...
```

4. `selection(?X,?L1,?L2)` : `X` est extrait de `L1` pour donner `L2`.

```
?- selection(b,[a,b,c],[a,c]).
true ;
false.
?- selection(b,[a,b,c],L2).
L2 = [a, c] ;
false.
?- selection(b,L1,[a,c]).
L1 = [b, a, c] ;
L1 = [a, b, c] ;
L1 = [a, c, b] ;
false.
?- selection(b,L1,L2).
L1 = [b|L2] ;
L1 = [_G291, b|_G295],
       L2 = [_G291|_G295] ;
L1 = [_G291, _G297, b|_G301],
       L2 = [_G291, _G297|_G301] ;
...
```

```
?- selection(X,[a,b,c],[a,c]).
X = b ;
false.
?- selection(X,[a,b,c],L2).
X = a, L2 = [b, c] ;
X = b, L2 = [a, c] ;
X = c, L2 = [a, b] ;
false.
?- selection(X,L1,[a,c]).
L1 = [X, a, c] ;
L1 = [a, X, c] ;
L1 = [a, c, X] ;
false.
?- selection(X,L1,L2).
L1 = [X|L2] ;
L1 = [_G306, X|_G310],
       L2 = [_G306|_G310] ;
L1 = [_G306, _G312, X|_G316],
       L2 = [_G306, _G312|_G316] ;
...
```

5. `supprimer(?X,+L1,?L2)` : toutes les occurences de `X` sont supprimées de `L1` pour donner `L2`.

```
?- supprimer(a,[a,b,a,c],[b,c]).
true ;
false.
?- supprimer(a,[a,b,a,c],L2).
L2 = [b, c] ;
false.
```

```
?- supprimer(X,[a,b,a,c],[b,c]).
X = a ;
false.
?- supprimer(X,[a,b,a,c],L2).
X = a, L2 = [b, c] ;
X = b, L2 = [a, a, c] ;
X = c, L2 = [a, b, a] ;
false.
```

6. `inverser(+L1,?L2)` : inverser `L1` donne `L2`.

```
?- inverser([a,b,c],[c,b,a]).
true.
```

```
?- inverser([a,b,c],L2).
L2 = [c, b, a].
```

7. `substituer(?X,?Y,+L1,?L2)` : substituer toutes les occurences de `X` par `Y` dans `L1` donne `L2`.

```
?- substituer(a,e,[a,b,a,c],
              [e,b,e,c]).
true ;
false.
?- substituer(a,e,[a,b,a,c],L2).
L2 = [e, b, e, c] ;
false.
?- substituer(a,Y,[a,b,a,c],
              [e,b,e,c]).
Y = e ;
false.
?- substituer(a,Y,[a,b,a,c],L2).
L2 = [Y, b, Y, c] ;
false.
```

```
?- substituer(X,e,[a,b,a,c],
              [e,b,e,c]).
X = a ;
false.
?- substituer(X,e,[a,b,a,c],L2).
X = a, L2 = [e, b, e, c] ;
X = b, L2 = [a, e, a, c] ;
X = c, L2 = [a, b, a, e] ;
false.
?- substituer(X,Y,[a,b,a,c],[e,b,e,c]).
X = a, Y = e ;
false.
?- substituer(X,Y,[a,b,a,c],L2).
X = a, L2 = [Y, b, Y, c] ;
X = b, L2 = [a, Y, a, c] ;
X = c, L2 = [a, b, a, Y] ;
false.
```

8. `decaler(+L,?LD)` : `LD` est la liste `L` où les éléments ont été décalés circulairement vers la droite.

```
?- decaler([a,b,c],[c,a,b]).
true ;
false.
```

```
?- decaler([a,b,c],L).
L = [c, a, b] ;
false.
```

9. `permutation(+L,?LP)` : `LP` est une permutation de la liste `L` (si n est le nombre d'éléments de la liste, il y a $n!$ permutations possibles).

```
?- permutation([a,b,c],[b,c,a]).
true ;
false.
```

```
?- permutation([a,b,c],LP).
LP = [a, b, c] ;
LP = [a, c, b] ;
LP = [b, a, c] ;
LP = [b, c, a] ;
LP = [c, a, b] ;
LP = [c, b, a] ;
false.
```

10. `convertir(?L,?LC)` : `LC` est la liste `L ( = [t1,t2,...,tn] )` écrite sous la forme `t1 et t2 et ... et tn et fin`.

```
?- convertir([a,b,c],
             a et b et c et fin). (2)
true.
?- convertir([a,b,c],LC).
LC = a et b et c et fin.
```

```
?- convertir(L,a et b et c et fin).
L = [a, b, c] ;
false.
?- convertir(L,LC).
L = [], LC = fin ;
L = [_G289], LC = _G289 et fin ;
L = [_G289, _G295],
        LC = _G289 et _G295 et fin ;
...
```

11. `aplatir(+L,?LA)` : `LA` est la liste `L` qui ne contient plus qu'un seul niveau de crochets `( [...] )`.

```
?- aplatir([a,[b,[[]]],[c]],
           [a,b,c]).
true ;
false.
```

```
?- aplatir([a,[b,[[]]],[c]],LA).
LA = [a, b, c] ;
false.
```

12. `transposer(+L,?LT)` : `LT` est la matrice transposée de la matrice `L`.

```
?- transposer([[1,2,3],[4,5,6]],L).
L = [[1, 4], [2, 5], [3, 6]] ;
false.
```

```
?- transposer([[1,4],[2,5],[3,6]],L).
L = [[1, 2, 3], [4, 5, 6]] ;
false.
```

13. `creerListe(?X,+N,-L)` : `L` est la liste créée à partir de `N` occurences du terme `X`.

```
?- creerListe(0,3,L).
L = [0, 0, 0].
```

```
?- creerListe(X,3,L).
L = [X, X, X].
```

14. `univ(?F,?L)` : `L` est la liste dont la tête est le nom du terme `F` et la queue est composée des arguments de `F`. Chacun des arguments peut être libre, mais pas les deux simultanément.

```
?- univ(f(a,f(b)),[f,a,f(b)]).
true ;
false.
?- univ(f(a,f(b)),L).
L = [f, a, f(b)] ;
false.
```

```
?- univ(F,[f,a,f(b)]).
F = f(a, f(b)) ;
false.
```

▽ 70

TD 3.4 : Tris de listes

Définir les prédicats suivants :

1. `triPermutation(+L,?LT)` : `LT` est la liste `L` triée selon la méthode naïve des permutations (méthode « générer et tester »).

```
?- triPermutation([c,a,b],[a,b,c]).
true ;
false.
```

```
?- triPermutation([c,a,b],LT).
LT = [a, b, c] ;
false.
```

2. `triSelection(+L,?LT)` : `LT` est la liste `L` triée selon la méthode du tri par sélection.

```
?- triSelection([c,a,b],[a,b,c]).
true ;
false.
```

```
?- triSelection([c,a,b],LT).
LT = [a, b, c] ;
false.
```

3. `triBulles(+L,?LT)` : `LT` est la liste `L` triée selon la méthode du tri à bulles.

```
?- triBulles([c,b,a],[a,b,c]).
true ;
false.
```

```
?- triBulles([c,b,a],LT).
LT = [a, b, c] ;
false.
```

4. `triDicho(+L,?LT)` : `LT` est la liste `L` triée selon la méthode de tri par insertion dichotomique.

```
?- triDicho([c,b,a],[a,b,c]).
true ;
false.
```

```
?- triDicho([c,b,a],LT).
LT = [a, b, c] ;
false.
```

5. `triFusion(+L,?LT)` : `LT` est la liste `L` triée selon la méthode de tri par fusion.

```
?- triFusion([c,b,a],[a,b,c]).
true ;
false.
```

```
?- triFusion([c,b,a],LT).
LT = [a, b, c] ;
false.
```

6. `triRapide(+L,?LT)` : `LT` est le liste `L` triée selon la méthode de tri rapide.

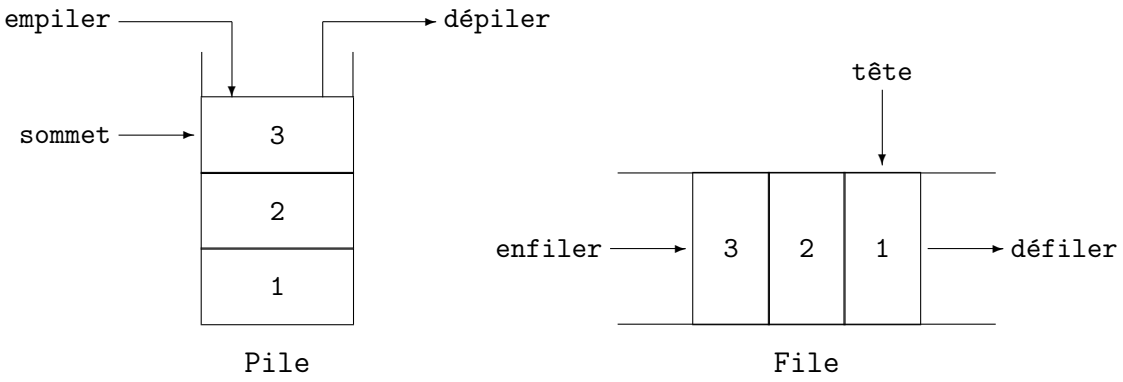
```
?- triRapide([c,b,a],[a,b,c]).
true ;
false.
```

```
?- triRapide([c,b,a],LT).
LT = [a, b, c] ;
false.
```

▽ 72

TD 3.5 : Piles et files

Les piles et les files sont représentées ici par des listes Prolog.



Définir les principales opérations sur ces structures :

1. `empiler(?X,?L1,?L2)` (`enfiler(?X,?L1,?L2)`) : empiler (enfiler) `X` dans la pile (file) `L1` donne `L2`.

```
?- empiler(a,[b,c],[a,b,c]).
true.
?- empiler(a,[b,c],L2).
L2 = [a, b, c].
?- empiler(a,L1,[a,b,c]).
L1 = [b, c].
?- empiler(a,L1,L2).
L2 = [a|L1].
```

```
?- empiler(X,[b,c],[a,b,c]).
X = a.
?- empiler(X,[b,c],L2).
L2 = [X, b, c].
?- empiler(X,L1,[a,b,c]).
X = a, L1 = [b, c].
?- empiler(X,L1,L2).
L2 = [X|L1].
```

2. `depiler(?X,?P1,?P2)` (`defiler(?X,?F1,?F2)`) : dépiler (défiler) `X` de la pile (file) `L1` donne `L2`.

```
?- defiler(a,[c,b,a],[c,b]).
true.
?- defiler(a,[c,b,a],L2).
L2 = [c, b] ;
false.
?- defiler(a,L1,[c,b]).
L1 = [c, b, a].
?- defiler(a,L1,L2).
L1 = [a], L2 = [] ;
L1 = [_G291, a], L2 = [_G291] ;
...
```

```
?- defiler(X,[c,b,a],[c,b]).
X = a.
?- defiler(X,[c,b,a],L2).
X = a, L2 = [c, b] ;
false.
?- defiler(X,L1,[c,b]).
L1 = [c, b, X].
?- defiler(X,L1,L2).
L1 = [X], L2 = [] ;
L1 = [_G306, X], L2 = [_G306] ;
...
```

3. `pileVide(?L)` (`fileVide(?L)`) : tester si la pile (file) `L` est vide.

```
?- pileVide([]).
true.
```

```
?- pileVide(L).
L = [].
```

4. `sommet(?X,?L)` (`tete(?X,?L)`) : `X` est le sommet (la tête) de la pile (file) `L`.

```
?- tete(a,[c,b,a]).
true ;
false.
?- tete(a,L).
L = [a] ;
L = [_G271, a] ;
L = [_G271, _G274, a] ;
...
```

```
?- tete(X,[c,b,a]).
X = a ;
false.
?- tete(X,L).
L = [X] ;
L = [_G286, X] ;
L = [_G286, _G289, X] ;
...
```

▽ 74

TD 3.6 : Ensembles

Un ensemble est représenté ici par une liste Prolog dont les éléments sont tous différents les uns des autres. Définir les prédicats suivants :

1. `ensemble(+L)` : la liste `L` repésente un ensemble.

```
?- ensemble([a,b,c]).
true ;
false.
```

```
?- ensemble([a,b,a,c]).
false.
```

2. `listeEnsemble(+L,?E)` : la liste `L` est transformée en un ensemble `E`.

```
?- listeEnsemble([a,b,a,c,a,a],
                 [a,b,c]).
true ;
false.
```

```
?- listeEnsemble([a,b,a,c,a,a],E).
E = [a, b, c] ;
false.
```

3. `intersection(+E1,+E2,?E3)` : intersection de 2 ensembles (`E3 = E1` $\cap$ `E2`).

```
?- intersection([a,b,c],[b,d,a],
                [a,b]).
true ;
false.
```

```
?- intersection([a,b,c],[b,d,a],E3).
E3 = [a, b] ;
false.
```

4. `union(+E1,+E2,?E3)` : union de 2 ensembles (`E3 = E1` $\cup$ `E2`).

```
?- union([a,b,c],[b,d,a],
         [a,b,c,d]).
true ;
false.
```

```
?- union([a,b,c],[b,d,a],E3).
E3 = [a, b, c, d] ;
false.
```

5. `sousEnsemble(+E1,+E2)` : `E1` est un sous-ensemble de `E2` (`E1` $\subset$ `E2`).

```
?- sousEnsemble([b],[a,b,c]).
true ;
false.
```

```
?- sousEnsemble([d,b],[a,b,c,d]).
true ;
false.
```

▽ 75

TD 3.7 : Puzzle logique

On reprend ici le puzzle du TD 1.1 page 9 pour le résoudre automatiquement en Prolog.

1. Trois personnes, de nationalités différentes (anglaise, espagnole et française) et pratiquant des sports différents (football, natation et tennis), habitent dans trois maisons de couleurs distinctes (blanc, bleu, vert). Ces trois maisons sont situées dans la même rue ; une des maisons est située au début de la rue, une autre au milieu, et la troisième au bout de la rue.

   Chacune des 3 maisons est caractérisée par un triplet `m(C,N,S)`, où `C` est la couleur de la maison, `N` et `S` la nationalité et le sport pratiqué par son occupant.

   Définir le prédicat `tableau(?Rue)` où `Rue` est la liste des maisons qui satisfont à la description précédente. L'ordre des éléments dans la liste correspond à l'ordre des maisons dans la rue.

   Il y a *a priori* 216 combinaisons possibles.

```
?- tableau(R).
R = [m(blanc, anglais, football),
     m(bleu, espagnol, natation),
     m(vert, francais, tennis)] ;
...
R = [m(bleu, espagnol, natation),
     m(blanc, anglais, football),
     m(vert, francais, tennis)] ;
...
R = [m(vert, francais, tennis),
     m(bleu, espagnol, natation),
     m(blanc, anglais, football)] ;
false.
```

2. On donne maintenant les indices suivants :
   - Dans la maison verte on pratique la natation.
   - La maison verte est située avant la maison de l'espagnol.
   - L'anglais habite la maison blanche.
   - La maison blanche est située avant la maison où on pratique le football.
   - Le tennisman habite au début de la rue.

   Définir le prédicat `puzzle(?Rue)` où `Rue` est la liste des maisons qui vérifient les indices précédents. L'ordre des éléments dans la liste correspond à l'ordre des maisons dans la rue. Chaque indice doit trouver sa traduction Prolog dans la définition du prédicat `puzzle/1`.

```
?- puzzle(R).
R = [m(blanc, anglais, tennis),
     m(vert, francais, natation),
     m(bleu, espagnol, football)] ;
false.
```

On retrouve bien sûr ici la solution obtenue page 55.

▽ 76

TD 3.8 : Listes et arithmétique

On considère ici des listes de nombres. Définir les prédicats suivants :

1. `maximum(+L,?Max)` : `Max` est le maximum des nombres qui composent la liste `L`.

```
?- maximum([7,2,5,8,1,3],8).
true ;
false.
```

```
?- maximum([7,2,5,8,1,3],M).
M = 8 ;
false.
```

2. `somme(+L,?N)` : `N` est la somme des éléments de la liste `L`.

```
?- somme([1,2,3],6).
true.
```

```
?- somme([1,2,3],N).
N = 6.
```

3. `entiers(+Min,+Max,?L)` : `L` est la liste des entiers compris entre `Min` et `Max` inclus (`Min <= Max`).

```
?- entiers(1,3,[1,2,3]).
true ;
false.
```

```
?- entiers(1,3,L).
L = [1, 2, 3] ;
false.
```

4. `produitScalaire(+A,+B,?P)` : `P` est le produit scalaire de deux vecteurs `A` et `B`. Un vecteur est représenté ici par une liste de composantes (`[X,Y,Z]`).

$\vec{A} \cdot \vec{B} = \sum_i A_i \cdot B_i$.

```
?- produitScalaire([1,2,3],
                   [-1,2,4],15).
true.
```

```
?- produitScalaire([1,2,3],
                   [-1,2,4],P).
P = 15.
```

5. `surface(+P,?S)` : `S` est la surface du polygone `P` représenté ici par une liste de points (`[(X1,Y1),(X2,Y2)...]`).
   $S = \frac{1}{2}\int xdy - ydx$. Le signe de S est négatif si l'orientation du polygone est dans le sens des aiguilles d'une montre, positif sinon.

```
?- surface([(0,0),(1,0),(1,1),
            (0,1),(0,0)],1.0).
true ;
false.
?- surface([(4,6),(4,2),
            (0,8),(4,6)],-8).
true ;
false.
```

```
?- surface([(0,0),(1,0),(1,1),
            (0,1),(0,0)],S).
S = 1.0 ;
false.
?- surface([(4,6),(4,2),
            (0,8),(4,6)],S).
S = -8 ;
false.
```

6. `premiers(+N,?L)` : `L` est la liste des `N` premiers nombres premiers (on pourra utiliser la méthode du crible d'Eratosthène).

```
?- premiers(14,
   [1, 2, 3, 5, 7, 11, 13]).
true ;
false.
```

```
?- premiers(14,L).
L = [1, 2, 3, 5, 7, 11, 13] ;
false.
```

▽ 76

TD 3.9 : Le compte est bon

Le jeu du « compte est bon x consiste ici à déterminer une combinaison arithmétique entre 4 nombres connus x, y, z et t pour obtenir un cinquième nombre r donné. Les nombres x, y, z et t peuvent apparaître dans n'importe quel ordre mais ne doivent intervenir qu'une seule fois dans la combinaison. Seuls les opérateurs arithmétiques $+$, $-$ et $*$ seront utilisées.

Exemples : $x = 2, y = 5, z = 4, t = 6, r = 41 \quad t(x + z) + y = r$
$x = 2, y = 5, z = 4, t = 1, r = 41 \quad xyz + t = r$
$x = 2, y = 2, z = 3, t = 5, r = 1 \quad x + t - yz = r$

Définir les prédicats suivants :

1. `operation(+X,+Y,?Op,+Z)` : `X Op Y = Z` où `Op` représente l'un des 3 opérateurs arithmétiques `+/2`, `-/2` et `*/2`.

```
?- operation(2,3,+,5).
true ;
false.
```

```
?- operation(2,3,Op,5).
Op = + ;
false.
```

2. `calcul(+X,+Y,+Z,+T,+R,?Expr)` : `(((X Op1 Y) Op2 Z) Op3 T) = R` ou `((X Op4 Y) Op5 (Z Op6 T)) = R`, où `Op1`, `Op2`, ... `Op6` représentent chacun l'un des 3 opérateurs arithmétiques $+$, $-$ et $*$. `Expr` la combinaison trouvée.

```
?- calcul(2,4,6,5,41,((2+4)*6)+5).
true ;
false.
```

```
?- calcul(2,4,6,5,41,E).
E = (2+4)*6+5 ;
false.
```

3. `compte(+X,+Y,+Z,+T,+R,?Expr)` : `Expr` est une expression arithmétique des nombres `X`, `Y`, `Z` et `T` égale au nombre `R`.

```
?- compte(2,4,6,5,41,((4+2)*6)+5).
true ;
false.
```

```
?- compte(2,4,6,5,41,E).
E = (2+4)*6+5 ;
E = (4+2)*6+5 ;
false.
```

▽ 77

TD 3.10 : Casse-tête cryptarithmétique

Dans cet exercice, on cherche à résoudre des casse-têtes cryptarithmétiques du type :

$$\begin{array}{cccccc} & z & e & r & o \\ + & z & e & r & o \\ \hline = & r & i & e & n \end{array} \qquad \begin{array}{cccccc} & m & a & p & p & e \\ + & m & o & n & d & e \\ \hline = & a & t & l & a & s \end{array} \qquad \begin{array}{cccccc} & g & r & a & n & d \\ + & e & c & r & a & n \\ \hline = & c & i & n & e & m & a \end{array}$$

où chaque lettre représente un chiffre ($\in \{0, 1, 2 \ldots, 8, 9\}$) et où deux lettres différentes représentent deux chiffres différents (10 lettres différentes au maximum). Les casse-têtes seront codés par les faits Prolog suivants :

```
% casseTete/4
casseTete(1,[Z,E,R,O],[Z,E,R,O],[R,I,E,N]).
casseTete(2,[D,O,N,A,L,D],[G,E,R,A,L,D],[R,O,B,E,R,T]).
casseTete(3,[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]).
casseTete(4,[L,A,R,D],[S,A,L,E],[B,A,C,O,N]).
casseTete(5,[C,O,U,P],[D,O,N,N,E],[M,A,S,S,E]).
casseTete(6,[M,A,P,P,E],[M,O,N,D,E],[A,T,L,A,S]).
casseTete(7,[G,R,A,N,D],[E,C,R,A,N],[C,I,N,E,M,A]).
casseTete(8,[D,O,U,Z,E],[M,O,I,S],[A,N,N,E,E]).
```

Définir le prédicat `somme(+L1,+L2,+L3)` où `L1` représente le premier nombre d'un casse-tête (exemple : `[G,R,A,N,D]`), `L2` le deuxième nombre du casse-tête (exemple : `[E,C,R,A,N]`) et `L1` la somme du casse-tête (exemple : `[C,I,N,E,M,A]`).

```
?- casseTete(2,N1,N2,N3),
   somme(N1,N2,N3).
N1 = [5, 2, 6, 4, 8, 5],
N2 = [1, 9, 7, 4, 8, 5],
N3 = [7, 2, 3, 9, 7, 0] ;
false.
```

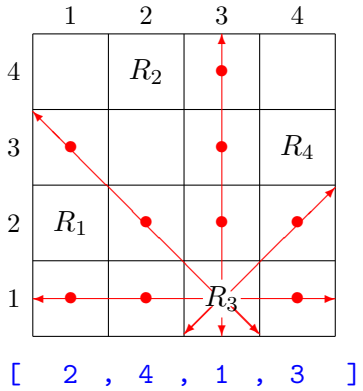
```
?- casseTete(7,N1,N2,N3),
   somme(N1,N2,N3).
N1 =    [4, 6, 2, 7, 5],
N2 =    [9, 1, 6, 2, 7],
N3 = [1, 3, 7, 9, 0, 2] ;
N1 =    [8, 5, 6, 7, 9],
N2 =    [2, 1, 5, 6, 7],
N3 = [1, 0, 7, 2, 4, 6] ;
false.
```

▽ 78

TD 3.11 : Problème des n reines

On cherche à placer n reines sur un échiquier ($n \times n$) sans qu'aucune des reines ne puisse en attaquer une autre. Ainsi deux reines ne peuvent pas se situer sur la même horizontale, ni sur la même verticale, ni sur les mêmes diagonales. Si le problème des n reines a une solution, les n reines se trouvent nécessairement à des abscisses différences puisqu'elles ne peuvent pas être situées sur une même verticale. On représentera alors la solution par la liste de leurs ordonnées respectives, leur position dans la liste codant pour sa part implicitement leurs abscissees respectives (`[y1,y2...,yn]` → `[(1,y1),(2,y2)...,(n,yn)]`).

La figure ci-contre illustre le problème des n reines pour $n = 4$. Dans cette figure les quatre reines $R_1(1,2)$, $R_2(2,4)$, $R_3(3,1)$ et $R_4(4,3)$ ne s'attaquent pas mutuellement. A titre d'exemple, les positions marquées d'un point rouge (•) sont attaquables par la reine R_3 ; on vérifie ainsi qu'elle n'attaque pas les trois autres reines.
La solution présentée est donc codée par la liste des ordonnées des 4 reines (`[2,4,1,3]`), leur position dans la liste codant implicitement leurs abscisses respectives (sous-entendu : `[(1,2),(2,4),(3,1),(4,3)]`).



Définir le prédicat `nreines(+N,?L)` où `L` représente la liste des ordonnées de `N` reines placées sur un échiquier `(N×N)` de telle manière qu'elles ne s'attaquent pas mutuellement.

```
?- nreines(4,[1,2,3,4]).
false.
?- nreines(4,L).
L = [2, 4, 1, 3] ;
L = [3, 1, 4, 2] ;
false.
```

```
?- nreines(5,L).
L = [1, 3, 5, 2, 4] ;
L = [1, 4, 2, 5, 3] ;
L = [2, 4, 1, 3, 5] ;
L = [2, 5, 3, 1, 4] ;
L = [3, 1, 4, 2, 5] ;
L = [3, 5, 2, 4, 1] ;
L = [4, 1, 3, 5, 2] ;
L = [4, 2, 5, 3, 1] ;
L = [5, 2, 4, 1, 3] ;
L = [5, 3, 1, 4, 2] ;
false.
```

▽ 79

TD 3.12 : Les mutants

Définir le prédicat `mutant(+Mot1,+Mot2,?Mot3)` qui, à partir de deux mots `Mot1` et `Mot2`, crée un troisième mot `Mot3` en faisant coïncider la fin du premier (`Mot1`) avec le début du deuxième (`Mot2`).

```
?- mutant(caribou,ours,Mutant).
Mutant = caribours ;
false.
?- mutant(vache,cheval,Mutant).
Mutant = vacheval ;
false.
?- mutant(koala,lapin,Mutant).
Mutant = koalapin ;
false.
```

```
?- mutant(alligator,tortue,Mutant).
Mutant = alligatortue ;
false.
?- mutant(caiman,mangouste,Mutant).
Mutant = caimangouste ;
false.
?- mutant(poule,lemming,Mutant).
Mutant = poulemming ;
false.
```

▽ 79

TD 3.13 : Mots croisés

Un mot sera représenté ici par la liste des caractères qui le composent.
Exemples : `mot` → `[m,o,t]`,
`liste` → `[l,i,s,t,e]`,
`caracteres` → `[c,a,r,a,c,t,e,r,e,s]`.
Pour stocker ces mots, on dispose d'un dictionnaire représenté en Prolog par un ensemble de faits `dico/1` tels que :

```
% dico/1
dico([c,a,r,a,c,t,e,r,e,s]). dico([l,i,s,t,e]). dico([m,o,t]). ...
```

Définir les prédicats suivants :

1. `nbCars(+Mot,?NbCars)` : `NbCars` est le nombre de caractères qui composent le `Mot`.

```
?- nbCars([m,o,t],3).
true ;
false.
```

```
?- nbCars([m,o,t],N).
N = 3 ;
false.
```

2. `selectMot(?Mot,+NbCars)` : `Mot` est un mot de `NbCars` caractères dans le dictionnaire.

```
?- selectMot([m,o,t],3).
true ;
false.
```

```
?- selectMot(Mot,3).
Mot = [m, o, t] ;
false.
```

3. `rang(+Mot,?N,?Car)` : `Car` est le caractère de rang `N` dans le mot `Mot`.

```
?- rang([m,o,t],2,o).
true ;
false.
?- rang([m,o,t],2,C).
C = o ;
false.
```

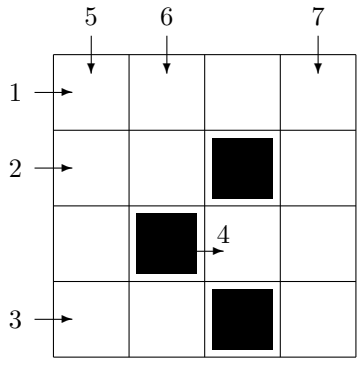
```
?- rang([m,o,t],N,o).
N = 2 ;
false.
?- rang([m,o,t],N,C).
N = 1, C = m ;
N = 2, C = o ;
N = 3, C = t ;
false.
```

4. `interCar(+Mot1,?N1,+Mot2,?N2)` : le caractère de rang `N1` dans le mot `Mot1` est identique au caractère de rang `N2` dans le mot `Mot2`.

```
?- interCar([m,o,t],2,
            [t,e,x,t,o],5).
true ;
false.
?- interCar([m,o,t],2,
            [t,e,x,t,o],N2).
N2 = 5 ;
false.
```

```
?- interCar([m,o,t],N1,
            [t,e,x,t,o],5).
N1 = 2 ;
false.
?- interCar([m,o,t],N1,
            [t,e,x,t,o],N2).
N1 = 2, N2 = 5 ;
N1 = 3, N2 = 1 ;
N1 = 3, N2 = 4 ;
false.
```

5. `motsCroises(?M1,?M2,?M3,?M4,?M5,?M6,?M7)` : les mots `M1`, `M2`, `M3`, `M4`, `M5`, `M6` et `M7` du dictionnaire permettent de compléter la grille ci-dessous composée de 7 mots numérotés de 1 à 7.




```
?- motsCroises(M1,M2,M3,M4,M5,M6,M7).
M1 = [c, h, a, t],
M2 = [l, a],
M3 = [n, e],
M4 = [d, u],
M5 = [c, l, i, n],
M6 = [h, a],
M7 = [t, r, u, c] ;
false.
```

▽ 79

TD 4

Contrôle de la résolution

TD 4.1 : Dérivation symbolique

Définir le prédicat `d(+FV,+V,?DFV)` où `DFV` est la dérivée de l'expression `FV` par rapport à la variable `V` (variable au sens mathématique du terme).

```
?- d(sin(cos(x)),x,D).
D = -1*sin(x)*cos(cos(x)).
?- d(exp(x^2),x,D).
D = 2*x^(2-1)*exp(x^2).
```

```
?- d(sin(ln(x))+x^2,x,D).
D = 1/x*cos(ln(x))+2*x^(2-1).
?- d(cos(x)^2,cos(x),D).
D = 2*cos(x)^(2-1).
```

On rappelle ci-dessous les principaux résultats concernant les dérivées de fonctions :

$$\frac{dC}{dx} = 0 \text{ si } C = Cte_{/x}$$
$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$
$$\frac{d(u \cdot v)}{dx} = v \cdot \frac{du}{dx} + u \cdot \frac{dv}{dx}$$
$$\frac{d(f(u))}{dx} = \frac{du}{dx} \cdot \frac{d(f(u))}{du}$$

$$\frac{d(x^n)}{dx} = nx^{n-1}$$
$$\frac{d(\sin(x))}{dx} = \cos(x)$$
$$\frac{d(\cos(x))}{dx} = -\sin(x)$$
$$\frac{d(\exp(x))}{dx} = \exp(x)$$
$$\frac{d(\log(x))}{dx} = \frac{1}{x}$$
$$\ldots$$

▽ 81

TD 4.2 : Traitements génériques

Définir le prédicat `map(+F,+L1,?L2)` où `L2` est la liste `L1` où chaque élément a été traité à l'aide du prédicat de symbole fonctionnel `F`.

```
?- map(abs,[-1,2,-3],L).
L = [1, 2, 3] ;
false.
```

```
?- map(triRapide,[[4,3],[9,5]],L).
L = [[3, 4], [5, 9]] ;
false.
```

▽ 81

TD 4.3 : Du mot aux syllabes

1. Définir les 3 prédicats suivants :
   (a) `codes(?Codes,?Lettres)` : `Codes` est la liste des codes ASCII correspondant à la liste des lettres `Lettres`

```
?- codes([65,66,97,98],L).
L = ['A', 'B', a, b].
```

```
?- codes(L,['A','B',a,b]).
L = [65, 66, 97, 98] ;
false.
```

(b) `decomposer(+Mot,?Lettres)` : `Lettres` est la liste des lettres qui composent le terme atomique `Mot`

```
?- decomposer(mot,L).
L = [m, o, t].
```

```
?- decomposer('ABab',L).
L = ['A', 'B', a, b].
```

(c) `recomposer(?Mot,+Lettres)` : `Mot` est le terme atomique dont la liste des lettres est `Lettres`

```
?- recomposer(M,[m,o,t]).
M = mot ;
false.
```

```
?- recomposer(M,['A','B',a,b]).
M = 'ABab' ;
false.
```

2. On admet maintenant les règles de césure suivantes :
   - Une consonne située entre deux voyelles introduit une nouvelle syllabe :
     `tra-vaux di-ri-gés pro-log`
   - Si deux consonnes sont situées entre deux voyelles, la première appartient à la syllabe précédente, la seconde à la syllabe suivante :
     `syl-la-bes en ar-gent mas-sif`
     Exceptions : les consonnes `l` ou `r` précédées d'une consonne autre que `l` ou `r`, forment avec cette consonne qui les précède un groupe inséparable (`bl, cl, fl, gl, pl, br, cr, dr, fr, gr, pr, tr, vr`), qui commence une syllabe :
     `o-gre li-bre`
     Les groupes `ch, ph, gn` et `th` sont inséparables :
     `or-tho-gra-phe so-phis-ti-quée`
   - Quand il y a trois consonnes consécutives à l'intérieur d'un mot, les deux premières terminent une syllabe, l'autre commence une nouvelle syllabe :
     `rith-me obs-ti-né`
     Exceptions : les groupes inséparables (`bl, cl, fl, gl, pl, br, cr, dr, fr, gr, pr, tr, vr, ch, ph, gn, th`) sont assimilés à une seule consonne :
     `af-flux ins-truit`

   On suppose par ailleurs que les prédicats suivants sont définis :
   - `voyelle(?V)` : `V` est une voyelle.
   - `consonne(?C)` : `C` est une consonne.
   - `inseparables([?C1,?C2])` : les consonnes `C1` et `C2`, prises dans cet ordre, sont inséparables.

   Définir le prédicat `cesures(+M,+Sep,?S)` où `S` est le mot `M` dans lequel les syllabes sont séparées par le caractère séparateur `Sep`.

```
?- cesures(inseparable,-,M).
M = 'in-se-pa-ra-ble' ;
false.
?- cesures(orthographe,-,M).
M = 'or-tho-gra-phe' ;
false.
```

```
?- cesures(algorithme,+,M).
M = 'al+go+rith+me' ;
false.
?- cesures(syllabe,+,M).
M = 'syl+la+be' ;
false.
```

```
?- cesures(anticonstitutionnellement,/,M).
M = 'an/ti/cons/ti/tu/tion/nel/le/ment' ;
false.
```

▽ 81

TD 4.4 : Gestion d'un compteur

On veut pouvoir gérer un compteur comme une variable globale d'un langage impératif (comme C ou Pascal). Définir les prédicats suivants :

1. `cptInit(+Cpt,+Val)` : le compteur `Cpt` est initialisé à `Val`.

```
?- cptInit(c,0).
true.
```

```
?- cptInit(d,3).
true.
```

2. `cptFin(+Cpt)` : le compteur `Cpt` est détruit.

```
?- cptFin(c).
true.
```

```
?- cptFin(d).
true.
```

3. `cptVal(+Cpt,?Val)` : la valeur du compteur `Cpt` est `Val`.

```
?- cptVal(c,3).
true.
```

```
?- cptVal(c,V).
V = 3.
```

4. `cptInc(+Cpt,+Inc)` : le compteur `Cpt` est incrémenté de `Inc`.

```
?- cptVal(c,V).
V = 3.
?- cptInc(c,4), cptVal(c,V).
V = 7.
```

```
?- cptVal(d,V).
V = 6.
?- cptInc(d,-5), cptVal(d,V).
V = 1.
```

5. `alphabet` : affiche les lettres de l'alphabet en utilisant un compteur et une boucle `repeat/0`.

```
?- alphabet.
a b c d e f g h i j k l m n o p q r s t u v w x y z
true.
```

▽ 82

TD 4.5 : Structures de contrôle

On veut définir en PROLOG les différentes structures de contrôle d'un langage impératif (comme C ou Pascal) :

- `si P alors Q`
- `si P alors Q sinon R`
- `selon X dans [X1 :Q1,X2 :Q2,...,Xn :Qn]`
- `selon X dans [X1 :Q1,X2 :Q2,...,Xn :Qn autrement R]`
- `repeter Q jusqua P`
- `pour Cpt := Min to Max faire Q`
- `pour Cpt := Max downto Min faire Q`

On définit pour cela les opérateurs suivants :

```
:- op(900,fx ,si), op(850,xfx,alors), op(800,xfx,sinon).
:- op(900,fx ,repeter), op(850,xfx,jusqua).
:- op(900,fx ,pour), op(850,xfx,to), op(850,xfx,downto), op(800,xfx,faire).
:- op(900,fx ,selon), op(850,xfx,dans), op(800,xfx,autrement).
:- op(750,xfx,:=), op(750,xfx,:).
```

1. Représenter par des arbres les 7 types de structures compte-tenu de la définition des opérateurs.
2. Définir les prédicats correspondants aux 4 structures de base :
   (a) `si/1`

```
?- X = 3, Y = 5,
   si X > Y
   alors M = X
   sinon M = Y.
X = 3, Y = 5, M = 5.
```

```
?- X = 3, Y = 2,
   si X > Y
   alors M = X
   sinon M = Y.
X = 3, Y = 2, M = 3.
```

(b) `selon/1`

```
?- X = 3,                                  ?- X = 0,
   selon X                                    selon X
         dans [1: Y = -1,                           dans [1: Y = -1,
               2: Y = 0,                                  2: Y = 0,
               3: Y = 1                                   3: Y = 1
               autrement Y = 5].                          autrement Y = 5].
X = 3, Y = 1.                              X = 0, Y = 5.
```

(c) `repeter/1`

```
?- cptInit(i,0),
   repeter
     (
     cptVal(i,V), write(V), write(' '), cptInc(i,1)
     )
   jusqua V =:= 3.
0 1 2 3
V = 3.
```

(d) `pour/1`

```
?- pour i := 0 to 3 faire (cptVal(i,V), write(V), write(' ')).
0 1 2 3
V = 3.
```

3. Définir un prédicat `menu/0` qui permet d'afficher un menu, d'entrer une sélection et d'effectuer l'action associée à la sélection.

```
?- menu(test).
EXEMPLE DE MENU
1. un
2. deux
3. trois
4. quitter

test > votre choix : 2.
deux

EXEMPLE DE MENU
1. un
2. deux
3. trois
4. quitter

test > votre choix : 4.

... fin de l'exemple
true.
```

Les différentes actions devront illustrer le fonctionnement des structures de contrôle définies dans la question précédente.

▽ 83

TD 4.6 : Unification

1. Définir le prédicat `unifier(?Terme1,?Terme2)` vrai si et seulement si `Terme1` et `Terme2` sont unifiables :
   – deux termes inconnus sont unifiables ;

- un terme inconnu est unifiable à un terme instancié ;
- deux termes atomiques sont unifiables s'ils sont identiques ;
- deux termes composés sont unifiables s'ils ont même foncteur (atome/arité) et si leurs arguments sont unifiables.

```
?- unifier(X,Y).
X = Y ;
false.
?- unifier(X,a).
X = a ;
false.
?- unifier(b,a).
false.
```

```
?- unifier(f(a),g(a)).
false.
?- unifier(f(a,f(b)),f(a,g(b))).
false.
?- unifier(f(X,a,g(Y,c)),f(b,Y,Z)).
X = b, Y = a, Z = g(a, c) ;
false.
```

2. A la question `X = f(X)`, certains interpréteurs Prolog donnent pour réponse :
   `X = f(f(f(f(f( ...Débordement de pile.`
   Définir le prédicat `nonOccur(-Terme1,?Terme2)` où le terme inconnu `Terme1` n'apparaît pas dans `Terme2` (test d'occurence).

```
?- nonOccur(X,f(f(f(Y,X)))).
false.
```

```
?- nonOccur(X,f(f(f(Z,Y)))).
true ;
false.
```

3. Redéfinir le prédicat `unifier(?Terme1,?Terme2)` de la première question en effectuant le test d'occurence de la question précédente.

```
?- unifier(X,f(X)).
X = f(**) ;
false.
```

```
?- unifier2(X,f(X)).
false.
```

4. Définir le prédicat `varLiees(-Terme1,-Terme2)` où `Terme1` et `Terme2` sont liés.

```
?- varLiees(X,Y).
false.
?- varLiees(X,X).
true.
```

```
?- X = Y, varLiees(X,Y).
X = Y.
?- X = Y, Y = Z, varLiees(X,Z).
X = Z, Y = Z.
```

5. Définir le prédicat `nbVars(+Terme,+N1,?N2)` où les arguments inconnus de `Terme` sont successivement instanciés au terme `v(N)`, où `N` varie de `N1` à `(N2 - 1)`.

```
?- nbVars(f(X,Y),1,N).
X = v(1), Y = v(2), N = 4 ;
false.
```

```
?- nbVars(f(g(X,Y),h(g(X),Z)),1,N).
X = v(1), Y = v(2), Z = v(3),
N = 4 ;
false.
```

▽ 88

TD 4.7 : Ensemble de solutions

1. On considère les faits suivants : `s(1). s(2). s(3). s(4). s(5).`
   (a) Définir le prédicat `s1(?Terme)` où `Terme` est la première solution de `s(Terme)`.

```
?- s1(1).
true.
```

```
?- s1(X).
X = 1.
```

   (b) Définir le prédicat `s2(?Terme)` où `Terme` est la deuxième solution de `s(Terme)`.

```
?- s2(1).
false.
```

```
?- s2(X).
X = 2 ;
false.
```

(c) Définir le prédicat `sd( ?Terme)` où `Terme` est la dernière solution de `s(Terme)`.

```
?- sd(1).
false.
```

```
?- sd(X).
X = 5 ;
false.
```

(d) Définir le prédicat `sn( ?Terme,+N)` où `Terme` est la $N^{ème}$ solution de `s(Terme)`.

```
?- sn(1,1).
true ;
false.
?- sn(X,4).
X = 4 ;
false.
?- sn(3,N).
N = 3 ;
false.
```

```
?- sn(X,N).
X = 1, N = 1 ;
X = 2, N = 2 ;
X = 3, N = 3 ;
X = 4, N = 4 ;
X = 5, N = 5 ;
false.
```

(e) Définir le prédicat `sall(-Liste)` où `Liste` contient toutes les solutions de `s(Terme)`.

```
?- sall(L).
L = [1, 2, 3, 4, 5].
```

2. Définir le prédicat `toutes( ?Terme,+But, ?Liste)` où `Liste` contient toutes les valeurs du `Terme` qui vérifient le `But`.

```
?- toutes(X,s(X),L).
L = [1, 2, 3, 4, 5].
?- toutes(X,(s(X),X =\= 4),L).
L = [1, 2, 3, 5].
```

```
?- toutes(X,(s(X),X > 5),L).
L = [].
```

▽ 89

TD 4.8 : Chaînage avant

Ecrire un interpréteur de règles en chaînage avant (`deduire/0`), qui permet d'effectuer des déductions à partir de règles et de faits.
Les règles seront de la forme : `regle(Conclusion,Conditions)`, et les faits : `regle(Fait,vrai)`.

```
?- listing(regle/2).
:- dynamic regle/2.
regle(pere(e,j),vrai).
regle(pere(j,f),vrai).
regle(gdPere(A,C),
      (pere(A,B),pere(B,C))).
regle(ancetre(A,B),pere(A,B)).
regle(ancetre(A,C),
      (pere(A,B),ancetre(B,C))).
true.
```

```
?- deduire.
gdPere(e, f) affirmé
ancetre(e, j) affirmé
ancetre(j, f) affirmé
ancetre(e, f) affirmé
true.
```

▽ 90

TD 4.9 : Mise sous forme clausale

Une forme clausale est une formule logique ne contenant que des disjonctions de littéraux positifs (forme prédicative $P(t_1, t_2, \ldots, t_n)$) ou négatifs (négation d'une forme prédicative $\neg P(t_1, \ldots, t_n)$).
Pour obtenir une forme clausale à partir d'une formule logique quelconque, on applique l'algorithme suivant (voir TD 1.2 page 9) :

1. élimination des $\Rightarrow$ $\quad a \Rightarrow b \rightarrow \neg a \vee b$
2. déplacement des $\neg$ vers l'intérieur $\quad \neg(a \vee b) \rightarrow \neg a \wedge \neg b$; $\neg(a \wedge b) \rightarrow \neg a \vee \neg b$

3. élimination des $\exists$ (“skolémisation”) $\forall x, \exists y P(y) \rightarrow P(f(x))$
4. déplacement des $\forall$ vers l'extérieur $\forall x, P(x) \vee \forall y, Q(y) \rightarrow \forall x \forall y (P(x) \vee Q(y))$
5. distribution de $\wedge$ sur $\vee$ $a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)$

On introduira les termes `tout/2` et `existe/2` ainsi que les opérateurs logiques `op(100,fx,non)`, `op(150,xfy,et)`, `op(200,xfy,ou)`, `op(250,xfy,imp)` et `op(300,xfy,eq)`.
Les exemples ci-dessous illustrent alors la traduction Prolog de formules logiques :

$\forall x\, P(x)$	$\rightarrow$ `tout(x,p(x))`
$\forall x\, (P(x) \Rightarrow Q(x))$	$\rightarrow$ `tout(x,imp(p(x),q(x)))`
$\forall x\, (\exists y\, (P(x) \Rightarrow Q(x, f(y))))$	$\rightarrow$ `tout(x,existe(y,imp(p(x),q(x,f(y)))))`

Pour les exemples ci-dessus, la mise sous forme clausale conduit aux clauses suivantes (voir le corrigé du TD 1.2 page 55) :

$\forall x\, P(x)$	$\rightarrow P(x)$	$\rightarrow$ `p(X).`
$\forall x\, (P(x) \Rightarrow Q(x))$	$\rightarrow \neg P(x) \vee Q(x)$	$\rightarrow$ `q(X) :- p(X).`
$\forall x\, (\exists y\, (P(x) \Rightarrow Q(x, f(y))))$	$\rightarrow \neg P(x) \vee Q(x, f(g(x)))$	$\rightarrow$ `q(X,f(g(X))) :- p(X).`

Définir le prédicat `toClause(+F,-L)` où `L` est la liste des formes clausales correspondant à la formule logique `F`.

```
?- listing(formule/2).
formule(1, tout(x, p(x))).
formule(2, tout(x, p(x)imp q(x))).
formule(3, tout(x, existe(y, p(x)imp q(x, f(y))))).
formule(4, tout(x, tout(y, p(x)imp (q(x, y)imp
        non existe(u, p(f(u))))ou (q(x, y)imp non r(y))))).
formule(5, tout(x, p(x)imp non tout(y, q(x, y)imp
        existe(u, p(f(u))))et tout(y, q(x, y)imp p(x)))).
true.

?- formule(N,F), toClause(F,C).
N = 1, F = tout(x, p(x)),
C = [c([p(x)], [])] ;
N = 2, F = tout(x, p(x)imp q(x)),
C = [c([q(x)], [p(x)])] ;
N = 3, F = tout(x, existe(y, p(x)imp q(x, f(y)))),
C = [c([q(x, f('$f2'(x)))], [p(x)])] ;
N = 4, F = tout(x, tout(y, p(x)imp (q(x, y)imp
                  non existe(u, p(f(u))))ou (q(x, y)imp non r(y)))),
C = [c([], [p(x), p(f(u)), r(y)])] ;
N = 5,
F = tout(x, p(x)imp non tout(y, q(x, y)imp
           existe(u, p(f(u))))et tout(y, q(x, y)imp p(x))),
C = [c([q(x, '$f3'(x))], [p(x)]), c([], [p(x), p(f(u))])] ;
false.
```

▽ 90

TD 5

Bases de données

On considère une base de données relationnelle[1] « fournisseurs–pièces–projets ». Elle est composée de 4 tables :

– La table des fournisseurs affecte à chaque fournisseur un numéro d'identification, un nom, une priorité et une ville.

Identification	Nom	Priorité	Ville
f1	martin	20	rennes
f2	albin	10	paris
f3	dupont	30	paris
f4	morin	20	rennes
f5	leach	30	brest

– La table des pièces détachées précise la référence, le nom, la couleur, le poids et le lieu de stockage de chaque pièce.

Référence	Nom	Couleur	Poids	Entrepôt
p1	ecrou	rouge	12	rennes
p2	boulon	vert	17	paris
p3	vis	bleu	17	grenoble
p4	vis	rouge	14	rennes
p5	came	bleu	12	paris
p6	engrenage	rouge	19	rennes

– La table des projets indique la référence, la nature et le lieu d'assemblage de chaque projet.

Référence	Nature	Site
pj1	disque	paris
pj2	scanner	grenoble
pj3	lecteur	brest
pj4	console	brest
pj5	capteur	rennes
pj6	terminal	bordeaux
pj7	bande	rennes

– La table des livraisons contient le numéro d'identification d'un fournisseur, la référence de la pièce à livrer, la quantité livrée ainsi que la référence du projet auquel cette pièce est

[1] Cet exemple est adapté de l'ouvrage de C. Date (1989) *Introduction au standard SQL*, InterEditions .

affectée.

Fournisseur	Pièce	Projet	Quantité
f1	p1	pj1	200
f1	p1	pj4	700
f2	p3	pj1	400
f2	p3	pj2	200
f2	p3	pj3	200
f2	p3	pj4	500
f2	p3	pj5	600
f2	p3	pj6	400
f2	p3	pj7	800
f2	p5	pj2	100
f3	p3	pj1	200
f3	p4	pj2	500
f4	p6	pj3	300
f4	p6	pj7	300
f5	p2	pj2	200
f5	p2	pj4	100
f5	p5	pj5	500
f5	p5	pj7	100
f5	p6	pj2	200
f5	p1	pj4	100
f5	p3	pj4	200
f5	p4	pj4	800
f5	p5	pj4	400
f5	p6	pj4	500

Pour chacune des questions suivantes, on pourra comparer les requêtes Prolog aux requêtes SQL[2] correspondantes.

TD 5.1 : Définition des données

1. Créer les 4 tables précédentes sous forme de faits Prolog.
2. Définir le prédicat `creerVue(+Vue,+But)` vrai si et seulement si la `Vue` (table temporaire) satisfaisant au `But`, a pu être créée à partir des tables existantes (tables permanentes ou autres vues).

   On veillera à ce qu'il n'y ait pas de doublons dans la vue.

   (a) Crér une vue `pjBrest(Reference,Nature)` donnant les informations concernant les projets en cours à `brest`.

   (b) Créer une vue `fpDistincts(Fournisseur,Piece)` faisant apparaître les numéros d'identification des fournisseurs et des pièces pour tous les fournisseurs et toutes les pièces non situés dans la même ville.

   (c) Créer une vue `pjF1P1(Projet,Ville)` faisant apparaître tous les projets alimentés par le fournisseur `f1` ou utilisant la pièce `p1`.

▽ 93

[2]La définition de la norme SQL ISO/IEC 9075 est donnée dans le document de l'International Standards Organization (1986), *Database language SQL*, ISO 1986.

TD 5.2 : Requêtes simples

1. Trouver tous les détails de chacun des projets.
2. Trouver tous les détails des projets rennais.
3. Trouver les références des fournisseurs du projet `pj1` .
4. Quelles sont les livraisons dont la quantité est comprise entre 300 et 750 ?
5. Quels sont les projets qui se déroulent dans une ville dont la quatrième lettre est un `n` *(ascii(*`n`*) = 110)* ! ?

▽ 95

TD 5.3 : Jointures

1. Trouver tous les triplets `(fournisseur,piece,projet)` tels que le fournisseur, la pièce et le projet soient situés dans la même ville.
2. Trouver tous les triplets `(fournisseur,piece,projet)` tels qu'un des éléments ne soit pas situé dans la même ville que les deux autres.
3. Trouver tous les triplets `(fournisseur,piece,projet)` tels que les trois éléments soient situés dans des villes différentes.
4. Trouver les références des pièces provenant d'un fournisseur rennais.
5. Trouver les références des pièces provenant d'un fournisseur rennais, et destinées à un projet rennais.
6. Trouver les numéros des projets dont au moins un des fournisseurs ne se trouve pas dans la même ville que celle où se déroule le projet.

▽ 96

TD 5.4 : Unions

1. Trouver la liste de toutes les villes dans lesquelles sont localisés au moins un fournisseur, une pièce ou un projet.
2. Trouver la liste de toutes les couleurs de pièces.

▽ 98

TD 5.5 : Mises à jour

1. Introduire un nouveau fournisseur : `(f10,ibm,100,lyon)`.
2. Changer la couleur de toutes les pièces rouges en orange.
3. Augmenter de 10% toutes les livraisons effectuées par les fournisseurs de pièces détachées oranges.
4. Ajouter 10 à la priorité de tous les fournisseurs dont la priorité est actuellement inférieure à celle du fournisseur `f4` .
5. Supprimer tous les projets se déroulant à Grenoble et toutes les livraisons correspondantes.

▽ 99

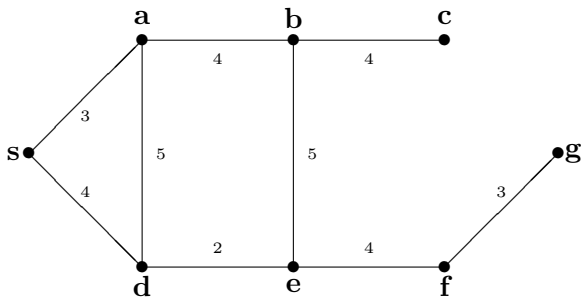
TD 6

Recherche dans les graphes

TD 6.1 : Réseau routier

On considère le graphe du réseau routier suivant :

```
route(s,a,3).
route(s,d,4).
route(b,a,4).
route(b,c,4).
route(b,e,5).
route(d,a,5).
route(e,d,2).
route(f,e,4).
route(f,g,3).
```



réseau routier

```
estim(a,g,10.4). estim(b,g,6.7). estim(c,g,4.0). estim(d,g, 8.9).
estim(e,g, 6.9). estim(f,g,3.0). estim(g,g,0.0). estim(s,g,12.5).
```

`route(?Ville1,?Ville2,?N)` : il existe une route directe de `N` km entre `Ville1` et `Ville2`.

`estim(?Ville1,?Ville2,?N)` : la distance à vol d'oiseau entre `Ville1` et `Ville2` est de `N` km.

Nous chercherons par la suite un chemin pour aller de `s` en `g` ; c'est pourquoi nous ne donnons ici que les distances à vol d'oiseau entre une ville quelconque du réseau et la ville `g` .

1. Dessiner l'arbre de recherche associé à ce réseau lorsqu'on veut aller de `s` à `g`.
2. Définir le prédicat `successeur(?Ville1,?Ville2,?N)` où `Ville1` et `Ville2` sont reliées par une route directe de `N` km.

```
?- successeur(s,a,N).
N = 3 ;
false.
```

```
?- successeur(s,X,N).
X = a, N = 3 ;
X = d, N = 4.
```

3. Définir le prédicat `chemin(+D,+A,-C,-K)` où `C` est la liste des villes par lesquelles il faut passer pour aller de la ville de départ `D` à la ville d'arrivée `A`, et `K` le nombre de kilomètres parcourus en passant par ce chemin.

```
?- chemin(s,g,C,K).
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, d, s], K = 25 ;
C = [g, f, e, d, s], K = 13 ;
false.
```

4. Afin de tester la généricité du prédicat `chemin/4` de la question précédente, on considère dans cette question le problème suivant.

   Soit un système composé de deux récipients non gradués R_1 et R_2, respectivement de capacité C_1 et C_2. L'état du système sera représenté par la paire (V_1, V_2) caractérisant la quantité d'eau contenue dans chacun des récipents. On veut faire passer le système de l'état initial (V_1^i, V_2^i) à l'état final (V_1^f, V_2^f) sachant que les seules opérations élémentaires autorisées sont :
   – remplir complètement un récipient,
   – vider complètement un récipient,
   – transvaser un récipient dans l'autre sans perdre une seule goutte.
   Pour fixer les idées, on prendra $C_1 = 8l$ et $C_2 = 5l$.

   (a) Définir les prédicats `remplir(+E1,-E2)`, `vider(+E1,-E2)` et `transvaser(+E1,-E2)` qui font passer le système de l'état `E1` à l'état `E2` en effectuant l'une des opérations élémentaires. En déduire la définition du prédicat `successeur(+E1,-E2,-N)` pour ce problème des vases.

```
?- remplir((1,3),(W1,W2)).
W1 = 8, W2 = 3 ;
W1 = 1, W2 = 5.
?- vider((1,3),(W1,W2)).
W1 = 0, W2 = 3 ;
W1 = 1, W2 = 0.
?- transvaser((1,3),(W1,W2)).
W1 = 0, W2 = 4 ;
W1 = 4, W2 = 0 ;
false.
```

```
?- successeur((1,3),(W1,W2),N).
W1 = 8, W2 = 3, N = 1 ;
W1 = 1, W2 = 5, N = 1 ;
W1 = 0, W2 = 3, N = 1 ;
W1 = 1, W2 = 0, N = 1 ;
W1 = 0, W2 = 4, N = 1 ;
W1 = 4, W2 = 0, N = 1 ;
false.
```

   (b) Montrer que le prédicat `chemin/4` permet de déterminer la suite des opérations élémentaires pour faire passer le système de l'état $E_1 = (0, 0)$ à l'état $E_2 = (4, 0)$.

```
?- chemin((0,0),(4,0),C,K).
C = [ (4, 0), (0, 4), (8, 4), (7, 5), (7, 0), (2, 5), (2, 0), (0, 2), (8, 2),
      (5, 5), (5, 0), (0, 5), (8, 5), (8, 0), (0, 0)],
K = 14 ;
...
C = [ (4, 0), (0, 4), (8, 4), (7, 5), (7, 0), (2, 5), (2, 0), (0, 2), (8, 2),
      (5, 5), (5, 0), (0, 5), (8, 5), (4, 5), (8, 1), (0, 1), (1, 0), (1, 5),
      (6, 0), (6, 5), (8, 3), (0, 3), (3, 0), (3, 5), (8, 0), (0, 0)],
K = 25 ;
...
C = [ (4, 0), (0, 4), (8, 4), (7, 5), (7, 0), (2, 5), (2, 0), (0, 2), (8, 2),
      (5, 5), (5, 0), (0, 5), (0, 0)],
K = 12 ;
...
C = [ (4, 0), (4, 5), (8, 1), (0, 1), (1, 0), (1, 5), (6, 0), (6, 5), (8, 3),
      (0, 3), (3, 0), (3, 5), (8, 0), (8, 2), (5, 5), (5, 0), (0, 5), (0, 0)],
K = 17 ;
false.
```

5. Il existe en fait plusieurs méthodes de parcours de l'arbre de recherche pour aller d'une ville à une autre du réseau. Le prédicat ci-dessous propose de généraliser le prédicat `chemin` de la question précédente en précisant la méthode de parcours de l'arbre de recherche (par exemple : recherche en `profondeur`, recherche en `largeur`, recherche du `meilleur` premier ou recherche `heuristique`).

```
% chemin/5
chemin(Methode,I,F,Chemin,Cout) :-
        initialiser(Methode,I,F,PilFil),
        rechercher(Methode,F,PilFil,Chemin,Cout).

% initialiser/4
initialiser(heuristique,I,F,[E-0-[I]]) :- !, estim(I,F,E).
initialiser(_,I,_,[0-[I]]).
```

La variable `PilFil` est une représentation de l'état d'avancement du parcours du graphe. C'est une liste dont les éléments sont de la forme `E-K-Villes` où `Villes` est la liste des villes par lesquelles on est déjà passé (exemple : `Villes = [b,a,s]`). `K` est le nombre de kilomètres effectifs qu'il a fallu parcourir pour suivre le chemin des `Villes` (exemple : `K = 7` pour `Villes = [b,a,s]`). `E` n'est utile que pour la méthode `heuristique` : elle représente la somme (`K + Estim`) de la distance déjà parcourue (`K`) pour arriver là où on en est dans l'arbre (exemple : en `b` pour `Villes = [b,a,s]` ; plus généralement en `T` pour `Villes = [T|Q]`) et d'une estimation (`Estim`) du chemin qui reste à parcourir entre la ville où on se trouve et la ville d'arrivée (`A`). On prendra `E = K` pour les méthodes autres qu'`heuristique`. Ainsi, pour une recherche heuristique d'un chemin entre `s` et `g`, l'état initial sera `[12.5-0-[s]]` et pour les autres méthodes il prendra la valeur `[0-0-[s]]`. L'état `[12.9-6-[e,d,s], 13.4-3-[a,s], 19.4-9-[a,d,s]]` signifie qu'on a déjà exploré 3 chemins `[e,d,s]`, `[a,s]` et `[a,d,s]` qui correspondent respectivement à des distances effectivement parcourues de `6`, `3` et `9` km et à des estimations de `12.9`, `13.4` et `19.4` km.

   (a) Définir le prédicat `successeurs(+Methode,+A,+E-K-[V|Q],-S)` où, selon la `Methode` considérée et pour la ville d'arrivée `A`, S est la liste des villes directement accessibles à partir de la ville `V` et par lesquelles on n'est pas déjà passé (ie. qui ne sont pas dans `[V|Q]`).

```
?- successeurs(profondeur,
                0-0-[s],g,S).
S = [3-3-[a, s], 4-4-[d, s]].
```

```
?- successeurs(heuristique,
                 12.5-0-[s],g,S).
S = [13.4-3-[a, s], 12.9-4-[d, s]].
```

   (b) Définir le prédicat `rechercher(+Methode,+A,+E,-C,-K)` où, selon la `Methode` considérée, `C` est le chemin qui mène à la ville d'arrivée `A` en `K` km pour un arbre de recherche dans l'état `E`.

```
?- rechercher(profondeur,
              g,[0-0-[s]],C,K).
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, d, s], K = 25 ;
C = [g, f, e, d, s], K = 13 ;
false.
?- rechercher(largeur,
              g,[0-0-[s]],C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
```

```
?- rechercher(meilleur,
              g,[0-0-[s]],C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
?- rechercher(heuristique,
              g,[0-0-[s]],C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, d, a, s],K = 17 ;
C = [g, f, e, b, a, s],K = 19 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
```

(c) En déduire la définition du prédicat `chemin(Methode,D,A,C,K)`.

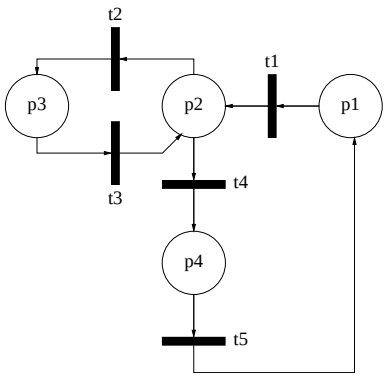
```
?- chemin(profondeur,s,g,C,K).
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, d, s], K = 25 ;
C = [g, f, e, d, s], K = 13 ;
false.
?- chemin(largeur,s,g,C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
```

```
?- chemin(meilleur,s,g,C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
?- chemin(heuristique,s,g,C,K).
C = [g, f, e, d, s], K = 13 ;
C = [g, f, e, d, a, s], K = 17 ;
C = [g, f, e, b, a, s], K = 19 ;
C = [g, f, e, b, a, d, s], K = 25 ;
false.
```

▽ 103

TD 6.2 : Réseaux de petri

Un réseau de Petri se représente par un graphe biparti orienté qui comporte deux types de nœuds : les places (représentées par des cercles) et les transitions (représentées par des traits). Les places et les transitions sont reliées par des arcs orientés qui lient soit une place à une transition, soit une transition à une place (voir figure ci-dessous).

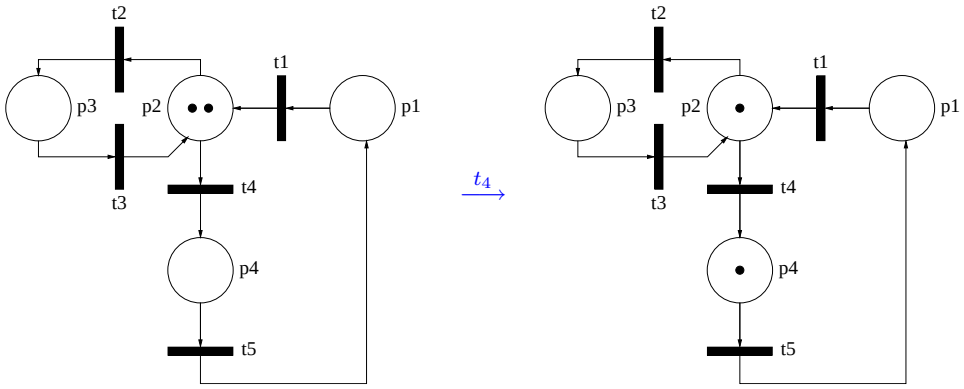


D'un point de vue formel, un réseau de Petri ordinaire est un quintuplet $< P, T, \text{Pre}, \text{Post}, M >$ tel que :

1. $P = \{p_1, p_2, \ldots, p_n\}$ est un ensemble fini et non vide de n places ($n > 0$).
2. $T = \{t_1, t_2, \ldots, t_m\}$ est un ensemble fini et non vide de m transitions ($m > 0$).
3. Les ensembles P et T sont disjoints : $P \cap T = \{\}$.
4. $\text{Pre}(p_i, t_j)$ est une application qui à chaque couple (p_i, t_j) du produit cartésien $P \times T$ associe un élément de l'ensemble $\{0, 1\}$ ($\text{Pre} : P \times T \rightarrow \{0, 1\}$). Elle traduit l'existence (1) ou non (0) d'un arc orienté d'une place vers une transition.
5. $\text{Post}(p_i, t_j)$ est une application qui à chaque couple (p_i, t_j) du produit cartésien $P \times T$ associe un élément de l'ensemble $\{0, 1\}$ ($\text{Post} : P \times T \rightarrow \{0, 1\}$). Elle traduit l'existence (1) ou non (0) d'un arc orienté d'une transition vers une place.
6. M est une application qui à chaque place p_i associe un nombre entier m_i (le marquage de la place) positif ou nul ($m_i \geq 0$) ($M : P \rightarrow \mathcal{N}$). Une marque est représentée par un $\bullet$ dans la place considérée.

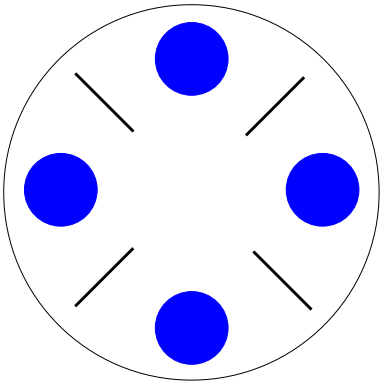
Une transition t_j d'un réseau de Petri $< P, T, \text{Pre}, \text{Post}, M >$ est valide si elle vérifie la condition $t_j \in T, \forall p_i \in P, m_i \geq \text{Pre}(p_i, t_j)$ Ce qui revient à dire que chacune des places amont de la transition t_j contient au moins une marque. Lorsqu'une transition est valide, on peut alors la franchir et, lorsque plusieurs transitions sont valides au même instant, une seule de ces transitions sera franchie. Le choix de la transition à franchir est *a priori* aléatoire : on parle de non-déterminisme des réseaux de Petri.

La règle de franchissement d'une transition valide consiste à retirer une marque dans chacune de ses places amont et à ajouter une marque dans chacune de ses places aval, comme dans le cas du franchissement de la transition t_4 de la figure ci-dessous.



Dans la suite, chaque place sera définie par un fait `place/2` où le premier argument est le nom de la place et le deuxième la description de l'état du système qu'elle représente (`place(nom, description)`). Une transition sera définie par un fait `transition/3` où le premier argument est le nom de la transition, le deuxième la liste de ses places amont et le troisième la liste de ses places aval (`transition(nom,amont,aval)`). Le marquage d'une place est stocké dans un fait `marquage/2` où le premier argument est le nom de la place considérée et le deuxième le marquage associé (`marquage(place,marquage)`). Pour fixer les idées, on considère ici le problème dit du dîner des philosophes chinois.

Quatre philosophes chinois se retrouvent autour d'une table circulaire ; chacun a devant lui un plat de nouilles et à sa droite une baguette. Pour pouvoir manger, chaque philosophe ne peut le faire que si les baguettes à sa gauche et à sa droite sont disponibles. Quand il ne mange pas, le philosophe pense.
Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger et penser.



Un réseau de Petri qui illustre cette situation peut être défini par les faits suivants.

```
% place/2
place(p11,'philosophe 1 pense').
place(p12,'philosophe 1 mange').
place(p21,'philosophe 2 pense').
place(p22,'philosophe 2 mange').
place(p31,'philosophe 3 pense').
place(p32,'philosophe 3 mange').
place(p41,'philosophe 4 pense').
place(p42,'philosophe 4 mange').
place(b1,'baguette 1 libre').
place(b2,'baguette 2 libre').
place(b3,'baguette 3 libre').
place(b4,'baguette 4 libre').
```

```
% transition/3
transition(t11,[p11,b1,b2],[p12]).
transition(t12,[p12],[p11,b1,b2]).
transition(t21,[p21,b2,b3],[p22]).
transition(t22,[p22],[p21,b2,b3]).
transition(t31,[p31,b3,b4],[p32]).
transition(t32,[p32],[p31,b3,b4]).
transition(t41,[p41,b4,b1],[p42]).
transition(t42,[p42],[p41,b4,b1]).
```

Dans l'état initial, les baguettes sont libres et les 4 philosophes pensent.

```
% marquage/2
:- dynamic marquage/2.

marquage(b1,1).
marquage(b2,1).
marquage(b3,1).
marquage(b4,1).
marquage(p11,1).
marquage(p12,0).
marquage(p21,1).
marquage(p22,0).
marquage(p31,1).
marquage(p32,0).
marquage(p41,1).
marquage(p42,0).
```

1. Représenter graphiquement le réseau de Petri ainsi défini.
2. Définir les prédicats suivants :
   (a) `putState` : affiche l'état du système.

```
?- putState.
philosophe 1 pense
philosophe 2 pense
philosophe 3 pense
philosophe 4 pense
baguette 1 libre
baguette 2 libre
baguette 3 libre
baguette 4 libre
true.
```

   (b) `valids(-Ts)` : `Ts` est la liste des transitions valides dans un état donné.

```
?- valids(Ts).
Ts = [t11, t21, t31, t41].
```

   (c) `tir(+T)` : franchir la transition `T`.

```
?- putState.
philosophe 1 pense
philosophe 2 pense
philosophe 3 pense
philosophe 4 pense
baguette 1 libre
baguette 2 libre
baguette 3 libre
baguette 4 libre
true.
?- valids(Ts).
Ts = [t11, t21, t31, t41].
```

```
?- tir(t11).
true.
?- putState.
philosophe 2 pense
philosophe 3 pense
philosophe 4 pense
baguette 3 libre
baguette 4 libre
philosophe 1 mange
true.
```

3. Définir le prédicat `simulRdp(+C)` qui simule pas à pas un réseau de Petri. A chaque pas de simulation, on demande à l'utilisateur s'il veut continuer (`C = continue`) ou non (`C = stop`).

```
?- putState.
philosophe 1 pense
philosophe 2 pense
philosophe 3 pense
philosophe 4 pense
baguette 1 libre
baguette 2 libre
baguette 3 libre
baguette 4 libre
true.

?- simulRdp(continue).
Transitions valides : [t11,t21,t31,t41]
Transition choisie  : t31
Etat du systeme     :
philosophe 1 pense
philosophe 2 pense
philosophe 4 pense
baguette 1 libre
baguette 2 libre
philosophe 3 mange

Continue (y/n) ? y.

Transitions valides : [t11,t32]
Transition choisie  : t11
Etat du systeme     :
philosophe 2 pense
philosophe 4 pense
philosophe 3 mange
philosophe 1 mange

Continue (y/n) ? y.

Transitions valides : [t12,t32]
Transition choisie  : t12
Etat du systeme     :
philosophe 2 pense
philosophe 4 pense
philosophe 3 mange
philosophe 1 pense
baguette 1 libre
baguette 2 libre

Continue (y/n) ? n.
true.
```

▽ 105

Deuxième partie

Corrigés

Corrigés TD 1

De la logique à Prolog

TD 1.1 : Puzzle logique

△ 9

Chaque maison est représentée ici par un quadruplet (N, C, P, S) où N représente le numéro dans la rue ($N \in \{1, 2, 3\}$), C la couleur ($C \in \{\text{blanc}, \text{bleu}, \text{vert}\}$), P le pays d'origine ($P \in \{\text{anglais}, \text{espagnol}, \text{français}\}$) et S le sport pratiqué ($S \in \{\text{football}, \text{natation}, \text{tennis}\}$). Ainsi, les 5 indices peuvent s'écrire :

- Dans la maison verte on pratique la natation : $(n_1, \text{vert}, p_1, \text{natation})$
- La maison verte est située avant la maison de l'espagnol : $(n_2, c_2, \text{espagnol}, s_2)$ avec $n_1 < n_2$
- L'anglais habite la maison blanche : $(n_3, \text{blanc}, \text{anglais}, s_3)$
- La maison blanche est située avant la maison où on pratique le football : $(n_4, c_4, p_4, \text{football})$ avec $n_3 < n_4$
- Le tennisman habite au début de la rue : $(1, c_5, p_5, \text{tennis})$

Des indices 1 et 5, on déduit : $n_1 \neq 1$. De l'indice 2, on déduit alors $n_1 = 2$ et $n_2 = 3$; d'où :

$$(1, c_5, p_5, \text{tennis}), (2, \text{vert}, p_1, \text{natation}), (3, c_2, \text{espagnol}, s_2)$$

De l'indice 4, on peut maintenant affirmer que $n_4 = 3$ et donc que $s_2 = s_4 = \text{football}$; d'où :

$$(1, c_5, p_5, \text{tennis}), (2, \text{vert}, p_1, \text{natation}), (3, c_2, \text{espagnol}, \text{football})$$

De l'indice 3, on déduit que la seule possibilité qui reste pour l'anglais est la maison 1 :

$$(1, \text{blanc}, \text{anglais}, \text{tennis}), (2, \text{vert}, p_1, \text{natation}), (3, c_2, \text{espagnol}, \text{football})$$

Enfin de l'indice 2, $p_1 \neq \text{espagnol}$ or $p_1 \neq \text{anglais}$, on en déduit la solution :

$$(1, \text{blanc}, \text{anglais}, \text{tennis}), (2, \text{vert}, \text{français}, \text{natation}), (3, \text{bleu}, \text{espagnol}, \text{football})$$

→ voir TD 3.7 pour une résolution automatique de ce problème.

TD 1.2 : Formes clausales

△ 9

1. $\forall x\, P(x)$
   (a) $\forall x\, P(x)$
   (b) $\forall x\, P(x)$
   (c) $\forall x\, P(x)$
   (d) $P(x)$
   (e) $P(x)$
   (f) 1 clause : $P(x)$
   (g) **Prolog** : `p(X).`

2. $\forall x\,(P(x) \Rightarrow Q(x))$
   (a) $\forall x\,(\neg P(x) \vee Q(x))$
   (b) $\forall x\,(\neg P(x) \vee Q(x))$
   (c) $\forall x\,(\neg P(x) \vee Q(x))$
   (d) $\neg P(x) \vee Q(x)$
   (e) $\neg P(x) \vee Q(x)$
   (f) 1 clause : $\neg P(x) \vee Q(x)$
   (g) **Prolog** : `q(X) :- p(X).`
3. $\forall x\,(\exists y\,(P(x) \Rightarrow Q(x, f(y))))$
   (a) $\forall x\,(\exists y\,(\neg P(x) \vee Q(x, f(y))))$
   (b) $\forall x\,(\exists y\,(\neg P(x) \vee Q(x, f(y))))$
   (c) $\forall x\,(\neg P(x) \vee Q(x, f(g(x))))$
   (d) $\neg P(x) \vee Q(x, f(g(x)))$
   (e) $\neg P(x) \vee Q(x, f(g(x)))$
   (f) 1 clause : $\neg P(x) \vee Q(x, f(g(x)))$
   (g) **Prolog** : `q(X,f(g(X))) :- p(X).`
4. $\forall x, y\,(P(x) \Rightarrow ((Q(x, y) \Rightarrow \neg(\exists u\,P(f(u)))) \vee (Q(x, y) \Rightarrow \neg R(y))))$
   (a) $\forall x, y\,(\neg P(x) \vee ((\neg Q(x, y) \vee \neg(\exists u\,P(f(u)))) \vee (\neg Q(x, y) \vee \neg R(y))))$
   (b) $\forall x, y\,(\neg P(x) \vee ((\neg Q(x, y) \vee (\forall u\,\neg P(f(u)))) \vee (\neg Q(x, y) \vee \neg R(y))))$
   (c) $\forall x, y\,(\neg P(x) \vee ((\neg Q(x, y) \vee (\forall u\,\neg P(f(u)))) \vee (\neg Q(x, y) \vee \neg R(y))))$
   (d) $\neg P(x) \vee \neg Q(x, y) \vee \neg P(f(u)) \vee \neg Q(x, y) \vee \neg R(y)$
   (e) $\neg P(x) \vee \neg Q(x, y) \vee \neg P(f(u)) \vee \neg Q(x, y) \vee \neg R(y)$
   (f) 1 clause : $\neg P(x) \vee \neg Q(x, y) \vee \neg P(f(u)) \vee \neg Q(x, y) \vee \neg R(y)$
   (g) **Prolog** : `:- p(X), q(X,Y), p(f(U)), q(X,Y), r(Y).`
5. $\forall x\,(\,P(x) \Rightarrow (\,(\neg\forall y\,(Q(x, y) \Rightarrow (\exists u\,P(f(u))))) \wedge (\forall y\,(Q(x, y) \Rightarrow P(x)))\,)\,)$
   (a) $\forall x\,(\,\neg P(x) \vee (\,(\neg\forall y\,(\neg Q(x, y) \vee (\exists u\,P(f(u))))) \wedge (\forall y\,(\neg Q(x, y) \vee P(x)))\,)\,)$
   (b) $\forall x\,(\,\neg P(x) \vee (\,(\exists y\,(Q(x, y) \wedge (\forall u\,\neg P(f(u))))) \wedge (\forall y\,(\neg Q(x, y) \vee P(x)))\,)\,)$
   (c) $\forall x\,(\,\neg P(x) \vee (\,((Q(x, g(x)) \wedge (\forall u\,\neg P(f(u))))) \wedge (\forall y\,(\neg Q(x, y) \vee P(x)))\,)\,)$
   (d) $\neg P(x) \vee (\,(Q(x, g(x)) \wedge \neg P(f(u))) \wedge (\neg Q(x, y) \vee P(x))\,)$
   (e) $(\neg P(x) \vee Q(x, g(x))) \wedge (\neg P(x) \vee \neg P(f(u))) \wedge (\neg P(x) \vee \neg Q(x, y) \vee P(x))$
   (f) 2 clauses :
   $\neg P(x) \vee Q(x, g(x))$,
   $\neg P(x) \vee \neg P(f(u))$ et
   $\neg P(x) \vee \neg Q(x, y) \vee P(x)$ toujours vraie :
   $\neg P(x) \vee \neg Q(x, y) \vee P(x) = (\neg P(x) \vee P(x)) \vee \neg Q(x, y) = 1 \vee \neg Q(x, y) = 1$
   (g) **Prolog** :
   ```
   q(X,g(X)) :- p(X).
   :- p(X), p(f(U)).
   ```

TD 1.3 : Des relations aux prédicats

△ 10

Classiquement, dans une phrase, on choisit le verbe comme prédicat, le sujet et les attributs comme arguments du prédicat ; ce qui peut donner :

Listing 1.1 – des relations aux prédicats

```
% est/3
est(thomson,entreprise,dynamique).
est(enib,ecole,ingenieurs).
est('champ magnetique',champ,'flux conservatif').
est(mozart,auteur,'La flute enchantée').
est(moliere,auteur,'L''avare').

% aime/2
aime(voisine,chat).
aime(voisin,chat).

% sonne/2
sonne(facteur,2).
```

TD 1.4 : Calculs booléens

△ 10

1. **Opérateurs logiques :** de simples faits suffisent pour définir les 3 opérateurs de base.

   Listing 1.2 – opérateurs logiques

   ```
   % non/2
   non(0,1). non(1,0).

   % et/3
   et(0,0,0). et(0,1,0). et(1,0,0). et(1,1,1).

   % ou/3
   ou(0,0,0). ou(0,1,1). ou(1,0,1). ou(1,1,1).
   ```

2. **Opérateurs dérivés :** pour définir les opérateurs dérivés à partir des 3 opérateurs de base, on utilise les relations suivantes : $a \oplus b = \overline{a} \cdot b + a \cdot \overline{b}$, $a \Rightarrow b = \overline{a} + b$ et $a \Leftrightarrow b = a \cdot b + \overline{a} \cdot \overline{b}$.

   Listing 1.3 – opérateurs dérivés

   ```
   % xor/3
   xor(X,Y,Z) :- non(X,U), et(U,Y,T), non(Y,V), et(X,V,W), ou(T,W,Z).

   % nor/3
   nor(X,Y,Z) :- ou(X,Y,T), non(T,Z).

   % nand/3
   nand(X,Y,Z) :- et(X,Y,T), non(T,Z).

   % imply/3
   imply(X,Y,Z) :- non(X,U), ou(U,Y,Z).

   % equiv/3
   equiv(X,Y,Z) :- non(X,U), non(Y,V), et(X,Y,T), et(U,V,W), ou(T,W,Z).
   ```

3. **Propriétés des opérateurs logiques :** de simples appels Prolog permettent de vérifier les différentes propriétés des opérateurs booléens.

commutativité	`?- et(X,Y,T), et(Y,X,T).` `?- ou(X,Y,T), ou(Y,X,T).`
associativité	`?- et(X,Y,U), et(U,Z,T), et(Y,Z,V), et(X,V,T).` `?- ou(X,Y,U), ou(U,Z,T), ou(Y,Z,V), ou(X,V,T).`
distributivité	`?- ou(Y,Z,W), et(X,W,T), et(X,Y,U), et(X,Z,V), ou(U,V,T).` `?- et(Y,Z,W), ou(X,W,T), ou(X,Y,U), ou(X,Z,V), et(U,V,T).`
idempotence	`?- et(X,X,X).` `?- ou(X,X,X).`
complémentarité	`?- non(X,T), et(X,T,0).` `?- non(X,T), ou(X,T,1).`
De Morgan	`?- et(X,Y,Z), non(Z,T), non(X,U), non(Y,V), ou(U,V,T).` `?- ou(X,Y,Z), non(Z,T), non(X,U), non(Y,V), et(U,V,T).`

4. **Circuits logiques**

Listing 1.4 – circuits logiques

```
% ou3/4
ou3(X,Y,Z,T) :- ou(X,Y,U), ou(U,Z,T).

% et3/4
et3(X,Y,Z,T) :- et(X,Y,U), et(U,Z,T).

% nonet3/
nonet(X,Y,Z,T) :- et3(X,Y,Z,U), non(U,T).

% circuit1/3
circuit1(A,B,S) :- non(A,U), non(B,V), et(A,U,W), et(U,B,T), ou(W,T,S).

% circuit2/3
circuit2(A,B,S) :- non(A,U), non(B,V), et(A,U,W), et(U,B,T), et(W,T,S).

% circuit3/4
circuit3(A,B,S,T) :- xor(A,B,S), non(A,U), et(U,B,T).

% circuit4/5
circuit4(A,B,C,S,T) :-
        et(B,C,U), xor(B,C,V), et(A,V,W), ou(U,W,T), xor(A,V,S).

% circuit5/4
circuit5(A,B,C,S) :- et(A,B,T), et(A,C,U), et(B,C,V), ou3(T,U,V,S)

% circuit6/5
circuit6(A,B,C,S,T) :-
        non(B,V), non(C,W), et(A,W,X), et3(A,V,C,U), et(B,C,Y),
        ou(X,U,S), ou(U,Y,T).

% circuit7/4
circuit7(A,B,C,S) :-
        non(A,X), non(B,Y), non(C,Z),
        nonet3(X,Y,Z,U), nonet3(A,Y,Z,V), nonet3(A,B,Z,W), nonet3(U,V,W,S).

% circuit8/11
circuit8(A,B,C,S0,S1,S2,S3,S4,S5,S6,S7) :-
```

```
38          non(A,X), non(B,Y), non(C,Z),
39          et3(X,Y,Z,S0), et3(X,Y,C,S1), et3(X,B,Z,S2), et3(X,B,C,S3),
40          et3(A,Y,Z,S4), et3(A,Y,C,S5), et3(A,B,Z,S6), et3(A,B,C,S7).
```

TD 1.5 : Une base de données Air-Enib

△ 11

1. **Vols Air-Enib**

Listing 1.5 – vols Air-Enib

```
1 vol(1,brest,paris,1400,1500,200).
2 vol(2,brest,lyon,0600,0800,100).
3 vol(3,brest,londres,0630,0800,75).
4 vol(4,lyon,paris,1200,1300,250).
5 vol(5,lyon,paris,1300,1400,250).
```

2. **Requêtes sur les vols Air-Enib**

   (a) `?- vol(N,brest,_,_,_,_).`

   (b) `?- vol(N,_,paris,_,_,_).`

   (c) `?- vol(N,brest,_,H,_,_), H =< 1200.`

   (d) `?- vol(N,_,paris,_,H,_), H > 1400.`

   (e) `?- vol(N,_,paris,_,H,P), H < 1700, P > 100.`

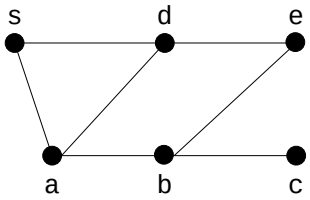
   (f) `?- vol(N1,V1,_,H,_,_), vol(N2,V2,_,H,_,_), V1 \== V2.`

   (g) `?- vol(N,_,_,D,A,_), A-D > 0200.`

TD 1.6 : Un petit réseau routier

△ 12

1. **Réseau routier**



2. **Questions simples :** on joue sur l'instanciation des arguments pour obtenir les questions les plus simples.

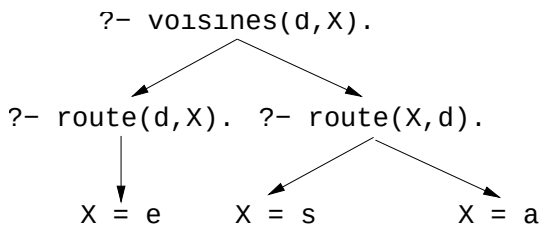
   `?- route(s,e).` `?- route(s,X).` `?- route(X,e).` `?- route(X,Y).`

3. **Villes voisines**

Listing 1.6 – villes voisines

```
1 voisines(X,Y) :- route(X,Y) ; route(Y,X).
```

4. **Arbre de résolution :**




```
[trace]  ?- voisines(d,X).
   Call: (7) voisines(d, _G336) ? creep
   Call: (8) route(d, _G336) ? creep
   Exit: (8) route(d, e) ? creep
   Exit: (7) voisines(d, e) ? creep
X = e ;
   Call: (8) route(_G336, d) ? creep
   Exit: (8) route(s, d) ? creep
   Exit: (7) voisines(d, s) ? creep
X = s ;
   Redo: (8) route(_G336, d) ? creep
   Exit: (8) route(a, d) ? creep
   Exit: (7) voisines(d, a) ? creep
X = a ;
   Redo: (8) route(_G336, d) ? creep
   Fail: (8) route(_G336, d) ? creep
   Fail: (7) voisines(d, _G336) ? creep
false.
```

TD 1.7 : Coloriage d'une carte

△ 12

Listing 1.7 – coloriage d'une carte

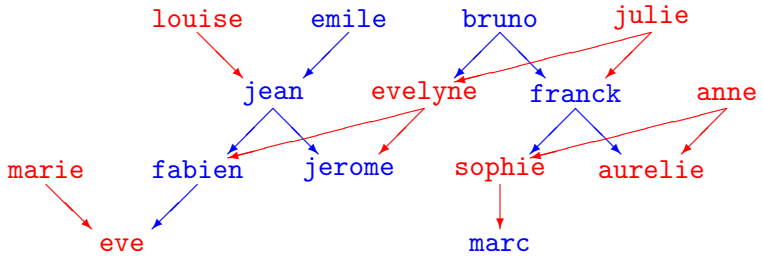
```
1 % couleur/1
2 couleur(rouge). couleur(jaune). couleur(bleu). couleur(vert).
3
4 % carte/6
5 carte(C1,C2,C3,C4,C5,C6) :-
6         couleur(C1), couleur(C2), couleur(C3),
7         couleur(C4), couleur(C5), couleur(C6),
8         C1 \== C2, C1 \== C3, C1 \== C5, C1 \== C6, C2 \== C3, C2 \== C4,
9         C2 \== C5, C2 \== C6, C3 \== C4, C3 \== C6, C5 \== C6.
```

```
?- carte(C1,C2,C3,C4,C5,C6).
C1 = rouge, C2 = jaune, C3 = bleu, C4 = rouge, C5 = bleu, C6 = vert ;
C1 = rouge, C2 = jaune, C3 = bleu, C4 = vert, C5 = bleu, C6 = vert ;
C1 = rouge, C2 = jaune, C3 = vert, C4 = rouge, C5 = vert, C6 = bleu ;
...
C1 = vert, C2 = bleu, C3 = jaune, C4 = vert, C5 = jaune, C6 = rouge ;
false.
```

TD 1.8 : Arbre généalogique

△ 13

1. **Représentation graphique**



2. **Prédicats de filiation :** à ce niveau introductif, et compte-tenu des données limitées du problème, certaines questions conduiront logiquement à des doublons. On ne cherchera pas à les éliminer.

Listing 1.8 – généalogie

```
% parent/2
parent(X,Y) :- mere(X,Y) ; pere(X,Y).

% fils/2
fils(X,Y) :- parent(Y,X), homme(X).

% fille/2
fille(X,Y) :- parent(Y,X), femme(X).

% femme/2
femme(X,Y) :- fils(Z,X), femme(X), fils(Z,Y), homme(Y).
femme(X,Y) :- fille(Z,X), femme(X), fille(Z,Y), homme(Y).

% mari/2
mari(X,Y) :- femme(Y,X).

% gdPere/2
gdPere(X,Y) :- pere(X,Z), parent(Z,Y).

% gdMere/2
gdMere(X,Y) :- mere(X,Z), parent(Z,Y).

% gdParent/2
gdParent(X,Y) :- parent(X,Z), parent(Z,Y).

% aGdPere/2
aGdPere(X,Y) :- pere(X,Z), gdParent(Z,Y).

% aGdMere/2
aGdMere(X,Y) :- mere(X,Z), gdParent(Z,Y).

% frereSoeur/2
frereSoeur(X,Y) :- fils(X,Z), fille(Y,Z).
frereSoeur(X,Y) :- fille(X,Z), fils(Y,Z).
frereSoeur(X,Y) :- fille(X,Z), fille(Y,Z), X \== Y.
frereSoeur(X,Y) :- fils(X,Z), fils(Y,Z), X \== Y.

% frere/2
frere(X,Y) :- frereSoeur(X,Y), homme(X).

% soeur/2
soeur(X,Y) :- frereSoeur(X,Y), femme(X).

% beauFrere/2
beauFrere(X,Y) :- mari(X,Z), frereSoeur(Z,Y).
beauFrere(X,Y) :- frere(X,Z), (mari(Z,Y) ; femme(Z,Y)).

% belleSoeur/2
belleSoeur(X,Y) :- femme(X,Z), frereSoeur(Z,Y).
belleSoeur(X,Y) :- soeur(X,Z), (mari(Z,Y) ; femme(Z,Y)).

% ancetre/2
ancetre(X,Y) :- parent(X,Y).
ancetre(X,Y) :- parent(X,Z), ancetre(Z,Y).
```

Corrigés TD 2

Termes

TD 2.1 : Etat-civil

△ 15

1. **Fait Prolog**

Listing 2.1 – fiches d'état-civil

```
% individu/2
% état-civil : ec(nom,prénom,date de naissance,nationalité,sexe)
% date       :  d(jour,mois,année)
% adresse    : ad(rue,ville,département)
individu(ec(ngaoundere,richard,d(15,2,1960),cameroun,m),
         ad('4 rue leclerc',brest,29)).
individu(ec(ngaoundere,ludovine,d(16,3,1953),cameroun,f),
         ad('4 rue leclerc',brest,29)).
individu(ec(martin,jean,d(19,10,1945),france,m),
         ad('21 rue de brest',lyon,59)).
```

2. **Requêtes Prolog**
   (a) `?- individu(ec(N,P,_,france,_),_).`
   (b) `?- individu(ec(N,P,_,Nation,_),_), Nation \== france.`
   (c) `?- individu(ec(N,P,_,Nation,_),ad(_,brest,29), Nation \== france.`
   (d) `?- individu(ec(N1,_,_,_,_),A), individu(ec(N2,_,_,_,_),A), N1 \== N2.`

TD 2.2 : Entiers naturels

△ 15

Listing 2.2 – entiers naturels

```
% entier/1
entier(z).
entier(s(X)) :- entier(X).

% plus/3
plus(X,z,X).
plus(X,s(Y),s(Z)) :- plus(X,Y,Z).

% fois/3
fois(_,z,z).
fois(X,s(Y),Res) :- fois(X,Y,Z) , plus(Z,X,Res).

% expo/3
```

```
14 expo(_,z,s(z)).
15 expo(X,s(Y),Res) :- expo(X,Y,Z) , fois(Z,X,Res).
16
17 % pred/2
18 pred(s(X),X).
19
20 % moins/3
21 moins(X,z,X).
22 moins(s(X),s(Y),Z) :- moins(X,Y,Z).
23
24 % lte/2
25 lte(z,_).
26 lte(X,Y) :- pred(X,PX), pred(Y,PY) , lte(PX,PY).
27
28 % lt/2
29 lt(X,Y) :- lte(X,Y) , X \== Y.
30
31 % quotient/3
32 quotient(X,Y,z) :- Y \== z , lt(X,Y).
33 quotient(Z,Y,s(Q)) :- Y \== z , plus(X,Y,Z) , quotient(X,Y,Q).
34
35 % reste/3
36 reste(X,Y,X) :- Y \== z , lt(X,Y).
37 reste(Z,Y,R) :- Y \== z , plus(X,Y,Z) , reste(X,Y,R).
38
39 % pgcd/3
40 pgcd(X,z,X).
41 pgcd(X,Y,P) :- Y \== z , reste(X,Y,R) , pgcd(Y,R,P).
42
43 % fact/2
44 fact(z,s(z)).
45 fact(s(X),F) :- fact(X,Y) , fois(s(X),Y,F).
```

TD 2.3 : Polynômes

△ 16

Listing 2.3 – polynômes

```
1 % polynome/2
2 polynome(C,X) :- atomic(C), C \== X.
3 polynome(X,X).
4 polynome(X^N,X) :- atomic(N), N \== X.
5 polynome(U+V,X) :- polynome(U,X), polynome(V,X).
6 polynome(U-V,X) :- polynome(U,X), polynome(V,X).
7 polynome(U*V,X) :- polynome(U,X), polynome(V,X).
```

TD 2.4 : Arithmétique

△ 16

1. **Fonctions mathématiques**

Listing 2.4 – fonctions mathématiques

```
1 % abs/2
2 abs(X,X) :- X >= 0.
3 abs(X,Y) :- X < 0, Y is -X.
4
5 % min/3
```

```
6  min(X,Y,X) :- X =< Y.
7  min(X,Y,Y) :- X > Y.
8
9  % pgcd/3
10 pgcd(X,0,X).
11 pgcd(X,Y,P) :- Y =\= 0, R is X mod Y, pgcd(Y,R,P).
12
13 % ppcm/3
14 ppcm(X,Y,P) :- pgcd(X,Y,D), P is X*Y/D.
15
16 % ch/2
17 ch(X,Y) :- Y is (exp(X) + exp(-X))/2.
```

2. **Suites récurrentes**

Listing 2.5 – suites récurrentes

```
1  % factorielle/2
2  factorielle(0,1).
3  factorielle(N,F) :-
4          N > 0, N1 is N-1, factorielle(N1,F1),
5          F is N*F1.
6
7  % fibonacci/2
8  fibonacci(0,1).
9  fibonacci(1,1).
10 fibonacci(N,F) :-
11         N > 1, N1 is N-1, N2 is N-2,
12         fibonacci(N1,F1), fibonacci(N2,F2),
13         F is F1 + F2.
```

3. **Nombres complexes**

Listing 2.6 – nombres complexes

```
1  % argC/2
2  argC(c(R,I),A) :- R =\= 0, A is atan(I/R).
3  argC(c(0,I),A) :- I > 0, A is 3.14159/2.
4  argC(c(0,I),A) :- I < 0, A is -3.14159/2.
5
6  % modC/2
7  modC(c(R,I),Mod) :- Mod is sqrt(R*R + I*I).
8
9  % addC/3
10 addC(c(R1,I1),c(R2,I2),c(R3,I3)) :- R3 is R1 + R2, I3 is I1 + I2.
11
12 % mulC/3
13 mulC(c(R1,I1),c(R2,I2),c(R3,I3)) :- R3 is R1*R2 - I1*I2, I3 is R1*I2 + R2*I1.
14
15 % conj/2
16 conj(c(R,I),c(R,IC)) :- IC is -I.
```

4. **Intervalles**

Listing 2.7 – intervalles

```
1  % dans/3
2  dans(Min,Min,Max) :- Min =< Max.
3  dans(I,Min,Max) :- Min < Max, Min1 is Min+1, dans(I,Min1,Max).
```

TD 2.5 : Arbres et dictionnaires binaires

△ 17

1. Arbres binaires

Listing 2.8 – arbres binaires

```
% arbreBinaire/1
arbreBinaire([]).
arbreBinaire(bt(G,_,D)) :- arbreBinaire(G), arbreBinaire(D).

% dansArbre/2
dansArbre(X,bt(G,X,D)).
dansArbre(X,bt(G,Y,D)) :- dansArbre(X,G).
dansArbre(X,bt(G,Y,D)) :- dansArbre(X,D).

% profondeur/2
profondeur([],0).
profondeur(bt(G,X,D),N) :-
        profondeur(G,NG), profondeur(D,ND),
        max(NG,ND,M), N is M+1.
```

2. Dictionnaires binaires

Listing 2.9 – dictionnaires binaires

```
% racine/2
racine(bt(_,X,_),X).
racine([],'racine vide').

% plusGrand/2
plusGrand('racine vide',X) :- X \== 'racine vide'.
plusGrand(X,'racine vide') :- X \== 'racine vide'.
plusGrand(X,Y) :- X \== 'racine vide', Y \== 'racine vide', X @> Y.

% dicoBinaire/1
dicoBinaire([]).
dicoBinaire(bt(G,X,D)) :-
        racine(G,RG), plusGrand(X,RG), dicoBinaire(G),
        racine(D,RD), plusGrand(RD,X), dicoBinaire(D).

% inserer/3
inserer(X,[],bt([],X,[])).
inserer(X,bt(G,X,D),bt(G,X,D)).
inserer(X,bt(G,Y,D),bt(G1,Y,D)) :- X @< Y, inserer(X,G,G1).
inserer(X,bt(G,Y,D),bt(G,Y,D1)) :- X @> Y, inserer(X,D,D1).

% supprimer/3
supprimer(X,bt([],X,D),D).
supprimer(X,bt(G,X,[]),G) :- G \== [].
supprimer(X,bt(G,X,D),bt(G,Y,D1)) :-
        G \== [], D \== [], transfert(D,Y,D1).
supprimer(X,bt(G,Y,D),bt(G,Y,D1)) :- X @> Y, supprimer(X,D,D1).
supprimer(X,bt(G,Y,D),bt(G1,Y,D)) :- X @< Y, supprimer(X,G,G1).

% transfert/3
transfert(bt([],Y,D),Y,D).
transfert(bt(G,Z,D),Y,bt(G1,Z,D)) :- transfert(G,Y,G1).
```

TD 2.6 : Termes de base

△ 17

Listing 2.10 – termes de base

```
% base/1
base(T) :- atomic(T).
base(T) :- compound(T), functor(T,F,N), base(N,T).

% base/2
base(N,T) :- N > 0, arg(N,T,Arg), base(Arg), N1 is N-1, base(N1,T).
base(0,T).

% sousTerme/2
sousTerme(T,T).
sousTerme(S,T) :- compound(T), functor(T,F,N), sousTerme(N,S,T).

% sousTerme/3
sousTerme(N,S,T) :- N > 1, N1 is N-1, sousTerme(N1,S,T).
sousTerme(N,S,T) :- arg(N,T,Arg), sousTerme(S,Arg).

% substituer/4
substituer(A,N,A,N).
substituer(A,N,T,T) :- atomic(T), A \== T.
substituer(A,N,AT,NT) :-
        compound(AT), functor(AT,F,Nb), functor(NT,F,Nb),
        substituer(Nb,A,N,AT,NT).

% substituer/5
substituer(Nb,A,N,AT,NT) :-
        Nb > 0, arg(Nb,AT,Arg),
        substituer(A,N,Arg,Arg1),
        arg(Nb,NT,Arg1), Nb1 is Nb-1,
        substituer(Nb1,A,N,AT,NT).
substituer(0,A,T,AT,NT).
```

TD 2.7 : Opérateurs

△ 18

1. Arbres syntaxiques

```
?- display(2 * a + b * c).
+(*(2, a), *(b, c))
true.
?- display(2 * (a + b) * c).
*(*(2, +(a, b)), c)
true.
?- display(2 * (a + b * c)).
*(2, +(a, *(b, c)))
true.
```

```
?- display(((2 * a) + b) * c).
*(+(*(2, a), b), c)
true.
?- display(2 + a + b + c).
+(+(+(2, a), b), c)
true.
```

2. Opérateurs « linguistiques »

```
?- display(diane est la secretaire de la mairie de brest).
ERROR: Syntax error: Operator expected
ERROR: diane
ERROR: ** here **
ERROR: est la secretaire de la mairie de brest .
```

```
?- op(300,xfx,est), op(200,yfx,de), op(100,fx,la).
true.
?- display(diane est la secretaire de la mairie de brest).
est(diane, de(de(la(secretaire), la(mairie)), brest))
true.
?- display(pierre est le maire de brest).
ERROR: Syntax error: Operator expected
ERROR: display(pierre est le
ERROR: ** here **
ERROR: maire de brest) .

?- op(100,fx,le).
true.
?- display(pierre est le maire de brest).
est(pierre, de(le(maire), brest))
true.

?- display(jean est le gardien de but des verts).
ERROR: Syntax error: Operator expected
ERROR: display(jean est le gardien de but
ERROR: ** here **
ERROR: des verts) .

?- op(200,yfx,des).
true.
?- display(jean est le gardien de but des verts).
est(jean, des(de(le(gardien), but), verts))
true.
```

3. **Opérateurs « de règles »**

```
?- op(800,fx,regle).
?- op(700,xfx,avant), op(700,xfx,arriere).
?- op(600,fx,si),  op(500,xfx,alors).
?- op(400,xfy,ou), op(300,xfy,et).

?- display(regle r1 avant si a et b ou c alors d et e).
regle(avant(r1, si(alors(ou(et(a, b), c), et(d, e)))))
true.
?- display(regle r2 arriere si f ou g alors h).
regle(arriere(r2, si(alors(ou(f, g), h))))
true.
```

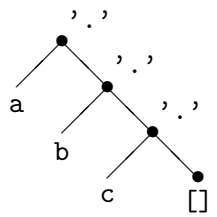
Corrigés TD 3

Listes

TD 3.1 : Représentations de listes

△ 19

1. **Représentation arborescente**



2. **Différentes écritures**

'.'(a,'.'(b,'.'(c,[])))
≡ [a|[b|[c|[]]]]
≡ [a|[b|[c]]]
≡ [a|[b,c|[]]]
≡ [a|[b,c]]
≡ [a,b|[c|[]]]
≡ [a,b|[c]]
≡ [a,b,c|[]]
≡ [a,b,c]

TD 3.2 : Informations sur les listes

△ 19

Listing 3.1 – listes (1)

```
1  % liste/1
2  liste([]).
3  liste([_|Q]) :- liste(Q).
4
5  % premier/2
6  premier(X,[X|_]).
7
8  % dernier/2
9  dernier(X,[X]).
10 dernier(X,[_|Q]) :- dernier(X,Q).
11
12 % nieme/3
13 nieme(1,T,[T|_]).
```

```
14 nieme(N,X,[_|Q]) :- nieme(M,X,Q), N is M + 1.
15
16 % longueur/2
17 longueur(0,[]).
18 longueur(N,[_|Q]) :- longueur(M,Q) , N is M + 1.
19
20 % dans/2
21 dans(X,[X|_]).
22 dans(X,[_|Q]) :- dans(X,Q).
23
24 % hors/2
25 hors(X,[]) :- nonvar(X).
26 hors(X,[T|Q]) :- X \== T , hors(X,Q).
27
28 % suivants/3
29 suivants(X,Y,[X,Y|_]).
30 suivants(X,Y,[_|Q]) :- suivants(X,Y,Q).
31
32 % unique/2
33 unique(X,[X|Q]) :- hors(X,Q).
34 unique(X,[T|Q]) :- unique(X,Q) , X \== T.
35
36 % occurence/3
37 occurences(X,0,[]) :- nonvar(X).
38 occurences(X,N,[X|Q]) :- occurences(X,M,Q) , N is M + 1.
39 occurences(X,N,[T|Q]) :- occurences(X,N,Q) , X \== T.
40
41 % prefixe/2
42 prefixe([],_).
43 prefixe([T|P],[T|Q]) :- prefixe(P,Q).
44
45 % suffixe/2
46 suffixe(L,L).
47 suffixe(S,[_|Q]) :- suffixe(S,Q).
48
49 % sousListe/2
50 sousListe([],_).
51 sousListe([T|Q],L) :- prefixe(P,L), suffixe([T|Q],P).
52 % sousListe([T|Q],L) :- suffixe(S,L), prefixe([T|Q],S).
```

Les implémentations de Prolog prédéfinissent certains des prédicats de'information sur les listes. Par exemple, dans SWI-PROLOG :

`is_list(+Term)`	: True if `Term` is bound to the empty list (`[]`) or a term with functor '`.`' and arity 2 and the second argument is a list.
`last(?List,?Elem)`	: Succeeds if `Elem` unifies with the last element of `List`.
`length(?List,?Int)`	: True if `Int` represents the number of elements of list `List`.
`member(?Elem,?List)`	: Succeeds when `Elem` can be unified with one of the members of `List`.
`nextto(?X,?Y,?List)`	: Succeeds when `Y` immediatly follows `X` in List.
`nth1(?Index,?List,?Elem)`	: Succeeds when the `Index`-th element of `List` unifies with `Elem`. Counting starts at 1.
`prefix(?Prefix,?List)`	: True iff `Prefix` is a leading substring of `List`.

TD 3.3 : MANIPULATION DE LISTES

△ 21

Listing 3.2 – listes (2)

```
% conc/3
conc([],L,L).
conc([T|Q],L,[T|QL]) :- conc(Q,L,QL).

% inserer/3
inserer(X,L,[X|L]).

% ajouter/3
ajouter(X,[],[X]).
ajouter(X,[T|Q],[T|L]) :- ajouter(X,Q,L).

% supprimer/3
supprimer(_,[],[]).
supprimer(X,[X|L1],L2) :- supprimer(X,L1,L2).
supprimer(X,[T|L1],[T|L2]) :- supprimer(X,L1,L2), X \= T.

% inverser/2
inverser([],[]).
inverser([T|Q],L) :- inverser(Q,L1), conc(L1,[T],L).

% substituer/3
substituer(_,_,[],[]).
substituer(X,Y,[X|L1],[Y|L2]) :- substituer(X,Y,L1,L2).
substituer(X,Y,[T|L1],[T|L2]) :- substituer(X,Y,L1,L2), X \= T.

% decaler/2
decaler(L,LD) :- conc(L1,[D],L), conc([D],L1,LD).

% convertir/2
:- op(100,xfy,et).

convertir([],fin).
convertir([T|Q],T et L) :- convertir(Q,L).

% aplatir/2
aplatir([],[]).
aplatir(T,[T]) :- T \= [], T \= [_|_].
aplatir([T|Q],L) :-
        aplatir(T,TA), aplatir(Q,QA), conc(TA,QA,L).

% transposer/2
transposer([T|Q],L) :- longueur(N,T), transposer(N,[T|Q],L).

% transposer/3
transposer(0,_,[]).
transposer(N,[T|Q],[Ts|Qs]) :-
        N > 0, transposerPremiers([T|Q],Ts,L1), N1 is N - 1,
        transposer(N1,L1,Qs).

% transposerPremiers/3
transposerPremiers([],[],[]).
transposerPremiers([[T|Q]|R],[T|TQ],[Q|QQ]) :-
        transposerPremiers(R,TQ,QQ).

% creerListe/3
creerListe(X,1,[X]) :- !.
creerListe(X,N,[X|L]) :- N > 1, N1 is N-1, creerListe(X,N1,L).

```

```
59 % univ/2
60 univ(F,[T|Q]) :-
61         nonvar(F), functor(F,T,N), univ(F,1,N,Q).
62 univ(F,[T|Q]) :-
63         var(F), longueur(N,Q), functor(F,T,N), univ(F,1,N,Q).
64
65 % univ/4
66 univ(F,Accu,N,[Arg|Q]) :-
67         N > 0, Accu < N, arg(Accu,F,Arg),
68         Accu1 is Accu + 1, univ(F,Accu1,N,Q).
69 univ(F,N,N,[Arg]) :- arg(N,F,Arg).
70 univ(_,_,0,[]).
```

Les implémentations de Prolog prédéfinissent certains des prédicats de manipulation de listes. Par exemple, dans SWI-PROLOG :

`?Term =.. ?List`	: `List` is a list which head is the functor of `Term` and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both.
`append(?List1,?List2,?List3)`	: Succeeds when `List3` unifies with the concatenation of `List1` and `List2`.
`append(?ListOfLists,?List)`	: Concatenate a list of lists.
`delete(+List1,?Elem,?List2)`	: Delete all members of `List1` that simultaneously unify with `Elem` and unify the result with `List2`.
`flatten(+List1,-List2)`	: Transform `List1`, possibly holding lists as elements into a 'flat' list by replacing each list with its elements (recursively). Unify the resulting flat list with `List2`.
`permutation(?List1,?List2)`	: Permuation is true when `List1` is a permutation of `List2`.
`reverse(+List1,-List2)`	: Reverse the order of the elements in `List1` and unify the result with the elements of `List2`.
`select(?Elem,?List,?Rest)`	: Select `Elem` from `List` leaving `Rest`.

TD 3.4 : TRIS DE LISTES

△ 25

Listing 3.3 – tris de listes

```
1  % triPermutation/2
2  triPermutation(L,LT) :-
3          permutation(L,LT),
4          enOrdre(LT).
5
6  % enOrdre/1
7  enOrdre([]).
8  enOrdre([_]).
9  enOrdre([T1,T2|Q]) :- T1 @=< T2, enOrdre([T2|Q]).
10
11 % triSelection/2
12 triSelection([],[]).
13 triSelection(L,[Min|LT]) :-
14         minimum(Min,L),
15         selection(Min,L,L1),
16         triSelection(L1,LT).
```

```
17
18 % minimum/2
19 minimum(T,[T]).
20 minimum(T1,[T1,T2|Q]) :- minimum(M2,[T2|Q]), T1 @=< M2.
21 minimum(M2,[T1,T2|Q]) :- minimum(M2,[T2|Q]), T1 @> M2.
22
23 % triBulles/2
24 triBulles(L,LT) :- bulles(L,L1), triBulles(L1,LT).
25 triBulles(L,L) :- \+ bulles(L,_).
26
27 % bulles/2
28 bulles([X,Y|Q],[Y,X|Q]) :- X @> Y.
29 bulles([X,Y|Q],[X|L]) :- X @=< Y, bulles([Y|Q],L).
30
31 % triDicho/2
32 triDicho([],[]).
33 triDicho([T|Q],LT) :- triDicho(Q,LQ), insertionDicho(T,LQ,LT).
34
35 % insertionDicho/3
36 insertionDicho(X,[],[X]).
37 insertionDicho(X,L,LX) :-
38         dicho(L,L1,[X|Q]), conc(L1,[X,X|Q],LX).
39 insertionDicho(X,L,LX) :-
40         dicho(L,L1,[T|Q]), X @> T,
41         insertionDicho(X,Q,LQ), conc(L1,[T|LQ],LX).
42 insertionDicho(X,L,LX) :-
43         dicho(L,L1,[T|Q]), X @< T,
44         insertionDicho(X,L1,L2), conc(L2,[T|Q],LX).
45
46 % dicho/3
47 dicho(L,L1,L2) :-
48         longueur(N,L),
49         R is N//2, % division entière
50         conc(L1,L2,L), longueur(R,L1).
51
52 % triFusion/2
53 triFusion([],[]).
54 triFusion([X],[X]).
55 triFusion([X,Y|Q],LT) :-
56         separation([X,Y|Q],L1,L2),
57         triFusion(L1,LT1),
58         triFusion(L2,LT2),
59         fusion(LT1,LT2,LT).
60
61 % separation/3
62 separation([],[],[]).
63 separation([X],[X],[]).
64 separation([X,Y|Q],[X|Q1],[Y|Q2]) :- separation(Q,Q1,Q2).
65
66 % fusion/3
67 fusion(L,[],L).
68 fusion([],[T|Q],[T|Q]).
69 fusion([X|QX],[Y|QY],[X|Q]) :- X @=< Y, fusion(QX,[Y|QY],Q).
70 fusion([X|QX],[Y|QY],[Y|Q]) :- X @> Y, fusion([X|QX],QY,Q).
71
72 % triRapide/2
73 triRapide([],[]).
74 triRapide([T|Q],LT) :-
75         partition(Q,T,LInf,LSup),
```

```
76          triRapide(LInf,LTInf),
77          triRapide(LSup,LTSup),
78          conc(LTInf,[T|LTSup],LT).
79
80  % partition/4
81  partition([],_,[],[]).
82  partition([T|Q],Pivot,[T|LInf],LSup) :-
83          T @=< Pivot, partition(Q,Pivot,LInf,LSup).
84  partition([T|Q],Pivot,LInf,[T|LSup]) :-
85          T @> Pivot, partition(Q,Pivot,LInf,LSup).
```

Certaines implémentations de Prolog proposent un (ou des) prédicat(s) de tri de listes. Par exemple, dans SWI-Prolog :

`sort(+List,-Sorted)`	: True if `Sorted` can be unified with a list holding the elements of `List`, sorted to the standard order of terms (see section 4.6). Duplicates are removed. The implementation is in C, using natural merge sort.
`msort(+List,-Sorted)`	: Equivalent to sort/2, but does not remove duplicates.
`keysort(+List,-Sorted)`	: `List` is a proper list whose elements are `Key-Value`, that is, terms whose principal functor is (-)/2, whose first argument is the sorting key, and whose second argument is the satellite data to be carried along with the key. `keysort/2` sorts `List` like `msort/2`, but only compares the keys.
`predsort(+Pred,+List,-Sorted)`	: Sorts similar to `sort/2`, but determines the order of two terms by calling `Pred(-Delta, +E1, +E2)` . This call must unify `Delta` with one of `<`, `>` or `=`. If built-in predicate `compare/3` is used, the result is the same as `sort/2`.

TD 3.5 : Piles et files

△ 25

Listing 3.4 – piles et files

```
1   % empiler/3
2   empiler(X,L1,L2) :- inserer(X,L1,L2).
3
4   % depiler/3
5   depiler(T,[T|Q],Q).
6
7   % pileVide/1
8   pileVide([]).
9
10  % sommet/2
11  sommet(X,L) :- premier(X,L).
12
13  % enfiler/3
14  enfiler(X,L1,L2) :- inserer(X,L1,L2).
15
16  % defiler/3
17  defiler(X,L1,L2) :- conc(L2,[X],L1).
18
19  % fileVide/1
20  fileVide([]).
```

```
21
22 % tete/2
23 tete(X,L) :- dernier(X,L).
```

TD 3.6 : Ensembles

△ 27

Listing 3.5 – ensembles

```
1  % ensemble/1
2  ensemble([]).
3  ensemble([T|Q]) :- hors(T,Q) , ensemble(Q).
4
5  % listeEnsemble/2
6  listeEnsemble([],[]).
7  listeEnsemble([T|Q1],[T|Q2]) :-
8          supprimer(T,Q1,Q),
9          listeEnsemble(Q,Q2).
10
11 % intersection/3
12 intersection([],_,[]).
13 intersection([T|Q],L,[T|Q1]) :-
14         dans(T,L),
15         supprimer(T,L,QL),
16         intersection(Q,QL,Q1).
17 intersection([T|Q],L,Q1) :-
18         hors(T,L),
19         intersection(Q,L,Q1).
20
21 % union/3
22 union([],L,L).
23 union([T|Q],L,[T|Q1]) :-They need not be ordered.
24         supprimer(T,L,QL),
25         union(Q,QL,Q1).
26
27 % sousEnsemble/2
28 sousEnsemble([],_).
29 sousEnsemble([T|Q],E) :-
30         dans(T,E),
31         sousEnsemble(Q,E).
```

Certaines implémentations de Prolog proposent un (ou des) prédicat(s) sur les ensembles. Par exemple, dans SWI-Prolog :

`is_set(+Set)`	: Succeeds if `Set` is a list without duplicates.
`list_to_set(+List,-Set)`	: Unifies `Set` with a list holding the same elements as `List` in the same order. If list contains duplicates, only the first is retained.
`intersection(+Set1,+Set2,-Set3)`	: Succeeds if `Set3` unifies with the intersection of `Set1` and `Set2`. `Set1` and `Set2` are lists without duplicates.
`subtract(+Set,+Delete,-Result)`	: Delete all elements of set `Delete` from `Set` and unify the resulting set with `Result`.
`union(+Set1,+Set2,-Set3)`	: Succeeds if `Set3` unifies with the union of `Set1` and `Set2`. `Set1` and `Set2` are lists without duplicates.
`subset(+Subset,+Set)`	: Succeeds if all elements of `Subset` are elements of `Set` as well.

TD 3.7 : Puzzle logique

△ 27

Listing 3.6 – puzzle logique

```
1  % tableau/1
2  tableau([m(C1,N1,S1),m(C2,N2,S2),m(C3,N3,S3)]) :-
3          dans(C1,[blanc,bleu,vert]),
4          dans(C2,[blanc,bleu,vert]),
5          dans(C3,[blanc,bleu,vert]),
6          C1 \== C2, C1 \== C3, C2 \== C3,
7          dans(N1,[anglais,espagnol,francais]),
8          dans(N2,[anglais,espagnol,francais]),
9          dans(N3,[anglais,espagnol,francais]),
10         N1 \== N2, N1 \== N3, N2 \== N3,
11         dans(S1,[football,natation,tennis]),
12         dans(S2,[football,natation,tennis]),
13         dans(S3,[football,natation,tennis]),
14         S1 \== S2, S1 \== S3, S2 \== S3.
15
16 % puzzle/1
17 puzzle(Tableau) :-
18         tableau(Tableau),
19         dans(m(vert,N1,natation),Tableau),
20         precede(m(vert,N2,S2),m(N3,espagnol,S3),Tableau),
21         dans(m(blanc,anglais,S4),Tableau),
22         precede(m(blanc,N5,S5),m(C6,N6,football),Tableau),
23         premier(m(C7,N7,tennis),Tableau).
24
25 % precede/2
26 precede(T,T1,[T|Q]) :- dans(T1,Q).
27 precede(T1,T2,[_|Q]) :- precede(T1,T2,Q).
```

On retrouve bien les 5 indices proposés dans le prédicat `puzzle/1` des lignes 19 à 23 :

19 : Dans la maison verte on pratique la natation.
20 : La maison verte est située avant la maison de l'espagnol.
21 : L'anglais habite la maison blanche.
22 : La maison blanche est située avant la maison où on pratique le football.
23 : Le tennisman habite au début de la rue.

Chaque indice diminue le nombre de solutions possibles : $216 \xrightarrow{1} 72 \xrightarrow{2} 24 \xrightarrow{3} 6 \xrightarrow{4} 2 \xrightarrow{5} 1$.

TD 3.8 : Listes et arithmétique

△ 28

Listing 3.7 – listes et arithmétique

```
1  % maximum/2
2  maximum([X],X).
3  maximum([T1,T2|Q],Max) :- T1 > T2, maximum([T1|Q],Max).
4  maximum([T1,T2|Q],Max) :- T2 >= T1, maximum([T2|Q],Max).
5
6  % somme/2
7  somme([],0).
8  somme([T|Q],N) :- somme(Q,NQ), N is T + NQ.
9
10 % entiers/3
11 entiers(Min,Max,[]) :- Min > Max.
```

```
12 entiers(Min,Max,[Min|E]) :-
13         Min =< Max, Min1 is Min + 1, entiers(Min1,Max,E).
14
15 % produitScalaire/3
16 produitScalaire([],[],0).
17 produitScalaire([T1|Q1],[T2|Q2],P) :-
18         produitScalaire(Q1,Q2,PQ), P is PQ + T1*T2.
19
20 % surface/2
21 surface([_],0).
22 surface([(X1,Y1),(X2,Y2)|Pts],S) :-
23         surface([(X2,Y2)|Pts],S1),
24         S is (X1*Y2 - Y1*X2)/2 + S1.
25
26 % premiers/2
27 premiers(1,[1]).
28 premiers(N,[1|LP]) :- N > 1, entiers(2,N,E), crible(E,LP).
29
30 % crible/2
31 crible([],[]).
32 crible([E|Es],[E|Ps]) :-
33         filtre(E,Es,Es1), crible(Es1,Ps).
34
35 % filtre/3
36 filtre(_,[],[]).
37 filtre(P,[T|Q],[T|PQ]) :- 0 =\=  T mod P, filtre(P,Q,PQ).
38 filtre(P,[T|Q],PQ) :- 0 =:= T mod P, filtre(P,Q,PQ).
```

Certaines implémentations de Prolog proposent un (ou des) prédicat(s) sur les listes de nombres. Par exemple, dans SWI-Prolog :

`between(+Low,+High,?Value)`	: `Low` and `High` are integers, `High >=Low`. If `Value` is an integer, `Low =<Value =<High`. When `Value` is a variable it is successively bound to all integers between `Low` and `High`.
`sumlist(+List,-Sum)`	: Unify `Sum` to the result of adding all elements in `List`. `List` must be a proper list holding numbers.
`max_list(+List,-Max)`	: True if `Max` is the largest number in `List`.
`min_list(+List,-Min)`	: True if `Min` is the smallest number in `List`.
`numlist(+Low,+High,-List)`	: If `Low` and `High` are integers with `Low =< High`, unify `List` to a list `[Low, Low+1, ...High]`.

TD 3.9 : Le compte est bon

△ 29

Listing 3.8 – « le compte est bon »

```
1  % operation/4
2  operation(X,Y,+,Z) :- Z is X+Y.
3  operation(X,Y,-,Z) :- Z is X-Y.
4  operation(X,Y,*,Z) :- Z is X*Y.
5
6  % calcul/5
7  calcul(X,Y,Z,T,R,Expr) :-
8          operation(X,Y,Op1,XY),
9          operation(XY,Z,Op2,XYZ),
10         operation(XYZ,T,Op3,R),
```

```
11          E1 =.. [Op1|[X,Y]],
12          E2 =.. [Op2|[E1,Z]],
13          Expr =.. [Op3|[E2,T]].
14
15
16  calcul(X,Y,Z,T,R,Expr) :-
17          operation(X,Y,Op4,XY),
18          operation(Z,T,Op6,ZT),
19          operation(XY,ZT,Op5,R),
20          E1 =.. [Op4|[X,Y]],
21          E2 =.. [Op6|[Z,T]],
22          Expr =.. [Op5|[E1,E2]].
23
24  % compte/5
25  compte(X,Y,Z,T,R,Expr) :-
26          permutation([X,Y,Z,T],[X1,X2,X3,X4]),
27          calcul(X1,X2,X3,X4,R,Expr).
```

TD 3.10 : Casse-tête cryptarithmétique

△ 30

Listing 3.9 – casse-têtes cryptarithmétiques

```
1   % somme/3
2   somme(N1,N2,N3) :-
3           normalise(N1,N2,N3,NN1,NN2,NN3),
4           somme(NN1,NN2,NN3,0,0,[0,1,2,3,4,5,6,7,8,9],_).
5
6   % normalise/6
7   normalise(N1,N2,N3,NN1,NN2,NN3) :-
8           longueur(Lgr1,N1),
9           longueur(Lgr2,N2),
10          longueur(Lgr3,N3),
11          maximum([Lgr1,Lgr2,Lgr3],Lgr),
12          normalise(0,N1,Lgr1,NN1,Lgr),
13          normalise(0,N2,Lgr2,NN2,Lgr),
14          normalise(0,N3,Lgr3,NN3,Lgr).
15
16  % normalise/5
17  normalise(X,N1,Lgr1,NN1,Lgr) :-
18          Lgr > Lgr1,
19          D is Lgr - Lgr1,
20          creerListe(X,D,LX),
21          conc(LX,N1,NN1).
22  normalise(_,N1,Lgr1,N1,Lgr) :- Lgr =< Lgr1.
23
24  % somme/7
25  somme([],[],[],0,0,Chiffres,Chiffres).
26  somme([C1|N1],[C2|N2],[C3|N3],R1,R,Ch1,Ch) :-
27          somme(N1,N2,N3,R1,R2,Ch1,Ch2),
28          sommeChiffre(C1,C2,R2,C3,R,Ch2,Ch).
29
30  % sommeChiffre/7
31  sommeChiffre(C1,C2,R1,C3,R,Ch1,Ch) :-
32          supprimer(C1,Ch1,Ch2), supprimer(C2,Ch2,Ch3), supprimer(C3,Ch3,Ch),
33          S is C1 + C2 + R1, C3 is S mod 10,
34          R is S // 10. % division entière
```

TD 3.11 : Problème des n reines

△ 30

Listing 3.10 – problème des n reines

```
% nreines/2
nreines(N,L) :-
        entiers(1,N,NCases),
        diagonales(N,NDiag),
        antiDiagonales(N,NAntiDiag),
        nreines(NCases,NCases,NDiag,NAntiDiag,L).

% nreines/5
nreines([],_,_,_,[]).
nreines([X|Xs],Ny,Nd,Na,[Y|Ys]) :-
        supp(Y,Ny,Ny1),
        D is X+Y, supp(D,Nd,Nd1),
        A is X-Y, supp(A,Na,Na1),
        nreines(Xs,Ny1,Nd1,Na1,Ys).

% diagonales/2
diagonales(N,NDiag) :- N2 is N+N, entiers(2,N2,NDiag).

% antiDiagonales/2
antiDiagonales(N,NAntiDiag) :-
        N1 is 1-N, N2 is N-1, entiers(N1,N2,NAntiDiag).

% supp/3
supp(T,[T|Q],Q).
supp(X,[T|Q],[T|L]) :- supp(X,Q,L).
```

TD 3.12 : Les mutants

△ 31

Listing 3.11 – mutants

```
% mutant/3
mutant(A1,A2,Mutant) :-
        name(A1,L1), name(A2,L2),
        conc(_,[T|Q],L1), conc([T|Q],S2,L2), conc(L1,S2,L),
        name(Mutant,L).
```

Les implémentations de Prolog proposent un prédicat permettant de transformer un atome en une liste de ses codes ASCII ou réciproquement. Par exemple, dans SWI-Prolog :

`name(?AtomOrInt,?String)` : `String` is a list of character codes representing the same text as `Atom`. Each of the arguments may be a variable, but not both. When `String` is bound to an character code list describing an integer and `Atom` is a variable, `Atom` will be unified with the integer value described by `String` (e.g. `name(N, "300"), 400 is N + 100` succeeds).

TD 3.13 : Mots croisés

△ 31

Listing 3.12 – mots croisés

```
% nbCars/2
nbCars(Mot,N) :- longueur(N,Mot).

% selectMot/2
selectMot(Mot,N) :- dico(Mot), nbCars(Mot,N).

% rang/3
rang(Mot,N,Car) :- nieme(Car,N,Mot).

% interCar/4
interCar(Mot1,N1,Mot2,N2) :-
        rang(Mot1,N1,Car), rang(Mot2,N2,Car).

% motsCroises/7
motsCroises(M1,M2,M3,M4,M5,M6,M7) :-
        selectMot(M1,4), selectMot(M2,2), selectMot(M3,2),
        selectMot(M4,2), selectMot(M5,4), selectMot(M6,2),
        selectMot(M7,4),
        interCar(M1,1,M5,1), interCar(M1,4,M7,1),
        interCar(M4,2,M7,3), interCar(M2,1,M5,2),
        interCar(M6,1,M1,2), interCar(M6,2,M2,2),
        interCar(M3,1,M5,4).
```

Corrigés TD 4

Contrôle de la résolution

TD 4.1 : Dérivation symbolique

△ 33

Listing 4.1 – dérivation symbolique

```
% d/3
d(C,X,0) :- C \== X , atomic(C) , !.
d(X^(N),X,N*(X^(N-1))) :- N \== X , atomic(N), !.
d(U+V,X,U1+V1) :- d(U,X,U1) , d(V,X,V1) , !.
d(U-V,X,U1-V1) :- d(U,X,U1) , d(V,X,V1) , !.
d(U*V,X,V*U1+U*V1) :- d(U,X,U1) , d(V,X,V1) , !.
d(U/V,X,(U1*V - V1*U)/(V^2)) :- d(U,X,U1) , d(V,X,V1) , !.
d(sin(X),X,cos(X)) :- !.
d(cos(X),X,(-1)*sin(X)) :- !.
d(exp(X),X,exp(X)) :- !.
d(ln(X),X,1/X) :- !.
d(F,X,DU*DFU) :- d(F,U,DFU), U \== X, d(U,X,DU).
```

TD 4.2 : Traitements génériques

△ 33

Listing 4.2 – traitements génériques

```
% map/3
map(_,[],[]).
map(F,[T|Q],[FT|FQ]) :-
        But =.. [F,T,FT], call(But), map(F,Q,FQ).
```

TD 4.3 : Du mot aux syllabes

△ 33

Listing 4.3 – césure de mots

```
% voyelle/1
voyelle(a). voyelle(e). voyelle(i). voyelle(o).
voyelle(u). voyelle(y).

% consonne/1
consonne(b). consonne(c). consonne(d). consonne(f).
consonne(g). consonne(h). consonne(j). consonne(k).
consonne(l). consonne(m). consonne(n). consonne(p).
consonne(q). consonne(r). consonne(s). consonne(t).
consonne(v). consonne(w). consonne(x). consonne(z).
```

```
11
12 % inseparables/1
13 inseparables([b,l]). inseparables([c,l]). inseparables([f,l]).
14 inseparables([g,l]). inseparables([p,l]). inseparables([b,r]).
15 inseparables([c,r]). inseparables([d,r]). inseparables([f,r]).
16 inseparables([g,r]). inseparables([p,r]). inseparables([t,r]).
17 inseparables([v,r]). inseparables([c,h]). inseparables([p,h]).
18 inseparables([g,n]). inseparables([t,h]).
19
20 % codes/2
21 codes([],[]).
22 codes([C|Cs],[L|Ls]) :- name(L,[C]), codes(Cs,Ls).
23
24 % decomposer/2
25 decomposer(Mot,Lettres) :- name(Mot,Codes), codes(Codes,Lettres).
26
27 % recomposer/2
28 recomposer(Mot,Lettres) :- codes(Codes,Lettres), name(Mot,Codes).
29
30 % cesures/3
31 cesures(Mot,Separateur,Syllabes) :-
32         decomposer(Mot,Lettres),
33         syllabes(Lettres,Separateur,Syllabes1),
34         recomposer(Syllabes,Syllabes1).
35
36 % syllabes/3
37 syllabes([V1,C,V2|L],Sep,[V1,Sep,C,V2|S]) :-
38         voyelle(V1), consonne(C), voyelle(V2),
39         !,
40         syllabes([V2|L],Sep,[V2|S]).
41 syllabes([V1,C1,C2,V2|L],Sep,[V1,C1,Sep,C2,V2|S]) :-
42         voyelle(V1), consonne(C1), consonne(C2), voyelle(V2),
43         \+ inseparables([C1,C2]),
44         !,
45         syllabes([V2|L],Sep,[V2|S]).
46 syllabes([V1,C1,C2,V2|L],Sep,[V1,Sep,C1,C2,V2|S]) :-
47         voyelle(V1), consonne(C1), consonne(C2), voyelle(V2),
48         inseparables([C1,C2]),
49         !,
50         syllabes([V2|L],Sep,[V2|S]).
51 syllabes([C1,C2,C3|L],Sep,[C1,C2,Sep,C3|S]) :-
52         consonne(C1), consonne(C2), consonne(C3),
53         \+ inseparables([C2,C3]),
54         !,
55         syllabes([C3|L],Sep,[C3|S]).
56 syllabes([C1,C2,C3|L],Sep,[C1,Sep,C2,C3|S]) :-
57         consonne(C1), consonne(C2), consonne(C3),
58         inseparables([C2,C3]),
59         !,
60         syllabes([C3|L],Sep,[C3|S]).
61 syllabes([L|Ls],Sep,[L|S]) :- syllabes(Ls,Sep,S).
62 syllabes([],_,[]).
```

TD 4.4 : Gestion d'un compteur

△ 34

Listing 4.4 – gestion d'un compteur

```
1 % cptInit/2
```

```
2  cptInit(Cpt,Val) :-
3          killCpt(Cpt), integer(Val), recorda(Cpt,Val,_).
4
5  % killCpt/1
6  killCpt(Cpt) :- recorded(Cpt,_,Ref), erase(Ref), fail.
7  killCpt(_).
8
9  % cptFin/2
10 cptFin(Cpt) :- killCpt(Cpt).
11
12 % cptVal/2
13 cptVal(Cpt,Val) :- recorded(Cpt,Val,_).
14
15 % cptInc/2
16 cptInc(Cpt,Inc) :-
17         integer(Inc), recorded(Cpt,Val,Ref), erase(Ref),
18         V is Val + Inc, recorda(Cpt,V,_).
19
20 % alphabet/0
21 alphabet :-
22         cptInit(c,97),
23         repeat,
24                 cptVal(c,V), put(V), put(32), cptInc(c,1),
25         V =:= 122, !,
26         cptFin(c).
```

TD 4.5 : Structures de contrôle

△ 35

1. **Représentation arborescente**

```
?- display(si p alors q).
si(alors(p, q))
true.
?- display(si p alors q sinon r).
si(alors(p, sinon(q, r)))
true.
?- display(selon x dans [a1:r1,a2:r2]).
selon(dans(x, .(:(a1, r1), .(:(a2, r2), []))))
true.
?- display(selon x dans [a1:r1,a2:r2] autrement r).
selon(dans(x, autrement(.(:(a1, r1), .(:(a2, r2), [])), r)))
true.
?- display(repeter r jusqua p).
repeter(jusqua(r, p))
true.
?- display(pour i := 0 to 5 faire r).
pour(to(:=(i, 0), faire(5, r)))
true.
?- display(pour i := 5 downto 0 faire r).
pour(downto(:=(i, 5), faire(0, r)))
true.
```

2. Structures de base

Listing 4.5 – structures de contrôle

```
:- op(900,fx ,si), op(850,xfx,alors), op(800,xfx,sinon).
:- op(900,fx ,repeter), op(850,xfx,jusqua).
:- op(900,fx ,pour), op(800,xfx,faire).
:- op(850,xfx,to), op(850,xfx,downto).
:- op(900,fx ,selon), op(850,xfx,dans), op(800,xfx,autrement).
:- op(750,xfx,:=), op(750,xfx,:).:- op(900,fx ,si).

% si/1
si P alors Q sinon R :- call(P), !, call(Q).
si P alors Q sinon R :- call(R), !.
si P alors Q :- call(P), !, call(Q).
si P alors Q.

% repeter/1
repeter Q jusqua P :-
        repeat,
                call(Q),
        call(P), !.

% selon/1
selon X dans [Xi : Qi autrement R] :- !,
        si X = Xi alors call(Qi) sinon call(R).
selon X dans [Xi : Qi] :- !,
        si X = Xi alors call(Qi).
selon X dans [Xi : Qi | XQ] :-
        si X = Xi alors call(Qi) sinon (selon X dans XQ).

% pour/1
pour X := Min to Max faire Q :- Min > Max, !.
pour X := Min to Max faire Q :-
        cptInit(X,Min),
        repeter
                ( cptVal(X,V),call(Q),cptInc(X,1) )
        jusqua(V =:= Max),
        cptFin(X).
pour X := Max downto Min faire Q :- Min > Max,!.
pour X := Max downto Min faire Q :-
        cptInit(X,Max),
        repeter
                ( cptVal(X,V),call(Q),cptDec(X,1) )
        jusqua(V =:= Min),
        cptFin(X).
```

3. Gestion d'un menu

Listing 4.6 – gestion d'un menu

```
% menu/1
menu(Menu) :-
        repeter (
                afficherMenu(Menu),
                choixMenu(Menu,Choix),
                itemMenu(Menu,Choix)
                )
        jusqua (finMenu(Menu,Choix,Fin)).

```

```
10 % choixMenu/2
11 choixMenu(Menu,Choix) :-
12         write(Menu), write(' > votre choix : '), read(Choix).
13
14 % finMenu/3
15 finMenu(Menu,Fin,Fin) :- nbItems(Menu,Fin).
```

Listing 4.7 – exemples de menus

```
1  % nbItems/2
2  nbItems(sdc,8).
3  nbItems(test,4).
4
5  % finmenu/3
6  finMenu(Menu,Fin,Fin) :- nbItems(Menu,Fin).
7
8  % itemMenu/2
9  itemMenu(sdc,Choix) :-
10         selon Choix dans
11                 [
12                 1 : exSdc(si P alors Q),
13                 2 : exSdc(si P alors Q sinon R),
14                 3 : exSdc(selon X dans [Xi : Qi]),
15                 4 : exSdc(selon X dans [Xi : Qi autrement R]),
16                 5 : exSdc(pour X := Min to Max faire Q),
17                 6 : exSdc(pour X := Max downto Min faire Q),
18                 7 : exSdc(repeter Q jusqua P),
19                 8 : quitter(sdc)
20                 autrement (write('*** choix incorrect ***'),nl)
21                 ].
22 itemMenu(test,Choix) :-
23         selon Choix dans
24                 [
25                 1 : write(un),
26                 2 : write(deux),
27                 3 : write(trois),
28                 4 : quitter(test)
29                 autrement (write('*** choix incorrect ***'),nl)
30                 ].
31
32 % choixMenu/2
33 choixMenu(Menu,Choix) :-
34         write(Menu), write(' > votre choix : '), read(Choix).
35
36 % quitter/1
37 quitter(sdc) :- nl,write('... fin de la demonstration'),nl.
38 quitter(test) :- nl,write('... fin de l''exemple'),nl.
39
40
41 % afficherMenu/1
42 afficherMenu(sdc) :-
43         nl,write('STRUCTURES DE CONTROLE'),nl,
44         tab(2),write('1. si ... alors ...'),nl,
45         tab(2),write('2. si ... alors ... sinon ...'),nl,
46         tab(2),write('3. selon ... dans [ ... ]'),nl,
47         tab(2),write('4. selon ... dans [ ... autrement ... ]'),nl,
48         tab(2),write('5. pour ... := ... to ... faire ...'),nl,
49         tab(2),write('6. pour ... := ... downto ... faire ...'),nl,
50         tab(2),write('7. repeter ... jusqua ...'),nl,
```

```
51          tab(2),write('8. quitter'),nl.
52  afficherMenu(test) :-
53          nl,write('EXEMPLE DE MENU'),nl,
54          tab(2),write('1. un'),nl,
55          tab(2),write('2. deux'),nl,
56          tab(2),write('3. trois'),nl,
57          tab(2),write('4. quitter'),nl.
58
59  % afficherClause/1
60  afficherClause(Tete) :-
61          clause(Tete,Corps),
62          tab(2),write(Tete),write(' :-'),nl,
63          afficherCorps(Corps),
64          fail.
65  afficherClause(_) :- nl.
66
67  % afficherCorps/1
68  afficherCorps((C1;C2)) :- !,
69          tab(8),write(C1),nl,
70          tab(8),write(';'),
71          afficherCorps(C2).
72  afficherCorps((C1,C2)) :- !,
73          tab(8),write(C1),write(','),nl,
74          afficherCorps(C2).
75  afficherCorps(C) :-
76          tab(8),write(C),write('.'),nl.
77
78  % exSdc/1
79  exSdc(TeteClause) :-
80          nl,write('REGLES DU '),write(TeteClause),write(' : '),
81          nl,nl,
82          fail.
83  exSdc(TeteClause) :-
84          afficherClause(TeteClause),
85          fail.
86  exSdc(TeteClause) :-
87          nl,write('EXEMPLE D''APPLICATION : '),fail.
88
89  exSdc(si P alors Q) :- !,
90          write('test si un nombre est negatif'),nl,
91          appel(test(si)).
92
93  exSdc(si P alors Q sinon R) :- !,
94          write('calcul de la valeur absolue d''un nombre'),nl,
95          appel(test(sinon)).
96
97  exSdc(selon X dans [Xi : Qi]) :- !,
98          write('teste une valeur entiere comprise entre
99                 0 et 4 exclus'),
100         nl,
101         appel(test(selon)).
102
103 exSdc(selon X dans [Xi : Qi autrement R]) :- !,
104         write('gestion d''un menu'),nl,
105         appel(test(autrement)).
106
107 exSdc(pour X := Min to Max faire Q) :-
108         write('calcul iteratif de n!'),nl,
109         appel(test(to)).
```

```
110
111 exSdc(pour X := Max downto Min faire Q) :-
112         write('calcul iteratif de la somme des n
113                premiers entiers'),nl,
114         appel(test(downto)).
115
116 exSdc(repeter Q jusqua P) :-
117         write('test d''un nombre entre au clavier'),nl,
118         appel(test(repeter)).
119
120 exSdc(_) :-
121         write('pas d''exemple,desole ...'),nl.
122
123 % appel/1
124 appel(But) :-
125         afficherClause(But),
126         write('| ?- '),write(But),write('.'),nl,
127         si (call(But),nl) alors write('true')
128         sinon write('false'),
129         nl.
130
131 % test/1
132 test(si) :-
133         entrer(X),
134         si X < 0 alors (write('nombre negatif'),nl).
135 test(sinon) :-
136         entrer(X),
137         write('valeur absolue : '),
138         si X >= 0 alors write(X) sinon (Y is - X,write(Y)),
139         nl.
140 test(selon) :-
141         repeter entrer(X) jusqua(X > 0 , X < 4),
142         selon X dans [
143                         1 : write(un),
144                         2 : write(deux),
145                         3 : write(trois)
146                       ],
147         nl.
148 test(autrement) :-
149         repeter (
150                 afficherMenu(test),
151                 choixMenu(test,Choix),
152                 itemMenu(test,Choix)
153                 )
154         jusqua(finMenu(Menu,Choix,4)),!,
155         si Choix == 4 alors fin sinon test(autrement).
156 test(to) :-
157         repeter entrer(X) jusqua(X >= 0),
158         recorda(fact,1,R0),
159         pour i := 1 to X faire (
160                                 recorded(fact,Val,R1),
161                                 cptVal(i,I),
162                                 Fact is Val * I,
163                                 erase(R1),
164                                 recorda(fact,Fact,R2)
165                                ),
166         recorded(fact,Fact,R3),
167         write(X),write('! = '),write(Fact),nl,
168         killCpt(fact).
```

```
169 test(downto) :-
170         repeter entrer(X) jusqua(X >= 0),
171         recorda(somme,0,R0),
172         pour i := X downto 1 faire (
173                                     recorded(somme,Val,R1),
174                                     cptVal(i,I),
175                                     Somme is Val + I,
176                                     erase(R1),
177                                     recorda(somme,Somme,R2)
178                                    ),
179         recorded(somme,Somme,R3),
180         write('la somme des '),write(X),
181         write(' premiers entiers = '),write(Somme),nl,
182         killCpt(somme).
183 test(repeter) :-
184         repeter entrer(X) jusqua(X =:= 0).
185
186 % entrer/1
187 entrer(X) :-
188         repeter
189         (
190         write('entrer un nombre entier (puis .) : '),
191         read(X)
192         )
193         jusqua integer(X).
```

TD 4.6 : UNIFICATION

△ 36

Listing 4.8 – unification

```
1  % unifier/2
2  unifier(X,Y) :- var(X), var(Y), X = Y.
3  unifier(X,Y) :- var(X), nonvar(Y), X = Y.
4  unifier(X,Y) :- nonvar(X), var(Y), Y = X.
5  unifier(X,Y) :- atomic(X), atomic(Y), X == Y.
6  unifier(X,Y) :-
7          compound(X), compound(Y),
8          functor(X,F,N), functor(Y,F,N),
9          unifierArguments(N,X,Y).
10
11 % unifierArguments/3
12 unifierArguments(N,X,Y) :-
13         N > 0, arg(N,X,ArgX), arg(N,Y,ArgY),
14         unifier(ArgX,ArgY),
15         N1 is N - 1, unifierArguments(N1,X,Y).
16 unifierArguments(0,_,_).
17
18 % unifier2/2
19 unifier2(X,Y) :- var(X), var(Y), X = Y.
20 unifier2(X,Y) :- var(X), nonvar(Y), nonOccur(X,Y), X = Y.
21 unifier2(X,Y) :- nonvar(X), var(Y), nonOccur(Y,X), Y = X.
22 unifier2(X,Y) :- atomic(X), atomic(Y), X == Y.
23 unifier2(X,Y) :-
24         compound(X), compound(Y),
25         functor(X,F,N), functor(Y,F,N),
26         unifierArguments2(N,X,Y).
27
28 % unifierArguments2/3
```

```
29 unifierArguments2(N,X,Y) :-
30         N > 0, arg(N,X,ArgX), arg(N,Y,ArgY),
31         unifier2(ArgX,ArgY),
32         N1 is N - 1, unifierArguments2(N1,X,Y).
33 unifierArguments2(0,_,_).
34
35 % varLiees/2
36 varLiees('$leurre$',Y) :- var(Y),!,fail.
37 varLiees(X,Y).
38
39 % nbVars/3
40 nbVars(v(N),N,N1) :- N1 is N + 1.
41 nbVars(Terme,N1,N2) :-
42         nonvar(Terme), functor(Terme,_,N),
43         nbVars(0,N,Terme,N1,N2).
44
45 % nbVars/5
46 nbVars(N,N,_,N1,N1).
47 nbVars(I,N,Terme,N1,N3) :-
48         I < N, I1 is I + 1, arg(I1,Terme,Arg),
49         nbVars(Arg,N1,N2), nbVars(I1,N,Terme,N2,N3).
```

TD 4.7 : Ensemble de solutions

△ 37

Listing 4.9 – ensemble de solutions

```
1  % s/1
2  s(1). s(2). s(3). s(4). s(5).
3
4  % s1/1
5  s1(X) :- s(X), !.
6
7  % s2/2
8  s2(X) :- sn(X,2).
9
10 % sall/1
11 sall(L) :- call(s(X)), assert(solution(X)), fail.
12 sall(L) :- assert(solution('$fin$')), fail.
13 sall(L) :- recuperer(L).
14
15 % sd/1
16 sd(X) :- sall(L), dernier(X,L).
17
18 % sn/2
19 sn(X,N) :- sall(L), nieme(N,X,L).
20
21 % toutes/3
22 toutes(X,But,L) :-
23         call(But),
24         assert(solution(X)),
25         fail.
26 toutes(_,_,_) :- assert(solution('$fin$')), fail.
27 toutes(X,But,L) :- recuperer(L).
28
29 % recuperer/1
30 recuperer([T|Q]) :-
31         retract(solution(T)), T \== '$fin$',
32         !,
```

```
33          recuperer(Q).
34  recuperer([]).
```

TD 4.8 : Chaînage avant

△ 38

Listing 4.10 – chaînage avant

```
1   % deduire/0
2   deduire :-
3           regle(B,C),
4           tester(C),
5           affirmer(B),!,
6           deduire.
7   deduire.
8
9   % tester/1
10  tester((C1,C2)) :- !, tester(C1), tester(C2).
11  tester((C1;C2)) :- !, (tester(C1) ; tester(C2)).
12  tester(C) :- regle(C,vrai) ; regle(C,affirme).
13
14  % affirmer/1
15  affirmer(B) :-
16          \+ tester(B),!,
17          assert(regle(B,affirme)),
18          write(B),write(' affirmé'),nl.
```

TD 4.9 : Mise sous forme clausale

△ 38

Listing 4.11 – mise sous forme clausale

```
1   :- op(100,fx,non), op(150,xfy,et), op(200,xfy,ou).
2   :- op(250,xfy,imp), op(300,xfy,eq).
3
4   % formule/2
5   formule(1,tout(x,p(x))).
6   formule(2,tout(x,imp(p(x),q(x)))).
7   formule(3,tout(x,existe(y,imp(p(x),q(x,f(y)))))).
8   formule(4,tout(x,tout(y, imp(p(x), ou(imp(q(x,y),non(existe(u,p(f(u))))),
9             imp(q(x,y),non(r(y)))))))).
10  formule(5,tout(x,imp(p(x),et(non(tout(y,imp(q(x,y),existe(u,p(f(u)))))),
11            tout(y,imp(q(x,y),p(x))))))).
12
13  % toClause/2 : mise sous forme clausale
14  toClause(In, Out) :-
15          implOut(In, In1),       % élimination des implications
16          negIn(In1, In2),        % déplacement des négations
17          skolem(In2, In3, []),   % skolémisation
18          toutOut(In3, In4),      % déplacement des 'quel que soit'
19          distrib(In4, In5),      % distribution de 'et' sur 'ou'
20          toClause(In5, Out, []).% mise en clauses
21
22  % implOut/2 : élimination des implications
23  implOut(P eq Q, (P1 et Q1) ou (non P1 et non Q1)) :- !,
24          implOut(P,P1), implOut(Q,Q1).
25  implOut(P imp Q, non P1 ou Q1) :- !, implOut(P,P1), implOut(Q,Q1).
26  implOut(tout(X,P), tout(X,P1)) :- !, implOut(P,P1).
```

```
27 implOut(existe(X,P), existe(X,P1)) :- !, implOut(P,P1).
28 implOut(P et Q, P1 et Q1) :- !, implOut(P,P1), implOut(Q,Q1).
29 implOut(P ou Q, P1 ou Q1) :- !, implOut(P,P1), implOut(Q,Q1).
30 implOut(non P, non P1) :- !, implOut(P,P1).
31 implOut(P,P).
32
33 % negIn/2 : déplacement des négations
34 negIn(non P, P1) :- !, negation(P, P1).
35 negIn(tout(X,P), tout(X,P1)) :- !, negIn(P, P1).
36 negIn(existe(X,P), existe(X,P1)) :- !, negIn(P, P1).
37 negIn(P et Q, P1 et Q1) :- !, negIn(P, P1), negIn(Q, Q1).
38 negIn(P ou Q, P1 ou Q1) :- !, negIn(P, P1), negIn(Q, Q1).
39 negIn(P, P).
40
41 % negation/2
42 negation(non P, P1) :- !, negIn(P, P1).
43 negation(tout(X,P), existe(X,P1)) :- !, negation(P, P1).
44 negation(existe(X,P), tout(X,P1)) :- !, negation(P, P1).
45 negation(P et Q, P1 ou Q1) :- !, negation(P, P1), negation(Q, Q1).
46 negation(P ou Q, P1 et Q1) :- !, negation(P, P1), negation(Q, Q1).
47 negation(P, non P).
48
49 % skolem/3 : skolémisation
50 skolem(tout(X,P), tout(X,P1), Vars) :- !, skolem(P, P1, [X|Vars]).
51 skolem(existe(X,P), P2, Vars) :- !,
52         genSymbol('$f', Sb), Skolem =.. [Sb|Vars],
53         substitute(X, Skolem, P, P1), skolem(P1,P2,Vars).
54 skolem(P et Q, P1 et Q1, Vars) :- !, skolem(P, P1, Vars), skolem(Q, Q1, Vars).
55 skolem(P ou Q, P1 ou Q1, Vars) :- !, skolem(P, P1, Vars), skolem(Q, Q1, Vars).
56 skolem(P, P, _).
57
58 % genSymbol/2
59 genSymbol(Racine, Symbol) :-
60         nombre(Racine, Nombre), name(Racine, L1),
61         intStr(Nombre, L2), conc(L1, L2, L),
62         name(Symbol, L).
63
64 % intStr/2
65 intStr(N, L) :- intStr(N, [], L).
66
67 % intStr/3
68 intStr(N, Accu, [C|Accu]) :- N < 10, !, C is N + 48.
69 intStr(N, Accu, L) :-
70         Quotient is N/10, Reste is N mod 10, C is Reste + 48,
71         intStr(Quotient, [C|Accu], L).
72
73 % nombre/2
74 nombre(Racine, N) :-
75         retract('$nombre'(Racine, N1)), !,
76         N is N1 + 1, asserta('$nombre'(Racine, N)).
77 nombre(Racine, 1) :- asserta('$nombre'(Racine, 1)).
78
79 % toutOut/2 : déplacement des 'quel que soit'
80 toutOut(tout(X,P), P1) :- !, toutOut(P, P1).
81 toutOut(P et Q, P1 et Q1) :- !, toutOut(P, P1), toutOut(Q, Q1).
82 toutOut(P ou Q, P1 ou Q1) :- !, toutOut(P, P1), toutOut(Q, Q1).
83 toutOut(P, P).
84
85 % distrib/2 : distribution de 'et' sur 'ou'
```

```
86  distrib(P ou Q, R) :- !, distrib(P, P1), distrib(Q, Q1), distrib1(P1 ou Q1, R).
87  distrib(P et Q, P1 et Q1) :- !, distrib(P, P1), distrib(Q, Q1).
88  distrib(P, P).
89
90  % distrib1/2
91  distrib1((P et Q) ou R, P1 et Q1) :- !,
92          distrib(P ou R, P1), distrib(Q ou R, Q1).
93  distrib1(P ou (Q et R), P1 et Q1) :- !,
94          distrib(P ou Q, P1), distrib(P ou R, Q1).
95  distrib1(P, P).
96
97  % toClause/3 : mise en clauses
98  toClause(P et Q, C1, C2) :- !,
99          toClause(P, C1, C3), toClause(Q, C3, C2).
100 toClause(P, [c(Pos,Neg)|Cs], Cs) :- inClause(P, Pos, [], Neg, []), !.
101 toClause(_, C, C).
102
103 % inClause/5
104 inClause(P ou Q, Pos, Pos1, Neg, Neg1) :- !,
105         inClause(P, Pos2, Pos1, Neg2, Neg1),
106         inClause(Q, Pos, Pos2, Neg, Neg2).
107 inClause(non P, Pos, Pos, Neg, Neg1) :- !,
108         notIn(P, Pos), putIn(P, Neg1, Neg).
109 inClause(P, Pos, Pos1, Neg, Neg) :- !,
110         notIn(P, Neg), putIn(P, Pos1, Pos).
111
112 % notIn/2
113 notIn(X, [X|_]) :- !, fail.
114 notIn(X, [_|L]) :- notIn(X, L).
115 notIn(X, []).
116
117 % putIn/3
118 putIn(X, [], [X]).
119 putIn(X, [X|L], L) :- !.
120 putIn(X, [Y|L], [Y|L1]) :- putIn(X, L, L1).
121
122 % putClause/1 : affichage des clauses
123 putClause([]).
124 putClause([c(Pos,Neg)|Cs]) :- putClause(Pos, Neg), putClause(Cs).
125 putClause(Pos, []) :- !, putDisjonction(Pos), write('.'), nl.
126 putClause([], Neg) :- !,
127         write(':- '), putConjonction(Neg,3),
128         write('.'), nl.
129 putClause(Pos, Neg) :-
130         putDisjonction(Pos),
131         write(' :- '),  nl,
132         tab(3), putConjonction(Neg,3),
133         write('.'), nl.
134
135 % putDisjonction/1
136 putDisjonction([T]) :- write(T).
137 putDisjonction([T1,T2|Q]) :-
138         write(T1), write(' ; '), putDisjonction([T2|Q]).
139
140 % putConjonction/2
141 putConjonction([T], _) :- write(T).
142 putConjonction([T1,T2|Q], N) :-
143         write(T1), write(', '), nl,
144         tab(N), putConjonction([T2|Q], N).
```

Corrigés TD 5

Bases de données

TD 5.1 : Définition des données

△ 42

1. **Création des tables :** les différentes tables de la base « fournisseurs-pièces-projets » se traduisent simplement par des faits Prolog.

Listing 5.1 – création des tables

```
% fournisseur/4 : table des fournisseurs
fournisseur(f1,martin,20,rennes).
fournisseur(f2,albin,10,paris).
fournisseur(f3,dupont,30,paris).
fournisseur(f4,morin,20,rennes).
% etc

% piece/5 : table des pièces détachées
piece(p1,ecrou,rouge,12,rennes).
piece(p2,boulon,vert,17,paris).
piece(p3,vis,bleu,17,grenoble).
piece(p4,vis,rouge,14,rennes).
% etc

% projet/3 : table des projets
projet(pj1,disque,paris).
projet(pj2,scanner,grenoble).
projet(pj3,lecteur,brest).
projet(pj4,console,brest).
% etc

% livraison/4 : table des livraisons
livraison(f1,p1,pj1,200).       livraison(f1,p1,pj4,700).
livraison(f2,p3,pj1,400).       livraison(f2,p3,pj2,200).
livraison(f2,p3,pj3,200).       livraison(f2,p3,pj4,500).
livraison(f2,p3,pj5,600).       livraison(f2,p3,pj6,400).
livraison(f2,p3,pj7,800).       livraison(f2,p5,pj2,100).
livraison(f3,p3,pj1,200).       livraison(f3,p4,pj2,500).
livraison(f4,p6,pj3,300).       livraison(f4,p6,pj7,300).
% etc
```

En SQL, chacune des tables sera d'abord créée par une requête du type :

```
-- table des fournisseurs
CREATE TABLE F (
      F CHAR(5) NOT NULL UNIQUE,
      NOM CHAR(20),
      PRIORITE DECIMAL(4),
      VILLE CHAR(15)
) ;
```

```
-- table des pièces détachées
CREATE TABLE P (
      P CHAR(6) NOT NULL UNIQUE,
      NOM CHAR(20),
      COULEUR CHAR(6),
      POIDS DECIMAL(3),
      VILLE CHAR(15)
) ;
```

```
-- table des projets
CREATE TABLE PJ (
      PJ CHAR(6) NOT NULL UNIQUE,
      NATURE CHAR(10),
      VILLE CHAR(15)
) ;
```

```
-- table des livraisons
CREATE TABLE L (
      F CHAR(5) NOT NULL,
      P CHAR(6) NOT NULL,
      PJ CHAR(4) NOT NULL,
      QUANTITE DECIMAL(5),
      UNIQUE (F,P,PJ)
) ;
```

Il faut ensuite insérer les données particulières dans chacune des tables :

```
INSERT INTO F (P,NOM,PRIORITE,VILLE)
VALUES ('f1','martin',20,'rennes') ;
-- etc
```

2. **Création de vues :** une vue est une table « virtuelle », c'est-à-dire sans existence physique sur le disque, mais qui apparaît à l'utilisateur comme une table réelle.

Listing 5.2 – création de vues

```
1  % creerVue/2
2  creerVue(Vue,But) :-
3          call(But), % instantier les arguments de la vue
4          affirmer(Vue).
5
6  affirmer(Vue) :-
7          \+ Vue, % éviter les doublons
8          !,
9          assertz(Vue).
10 affirmer(_).
```

La vue contenant les informations concernant les projets en cours à `brest` est obtenue par l'appel :

```
?- dynamic pjBrest/2.
true.
?- creerVue(pjBrest(R,N),projet(R,N,brest)).
R = pj3, N = lecteur ;
R = pj4, N = console ;
false.
```

```
-- en SQL
CREATE VIEW PJBREST (PJ,NATURE) AS
   SELECT PJ.PJ, PJ.NATURE, PJ.VILLE
      FROM PJ
      WHERE PJ.VILLE = 'brest' ;
```

On vérifie que la vue a bien été créée :

```
?- pjBrest(R,N).
R = pj3, N = lecteur ;
R = pj4, N = console.
```

On obtient de la même manière les vues `fpDistincts/2` et `pjF1P1/2` :

```
?- dynamic fpDistincts/2.
true
```

```
?- creerVue(fpDistincts(F,P),
              (fournisseur(F,_,_,VF), piece(P,_,_,_,VP), VF\==VP)
            ).
F = f1, P = p2, VF = rennes, VP = paris ;
F = f1, P = p3, VF = rennes, VP = grenoble ;
F = f1, P = p5, VF = rennes, VP = paris ;
F = f2, P = p1, VF = paris,  VP = rennes ;
F = f2, P = p3, VF = paris,  VP = grenoble ;
F = f2, P = p4, VF = paris,  VP = rennes ;
F = f2, P = p6, VF = paris,  VP = rennes ;
F = f3, P = p1, VF = paris,  VP = rennes ;
F = f3, P = p3, VF = paris,  VP = grenoble ;
F = f3, P = p4, VF = paris,  VP = rennes ;
F = f3, P = p6, VF = paris,  VP = rennes ;
F = f4, P = p2, VF = rennes, VP = paris ;
F = f4, P = p3, VF = rennes, VP = grenoble ;
F = f4, P = p5, VF = rennes, VP = paris ;
F = f5, P = p1, VF = brest,  VP = rennes ;
F = f5, P = p2, VF = brest,  VP = paris ;
F = f5, P = p3, VF = brest,  VP = grenoble ;
F = f5, P = p4, VF = brest,  VP = rennes ;
F = f5, P = p5, VF = brest,  VP = paris ;
F = f5, P = p6, VF = brest, VP = rennes.

?- dynamic pjF1P1/2.
true.
?- creerVue(pjF1P1(PJ,V),
              (
               (livraison(f1,_,PJ,_) ; livraison(_,p1,PJ,_)),
               projet(PJ,_,V)
              )
            ).
PJ = pj1, V = paris ;
PJ = pj4, V = brest ;
PJ = pj1, V = paris ;
PJ = pj4, V = brest ;
PJ = pj4, V = brest ;
false.
```

En SQL, les créations des vues correspondantes peuvent s'écrire :

```
CREATE VIEW FPDISTINCTS (F,P) AS
   SELECT F.F, P.P
      FROM F, P
      WHERE F.VILLE <> P.VILLE;
```

```
CREATE VIEW PFF1P1 (PJ,VILLE) AS
   SELECT DISTINCT PJ.PJ, PJ.VILLE
      FROM PJ, L
      WHERE PJ.PJ = L.PJ
         AND (L.F = 'f1' OR L.P = 'p1');
```

TD 5.2 : Requêtes simples

△ 43

1. **Détails de chacun des projets**

```
?- projet(PJ,N,V).
PJ = pj1, N = disque,   V = paris ;
PJ = pj2, N = scanner,  V = grenoble ;
PJ = pj3, N = lecteur,  V = brest ;
PJ = pj4, N = console,  V = brest ;
PJ = pj5, N = capteur,  V = rennes ;
PJ = pj6, N = terminal, V = bordeaux ;
PJ = pj7, N = bande,    V = rennes.
```

```
-- en SQL
SELECT PJ.PJ, PJ.NATURE, PJ.VILLE
      FROM PJ;
```

2. **Détails des projets rennais**

```
?- projet(PJ,N,rennes).
PJ = pj5, N = capteur ;
PJ = pj7, N = bande.
```

```
-- en SQL
SELECT PJ.PJ, PJ.NATURE, PJ.VILLE
      FROM PJ
      WHERE PJ.VILLE = 'rennes';
```

3. **Références des fournisseurs du projet** `pj1`

```
?- livraison(F,_,pj1,_).
F = f1 ;
F = f2 ;
F = f3 ;
false.
```

```
-- en SQL
SELECT DISTINCT L.F
      FROM L
      WHERE L.PJ = 'pj1'
      ORDER BY L.F;
```

4. **Livraisons dont la quantité est comprise entre** `600` **et** `750`

```
?- livraison(F,P,PJ,Q), Q >= 600, Q =< 750.
F = f1, P = p1, PJ = pj4, Q = 700 ;
F = f2, P = p3, PJ = pj5, Q = 600 ;
false.
```

```
-- en SQL
SELECT L.F, L.P, L.PJ, L.QUANTITE
      FROM L
      WHERE L.QUANTITE
            BETWEEN 600 AND 750;
```

5. **Projets qui se déroulent dans une ville dont la quatrième lettre est un** `n`

```
?- projet(PJ,_,V), name(V,L), nth1(4,L,110).
PJ = pj2, V = grenoble, L = [103, 114, 101, 110, 111, 98, 108, 101] ;
PJ = pj5, V = rennes,   L = [114, 101, 110, 110, 101, 115] ;
PJ = pj7, V = rennes,   L = [114, 101, 110, 110, 101, 115].
```

```
-- en SQL
SELECT PJ.PJ, PJ.VILLE
      FROM PJ
      WHERE PJ.VILLE LIKE '___n%';
```

TD 5.3 : JOINTURES

△ 43

1. **Triplets** `(fournisseur,piece,projet)` **tels que le** `fournisseur`**, la** `piece` **et le** `projet` **soient situés dans la même ville :**

Listing 5.3 – jointures (1)

```
1 jointure1(F,P,PJ) :-
2         livraison(F,P,PJ,_ ),
3         fournisseur(F,_,_,V),
4         piece(P,_,_,_,V),
5         projet(PJ,_,V).
```

```
?- jointure1(F,P,PJ).
F = f4, P = p6, PJ = pj7 ;
false.
```

```
-- en SQL
SELECT F.F, P.P, PJ.PJ
      FROM F, P, J
      WHERE F.VILLE = P.VILLE
            AND P.VILLE = PJ.VILLE ;
```

2. **Triplets (fournisseur,piece,projet) tels qu'un des éléments ne soit pas situé dans la même ville que les deux autres :**

Listing 5.4 – jointures (2)

```
jointure2(F,P,PJ) :-
        livraison(F,P,PJ,_),
        fournisseur(F,_,_,V1),
        piece(P,_,_,_,_,V2),
        projet(PJ,_,V3),
        \+ (V1 == V2,V2 == V3).
```

```
?- jointure2(F,P,PJ).
F = f1, P = p1, PJ = pj1 ;
F = f1, P = p1, PJ = pj4 ;
...
F = f5, P = p5, PJ = pj4 ;
F = f5, P = p6, PJ = pj4.
```

```
-- en SQL
SELECT F.F, P.P, PJ.PJ
      FROM F, P, J
      WHERE NOT
            ( F.VILLE = P.VILLE AND
              P.VILLE = PJ.VILLE) ;
```

3. **Triplets (fournisseur,piece,projet) tels que les trois éléments soient situés dans des villes différentes :**

Listing 5.5 – jointures (3)

```
jointure3(F,P,PJ) :-
        livraison(F,P,PJ,_),
        fournisseur(F,_,_,V1),
        piece(P,_,_,_,_,V2),
        projet(PJ,_,V3),
        V1 \== V2,
        V2 \== V3,
        V1 \== V3.
```

```
?- jointure3(F,P,PJ).
F = f2, P = p3, PJ = pj3 ;
F = f2, P = p3, PJ = pj4 ;
...
F = f5, P = p5, PJ = pj7 ;
F = f5, P = p6, PJ = pj2 ;
false.
```

```
-- en SQL
SELECT F.F, P.P, PJ.PJ
      FROM F, P, J
      WHERE F.VILLE <> P.VILLE
            AND P.VILLE <> PJ.VILLE
            AND F.VILLE <>
              PJ.VILLE ;
```

4. **Références des pièces provenant d'un fournisseur rennais :**

Listing 5.6 – jointures (4)

```
jointure4(P) :-
        livraison(F,P,_,_),
        fournisseur(F,_,_,rennes),
        affirmer('déjà vu'(P)),
        fail.
jointure4(P) :-
        retract('déjà vu'(P)).
```

```
?- dynamic 'déjà vu'/1.
true.
?- jointure4(P).
P = p1 ;
P = p6.
```

```
-- en SQL
SELECT DISTINCT L.P
      FROM L, F,
      WHERE PJ.F = F.F
            AND F.VILLE = 'rennes' ;
```

5. **Références des pièces provenant d'un fournisseur rennais, et destinées à un projet rennais :**

Listing 5.7 – jointures (5)

```
jointure5(P) :-
        livraison(F,P,PJ,_),
        fournisseur(F,_,_,rennes),
        projet(PJ,_,rennes),
        affirmer('déjà vu'(P)),
        fail.
jointure5(P) :-
        retract('déjà vu'(P)).
```

```
?- dynamic 'déjà vu'/1.
true.
?- jointure5(P).
P = p6.
```

```
-- en SQL
SELECT DISTINCT L.P
      FROM L, F, P
      WHERE PJ.F = F.F
            AND PJ.P = P.P
            AND P.VILLE = 'rennes'
            AND F.VILLE = 'rennes' ;
```

6. **Numéros des projets dont au moins un des fournisseurs ne se trouve pas dans la même ville que celle où se déroule le projet :**

Listing 5.8 – jointures (6)

```
jointure6(PJ) :-
        projet(PJ,_,V),
        fournisseur(F,_,_,VF),
        livraison(F,_,PJ,_),
        V \== VF,
        affirmer('déjà vu'(PJ)),
        fail.
jointure6(PJ) :-
        retract('déjà vu'(PJ)).
```

```
?- dynamic 'déjà vu'/1.
true.
?- jointure6(PJ).
PJ = pj1 ;
PJ = pj2 ;
PJ = pj3 ;
PJ = pj4 ;
PJ = pj5 ;
PJ = pj6 ;
PJ = pj7.
```

```
-- en SQL
SELECT DISTINCT PJ.PJ
      FROM L, F, PJ
      WHERE PJ.V <> F.V
            AND L.F = F.F
            AND L.PJ = PJ.PJ ;
```

TD 5.4 : UNIONS

1. **Villes dans lesquelles sont localisés au moins un fournisseur, une pièce ou un projet**

Listing 5.9 – union (1)

```
union1(V) :-
        (fournisseur(_,_,_,V); piece(_,_,_,_,V); projet(_,_,V)),
        affirmer('déjà vu'(V)),
        fail.
union1(V) :-
        retract('déjà vu'(V)).
```

```
?- dynamic 'déjà vu'/1.
true.
?- union1(V).
V = rennes ;
V = paris ;
V = brest ;
V = grenoble ;
V = bordeaux.
```

```
-- en SQL
SELECT F.VILLE FROM F
UNION
SELECT P.VILLE FROM P
UNION
SELECT J.VILLE FROM J
ORDER BY 1 ;
```

2. **Couleurs de pièces**

Listing 5.10 – union (2)

```
union2(C) :-
        piece(_,_,C,_,_),
        affirmer('déjà vu'(C)),
        fail.
union2(C) :-
        retract('déjà vu'(C)).
```

```
?- dynamic 'déjà vu'/1.
true.
?- union2(C).
C = rouge ;
C = vert ;
C = bleu.
```

```
-- en SQL
SELECT P.COULEUR FROM P
UNION
SELECT P.COULEUR FROM P ;
```

TD 5.5 : Mises à jour

△ 43

1. **Introduction d'un nouveau fournisseur :** l'insertion d'un nouveau fournisseur se fait simplement à l'aide du prédicat `assert/1`.

```
?- assert(fournisseur(f10,ibm,100,lyon)).
true.
```

```
-- en SQL
INSERT INTO F (P,NOM,PRIORITE,VILLE)
VALUES ('f10','ibm',100,'lyon') ;
```

2. **Changer la couleur rouge en orange :** le prédicat `modifier/2` permet de modifier un fait de la base de données.

Listing 5.11 – mises à jour (1)

```
modifier(Ancien,Nouveau) :-
        functor(Ancien,F,N),
        functor(Nouveau,F,N),
        clause(Ancien,true,RefAncien),
```

```
5           erase(RefAncien),
6           assert(Nouveau),
7           !.
```

En SWI-Prolog, les faits à modifier doivent avoir été déclarés dynamiques au préalable :

```
:- dynamic [fournisseur/4, piece/5, projet/3, livraison/4].
```

La modification d'une couleur de pièce utilise alors le prédicat `modifier/2` précédent.

Listing 5.12 – mises à jour (2)

```
1  modifCouleurPiece(Couleur1,Couleur2) :-
2           Couleur1 \== Couleur2,
3           piece(P,N,Couleur1,Pds,V),
4           modifier(piece(P,N,Couleur1,Pds,V),
5                    piece(P,N,Couleur2,Pds,V)),
6           fail.
7  modifCouleurPiece(_,_).
```

```
?- piece(P,_,rouge,_,_).
P = p1 ;
P = p4 ;
P = p6.
?- modifCouleurPiece(rouge,orange).
true.
?- piece(P,_,rouge,_,_).
false.
?- piece(P,_,orange,_,_).
P = p1 ;
P = p4 ;
P = p6.
```

```
-- en SQL
UPDATE P
SET COULEUR = 'orange'
WHERE P.COULEUR = 'rouge' ;
```

3. **Augmenter les livraisons pour les pièces détachées oranges :** ici encore, on utilisera le prédicat `modifier/2` pour augmenter les livraisons. Le taux d'augmentation sera écrit sous la forme d'un réel (exemple : 10% → 0.1).

Listing 5.13 – mises à jour (3)

```
1  modifQuantite(Couleur,Taux) :-
2           piece(P,_,Couleur,_,_),
3           livraison(F,P,PJ,Q),
4           Q1 is (1+Taux)*Q,
5           modifier(livraison(F,P,PJ,Q),livraison(F,P,PJ,Q1)),
6           fail.
7  modifQuantite(_,_).
```

```
?-  piece(P,_,orange,_,_),
    livraison(F,P,_,Q).
P = p1, F = f1, Q = 200 ;
P = p1, F = f1, Q = 700 ;
P = p1, F = f5, Q = 100 ;
...
P = p6, F = f5, Q = 500.
?- modifQuantite(orange,0.1).
true.
?- piece(P,_,orange,_,_),
   livraison(F,P,_,Q).
P = p1, F = f1, Q = 220.0 ;
P = p1, F = f1, Q = 770.0 ;
P = p1, F = f5, Q = 110.0 ;
...
P = p6, F = f5, Q = 550.0.
```

```
-- en SQL
```

4. **Modifier la priorité de fournisseurs :** de la même manière que dans les deux précédents prédicats, on utilise le prédicat `modifier/2` pour changer la priorité d'un fournisseur.

Listing 5.14 – mises à jour (4)

```
1 modifPriorite(F,Delta) :-
2         fournisseur(F,_,P,_),
3         fournisseur(F1,_,P1,_),
4         P1 < P,
5         P2 is P1 + Delta,
6         modifier(fournisseur(F1,N1,P1,V1),
7                  fournisseur(F1,N1,P2,V1)),
8         fail.
9 modifPriorite(_,_).
```

```
?- fournisseur(f4,_,P4,_),
   fournisseur(F1,_,P1,_),
   P1 < P4.
P4 = 20, F1 = f2, P1 = 10 ;
false.
?- modifPriorite(f4,10).
true.
?- fournisseur(f2,_,P2,_).
P2 = 20.
```

```
-- en SQL
SELECT F.PRIORITE FROM F
      WHERE F.F = 'f4';

-- on suppose que la requête
-- précédente donne 20
UPDATE F
SET PRIORITE = F.PRIORITE + 10
      WHERE F.PRIORITE < 20;
```

5. **Supprimer les projets se déroulant à Grenoble**

Listing 5.15 – suppression

```
1 supprimerProjet(Ville) :-
2         retract(projet(PJ,_,Ville)),
3         retract(livraison(_,_,PJ,_)),
4         fail.
5 supprimerProjet(_).
```

```
?- projet(PJ,_,grenoble).
PJ = pj2 ;
false.
?- supprimerProjet(grenoble).
true.
?- projet(pj2,_,_).
false.
?- livraison(_,_,pj2,_).
false.
```

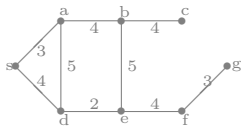
```
-- en SQL
DELETE FROM L
      WHERE 'grenoble' = (
            SELECT PJ.VILLE FROM PJ
                  WHERE PJ.PJ = L.PJ
      );

DELETE FROM PJ
      WHERE PJ.VILLE = 'grenoble';
```

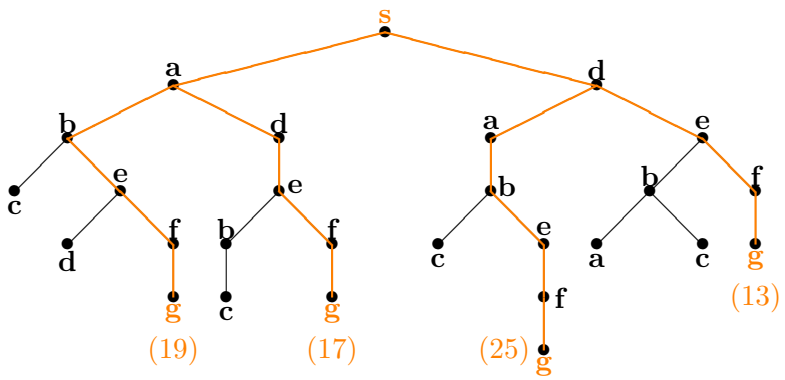
Corrigés TD 6

Recherche dans les graphes

TD 6.1 : Réseau routier

△ 45





Listing 6.1 – réseau routier

```
% route/3
route(s,a,3). route(s,d,4). route(a,b,4). route(b,c,4).
route(a,d,5). route(b,e,5). route(d,e,2). route(e,f,4). route(f,g,3).

% estim/3
estim(a,g,10.4). estim(b,g,6.7). estim(c,g,4.0). estim(d,g, 8.9).
estim(e,g, 6.9). estim(f,g,3.0). estim(g,g,0.0). estim(s,g,12.5).

% successeur/3
successeur(V1,V2,N) :- route(V2,V1,N) ; route(V1,V2,N).
```

Listing 6.2 – recherche simple

```
% chemin/4
chemin(D,A,C,K) :- chemin(D,A,[D],0,C,K).

% chemin/5
chemin(A,A,C,K,C,K).
chemin(D,A,P,KP,C,K) :-
        successeur(D,S,KDS), \+ dans(S,P),
        KP1 is KP + KDS, chemin(S,A,[S|P],KP1,C,K).
```

Listing 6.3 – transvasements

```
% capacite/2
capacite(1,8). capacite(2,5).

% remplir/1
remplir((V1,V2),(C1,V2)) :- capacite(1,C1), V1 =\= C1.
remplir((V1,V2),(V1,C2)) :- capacite(2,C2), V2 =\= C2.

% vider/1
vider((V1,V2),(0,V2)) :- V1 =\= 0.
vider((V1,V2),(V1,0)) :- V2 =\= 0.

% transvaser/2
transvaser((V1,V2),(0,W2)) :-
        capacite(2,C2), R2 is C2 - V2, V1 =< R2, W2 is V1 + V2.
transvaser((V1,V2),(W1,C2)) :-
        capacite(2,C2), R2 is C2 - V2, V1 > R2, W1 is V1 - R2.
transvaser((V1,V2),(W1,0)) :-
        capacite(1,C1), R1 is C1 - V1, V2 =< R1, W1 is V1 + V2.
transvaser((V1,V2),(C1,W2)) :-
        capacite(1,C1), R1 is C1 - V1, V2 > R1, W2 is V2 - R1.

% successeur/3
successeur(E1,E2,1) :- remplir(E1,E2) ; vider(E1,E2) ; transvaser(E1,E2).
```

Listing 6.4 – recherche avancée

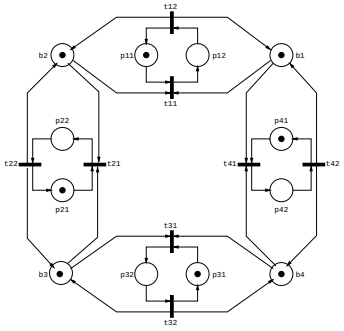
```
% chemin/5
chemin(Methode,I,F,Chemin,Cout) :-
        initialiser(Methode,I,F,PilFil),
        rechercher(Methode,F,PilFil,Chemin,Cout).

% initialiser/4
initialiser(heuristique,I,F,[E-0-[I]]) :- !, estim(I,F,E).
initialiser(_,I,_,[0-[I]]).

% rechercher/5
rechercher(profondeur,F,PilFil,Chemin,Cout) :-
        profondeur(PilFil,F,Chemin,Cout).
rechercher(largeur,F,PilFil,Chemin,Cout) :-
        largeur(PilFil,F,Chemin,Cout).
rechercher(meilleur,F,PilFil,Chemin,Cout) :-
        meilleur(PilFil,F,Chemin,Cout).
rechercher(heuristique,F,PilFil,Chemin,Cout) :-
        heuristique(PilFil,F,Chemin,Cout).

% profondeur/4
profondeur([K-K-[A|Passe]|_],A,[A|Passe],K).
profondeur([K-K-[D|Passe]|Reste],A,Chemin,Cout) :-
        successeurs(profondeur,A,K-K-[D|Passe],Suivants),
        conc(Suivants,Reste,Pile1), %------- empiler
        profondeur(Pile1,A,Chemin,Cout).

profondeur1(D,A,Chemin,Cout) :-
        profondeur([0-0-[D]],A,Chemin,Cout).

% largeur/4
largeur([K-K-[A|Passe]|_],A,[A|Passe],K).
largeur([K-K-[D|Passe]|Reste],A,Chemin,Cout) :-
```

```
33          successeurs(largeur,A,K-K-[D|Passe],Suivants),
34          conc(Reste,Suivants,File1), %------- enfiler
35          largeur(File1,A,Chemin,Cout).
36
37  largeur1(D,A,Chemin,Cout) :-
38          largeur([0-0-[D]],A,Chemin,Cout).
39
40  % meilleur/4
41  meilleur([K-K-[A|Passe]|_],A,[A|Passe],K).
42  meilleur([K-K-[D|Passe]|Reste],A,Chemin,Cout) :-
43          successeurs(meilleur,A,K-K-[D|Passe],Suivants),
44          conc(Reste,Suivants,File1), %------- enfiler
45          sort(File1,FileTriee), %------------- trier
46          meilleur(FileTriee,A,Chemin,Cout).
47
48  meilleur1(D,A,Chemin,Cout) :-
49          meilleur([0-0-[D]],A,Chemin,Cout).
50
51  % heuristique/4
52  heuristique([E-K-[A|Passe]|_],A,[A|Passe],K).
53  heuristique([E-K-[D|Passe]|Reste],A,Chemin,Cout) :-
54          successeurs(heuristique,A,E-K-[D|Passe],Suivants),
55          conc(Reste,Suivants,File1), %------- enfiler
56          sort(File1,FileTriee), %------------- trier
57          heuristique(FileTriee,A,Chemin,Cout).
58
59  heuristique1(D,A,Chemin,Cout) :-
60          estim(D,A,EDA),
61          heuristique([EDA-0-[D]],A,Chemin,Cout).
62
63  % successeurs/4
64  successeurs(heuristique,A,_-KP-[D|Passe],Suivants) :- !,
65          findall(E1-KP1-[S,D|Passe],
66                  (
67                  successeur(D,S,KDS), \+ dans(S,[D|Passe]), estim(S,A,ESA),
68                  KP1 is KP + KDS, E1 is KP1 + ESA), Suivants
69                  ).
70  successeurs(_,_,KP-KP-[D|Passe],Suivants) :-
71          findall(KP1-KP1-[S,D|Passe],
72                  (
73                  successeur(D,S,KDS), \+ dans(S,[D|Passe]), KP1 is KP + KDS
74                  ),
75                  Suivants).
```

TD 6.2 : Réseaux de petri

△ 48

1. **Dîner des philosophes chinois**



2. Simulateur de réseaux de petri

Listing 6.5 – réseaux de Petri

```
% simulRdp/1
simulRdp(stop) :- !.
simulRdp(continue) :-
        valids(Ts), aleaList(Ts,T), tir(T),
        !,
        putRdp(Ts,T,Suite), simulRdp(Suite).
simulRdp(continue) :- nl, write('!!! deadlock !!!').

% valids/1
valids(Ts) :- setof(T,valid(T),Ts).

% valid/1
valid(T) :- transition(T,Amont,_), preCond(Amont).

% preCond/1
preCond([]).
preCond([P|Ps]) :- marquage(P,N), N > 0, preCond(Ps).

% tir/1
tir(T) :- transition(T,Amont,Aval), maj(Amont, -1), maj(Aval, 1).

% maj/2
maj([],_).
maj([P|Ps],I) :-
        retract(marquage(P,M)), !,
        M1 is M + I, assert(marquage(P,M1)),
        maj(Ps,I).

% putRdp/3
putRdp(Ts,T,Suite) :-
        nl,
        tab(2), write('Transitions valides : '), write(Ts), nl,
        tab(2), write('Transition choisie  : '), write(T), nl,
        tab(2), write('Etat du systeme     : '), nl,
        putState,
        continue(Suite).

% putState/0
putState :-
        marquage(P,N), N > 0, place(P,State),
        tab(4), write(State), nl,
        fail.
putState :- nl.

% continue/2
continue(Suite) :-
        write('Continue (y/n) ? '), read(In), stopSimul(In,Stop).

% stopSimul/2
stopSimul(y,continue) :- !.
stopSimul(_,stop).

:- dynamic germeAlea/1.

% alea/1
alea(X) :-
```

```
57          retract(germeAlea(X1)), !,
58          X2 is (X1*824) mod 10657,
59          assert(germeAlea(X2)),
60          X is X2/10657.0 .
61
62  % alea/2
63  alea(N,X) :- alea(Y), X is floor(N*Y) + 1.
64
65  % aleaList/2
66  aleaList(L,T) :- longueur(N,L), alea(N,X), nieme(X,T,L).
67
68  :- use_module(library(date)).
69  :- abolish(germeAlea/1),
70          get_time(T),
71          stamp_date_time(T,date(_,_,_,_,_,_,S,_,_,_),0),
72          C is floor(S*10000),
73          assert(germeAlea(C)).
```

Troisième partie

Annexes

Liste des exercices

Listings

Bibliographie

[1] Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991

[2] Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007

[3] Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000

[4] Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003

[5] Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988

[6] Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996

[7] O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990

[8] Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990