Université des Sciences et de la Technologie d'Oran USTO–MB
Faculté des Mathématiques et de l'Informatique
Département d'informatique

Support de cours préparé pour les étudiants inscrits en 3ème Année LMD – S5
Diplôme préparé : Licence académique
Spécialité : Systèmes Informatique

Programmation logique: PROLOG

Cours & exercices corrigés

Auteur :
Karima Belmabrouk
(karima.belmabrouk@univ-usto.dz)
Maître de Conférences au département d'Informatique,
USTO-MB, Oran, Algérie.

Année universitaire 2019-2020

But du cours

- Découvrir un nouveau paradigme de programmation, la programmation logique.
- Initiation à la programmation en logique : application de certaines notions vues en cours de logique.
- Découverte de la programmation en Prolog: utilisation d'exemples liés aux cours de théorie des langages et d'analyse et compilation.

Plan du cours

I. Introduction à la programmation logique : langage PROLOG.
II. Principales caractéristiques de ce type de programmation.
III. Syntaxe et structures de données – opérateur de coupure.
IV. Sémantique des programmes PROLOG.
V. Le problème de la négation en PROLOG : l'hypothèse du monde clos et la négation par échec.

Références bibliographiques:

- Jacques Savoy, Introduction à la programmation logique : Prolog, 2006.
- Jacques Tisseau, Le langage Prolog, Ecole Nationale d'Ingénieurs de Brest, Révision : Fred Mesnard, université de la Réunion, 1991/2006.
- G. Gacôgne, 512 Problèmes corrigés – Prolog, Chapitre 15 «La programmation logique », Institut d'Informatique d'Entreprise, 2001.
- Martin Ludovic, Programmation fonctionnelle et logique - Dossier sur PROLOG, Licence informatique 2001/ 2002.
- Christine Solnon, Introduction à Prolog, 1997.
- Alain Colmerauer, Les Bases de Prolog IV, Laboratoire d'Informatique de Marseille, 1996.
- Alain Colmerauer, Philippe Roussel, La naissance de Prolog, juillet 1992.
- Pierre Nugues, La Programmation Logique et le langage Prolog, Notes de cours, 2000.
- Hoogger. Programmer en logique. Masson, 1987.
- Transparents du cours « Le langage Prolog : Un langage de programmation logique » du lien: http://www.inria.fr/oasis/Marjorie.Russo/, Marjorie Russo.

Partie I – Cours

I. Introduction à la programmation logique: Langage PROLOG

1. Introduction

La programmation logique est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes.
Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou requête.
La programmation logique est née de la découverte qu'une partie substantielle de la logique du premier ordre pouvait recevoir une interprétation procédurale qui est basée sur la stratégie de résolution SLD «Selected Lineary Defined» Resolution. Cette stratégie de résolution est une règle de simplification des buts, elle procède par une unification avec les têtes de clauses du programme.
La programmation logique est une forme de programmation dont le principe repose sur la définition des règles de logique mathématique au lieu de fournir la séquence d'instructions que l'ordinateur exécuterait.

2. Le modèle logique

Le modèle logique abandonne le principe de la programmation impérative (où le programmeur doit indiquer pas à pas à la machine ce qu'elle doit faire) au profit de la programmation déclarative qui consiste à représenter, via un formalisme approprié, les données du problème à partir desquelles un résultat pourra être déduit. Il est né de la découverte d'un sous-ensemble de la logique du premier ordre (la partie Hornienne de la logique des prédicats) associé à une interprétation procédurale correcte et complète.
L'idée sous-jacente de la programmation logique est la considération d'un programme comme ensemble de relations dont l'exécution consiste à 'démontrer' une nouvelle relation à partir de celles constituant le programme.
De ce fait, le calcul exécuté par un programme logique à travers une requête correctement fournie peut être vu comme une extraction d'un résultat à partir d'une preuve.

3. Langages de programmation logique

Parmi les langages de programmation logique nous pouvons citer: Prolog, Oz et Python. Prolog est le premier langage de programmation logique ;

4. Présentation de PROLOG

- Prolog est l'un des principaux langages de programmation logique.
- Le nom Prolog est un acronyme de PROgrammation LOGique.
- Il a été créé par Alain Colmerauer et Philippe Roussel vers 1972.
- Le but était de faire un langage de programmation qui permettait d'utiliser l'expressivité de la logique au lieu de définir pas à pas la succession d'instructions que doit exécuter un ordinateur.
- Prolog est utilisé dans de nombreux programmes d'intelligence artificielle et dans le traitement de la linguistique par ordinateur (surtout ceux concernant les langages naturels).

- Ses syntaxe et sémantique sont considérées comme très simples et claires(le but original était de procurer un outil pour les linguistes ignorant l'informatique).
- Beaucoup de recherches menant à l'implémentation actuelle de Prolog vinrent des effets du projet pour les ordinateurs de la cinquième génération qui utilisent comme base une variante.
- Prolog est basé sur le calcul des prédicats du premier ordre ; cependant il est restreint dans sa version initiale à n'accepter que les clauses de Horn(les versions modernes de Prolog acceptent des prédicats plus complexes, notamment avec le traitement de la négation par l'échec).
- L'exécution d'un programme Prolog est effectivement une application du théorème prouvant par résolution du premier ordre. Les concepts fondamentaux sont l'unification, la récursivité et le retour sur trace.
- L'algorithme de résolution de Prolog est basé sur une extension de la SLD-résolution.
- Une des particularités de Prolog est que l'on peut construire une base de connaissances dans un ordre indéterminé. Prolog peut ensuite résoudre des séries de problèmes logiques relatifs à une telle base de connaissances (notion base de données déductive).

5. Implémentations

Parmi les outils permettant de programmer en PROLOG, on peut citer:

- **SWI PROLOG** : possède un débuggeur graphique ainsi que plusieurs solveurs de contraintes, développé par l'Université d'Amsterdam(http://www.swi-prolog.org/).
- **GNU PROLOG** : développé par l'***INRIA***, propose un solveur de contraintes sur domaine fini(http://gnu-prolog.inria.fr/).
- **Sicstus PROLOG**(payant): possède des extensions en plus, dont plusieurs solveurs de contraintes(www.sics.se/sicstus) .
- **Turbo PROLOG** par Borland, désormais abandonné.
- **Open PROLOG**(http://www.cs.tcd.ie/open-prolog/).
- **Visual PROLOG**(http://www.visual-prolog.com/).
- **Prologia PROLOG II+** : est une version améliorée de PROLOG II, avec aide conséquente consultable (http://www.prologia.fr/).
- **Ciao Prolog**.
- **Prolog.NET**, développé à l'Institut de Technologie de l'Oregon.
- **Qu-Prolog**, un prolog multithread développé par l'Université du Queensland.
- **Quintus Prolog**, développé par le Swedish Institute for Computer Science.
- **Rebol Prolog**.
- **Strawberry Prolog**.
- **B-Prolog**.
- **YAP Prolog**, développé par l'université de Porto.
- **Squeak Prolog** Un Prolog intégré à Smalltalk dans l'environnement Squeak, issu du Prolog intégré à Smalltalk/V. Syntaxe non standard mais permet de mêler programmation objet impérative(Smalltalk) et logique(Prolog). Smalltalk peut poser des questions à Prolog et Prolog peut exécuter des ordres SmallTalk.

2. Principales caractéristiques de la Programmation logique

1. Principes de la programmation logique(Prolog)

La programmation en Prolog est très différente de la programmation dans un langage impératif. En Prolog, on alimente une base de connaissances de faits et de règles ; il est alors possible de faire des requêtes à la base de connaissances. L'unité de base de Prolog est le prédicat, qui est défini comme étant vrai. Un prédicat consiste en une tête et un nombre d'arguments. Par exemple : chat(tom).

2. Champs d'application de la programmation logique

Divers domaines d'application sont concernés par la programmation logique tels que :

- La conception de systèmes experts afin de simuler l'expertise humaine.
- La conception des SGBDR "*Systèmes de Gestion de Bases de Données Relationnelles*".
- Le traitement du langage naturel.
- L'enseignement assisté par ordinateur
- L'automatisme.
- La législation.

De plus, des issues théoriques telles que : le contrôle de la communication dans les systèmes multi-processus, la transformation et la synthèse de programmes et la programmation parallèle, sont basées sur la programmation logique.

3. Avantages de la programmation logique

Parmi les avantages de la programmation logique, on peut citer :

- **La simplicité :** grâce à l'aspect déclaratif qui réduit la tâche du développeur à la description des connaissances et du problème à résoudre.
- **La puissance :** grâce à l'utilisation du concept d'unification et de la résolution pour inférer la solution du problème à partir des descriptions.
- **Les procédures non directionnelles :** qui peuvent être utilisées pour résoudre divers types de problèmes et cela selon l'instanciation de leurs arguments.

4. Inconvénients de la programmation logique

Quant aux inconvénients de la programmation logique, on peut mentionner :

- **La lenteur :** due essentiellement à l'inadaptation du style logique à l'égard du modèle de Von Neumann qui répercute d'une manière irrémédiable sur le temps d'exécution des programmes.
- **La pauvreté en représentation des données :** les structures de donnée définies dans la programmation logique sont quelquefois inefficaces auprès des structures plus spécialisées comme les vecteurs par exemple.

III. Syntaxe et structures de données – opérateur de coupure

1. Introduction

Un programme PROLOG ne contient pas les types de données habituellement utilisés dans d'autres langages, du fait qu'il comporte une base de **faits** et un ensemble de **règles.**
Dès lors, un programme PROLOG est constitué d'un ensemble de **clauses**.
Une clause est une affirmation portant sur des **atomes logiques**. Ces derniers expriment une relation entre les termes.

2. Chargement d'un programme prolog:

On charge ou on substitue le fichier (mise à jour) dans l'interprète Prolog avec la commande :

```
?- consult(nom_de_fichier).
```

ou bien avec le raccourci :

```
?- [nom_de_fichier].
```

3. Les éléments fondamentaux du Prolog

3.1. Les faits

Les faits sont des affirmations qui décrivent des relations ou des propriétés, par exemple :

```
élève(jamel, 1975, info, 2).
élève(sarah, 1974, info, 2).
élève(ahmed, 1976, _, 1).

masculin(jamel).
masculin(ahmed).

féminin(sarah).

père(farouk, jamel).            % farouk est le père de jamel

mère(amel, jamel).
```

La forme générale d'un fait est la suivante :

```
prédicat(argument1, argument2, …).
```

Un prédicat est un symbole qui traduit une relation. L'arité est le nombre de ses arguments. On identifie un prédicat par son nom et son arité : prédicat/arité, par exemple mère/2, élève/4.

3.2. Les questions ou les requêtes

Une fois le programme chargé, on peut poser des questions sur les faits :

```
?- masculin(jamel).
Yes
?- masculin(farid).
No.                             (Pas dans la base de connaissances)
```

On peut aussi utiliser des variables. Elles peuvent s'identifier à toutes les autres valeurs : aux constantes, aux termes composés, aux variables elles-mêmes. Les constantes commencent par une minuscule, les variables commencent par une majuscule.

```
?- masculin(X).
X = jamel ;
X = ahmed ;
No
```

Le caractère « ; » permet de demander la solution suivante. Le caractère *Retour* arrête la recherche des solutions.

```
?- élève(X, Y, Z, 2)
X = jamel, Y = 1975, Z = info ;
X = sarah, Y = 1974, Z = info.
?-
```

3.3. Les types du prolog

En *Prolog*, tout est un terme :

- **Les constantes** ou les termes atomiques :
  1. ***Les atomes*** sont des chaînes alphanumériques qui commencent par une minuscule : jamel, farouk, tOtO1. On peut transformer une chaîne contenant des caractères spéciaux(point, espaces, etc.) dans un atome en l'entourant de caractères « ' ». Ainsi 'Cité Maraval', 'Bd des chasseurs' sont des atomes.
  2. ***Les nombres*** : 19, -25, -3.14, 23E-5
- **Les variables** commencent par une majuscule ou le signe _. _ tout seul est une variable anonyme : X XYZ Xyz _x _3 _. Le système renomme en interne ses variables et utilise la convention _nombre, comme _127 ou _G127.
- **Les structures** ou **termes composés:** se composent d'un foncteur avec une suite d'arguments. Les arguments peuvent être des atomes, des nombres, des variables, ou bien des structures comme par exemple élève(amine, 1975, info, 2, adresse(6, 'Cité Jamel Lafontaine', 'Oran')). Dans l'exemple, élève est le foncteur principal.

Lors d'une question, si elle réussit, les variables s'**unifient** en vis-à-vis aux autres termes. Les termes peuvent être composés comme par exemple adresse(6, 'Cité Jamel Lafontaine', 'Oran'). Ajoutons le terme élève(amine, 1975, info, 2, adresse(6, 'Cité Jamel Lafontaine', 'Oran')) dans le ficher et consultons-le. Posons la question :

```
?- élève(X, Y, Z, T, W).
X = amine
Y = 1975
Z = info
T = 2
W = adresse(6, 'Cité Jamel Lafontaine', 'Oran')
```

Prolog unifie le terme de la question au terme contenu dans la base de données. Pour ceci, il réalise la **substitution** des variables X, Y, Z, T, W par des termes, ici des constantes et un terme composé. On note cette substitution :
{X = amine, Y = 1975, Z = info, T = 2, W = adresse(6, 'Cité Jamel Lafontaine', 'Oran')}

On dit qu'un terme A est une **instance** de B s'il existe une substitution de A à B :

- masculin(jamel) et masculin(ahmed) sont des instances de masculin(X)
- {X = jamel} ou {X = ahmed} sont les substitutions correspondantes.

Un terme est **fondé***(ground term*) s'il ne comporte pas de variable : f(a, b) est fondé, f(a, X) ne l'est pas.
Une substitution est fondée si les termes qui la composent sont fondés : {X = a, Y = b} est fondée, {X = a, Y = f(b, Z)} n'est pas fondée.

3.4. Variables partagées

On utilise une même variable pour contraindre deux arguments à avoir la même valeur. Par exemple, pour chercher un élève qui porterait le nom de sa filière:

```
?- élève(X, Y, X, Z).
```

Les questions peuvent être des conjonctions et on peut partager des variables entre les buts. Pour chercher tous les élèves masculins, on partage la variable X entre élève et masculin :

```
?- élève(X, Y, Z, T), masculin(X).
```

3.5. Les règles

Les règles permettent d'exprimer des conjonctions de buts. Leur forme générale est :

```
TÊTE :- C1, C2,...,Cn.
```

La tête de la règle est vraie si chacun des éléments du corps de la règle C_1,...,C_n est vrai. On appelle ce type de règles des clauses de Horn.

```
fils(A, B):-
  père(B, A),
  masculin(A).
fils(A, B):-
  mère(B, A),
  masculin(A).

parent(X, Y):-
  père(X, Y).
parent(X, Y):-
  mère(X, Y).

grand_parent(X, Y):-
  parent(X, Z), % On utilise une variable intermédiaire Z
  parent(Z, Y).
```

Un prédicat correspond donc à un ensemble de règles ou de faits de même nom et de même arité : les clauses du prédicat. Plusieurs variantes de Prolog demandent que toutes les règles et tous les faits d'un même prédicat soient contigus – groupés ensembles – dans le fichier du programme. On note le prédicat par son nom et son arité, par exemple fils/2, parent/2, grand_parent/2. Les faits sont une forme particulière de règles qui sont toujours vraies. La notation :

```
fait.
```

est en effet équivalente à :

```
fait :- true.
```

Définissons un nouveau prédicat « ancêtre/2 » déterminant l'ancêtre X de Y par récursivité :

1. Condition de terminaison de la récursivité si c'est un parent direct.

```
ancêtre(X, Y):- parent(X, Y).
```

2. Sinon X est ancêtre de Y si et seulement si il existe Z, tel que X parent de Z et Z parent de Y.

```
ancêtre(X, Y):- parent(X, Z), ancêtre(Z, Y).
```

Lors de l'exécution d'une requête, Prolog examine les règles ou les faits correspondants dans l'ordre de leur écriture dans le programme : de haut en bas. Il utilise la première règle(ou le premier fait) du prédicat pour répondre. Si elle échoue, alors il passe à la règle suivante et ainsi de suite jusqu'à épuiser toutes les règles(ou tous les faits) définies pour ce prédicat. Lorsqu'une règle est récursive, l'interprète Prolog rappelle le prédicat du même nom en examinant les règles(ou les faits) de ce prédicat dans le même ordre.

Dans le corps d'une règle, la virgule «, » est le symbole représentant un ET logique : le conjonction de buts. Le symbole « ; » représente le OU logique, la disjonction de buts :

```
A :-
  B
  ;
  C.
```

est équivalent à

```
A :- B.
A :- C.
```

4. Mise en œuvre d'un programme

4.1. Chargement de fichiers

Les programmes sont dans des fichiers avec le suffixe « .pl » en général. Pour compiler et charger un programme :

```
?- consult(nom_du_fichier).
```

où nom_du_fichier est un atome, par exemple :

```
?- consult('file.pl').
```

ou bien le raccourci avec la commande :

```
?- [nom_du_fichier].
```

par exemple

```
?- ['file.pl'].
```

Autre raccourci avec SWI :

```
?- [file].
```

Chargement de plusieurs fichiers simultanément

```
?- ['file1.pl', 'file2.pl'].
```

Une fois que les fichiers sont chargés, on peut exécuter les commandes. Elles se terminent par un point « . » :

```
?- gnagnagna.
```

Les conjonctions de buts sont séparées par des «, » :

```
?- gna1, gna2, gna3.
```

On peut aussi inclure des directives dans un fichier par l'instruction :

```
:- clause_à_exécuter.
```

Les directives sont des clauses que l'interprète exécutera lors du chargement du programme.
L'affichage du contenu du fichier chargé se fait par :

```
?- listing.
```

L'affichage d'une clause particulière, ici père, se fait par :

```
?- listing(père).
```

Si on modifie le programme dans le fichier, on peut mettre à jour la base de données par(Pas avec SWI où on ne peut que consulter) :

```
?- reconsult('file.pl').
```

Le raccourci de rechargement est :

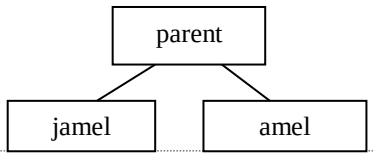
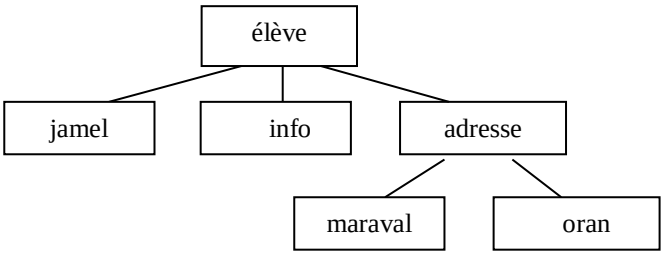
```
?- [-'file.pl'].
```

Finalement, on quitte Prolog avec

```
?- halt.
```

4.2. Termes et unification

Les termes peuvent se représenter par des arbres. Les nœuds sont les foncteurs.

Termes	Représentation graphique
`parent(jamel, amel).`	
`élève(jamel, info, adresse(maraval, oran)).`	

Formellement, t_1 est une instance de t_2 s'il existe une substitution *sigma* telle que : t_2[*sigma*] = t_1.

Exemple :

```
élève(jamel, info, adresse(maraval, Oran)).
```

est une instance de

```
élève(jamel, X, Y).
```

avec la substitution :

sigma = {(X, info),(Y, adresse(maraval, Oran)}

En effet, si on applique la substitution au dernier terme, on retrouve le premier.

On dit qu'un terme t_1 est plus général qu'un terme autre t_2 si t_2 est une instance de t_1.
On définit l'unification comme l'instance commune la plus générale de deux termes. L'opérateur Prolog de l'unification est « = ».

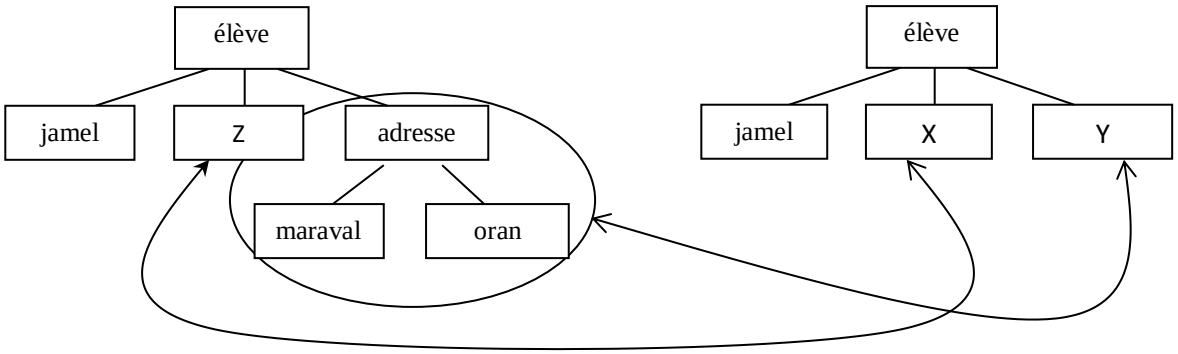
```
?- parent(X, farah) = parent(jamel, Y)
X = jamel, Y = farah
?- parent(jamel, farah) = parent(X, X).
No
?- élève(jamel, info, Z) = élève(X, info, adresse(maraval, Oran)).
X = jamel, Z = adresse(maraval, Oran)
?- élève(jamel, info, adresse(Z1, Z2)) = élève(Y, info, adresse(maraval,
Oran)).
Y = jamel, Z1 = maraval, Z2 = Oran
?- élève(X, Y, adresse(maraval, Oran)) = élève(T, info, adresse(maraval,
Oran)).
T = X, Y = info
?- parent(X, Y) = parent(T, amel), parent(X, Y), masculin(T).
X = jamel,
Y = amel,
```

```
T = jamel.
```

Cette dernière question s'interprète de la façon suivante : tout d'abord l'unification: X = T, Y = amel puis la recherche dans la base de T tel que parent(T, amel) ET masculin(T)(autrement dit, T est le père de amel). On doit avoir dans la base de données parent(jamel, amel) et masculin(jamel).
L'unification de deux termes peut se représenter de manière graphique :

```
élève(jamel, Z, adresse(maraval, Oran)) = élève(jamel, X, Y).
```

Elle se traduit par une superposition des deux arbres et une identification des variables à leurs branches :



4.3. Les listes

Une liste est une structure de la forme :

```
[a]
[a, b]
[a, X, adresse(X, Oran)]
[[a, b], [[[adresse(X, Oran)]]]]
```

`[]` est la liste vide.

La notation des listes est un raccourci. Le foncteur «.» est le foncteur de liste : `.(a,.(b,[]))`. Il est équivalent à `[a, b]`.

La notation «|» est prédéfinie et permet d'extraire la tête et la queue. La tête est le 1 er élément ; la queue est la liste restante sans le premier élément. Quelques exemples :

```
?- [a, b] = [X|Y].
X = a, Y = [b]
?- [a] = [X|Y].
X = a, Y = []
?- [a, [b]] = [X|Y].
X = a, Y = [[b]]
?- [a, b, c, d] = [X, Y|Z].
X = a, Y = b, Z = [c, d]
?- [[a, b, c], d, e] = [X|Y].
X = [a, b, c], Y = [d, e]
```

4.4. Quelques prédicats de manipulation de listes

Certains prédicats de manipulation de listes sont prédéfinis suivant les variantes de *Prolog*(c'est le cas à l'école).

Le prédicat *member/2*

Rôle : appartenance d'un élément à une liste :

```
?- member(a, [b, c, a]).
Yes
?- member(a, [c, d]).
No
```

Définition du prédicat

Programme	Commentaires

`member(X, [X|Y]).`
 	
`member(X, [Y|Ys]):-`
`  member(X, Ys).`
	Le cas trivial. On peut aussi écrire `member(X, [X|_]).` Les 2 versions se valent.
	La règle récursive

Exemples d'utilisation

```
?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c ;
No.
```

```
?- member(a, Z). % Déconseillé, car sans intérêt
Z = [a|Y] ;
Z = [Y, a|X] ;
```

etc.

```
?- not member(X, L).
```

Renvoie yes si member(X, L) vaut no et vice versa.

Le prédicat *append/3*

Rôle : Ajout de deux listes.

```
?- append([a, b, c], [d, e, f], [a, b, c, d, e, f]).
Yes
?- append([a, b], [c, d], [e, f]).
No
?- append([a, b], [c, d], L).
L = [a, b, c, d]
?- append(L, [c, d], [a, b, c, d]).
L = [a, b]
?- append(L1, L2, [a, b, c]).
L1 = [], L2 = [a, b, c] ;
L1 = [a], L2 = [b, c] ;
```

etc., avec toutes les combinaisons

Définition du prédicat

Programme	Commentaires

`append([], L, L).`
`append([X|Xs], Ys, [X|Zs]):-`
`  append(XS, YS, ZS).`
	La règle dans le cas trivial
	La règle générale

Le prédicat *intersection/3*

Rôle : ce prédicat réalise l'intersection de deux listes.

Sa syntaxe est : intersection(LEntrée1, LEntrée2, LIntersection).
Ici, on utilise une version approximée de ce prédicat pour simplifier le programme :

```
?- intersection([a, b, c],[d,b,e,a],L).
L = [a,b]
?- intersection([a,b,c,a],[d,b,e,a],L).
L = [a,b,a]
```

Définition du prédicat

Programme	Commentaires
`intersection([],_,[]).`	
`intersection([X|L1],L2,[X|L3]):- member(X,L2),intersection(L1,L2,L3).`	
`intersection([_|L1],L2,L3):- intersection(L1,L2,L3).`	si on arrive à cette règle, c'est que la tête de L_1 n'est pas dans L_2

Étude d'une exécution

```
?- intersection([a,b,c],[d,b,e,a],L).
```

N° appel	Clauses appelés	Variables lors de la descente			Dépilement récursif
1)	Clause 2	X = a L_1 = [b,c]	L_2 = [d,b,e,a]	[a|L_3] = L	L = [a,b]
2)	Clause 2	X' = b L'_1 = [c]	L'_2 = [d,b,e,a]	[b|L'_3] = L_3	L_3 = [b]
3)	Clause 3	L''_1 = []	L''_2 = [d,b,e,a]	L''_3 = L'_3	L'_3 = []
4)	Clause 1				L''_3 = []

Le prédicat *delete/3*

Rôle : Efface un élément d'une liste.

Sa syntaxe est : delete(Élément,Liste,ListeSansÉlément).

Définition du prédicat

Programme	Commentaires

`delete(_,[],[]).`
`delete(Élément,[Élément|Liste],ListeSansÉlément):-`
`delete(Élément,Liste,ListeSansÉlément).`
`delete(Élément,[T|Liste],[T|ListeSansÉlément]):-`
`delete(Élément,Liste,ListeSansÉlément).`
	Cas trivial
	Cas où Élément est en tête de liste
	Cas où Élément n'est pas en tête de liste(filtrage avec les règles précédentes.)

Le prédicat *reverse/2*

Rôle : inverse l'ordre d'une liste

1re solution coûteuse :

```
reverse([],[]).
reverse([X|Xs],Zs] :- reverse(Xs,,Ys),append(Ys,[X],Zs).
```

2e solution. On passe d'une arité 2 à une arité 3 :

```
reverse(X,Y):-reverse(X,[],Y).
reverse([X|XS],Accu,ZS):- reverse(XS,[X|Accu],ZS).
reverse([],ZS,ZS).
```

4.5. Arithmétique et opérateurs

Chaque *Prolog* dispose d'opérateurs infixés, préfixés et postfixés : +, -, *,/ . L'interprète les considère comme des foncteurs et transforme les expressions en termes :
2 * 3 + 4 * 2 est un terme identique à +(*(2, 3), *(4,2))).

Évaluer un terme représentant une expression arithmétique revient à appliquer les opérateurs. Ceci se fait par le prédicat prédéfini is/2.

```
?- X = 1 + 1 + 1.
X = 1 + 1 + 1(ou X = +(+(1,1),1)).
?- X = 1 + 1 + 1,Y is X.
X = 1 + 1 + 1,Y = 3.
?- X is 1 + 1 + a.
```

erreur(a pas un nombre)

```
?- X is 1 + 1 + Z.
```

erreur(Z non instancié à un nombre)

```
?- Z = 2,X is 1 + 1 + Z.
Z = 2
X = 4
```

Si on essaie

```
?- 1 + 2 < 3 + 4.
```

Il y a évaluation des 2 termes de gauche et de droite avant la comparaison. Il importe de bien distinguer les opérateurs arithmétiques des opérateurs littéraux,ainsi que de l'unification.

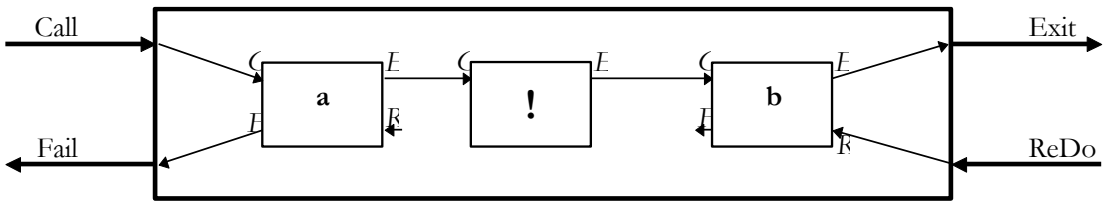
	Numérique	Littérale(terme à terme)
Opérateur d'égalité	=:=	==
Opérateur d'inégalité	=\=	\==
Plus petit	<	@<
Plus petit ou égal	=<	@=<

Exemple :

```
?- 1 + 2 =:= 2 + 1.
Yes.
?- 1 + 2 = 2 + 1.
No.
?- 1 + 2 = 1 + 2.
Yes.
?- 1 + X = 1 + 2.
X = 2.
?- 1 + 2 =:= 2 + 1.
Yes.
?- 1 + X =:= 1 + 2.
ERROR: =:=/2: Arguments are not sufficiently instantiated
?- 1 + 2 == 1 + 2.
Yes.
?- 1 + 2 == 2 + 1.
No.
?- 1 + X == 1 + 2.
No.
?- 1 + a == 1 + a.
Yes.
```

5. opérateur de coupure en prolog

Le predicat prédéfini « ! » – le *cut* – permet d'empêcher le retour en arrière.
La règle : `p(X):-a(X),!,b(X).` a la boîte d'exécution suivante :



Une fois un *cut* franchi :

1) ! coupe toutes les clauses en dessous de lui,
2) ! coupe tous les buts à sa gauche,
3) En revanche, possibilité de retour arrière sur les sous buts à la droite du *cut*.

```
P:- A1,...,Ai,!,Ai+1,...,An.
P:- B1,...,Bm.
```

Le cut permet d'améliorer l'efficacité d'un programme; mais il revient à faire fonctionner le Prolog comme un langage procédural et il peut introduire des erreurs de programmation vicieuses. Il faut donc l'utiliser avec prudence.

IV. Sémantique des programmes PROLOG

1. Dénotation d'un programme PROLOG

L'ensemble des atomes logiques qui sont des conséquences logiques d'un programme PROLOG P est appelé « la dénotation de P ».
De ce fait, la réponse de l'interprète PROLOG à une question est l'ensemble des instances de cette question. Cet ensemble est inclus dans la dénotation, il peut être calculé par une approche ascendante, dite 'en chaînage avant' : on part des faits, et on applique itérativement toutes les règles conditionnelles pour déduire de nouvelles relations vraies jusqu'à ce qu'on ait tout déduit.

Exemple :

Soit le programme PROLOG P suivant :

```
parent(farouk,jamel).
parent(jamel,amel).
parent(amel,meriem).
homme(farouk).
homme(jamel).
pere(X,Y):-parent(X,Y), homme(X).
grand_pere(X,Y):-pere(X,Z),parent(Z,Y).
```

L'ensemble des fais dans P est E_0 = {parent(farouk,jamel), parent(jamel,amel), parent(amel,meriem), homme(farouk), homme(jamel)}.
À partir de E_0 et P, on déduit E_1 l'ensemble des nouvelles relations vraies : E_1 = {pere(farouk,jamel), pere(jamel,amel)}.
À partir de E_0, E_1 et P, on déduit l'ensemble E_2 = {grand_pere(farouk,amel), grand_pere(jamel,meriem)}.
À partir de E_0, E_1, E_2 et P, on ne peut rien déduire.
La dénotation de P est donc l'union de E_0, E_1 et E_2.

2. Signification opérationnelle

Le calcul de la dénotation d'un programme par l'approche ascendante est généralement trop coûteux, voire infini. En contrepartie, on peut prouver qu'un but est une conséquence logique du programme en utilisant une approche descendante, dite en chaînage arrière :

Pour prouver qu'un but qui est composé d'une suite d'atomes logiques (But = [A_1, A_2, .., A_n]), l'interprète PROLOG commence par prouver le premier de ces atomes logiques(A_1). Pour cela, il cherche une clause dans le programme dont l'atome de tête s'unifie avec le premier atome logique à prouver(la clause A'_0 :- A'_1, A'_2, .., A'_r telle que l'unificateur le plus général upg(A_1,A'_0) = s). Puis l'interprète PROLOG remplace le premier atome logique à prouver(A_1) dans le but par les atomes logiques du corps de la clause, en leur appliquant la substitution(s). Le nouveau but à prouver devient But = [s(A'_1), s(A'_2), .., s(A'_r), s(A_2), .., s(A_n)]. L'interprète PROLOG recommence alors ce processus jusqu'à ce que le but à prouver soit vide (il n'y ait plus rien à prouver).
A ce moment, l'interprète PROLOG a prouvé le but initial. Si le but initial comportait des variables, il affiche la valeur de ces variables obtenue en leur appliquant les substitutions successivement utilisées pour la preuve.
D'une façon générale, il existe plusieurs clauses dans le programme dont l'atome de tête s'unifie avec le premier atome logique à prouver. Ainsi, l'interprète PROLOG répétera ce

processus de preuve pour chacune des clauses candidates. Par conséquent, il peut trouver plusieurs réponses à un but.
Le processus de preuve en chaînage arrière est résumé par la fonction prouver(But) suivante, cette procédure affiche l'ensemble des instances de But qui est inclus dans la dénotation du programme:

```
procedure prouver(But: liste d'atomes logiques )
   si But = [] alors
     /* le but initial est prouvé */
     /* afficher les valeurs des variables du but initial */
   sinon soit But = [A_1, A_2, .., A_n]
      pour toute clause(A'_0 :- A'_1, A'_2, ..,A'_r) du programme:
        (où les variables ont été renommées)
          s <- upg(A_1,A'_0)
          si s != echec alors
             prouver([s(A'_1), s(A'_2), .. s(A'_r), s(A_2), .. s(A_n)],
s(But-init}))
          finsi
      finpour
   finsi
fin prouver
```

L'interprète PROLOG exécute dynamiquement l'algorithme ci-dessus pour répondre aux questions posées. L'arbre constitué de l'ensemble des appels récursifs à la procédure est appelé 'arbre de recherche'.

V. Le problème de la négation en PROLOG :
l'hypothèse du monde clos et la négation par échec.

Un programme logique exprime ce qui est vrai. La négation, c'est donc ce que l'on ne peut pas prouver :
Le symbole de la négation est : « \+ »(anciennement not)

- Si G réussi ⇒ \+ G échoue,
- Si G échoue ⇒ \+ G réussit.

Le prédicat not se définit par :

```
not(P):-P, !, fail.
not(P):-true.
```

Exemple et précautions

Avec un \+(ou bien not), il vaut mieux que les variables soient instanciées :

```
?- \+ member(X, [a, b]).
No.
```

Le programme identifie X à une des variables, réussit et le not le fait échouer.

Si on ne trouve pas de règles ou de faits qui montrent qu'une demande est vraie, elle serait considérée comme fausse, c'est ce qui est appelé 'la prémisse du monde fermé'; on considère que la base de données contient tout ce qui doit être connu, alors il n'y a pas de monde extérieur qui pourrait contenir des preuves inconnues. Autrement dit, si un fait n'est pas connu comme étant vrai ou respectivement faux, il serait considéré comme faux ou respectivement vrai.

Exemple :

```
mangeable(X):-not indigeste(X).
```

Cette règle peut uniquement être évaluée en cherchant en premier lieu à prouver que 'indigeste(X)' est vrai. Si cette recherche échoue, alors "mangeable(X)" est vrai. C'est ce qu'on appelle la négation par échec.

Remarque :

not est en minuscule, sinon l'interpréteur PROLOG la considère comme étant variable.

Partie II – Exercices corrigés

Exercices

Exercice 1 : Qui a utilisé Quoi?

Cinq élèves ont utilisé trois outils de coloriage pour colorier leurs dessins à l'école comme suit:

- oussama a utilisé des crayons.
- amine a utilisé de la peinture à l'eau.
- farah a utilisé des pastelles.
- sarah a utilisé des crayons.
- ahmed a utilisé des pastelles.

1. Formuler la base de connaissances que nous devons interroger pour savoir qui a utilisé quoi.
2. Quelle est la question à poser si on veut connaître qui est ce qui a utilisé la peinture à l'eau.
3. Quelle est la question à poser si on veut connaître si farah a utilisé des crayons ou non.

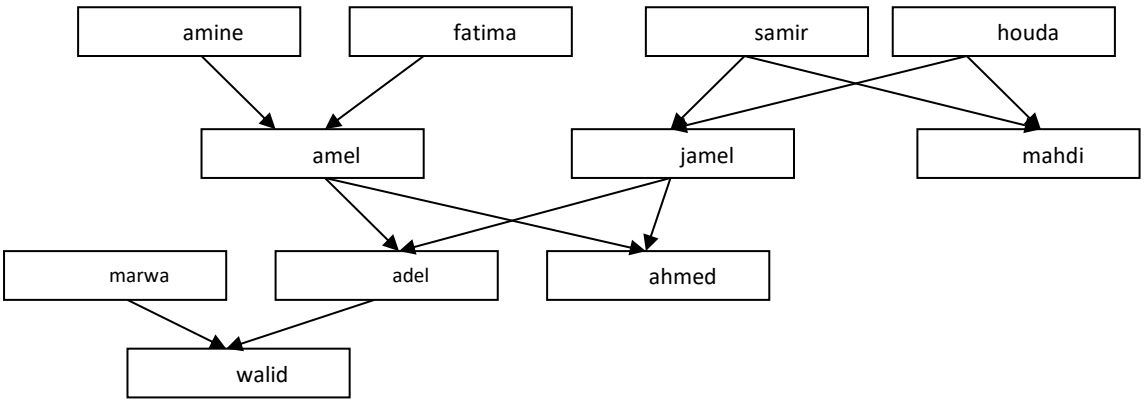
Exercice 2 : Un peu de logique

Soit l'ensemble des objets suivants : stylos, trousse, cartable, bureau.

Ecrire le programme prolog permettant de comparer ces objets par rapport à leurs volumes (par exemple X plus gros que Y ou X plus petit que Y,…) et d'en déduire également toutes les relations pouvant être obtenues par transitivité(X relation Y∧Y relation Z ⇒ X relation Z).

Exercice 3 : L'arbre généalogique d'une famille

On voudrait décrire la structure, d'une famille de 11 personnes, présentée par son arbre généalogique suivant:



Puis on voudrait que le programme nous donne la réponse à des questions du genre :

- Qui est le père ou la mère de jamel ?ou quels sont les parents de jamel ?
- Qui est le grand-père ou grand-mère de ahmed ?

Ecrire le programme prolog permettant de répondre à nos besoins.

On désire compléter notre programme en définissant de nouvelles relations de parenté au sein de notre arbre généalogique. Définissez pour cela les nouveaux prédicats suivants :

- fille(X,Y) qui réussit si X est la fille de Y
- frere(X,Y) qui réussit si X est le frère de Y
- soeur(X,Y) qui réussit si X est la soeur de Y
- gdparent(X,Y) qui réussit si X est un grand parent de Y(grand-père ou grand-mère)

Exercice 4 : Produit de deux entiers

En tenant compte que d'un sous ensemble E des nombre naturelle tel que

E ={0,1,2,..,10}

Ecrire le programme qui donne le produit de deux nombres entiers.

Exercice 5 : PGCD de deux entiers

Ecrire en prolog le programme qui calcul le PGCD en tenant compte du contexte suivant :

Propriétés du PGCD **D** de **X** et **Y**

- *si **X** et **Y** sont égaux, **D** vaut **X***
- *si **X** < **Y** alors **D** est le PGCD de **X** et de **Y** - **X***
- *si **Y** < **X** alors échanger le rôle de **X** et **Y***

Exercice 6 :

Ecrire le programme prolog permettant de calculer la somme de tous les éléments positifs d'une liste d'entiers(positifs et négatifs) ainsi que celle des éléments négatifs.

Exercice 7 :

Donner la définition des prédicats suivants :

- nb_éléments(L,N) permettant de calculer le nombre d'éléments N de la liste L.
- moyenne(L,M) permettant de calculer la moyenne M des éléments de la liste L.

Exercice 8 :

Donner le prédicat permettant de rechercher le dernier élément d'une liste donnée.

Exemple:

?- dernier_élément(X,[a,b,c,d]).

X = d

Exercice 9 :

1. Donner la définition du prédicat renverse_liste(L1,L2) qui permet de renverser l'ordre d'apparition des éléments de la liste L1 et de les mettre dans L2.
2. Utiliser le prédicat renverse_liste précédent pour donner la définition d'un autre prédicat palindrome(L) permettant de vérifier si la lise L est palindrome ou non.
   (Rappel: une liste palindrome est une liste qui peut être lue de droite à gauche ou de gauche à droite, par exemple L=[1,2,3,2,1] est une liste palindrome).

Exercice 10 :

Soit le programme prolog suivant :

```
predrop(L1,N,L2):-predrop(L1,N,L2,N).
predrop([],_,[],_).
predrop([_|Xs],N,Ys,1):-predrop(Xs,N,Ys,N).
predrop([X|Xs],N,[X|Ys],K):-K>1,K1 is K-1,predrop(Xs,N,Ys,K1).
prerange(I,I,[I]).
prerange(I,K,[I|L]):-I < K, I1 is I+1, prerange(I1,K,L).
```

1. Expliquer le rôle de chacun des prédicats **predrop** et **prerange** définis dans ce programme.
2. Donner un exemple d'appel pour chaque prédicat avec le résultat correspondant.

Exercice 11

Donner la définition du prédicat duplicate(L1,L2) qui permet de dupliquer tous les éléments de la liste L1 et les mettre dans L2.

Exemple:

```
?- duplicate([a,b,c,c,d],X).
   X = [a,a,b,b,c,c,c,c,d,d]
```

Exercice 12 :

Ecrire le programme qui permet de substituer un élément par un autre, dans une liste d'éléments.

Exemple :

```
?- substituer(a, b, [x, z, a, g, g, a], L).
L=[ x, z, b, g, g, b]
```

% toutes les occurrences de a sont remplacées par b dans la liste.

Exercice 13 :

Ecrire le programme qui permet de trier les éléments d'une liste par ordre croissant.

Exemple :

```
?-trier([20,-1,31,14,-22,11], L).
L=[-22,-1,11,14,20,31]
```

Exercice 14 :

Ecrire le programme contenant les prédicats suivants:

- element(X,L): X est un élément de la liste L,
- hors_de(X,L): X n'est pas un élément de la liste L,
- concat(L1,L2,L1L2): L1L2 est la concaténation des listes L1 et L2,
- renverse(L,Lrev): Lrev est la liste inverse de L,
- enleve(X,L,LX): X est un élément de la liste L, et LX est égal à la liste L sans l'élément X,

```
?- enleve(X,[a,b,a],LX).
   X=a          LX=[b,a]
   X=b          LX=[a,a]
   X=a          LX=[a,b]
   Yes
```

- partition(X,L,LinfX,LsupX): LinfX est la liste composée des éléments de L qui sont inférieurs à X, et LsupX est la liste composée des éléments de L qui sont supérieurs ou égaux à X.

```
?- partition(4,[3,8,4,1,6,5,2],LinfX,LsupX).
   LinfX = [3,1,2]
   LsupX = [8,4,6,5]
   Yes
```

Exercice 15 :

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

```
% homme/1
homme(jamel). % jamel est un homme
homme(fares). homme(ayoub). homme(emile).
homme(farouk). homme(badis). homme(malik).
% femme/1
femme(ines).   % ines est une femme
femme(louisa). femme(yasmine). femme(amina). femme(jihene).
femme(souad). femme(meriem). femme(maroua).
% père/2
père(emile,jamel).       % omar est le père de jamel
père(jamel,fares). père(fares,maroua). père(jamel,ayoub).
```

```
père(badis,ines). père(badis,farouk). père(farouk,souad).
père(farouk,jihene).
% mère/2
mère(louisa,jamel).      % louisa est la mère de jamel
mère(yasmine,ines). mère(yasmine,farouk).
mère(ines,fares). mère(ines,ayoub).
mère(amina,souad). mère(amina,jihene).
mère(meriem,maroua). mère(souad,malik).
```

1. Représenter l'arbre généalogique ainsi décrit, graphiquement.
2. Définir les prédicats suivants :
   - (a) fils(X,Y) % X est le fils de Y
   - (b) fille(X,Y) % X est la fille de Y
   - (c) femme(X,Y) % X est la femme de Y
   - (d) mari(X,Y) % X est le mari de Y
   - (e) frèreSœur(X,Y) % X est le frère ou la sœur de Y
3. Donner la sémantique dénotationnelle(Dénotation) du programme P défini par l'ensemble des faits donnés ainsi que par l'ensemble de prédicats à définir dans 2.

Exercice 16

Soit le programme P suivant :

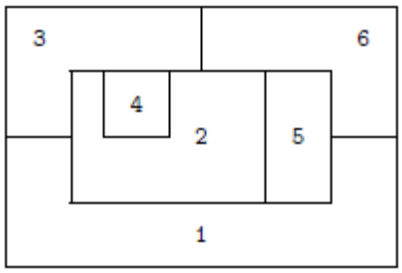
```
entrée(salade). entrée(gratin).
viande(steak). viande(escalope). viande(poulet).
poisson(calamar). poisson(merlan). poisson(crevette).
dessert(raisin). dessert(banane). dessert(orange).
plat(V):-viande(V).
plat(P):-poisson(P).
repas(E,P,D):-entrée(E), plat(P), dessert(D).
```

Quelles sont les questions prolog que doit poser l'utilisateur de ce programme s'il veut répondre aux questions suivantes:

- quels sont les repas comprenant du poisson ?
- le premier repas de la base comprend-il du poisson ?
- construire un repas avec le premier poisson de la base
- quel est le premier repas contenant du poisson ?

Exercice 17

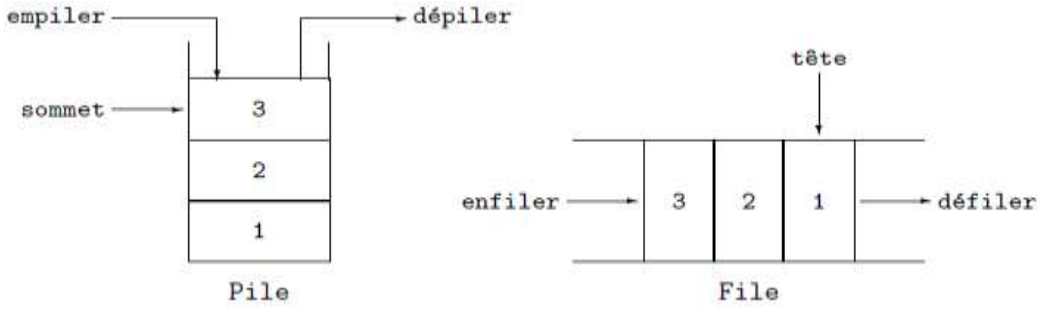
On dispose de 4 couleurs(rouge,jaune,vert,bleu) pour colorier la carte représentée ci-dessous.



1. Ecrire le 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.
2. Donner un exemple d'exécution de ce programme permettant de donner une solution à ce problème avec le résultat correspondant.

Exercice 18

Soient les structures de données présentées ci-dessous :



Sachant que les piles et les files sont représentées par des listes Prolog, définir les principales opérations sur ces structures :

a. **empiler(X,L1,L2)** : empiler X(ajouter X) à la pile L1 donne L2.
b. **dépiler(X,P1,P2)** : dépiler X(supprimer X) de la pile L1donne L2.
c. **pileVide(L)** : tester si la pile L est vide.
d. **sommet(X,L)** : X est le sommet de la pile L.
e. **enfiler(X,L1,L2)** : enfiler X(ajouter X) à la file L1 donne L2.
f. **défiler(X,L1,L2)** : défiler X(supprimer X) de la file L1donne L2.
g. **fileVide(L)** : tester si la file L est vide.
h. **tete(X,L)** : X est la tête de la file L.

Exercice 19(Automates finis)

On représente les mots sur un alphabet A par des listes, ainsi le mot "abaa" par [a, b, a, a] sur A = {a, b}. Un automate sur A est un ensemble d'états Q = {q_0, q_1, ...} ayant un état initial q_0, un(ou plusieurs) état final, et une relation de transition donnée par des triplets(q, x, q') où $x \in A$ et $q, q' \in Q$. Un mot m est reconnu par l'automate si et seulement si il existe une suite de transition de l'état initial à un état final au moyen des lettres de ce mot.

a) Ecrire le programme contenant les clauses nécessaires pour cette reconnaissance.

b) Ecrire les clauses particulières décrivant l'automate à deux états q0, q1, et la transition définie par tr(q0, a) = q1, tr(q1, b) = q0 avec q0 à la fois initial et final.

c) Décrire l'automate reconnaissant les mots contenant le sous-mot "iie" sur A = {a, e, i, o, u}.

Exercice 20(Triangle de Pascal)

Soit à calculer les **C(n,p)**, coefficients du binôme :

$\mathbf{(a+b)^n=C(n,0)a^nb^0+C(n,1)a^{n-1}b^1+...+C(n,p)a^{n-p}b^p+...+C(n,n-1)a^1b^{n-1}+ C(n,n)a^0b^n}$

où $\mathbf{C(n,p) = n!/(n!\times(n-p)!)}$.

Ce qui donne par exemple :$\mathbf{(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3}$.

La relation de *Pascal* nous dit que $\mathbf{C(n,p) = C(n-1,p)+C(n-1,p-1)}$.

La relation de *Pascal* énonce que chaque case est la somme de celle immédiatement au-dessus et de celle au-dessus à gauche. Par exemple le **70** de la dernière ligne est la somme des deux **35** de la ligne du dessus.(Vous pouvez vérifier, ça marche partout !). Ce qui permet de représenter les **C(n,p)** ainsi :

	p=0	**p=1**	**p=2**	**p=3**	**p=4**	**p=5**	**p=6**	**p=7**	**p=8**
n=0	1								
n=1	1	1							
n=2	1	2	1						
n=3	1	3	3	1					
n=4	1	4	6	4	1				
n=5	1	5	10	10	5	1			
n=6	1	6	15	20	15	6	1		
n=7	1	7	21	35	35	21	7	1	
n=8	1	8	28	56	70	56	28	8	1

1. En utilisant la relation de Pascal ainsi définie ; donner la définition du prédicat coef(N,P,C) qui a un sens si N>P et est vrai si C vaut N!/((N-P)!P!)
2. Donner le résultat d'exécution des requêtes suivantes :
   1) ?-coef(9,5,C). 2) ?-coef(2,7,21). 3) ?-coef(6,8,D).

Solutions des exercices

Exercice 1 : Qui a utilisé Quoi?
Remarque :
Il faut se rappeler qu'un **prédicat** commence toujours par une **minuscule** et non pas par une majuscule.

Oussama est différent de **o**ussama. Dans ce cas **Oussama** est considéré comme une **variable** et peut être substitué par n'importe quelle valeur. Et il ne faut pas oublier le point qui détermine la fin du prédicat.

Il faut se rappeler, également, que l'**arité** du prédicat est le **nombre d'arguments** de ce dernier.

1. La base de connaissances que nous devons interroger pour savoir qui a utilisé quoi :

```
utilise(oussama,crayons).
utilise(amine,peinture).
utilise(farah,pastelles).
utilise(sarah,crayons).
utilise(ahmed,pastelles).
```

Attention !!!

```
Utilise(oussama,crayons).
% faux car le prédicat ne doit pas commencer par une majuscule
utilise(Farah,pastelles).
% Farah<> farah
```

2. La question à poser est :

```
?- utilise(X,peinture).
```

3. La question à poser est :

```
?- utilise(farah,crayons).
```

Exercice 2 : Un peu de logique

```
plusGros('trousse','stylos').
plusGros('cartable','trousse').
plusGros('bureau','cartable').

estPlusGros(X,Y):-plusGros(X,Y).

estPlusGros(X, Z):-plusGros(X,Y),  estPlusGros(Y, Z).
```

Remarque :
Le prédicat *plusGros* peut être remplacé par le prédicat *plusPetit* en inversant l'ordre de ses arguments.

C'est-à-dire :

```
plusPetit('stylos','trousse').
plusPetit('trousse','cartable').
plusPetit('cartable','bureau').

estPlusPetit(X,Y):-plusPetit(X,Y).

estPlusPetit(X, Z):-plusPetit(X,Y),  estPlusPetit(Y,Z).
```

Exercice 3 :

```
homme(amine).
homme(samir).
homme(jamel).
homme(mahdi).
……..
femme(fatima).
femme(houda).
……..
père(amine,amel).
père(samir, jamel).
……
mère(fatima, amel).
mère(houda, jamel).
……..
parents(X,Y):-père(X,Y) ; mère(X,Y).
grand_père(X,Y):-père(X,Z), parents(Z,Y).
grand_mère(X,Y):-mère(X,Z), parents(Z,Y).
grand_parents(X,Y):-grand_père(X,Y) ; grand_mère(X,Y).
……
oncle(X,Y):-père(Z,Y), parents(T,X),parents(T,Z), homme(X).
oncle(X,Y):-mère(Z,Y), parents(T,X),parents(T,Z), homme(X).
tante(X,Y):-mère(Z,Y), parents(T,X),parents(T,Z), femme(X).
tante(X,Y):-père(Z,Y), parents(T,X),parents(T,Z), femme(X).
……
```

Exercice 4 : Produit de deux entiers

```
produit(X,Y,Z):-Z is X*Y.
```

(le signe « = » est remplacé par « is » en prolog**)**

Exercice 5 : PGCD de deux entiers

```
pgcd(X,X,X).
pgcd(X,Y,D):-X<Y,X1 is Y-X,pgcd(X,X1,D).
pgcd(X,Y,D):-X>Y,pgcd(Y,X,D).
```

Exercice 6 :

```
% sump : somme des éléments positifs
sump([], 0).
sump([T|Q], Sommep):- T<0,sump(Q, Sommep).
```

```
sump([T|Q], Sommep):-T>0,sump(Q, S),Sommep is T + S.
% sumn : somme des éléments négatifs
sumn([], 0).
sumn([T|Q], Sommen):- T>0,sumn(Q, Sommen).
sumn([T|Q], Sommen):-T<0,sumn(Q, S),Sommen is T + S.
% Il est également possible d'utiliser un seul prédicat sumPN d'arité
% 3 permettant de calculer la somme des % éléments positifs et celle
% des éléments négatifs en même temps, comme suit :
sumPN([],0,0).
sumPN([T|Q],SommeP,SommeN):-T>0,SommeP is T+S, sumPN(Q,S,SommeN).
sumPN([T|Q],SommeP,SommeN):-T<0,SommeN is T+S, sumPN(Q,SommeP,S).
```

Exercice 7 :

a) définition du prédicat **nb_éléments(L,N)**

```
nb_éléments([],0).
nb_éléments([_|Q],N):-nb_éléments(Q,N1), N is N1+1.
```

b) définition du prédicat **moyenne(L,M)**

1re solution:

```
moyenne(L,M):-moyenne(L,N,S,M).
moyenne([T],1,T,T).    %      moyenne([],0,0,0).
moyenne([T|L],N,S,M):- moyenne(L,N1,S1,_), N is N1+1, S is T+S1, M is
S/N.
```

2e solution:

```
moyenne(L,M):-moyenne(L,N,M).
moyenne([T],1,T).    %  moyenne([],0,0).
moyenne([T|L],N,M):-moyenne(L,N1,M1), N is N1 + 1, M is(T+(M1*N1))/N.
```

Exercice 8 :

```
% Dernier élément d'une liste donnée.
dernier_élément(X,[X]).
dernier_élément(X,[_|L]):-dernier_élément(X,L).
```

Exercice 9 :

% renverse_liste

% 1re solution:

```
renverse([],[]).
renverse([X|XS], ZS] :- renverse(XS, YS), append(YS, [X], ZS).
```

% 2e solution. On passe d'une arité 2 à une arité 3 :

```
renverse(X, Y):-renverse(X, [], Y).
renverse([X|XS], Accu, ZS):- renverse(XS,[X|Accu], ZS).
renverse([], ZS, ZS).
```

% palindrome

```
palindrome(L):-renverse(L,L).
```

Exercice 10 :

1. **Rôles des prédicats :**
   - predrop(L1, N, L2): élimine l'élément d'ordre N de la liste L1 et met le reste des éléments dans L2.

- prerange(I,K,L) : affiche dans la liste L tous les éléments compris entre I et K.

2. Exemples :

```
   ?-predrop([a,b,c,d,e,f,g,h,i,k],3,X).
X = [a,b,d,e,g,h,k]
```

```
   ?-prerange(4,9,L).
L = [4,5,6,7,8,9]
```

Exercice 11 :

```
   % duplicate/2
   duplicate([],[]).
   duplicate([X|Xs],[X,X|Ys]):-duplicate(Xs,Ys).
```

Exercice 12

```
   % 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.
```

Exercice 13

Tris de listes : il existe plusieurs méthodes de tris. La résolution de l'exercice revient à définir les prédicats suivants correspondant aux différentes méthodes :

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]).
   Yes ;
   No.
   ?- triPermutation([c,a,b],LT).
   LT = [a, b, c];
   No.
```

```
   % triPermutation/2
   triPermutation(L,LT):-permutation(L,LT), enOrdre(LT).
    % enOrdre/1
   enOrdre([]).
   enOrdre([_]).
   enOrdre([T1,T2|Q]):- T1 @=< T2, enOrdre([T2|Q]).
```

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

```
   ?- triSelection([c,a,b],[a,b,c]).
   Yes ;
   No.
   ?- triSelection([c,a,b],LT).
   LT = [a,b,c] ;
   No.
```

```
   % triSelection/2
   triSelection([],[]).
   triSelection(L,[Min|LT]):-minimum(Min,L), select(Min,L,L1),
triSelection(L1,LT ).
```

```
   % minimum/2
   minimum(T,[T]).
   minimum(T1,[T1,T2|Q]):- minimum(M2,[T2|Q]),T1 @=< M2.
   minimum(M2,[T1,T2|Q]):- minimum(M2,[T2|Q]),T1 @> M2.
```

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

```
   ?- triBulles([c,b,a],[a,b,c]).
   Yes;
   No.
   ?- triBulles([c,b,a],LT).
   LT = [a,b,c];
   No.
```

```
   % triBulles/2
   triBulles(L,LT):- bulles(L,L1),triBulles(L1,LT ).
   triBulles(L,L):-\+bulles(L,_).
   % bulles/2
   bulles([X,Y|Q],[Y,X|Q]):- X @> Y.
   bulles([X,Y|Q],[X|L]):- X @=< Y, bulles([Y|Q],L).
```

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]).
   Yes ;
   No.
   ?-triDicho([c,b,a],LT).
   LT=[a,b,c] ;
   No.
```

```
   % triDicho/2
   triDicho([],[]).
   triDicho([T|Q],LT):-triDicho(Q,LQ),insertionDicho(T,LQ,LT).

   % insertionDicho/3
   insertionDicho(X,[],[X]).
   insertionDicho(X,L,LX):-dicho(L,L1,[X|Q]),conc(L1,[X,X|Q],LX).
   insertionDicho(X,L,LX):-dicho(L,L1,[T|Q]),X @>T,
insertionDicho(X,Q,LQ),conc(L1,[T|LQ],LX).
   insertionDicho(X,L,LX):-dicho(L,L1,[T|Q]),X
@<T,insertionDicho(X,L1,L2),conc(L2,[T|Q],LX ).

   % dicho/3
   dicho(L,L1,L2):-longueur(N,L), R is N//2,   % division entière
   conc(L1,L2,L),longueur(R,L1 ).
```

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

```
   ?- triFusion([c,b,a],[a,b,c]).
   Yes;
   No.
   ?- triFusion([c,b,a],LT).
   LT=[a,b,c] ;
   No.
```

```
    % triFusion/2
    triFusion([],[]).
    triFusion([X],[X]).
    triFusion([X,Y|Q],LT):-separation([X,Y|Q],L1,L2),triFusion(L1,LT1),
triFusion(L2,LT2),fusion(LT1,LT2,LT ).
    % separation/3
    separation([],[],[]).
```

```
    separation([X],[X],[]).
    separation([X,Y|Q],[X|Q1],[Y|Q2]):-separation(Q,Q1,Q2 ).
    % fusion/3
    fusion(L,[],L).
    fusion([],[T|Q],[T|Q]).
    fusion([X|QX],[Y|QY],[X|Q]):- X @=< Y, fusion(QX,[Y|QY],Q).
    fusion([X|QX],[Y|QY],[Y|Q]):- X @> Y, fusion([X|QX],QY,Q).
```

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

```
   ?-triRapide([c,b,a],[a,b,c]).
   Yes;
   No.
   ?-triRapide([c,b,a],LT).
   LT=[a,b,c];
   No.
```

```
   % triRapide/2
   triRapide([],[]).
   triRapide([T|Q],LT):-partition(Q,T,LInf,LSup),triRapide(LInf,LTInf),
triRapide(LSup,LTSup),conc(LTInf,T|LTSup],LT).
   % partition/4
   partition([],_,[],[]).
   partition([T|Q],Pivot,[T|LInf],LSup):- T @=< Pivot,
partition(Q,Pivot,LInf,LSup).
   partition([T|Q],Pivot,LInf,[T|LSup]):- T @> Pivot,
partition(Q,Pivot,LInf,LSup).
```

Exercice 14

```
   % élément(X,L)
   élément(X,[X|_]).
   élément(X,[_|Q]):- élément(X,Q).
```

```
   % hors_de(X,L)
   hors_de(X,L):-not élément(X,L).
```

```
   % concat(L1,L2,L1L2)
   concat([],L,L).
   concat([T|Q],L,[T|QL]):- concat(Q,L,QL).
```

```
   % renverse(L1,L2)
   renverse([],[]).
   renverse([T|Q],L):- renverse(Q,L1), concat(L1,[T],L).
                       % concat peut être remplacé par append
```

```
   % enlève(X,L,LX)
   enlève(_,[],[]).
   enlève(X,[X|L1],L2):- enlève(X,L1,L2).
   enlève(X,[T|L1],[T|L2]):- enlève(X,L1,L2),X\=T.
```

```
   % partition(X,L,LinfX,LsupX)
   partition([],_,[],[]).
   partition([T|Q],Pivot,[T|LInf],LSup):- T @=< Pivot,
partition(Q,Pivot,LInf, LSup ).
   partition([T|Q],Pivot,LInf,[T|LSup]):- T @> Pivot,
partition(Q,Pivot,LInf,LSup).
```

Exercice 15 :

Définition des prédicats

```
% fils/2
fils(X,Y):-père(Y,X), homme(X).
fils(X,Y):-mère(Y,X), homme(X).

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

% 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.
```

Remarque :

Cette solution n'est pas unique. L'étudiant peut proposer d'autres solutions(en définissant des prédicats intermédiaires par exemple).

1. Dénotation du programme P :

Soit E0 l'ensemble de faits élémentaires

E0= **{** homme(jamel), homme(fares), homme(ayoub), homme(emile), homme(farouk), homme(badis), homme(malik), femme(ines), femme(louisa), femme(yasmine), femme(amina), femme(jihene), femme(souad), femme(meriem), femme(maroua), père(emile,jamel), père(jamel,fares), père(fares,maroua), père(jamel,ayoub), père(badis,ines), père(badis,farouk), père(farouk,souad), père(farouk,jihene), mère(louisa,jamel), mère(yasmine,ines), mère(yasmine,farouk), mère(ines,fares), mère(ines,ayoub), mère(amina,souad), mère(amina,jihene), mère(meriem,maroua), mère(souad,malik) **}**.

Soit E1 l'ensemble de faits obtenus à partir de E0 et de P :

E1= **{** fils(jamel,louisa), fils(jamel,emile), fils(farouk,badis), fils(farouk,yasmine), fils(fares,jamel), fils(ayoub,jamel), fils(jérome,ines), fils(malik,souad), fille(ines,badis), fille(ines,yasmine), fille(souad,farouk), fille(souad,amina), fille(jihene,farouk), fille(jihene,amina), fille(maroua,meriem), fille(maroua,fares)}

Soit E2 l'ensemble de faits obtenus à partir de E0,E1 et P :

E2= **{** femme(louisa,emile), femme(yasmine,badis), femme(ines,jamel), femme(amina,farouk), femme(meriem,fares), frereSoeur(ines,farouk), frereSoeur(farouk,ines), frereSoeur(souad,jihene), frereSoeur(jihene,souad), frereSoeur(fares,ayoub), frereSoeur(jerom,fares) **}**.

Soit E3 l'ensemble de faits obtenus à partir de E0,E1,E2 et de P :

E3= **{** mari(emile,louisa), mari(badis,yasmine), mari(jamel,ines), mari(farouk,amina), mari(fares,meriem)**}.**

A partir de E0, E1,E2,E3 et P ont ne peut rien en déduire donc :

Dénotation(P)=E0∪E1∪E2∪E3 = { homme(jamel), homme(fares), homme(ayoub), homme(emile), homme(farouk), homme(badis), homme(malik), femme(ines), femme(louisa), femme(yasmine), femme(amina), femme(jihene), femme(souad), femme(meriem), femme(maroua), père(emile,jamel), père(jamel,fares), père(fares,maroua), père(jamel,ayoub), père(badis,ines), père(badis,farouk), père(farouk,souad), père(farouk,jihene), mère(louisa,jamel), mère(yasmine,ines), mère(yasmine,farouk), mère(ines,fares), mère(ines,ayoub), mère(amina,souad), mère(amina,jihene), mère(meriem,maroua), mère(souad,malik), fils(jamel,louisa), fils(jamel,emile), fils(farouk,badis), fils(farouk,yasmine), fils(fares,jamel), fils(ayoub,jamel), fils(jérome,ines), fils(malik,souad), fille(ines,badis), fille(ines,yasmine), fille(souad,farouk), fille(souad,amina), fille(jihene,farouk), fille(jihene,amina), fille(maroua,meriem), fille(maroua,fares), femme(louisa,emile), femme(yasmine,badis), femme(ines,jamel), femme(amina,farouk), femme(meriem,fares), mari(emile,louisa), mari(badis,yasmine), mari(jamel,ines), mari(farouk,amina), mari(fares,meriem), frereSoeur(ines,farouk), frereSoeur(farouk,ines), frereSoeur(souad,jihene), frereSoeur(jihene,souad), frereSoeur(fares,ayoub), frereSoeur(jerom,fares) **}**.

Exercice 16

/* quels sont les repas comprenant du poisson ? */

```
?- repas(E,P,D),poisson(P).
?- poisson(P),repas(E,P,D).
```

/* le premier repas de la base comprend-il du poisson ? */

```
?- repas(E,P,D),!,poisson(P).
```

/* construire un repas avec le premier poisson de la base */

```
?- poisson(P),!,repas(E,P,D).
```

/* quel est le premier repas contenant du poisson ? */

```
?- repas(E,P,D),poisson(P),!.
?- poisson(P),repas(E,P,D),!.
```

Exercice 17

1) Programme prolog

```
% couleur/1
couleur(rouge).
couleur(jaune).
couleur(bleu).
couleur(vert).
% carte/6
carte(C1,C2,C3,C4,C5,C6):-couleur(C1), couleur(C2),
couleur(C3),couleur(C4), couleur(C5), couleur(C6),C1 \== C2,
```

```
C1 \== C3, C1 \== C5, C1 \== C6, C2 \== C3, C2 \== C4,C2 \==
C5, C2 \== C6, C3 \== C4, C3 \== C6, C5 \== C6.
```

2) Exemple d'exécution

```
?- 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 ;
No.
```

Exercice 18

a)

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

b)

```
  % dépiler/3
dépiler(T,[T|Q],Q).
```

c)

```
  % pileVide/1
pileVide([]).
```

d)

```
  % sommet/2
sommet(X,[X|_]).
```

e)

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

f)

```
  % défiler/3
conc([],L,L).
conc([T|Q],L,[T|QL]):- conc(Q,L,QL ).
défiler(X,L1,L2):- conc(L2,[X],L1 ).
```

g)

```
  % fileVide/1
fileVide([]).
```

h)

```
  % tete/2
tete(X,[X]).
tete(X,[_|Q]):- tete(X,Q).
```

Exercice 19(Automates finis)

a) Le programme contenant les clauses générales pour cette reconnaissance.

```
reconnu(M):-init(E), continue(M, E).
continue([A|M], E):-trans(E, A, F), continue(M, F).
continue([], E):-final(E).
```

b) Les clauses particulières décrivant l'automate à deux états q0, q1, et la transition définie par tr(q0, a) = q1, tr(q1, b) = q0 avec q0 à la fois initial et final.

```
  % automate du petit b
init(q0).
final(q0).
trans(q0, a, q1).
trans(q1, b, q0). % il reconnait les mots de la forme(ab)n
```

```
?-reconnu([a, b, a, b, a, b, a, b]).
Yes
```

```
?-reconnu([a, b, b, b, a, a]).
No
```

c) L'automate reconnaissant les mots contenant le sous-mot "iie" sur A = {a, e, i, o, u}.

```
% automate du petit c
init(q0).
trans(q0,i,q1).
trans(q1,i,q2).
trans(q2,e,q3).
trans(q2,i,q2).
trans(q0,V,q0):-membre(V,[a,e,o,u]).
trans(q1,V,q0):-membre(V,[a,e,o,u]).
trans(q2,V,q1):-membre(V,[a,o,u]).
trans(q3,V,q3):-membre(V,[a,e,i,o,u]).%ou trans(q3,_,q3).
final(q3).
membre(X,[X|_]).
membre(X,[_|L]):-membre(X,L).
```

```
?-reconnu([a,a,i,i,i,o,i,i,e,e,a,u,u,o,i,a]).
Yes
?-reconnu([a,a,i,e,i,i,o,i,a]).
No
```

Exercice 20(Triangle de Pascal)

1. Définition du prédicat coef(N,P,C)

```
% coef(N,P,C)
coef(_,0,1).
coef(P,P,1).
coef(N,P,C):- N>P, P>0, N>0, N1 is N-1, P1 is P-1,
coef(N1,P1,C1), coef(N1,P,C2), C is C1+C2.
```

2. Résultat d'exécution des requêtes:

```
?-coef(9,5,C).
C=126 ;
No
```

```
?-coef(2,7,21).
No
```

```
?-coef(6,8,D).
No
```