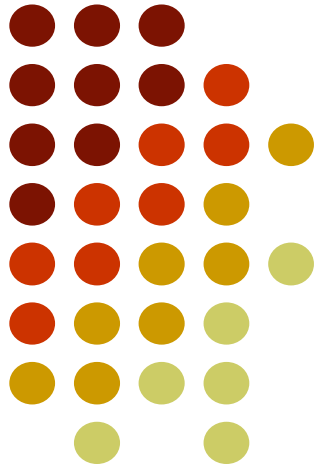
Le langage Prolog

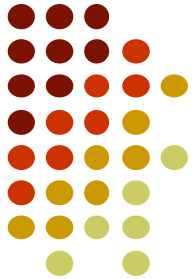
Partie 2 : description du langage

Jean-Michel Adam

Université Grenoble Alpes
L2 - MIASHS
Grenoble - France



Commentaires dans un programme Prolog



- Commentaire sur plusieurs lignes

```
/*  lignes
    de commentaires
    l'interprète ignore
    le texte entre slash étoile et étoile slash
*/
```

- Commentaire sur une seule ligne

```
% l'interprète ignore le texte après le caractère pourcent
```

Structures de données

- Objets atomiques : atomes, nombres, variables et chaines de caractères
- Prolog connaît aussi les **listes** : listes de valeurs entre [ ], exemple : [a, 2, c]
- Il existe aussi une structure de données plus complexe appelée arbre Prolog ou structure Prolog, ainsi que des dictionnaires.

Chaines de caractères

- En Prolog, les chaines de caractères sont notées entre " "
- Les chaines sont considérées comme des éléments atomiques, mais ne sont pas des atomes.
- Il existe des prédicats prédéfinis permettant la manipulation des chaines.

Partie 1
-
Opérations arithmétiques et logiques

Les expressions arithmétiques

- Prolog connaît les entiers et les nombres flottants.
- Il n'y a pas de valeur maximale pour les entiers !
- Les expressions arithmétiques sont construites à partir de nombres, de variables et d'opérateurs arithmétiques.
- L'évaluation d'une expression se fait par l'utilisation de l'opérateur **is** par exemple dans

  *X **is** 3 - 2*.

Expressions arithmétiques

- Opérations habituelles :

  addition (+), soustraction (-),

  multiplication (*), division entière (//),

  division flottante (/), modulo (mod), puissance(^)

- Fonctions mathématiques prédéfinies :

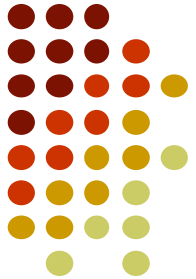
  abs(X), log(X), sqrt(X), exp(X), sign(X), random(X), sin(X), cos(X), tan(X), min(X,Y), max(X,Y), pi, etc.

Exemples :

```
?- X is 2^20, Y is exp(1)
X = 1048576,
Y = 2.718281828459045,
?- Z is random(100)+100.
Z = 151.
```

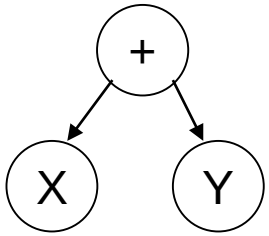
```
?- X is sin(pi/2), Y is cos(pi).
X = 1.0,
Y = -1.0.
?- S1 is sign(20), S2 is sign(-12).
S1 = 1,
S2 = -1.
```

Expressions arithmétiques



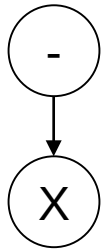
- Les expressions sont représentées par des arbres Prolog
- Opérateur binaire infixé : figure entre ses 2 arguments et désigne un arbre binaire

X + Y



- Opérateur unaire préfixé : figure avant son argument et désigne un arbre unaire

-X



Les expressions arithmétiques

- Expressions et unification : attention à certaines tentatives d'unification
  - la tentative d'unification entre *3+2* et *5* échouera. En effet, l'expression *3+2* est un arbre alors que 5 est un nombre.
  - L'évaluation des expressions ne fait pas partie de l'algorithme d'unification.

```
?- 2+3 = 5.
false.
```

Les prédicats de comparaison

- Comparaison des expressions arithmétiques
  - X =:= Y se traduit par X est égal à Y
  - X =\= Y se traduit par X est différent de Y
  - X < Y
  - X =< Y
  - X > Y
  - X >= Y

  Il y a évaluation puis comparaison.

```
?- 5 = 3 + 2.
false
?- 5 =:= 3 + 2.
true
```

Comparaison et unification de termes

- Comparer deux termes :
  - T1==T2 réussit si T1 est identique à T2
  - T1\==T2 réussit si T1 n'est pas identique à T2
  - T1=T2 réussit si T1 s'unifie avec T2
  - T1\=T2 réussit si T1 n'est pas unifiable à T2

Exemples :

```
?- f(X)==f(x).
false.

?- f(X)=f(x).
X = x.

?- f(X)\=f(x).
false.

?- f(X)\==f(x).
true.
```

Fonctions de vérification de type

- var, nonvar

```
?- var(X).
true.
?- X=2,var(X).
false.
```

```
?- var(X), nonvar(3).
true.
?- nonvar(X).
false.
```

- integer, float, number

```
?- float(exp(1)).
false.
?- X is exp(1), float(X), number(X).
X = 2.718281828459045.
```

```
?- X is sign(-2.3), integer(X).
false.
?- X is sign(-2.3).
X = -1.0.
```

- atom, string, atomic

```
?- atom(toto).
true.
?- atomic(toto).
true.
```

```
?- atom(3).
false.
?- atomic(3).
true.
```

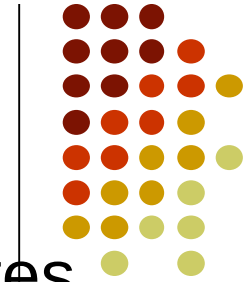
```
?- string('toto').
false.
?- string("toto").
true.
```

Exercice

- Donnez les réponses de Prolog aux requêtes suivantes

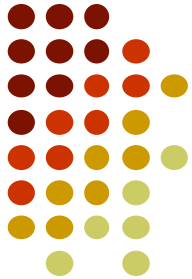
```
?- X is 9 mod 4.
X = 1
?- X = 9 mod 4.
X = 9 mod 4
?- X == 9 mod 4.
False.
?- f(x) =:= X+3.
ERROR: Arithmetic: `f(x)' is not a function
?- f(X) = X+3.
false
?- f(X) = f(X+3).
X = X+3
```

Exercice



- Donnez les réponses de Prolog aux requêtes suivantes

```
?- X = 1+1+1, Y = X.
X = Y, Y = 1+1+1.

?- X = 1+1+1, Y is X.
X = 1+1+1, Y = 3.

?- X is X+1.
ERROR: Arguments are not sufficiently instantiated

?- X is 2, X is X+1.
false

?- X is 2, X1 is X+1.
X = 2, X1 = 3.
```



Partie 2
-
Les opérations d’Entrée / Sortie

Les entrées-sorties

- Affichage sur l'écran
- Enregistrement dans un fichier
- Lecture à partir du clavier
- Lecture d’un fichier

Affichage de termes

- *write(1+2)*
  - affiche 1+2
- *write(X)*
  - *Si X n'est pas instanciée,* affiche X sous la forme _2451 qui est le nom interne de la variable.
  - Affiche la valeur de X si X est instanciée.
- nl permet de passer à la ligne (nl/0)
- tab(N) affiche N espaces (tab/1)

Exemple :

```
?- write(3+4), nl, tab(10), write("hello !").
3+4
          hello !
true.
```

Affichage de termes

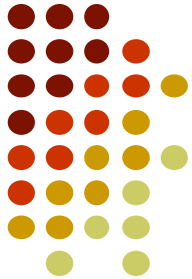
- writeln/1 est équivalent à write suivi de nl

```
?- writeln("hello"), writeln("world !").
hello
world !
true.
```

- display/1 agit comme write/1 mais en affichant la représentation sous forme d'arbre.

```
?- writeln(3*4+(6-(a+3))), display(3*4+(6-(a+3))).
3*4+(6-(a+3))
+(*(3,4),-(6,+(a,3)))
true.
```

Affichage de termes



- writeq/1 affiche la valeur d'un terme telle qu'elle est représentée en Prolog :

```
?- write('hello world').
hello world
true.

?- writeq('hello world').
'hello world'
true.

?- write("hello world").
hello world
true.

?- writeq("hello world").
"hello world"
true.
```

Affichage d’un caractère

- *put/1* s’utilise en donnant un caractère en argument, quelle que soit sa forme : code Unicode, atome, chaine de longueur 1 :

```
?- put(97).
a
true.
?- put(a).
a
true.
?- put("a").
a
true.
?- ?- put(0x2660),put(0x2663),put(0x2665),put(0x2666).
♠♣♥♦
true.
```

Affichages des chaines

- Dans les anciennes versions de Swi-Prolog et dans d'autres interprètes Prolog, la chaîne est représentée par la liste des codes ASCII ou Unicode des caractères la composant.

```
?- write("toto").
[116,111,116,111]
true.
```

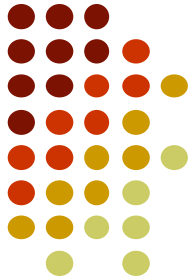
- Depuis la version 7 de Swi-Prolog gère les chaines de manière analogue aux autres langages de programmation, une chaine est un objet atomique.

```
?- atomic("toto").
true.
?- atom("toto").
false.
```

- Pour retrouver l'ancienne représentation des chaines :

```
set_prolog_flag(double_quotes, codes).
```

Saisie de données



- Lecture de termes :

  read/1 admet n'importe quel terme en argument.

  read lit un terme au clavier et l'unifie avec son argument. Le terme lu doit être obligatoirement suivi d'un point. Certains interprètes Prolog affichent un signe d'invite lorsque le prédicat read/1 est utilisé, c'est le cas de Swi-Prolog.

Exemple :

```
?- read(X).
|: a(1,2).
X = a(1,2).
?- read(A).
|: toto.
A = toto.
?- read(A).
|: "toto".
A = "toto".
```

Saisie de données

- Lecture d'un caractère :
  - *get/1* prend en argument un terme unifié avec le code Unicode du caractère lu. L'argument est une variable, qui prendra pour valeur le code Unicode du caractère lu. *get/1* ne lit que les caractères affichables, pour les autres caractères (tabulation, retour à la ligne, etc.) utiliser get_single_char/1.

```
?- get(A).
|: a
A = 97.
```

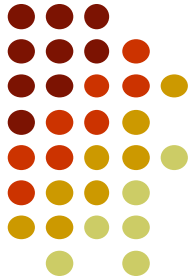
Fichiers

- Un fichier peut être ouvert en lecture (read) ou écriture (write ou append).
  - En écriture :
    - mode *write* : son contenu est effacé avant que Prolog y écrive.
    - mode *append* : Prolog écrira à partir de la fin du fichier.
  - Ouverture d'un fichier : prédicat *open/3*
    - argument 1 : nom du fichier
    - argument 2 : mode d'ouverture *write*, *append* ou *read*
    - argument 3 : variable qui va recevoir un identificateur de fichier appelé *flux* ou *stream*.

Fichiers

- Les prédicats *read, write, writeq, writeln, get, put,* admettent un second argument : le flux identifiant le fichier.
  - Ex : *writeq(Flux, X).* où Flux est un identificateur de fichier
  - Ex : *read(Flux,X), get(Flux, X)*
  - Fermeture du fichier : prédicat *close/1* qui prend en argument le flux associé au fichier.
- Le prédicat *get0/2* permet de lire tous les caractères d'un fichiers, y compris les tabulations, et caractères de fin de ligne.

Exemple d’usage d’un fichier



```
ecrire(T) :-
    open('a.pl', append, Flux), /* ouverture du fichier a.pl */
    write(Flux, T), nl(Flux),   /* enregistrement de T */
    close(Flux).                /* fermeture du fichier */

?- ecrire('etoile(soleil).'). /* ajout de etoile(soleil). au fichier a.pl */
```

Partie 3
-
Négation

La négation

- Problème de la négation et de la différence en Prolog
- Pas de moyen en Prolog de démontrer qu'une formule n'est pas déductible.

La négation

- Négation par l'échec
  - Si *F* est une formule, sa négation est notée \+ *F*. L'opérateur \+ est préfixé.
  - Prolog pratique la **négation par l'échec**, c'est-à-dire que pour démontrer \+ *F* Prolog va tenter de démontrer *F*. Si la démonstration de *F* échoue, Prolog considère que \+ *F* est démontrée.
  - Pour Prolog, \+ *F* signifie que la formule F n'est pas démontrable, et non que c'est une formule fausse. C'est ce que l'on appelle **l'hypothèse du monde clos**.

La négation

- \+(X) ne veut pas dire que X est faux mais que X ne peut pas être prouvé.
- Elle est utilisée pour vérifier qu'une formule n'est pas vraie.
- La négation par l'échec ne doit être utilisée que sur des prédicats dont les arguments sont déterminés et à des fins de vérification.
  - Son utilisation ne détermine jamais la valeur d'une variable

La négation

- Dangers :
  - Considérez le programme suivant :

    p(a).
  - Et interrogez Prolog avec :

    ?- X = b, \+(p(X)).

    X = b
  - Prolog répond positivement car *p(b)* n'est pas démontrable.

La négation

- Par contre, interrogez le avec :

  ?- \+(p(X)), X=b.

  false

- Prolog répond négativement car *p(X)* étant démontrable avec *X = a*, *\+ p(X)* est considéré comme faux.

  => Incohérence qui peut être corrigée si l'on applique la règle vue précédemment : n'utiliser la négation que sur des prédicats dont les arguments sont déterminés.

La négation

- L’unification :
  - Prédicat binaire infixé : *X* = *Y*
  - Pour démontrer *X* = *Y*, Prolog unifie *X* et *Y* ; s’il réussit, la démonstration est réussie, et le calcul continue avec les valeurs des variables déterminées par l’unification.

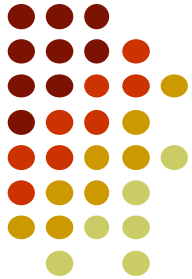
    ?- X = 2.

    true

    ?- X = 2, Y = 3, X = Y.

    false

La négation (fin)



- La **différence** est définie comme le contraire de l'unification
  - Elle est notée : *X* \= *Y*. Elle signifie que X et Y ne sont pas unifiables, et non qu'ils sont différents.

    Ex : *Z* \= *3.* Sera faux car Z et 3 sont unifiables.
  - Elle peut être définie comme :

    X \= *Y* :- \+*(X* = *Y).*

Partie 4
-
Récursivité

Programmation récursive

- Rappel : un programme récursif est un programme qui s’appelle lui-même.
- Exemple : factorielle
  - factorielle(0) = 1 (Cas de base)
  - factorielle(n) = n * factorielle(n-1) si n≠0
- En Prolog, un prédicat est récursif si au moins une de ses règles associées réfère à lui-même.

Exemple de Récursion simple

```
digere(X,Y) :- vientDeManger(X,Y).
digere(X,Y) :- vientDeManger(X,Z), digere(Z,Y).
vientDeManger(moustique,sang(marie)).
vientDeManger(grenouille,moustique).
vientDeManger(marie,grenouille).
```

```
?- digere(marie,moustique).
true.
?- digere(grenouille,marie)
false
?- digere(marie,sang(marie)).
true.
```

Intérêt de la récursion

- Permet d'exprimer de manière succincte un processus réitéré n fois
- Expl : Exprimer la relation de descendance
- Non récursif : Nombre infini de règles

```
descend(X,Y):- enfant(X,Y).
descend(X,Y):- enfant(X,Z), enfant(Z,Y).
descend(X,Y):- enfant(X,Z), enfant(Z,A), enfant(A,Y).      ...
```

- Récursif : Deux règles

```
descend(X,Y):- enfant(X,Y).
descend(X,Y):- enfant(X,Z), descend(Z,Y).
```

Le cas de base

- Sans cas de base, la récursion ne s'arrête pas

```
p :-p.
```

```
?- p.
```

- Il faut placer le cas de base avant la relation de récurrence dans la règle, sinon pas d'arrêt

```
descend(X,Y):- descend(X,Z), enfant(Z,Y).
descend(X,Y):- enfant(X,Y).
enfant(marie,paul).
```

```
?- descend(marie,paul).
ERROR: Stack limit (1.0Gb) exceeded
```

Exercice

- Ecrire l'arbre de recherche pour la requête

  ?- descend(marie,paul).

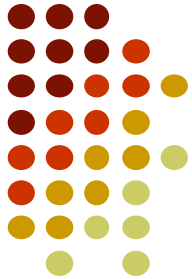
Avec la base :

```
descend(X,Y):- enfant(X,Y).
descend(X,Y):- enfant(X,Z), descend(Z,Y).
enfant(marie,paul).
```

- Trouvez-vous problématique la réécriture suivante du prédicat descend ?

```
descend(X,Y):- enfant(X,Y).
descend(X,Y):- descend(X,Z), descend(Z,Y).
```

Solution



- Réécriture: Donne les bonnes solutions
- Mais finit par une erreur : Out of local stack

Pour écrire un programme récursif

Il faut :

- Choisir sur quoi faire la récurrence
- Définir la relation de récurrence
- Définir le(s) cas de base

Exemple: Factorielle en Prolog

```
fact(0, 1).
fact(N, F) :- N1 is N-1, fact(N1, F1), F is N*F1.
```

```
?- fact(5,F).
F = 120 ;
ERROR: Out of local stack
```

Il faut des cas exclusifs !

Exemple : Factorielle

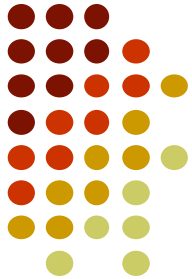
```
fact(0, 1).
fact(N, F) :- N >= 1, N1 is N-1, fact(N1, F1), F is N*F1.
?- fact(5,F).
F = 120 ;
false

?- fact(X,120).
ERROR: is/2: Arguments are not sufficiently instantiated
```

On a conçu le prédicat fact par rapport au premier argument N
N doit obligatoirement être instancié.

Spécification du prédicat fact

fact/2: fact(+N, -F)

- Le + indique que pour utiliser ce prédicat, il faut obligatoirement que l'argument N soit instancié.
- Le - indique qu'il faut obligatoirement que l'argument F ne soit pas instancié.
- Si dans une spécification de prédicat, un argument n'est précédé ni de +, ni de -, l'argument peut indifféremment être instancié ou non.



Partie 5
-
Bases de données et Prolog

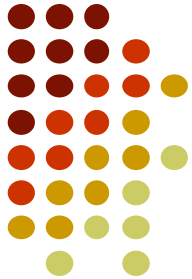
Bases de données et Prolog

- On peut facilement représenter des tuples de relations par des faits Prolog.
- Les diverses opérations de l’algèbre relationnelle se réalisent facilement par des requêtes en Prolog.
- Algèbre relationnelle : Ensemble d’opérations logiques permettant de manipuler des relations. Le résultat de toute opération de l'algèbre est une relation.

Algèbre relationnelle

- Opérations ensemblistes :
  - l'union notée $\cup$,
  - l'intersection notée $\cap$,
  - la différence notée -
  - le produit cartésien noté X
- Opérations propres aux bases de données :
  - la sélection notée σ
  - la projection notée Π
  - le produit noté $*$

Algèbre relationnelle en Prolog



- Opérations ensemblistes : l'union

  $R1 \cup R2$ représente l'union des lignes des relations R1 et R2.

  Pour pouvoir effectuer cette opération, les relations R1 et R2 doivent avoir les mêmes attributs.

```
r(a,1).        s(a,1).
r(b,2).        s(e,2).
r(c,3).        s(f,1).
```

```
union(X,Y) :- r(X,Y) ; s(X,Y).
```

SQL: select * from r union all select * from s;

Algèbre relationnelle en Prolog

- Opérations ensemblistes : l'intersection

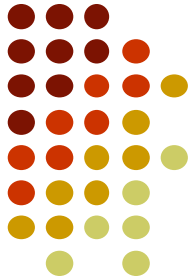
  $R1 \cap R2$ représente l'intersection des lignes des relations R1 et R2.

  R1 et R2 doivent avoir les mêmes attributs.

```
r(a,1).          s(a,1).
r(b,2).          s(e,2).
r(c,3).          s(f,1).

intersection(X,Y) :- r(X,Y) , s(X,Y).
```

SQL: select * from r intersect select * from s;

Algèbre relationnelle en Prolog



- Opérations ensemblistes : la différence

  R1–R2 représente la différence des relations R1 et R2 : lignes de R1 qui ne sont pas présentes dans R2. On a :

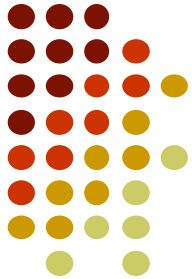
  $$R1\text{-}R2 = R1 - (R1 \cap R2)$$

  R1 et R2 doivent avoir les mêmes attributs.

```
r(a,1).        s(a,1).
r(b,2).        s(e,2).
r(c,3).        s(f,1).

difference(X,Y) :- r(X,Y) , \+(s(X,Y)).
```

SQL: select * from r except select * from s;

Algèbre relationnelle en Prolog



- Le produit cartésien

  R1 X R2 représente la relation formée des attributs des relations R1 et R2, construite en combinant chaque n-uplet de R1 avec tous les n-uplets de R2.

```
r(a,1).        s(a,1).
r(b,2).        s(e,2).
r(c,3).        s(f,1).

prodcart(A,B,C,D) :- r(A,B) , s(C,D).
```

SQL: select * from r, s;

Algèbre relationnelle en Prolog

- La sélection :

  σ(P)(R) représente les lignes de la relation R qui vérifient la propriété P exprimée sous la forme d'une expression logique :

```
r(a,1).
r(b,2).
r(c,3).
r(d,5).
r(a,3).
```

$\sigma_{(A2<3)}(R)$

```
select(X,Y) :- r(X,Y) , Y<3.
```

SQL: select * from r where r.Y<3;

Algèbre relationnelle en Prolog

- La projection : sélection de colonnes.

  Π(A1,..., Ak)(R) représente la sous-table de R formée uniquement des colonnes dont les attributs ont été spécifiés (A1,..., Ak).

  Si après cette opération plusieurs lignes sont identiques, les doublons sont éliminés (pas en Prolog).

r(a,1,xxx,12).
r(b,2,c,15).
r(c,3,as,123).
r(d,5,aaa,23).
r(a,3,ddd,7).

Π (A1,A4)(R)

projection(X,Y) :- r(X,_,_,Y).

SQL: select X,Y from r;

Algèbre relationnelle en Prolog

- La jointure naturelle ⋈

  R1 ⋈ **R2** représente la relation formée de l'union des attributs de R1 et de R2, les lignes de cette relation sont les n-uplets de R1 concaténés aux n-uplets de R2 si les attributs communs à R1 et R2 ont la même valeur.

  Cette opération n'est possible que si les schémas de R1 et de R2 ont une intersection non vide.

```
r(d,3).        s(d,3).
r(b,2).        s(c,2).
r(c,3).        s(a,1).
```

```
jointure(X,Y,Z) :- r(X,Y), s(X,Z).
```

SQL: select * from r natural join s;

Algèbre relationnelle

- La jointure ⋈

  R1 ⋈R2 [Q] représente la relation formée de l'union des attributs de R1 et de R2, les lignes de cette relation sont les n-uplets de R1 concaténés aux n-uplets de R2 pour lesquels la qualification Q est vérifiée.

- Ceci constitue la forme générale d'une jointure.

jointure(A,B,C,D) :- r(A,B), s(C,D), B==D.

SQL: select * from r inner join s on r.B = s.B;

r	s
r(d,3).	s(d,3).
r(b,2).	s(c,2).
r(c,3).	s(a,1).

Exercice

- Quelle question Prolog permet de représenter la requête SQL suivante?
  - *select * from CV union select * from Compte*
    avec les tables CV et Compte ayant 2 colonnes Nom, Age
  - *select * from Compte where Age>18*

Exercice

- Quelle question Prolog permet de représenter la requête SQL suivante?
  - *select * from CV union select * from Compte*
    avec les tables CV et Compte ayant 2 colonnes Nom, Age
  - *select * from Compte where Age>18*

```
cv(Nom,Age);compte(Nom,Age).

compte(Nom,Age), Age > 18.
```

Partie 6
-
Les Listes

Les Listes

- Une liste est une suite de termes entre crochets, ordonnée ou non, séparés par des virgules :
  - La suite e1, e2, …, eN est représentée en Prolog par la liste [e1, e2, …. , eN]
  - Exemples :
    - suite des variables X et Y est représentée par [X,Y]
    - suite fraises, tarte, glace par [fraises, tarte, glace]
- Liste vide : [ ]
- Une liste peut contenir n fois le même élément : [a, a]

Unification de liste

- Exemple de programme *menu* :

  entrees([artichauts, crevettes, oeufs]).

  viandes([grillade_de_boeuf, poulet]).

  poissons([sole, loup]).

  desserts([glace, tarte, fraises]).

  - Et si on pose la question :

    entrees(E).

  - Par unification, Prolog trouve la solution suivante :

    E= [artichauts, crevettes, oeufs]

- Nous pouvons donc unifier une liste et une variable.

Découpage Tête Queue

- La notation [X|Y] représente une liste non-vide dont la **tête** (le 1er élément) est X et la **queue** (le reste de la liste) est Y.
  - Cela constitue la base de l'utilisation des listes dans les programmes Prolog.
  - | symbole spécial décomposant la liste en Tête et Queue : [Tête | Queue] correspond à [Car | Cdr] (cf. Scheme)
  - [a, b, c] ≡ [a | [b | [c | [ ]]]]
  - La tête est un élément et la queue est une liste :

```
?- [a, b] = [X|Y].
X = a, Y = [b].
```

Contenu d'une liste

- Une liste peut contenir des :
  - Constantes (atomes, nombres, chaines)
  - Variables
  - Termes complexes
  - Listes

Listes et sous-listes

- Modification du programme **menu** :
  - On veut poursuivre le regroupement des données.
  - Liste unique qui regrouperait tous les mets du menu
    - L= [artichauts, crevettes, … , fraises]
  - Cette représentation n'est pas adéquate car il n'y a pas de distinction entre entrée, plat ou dessert.
  - Il est donc nécessaire de découper la liste en sous-listes qui rassembleront les entrées, plats et desserts. La sous-liste des plats sera elle-même divisée en deux parties, les viandes et les poissons.
  - Cela nous donne la structure :

    L = [[artichauts, crevettes, oeufs], [grillade_de_boeuf, poulet],
    [sole, loup], [*glace, tarte, fraises*]]

    L est de la forme : L = [L1, L2, L3, L4]

    Ou L = [E, V, P, D]

Exercice

- Dire si les expressions suivantes unifient et comment :

```
?- [X|Y] = [jean, marie,  leo,  lea].
?- [X|Y] = [].
?- [X|Y] = [a].
?- [X|Y] = [[], dort(jean), [2], [], Z].
?- [X,Y|W]=[[], dort(jean), [2], [], Z].
?- [_,X,_,Y|_] = [[], dort(jean), [2], [], Z].
?- [_,_,_,[ _|X ]|_] = [1,2, dort(jean), [2,3], [], Z].
```

Elément d'une liste (1)

- Nous avons besoin de pouvoir considérer les éléments d'une liste de façon individuelle, par exemple pour récupérer un élément particulier.
  - Supposons que l'on dispose du prédicat :

    `element_de(X,L)` capable d'extraire un objet `X` d'une liste `L`.

Elément d'une liste (2)

- Nous voulons adapter le programme suivant aux listes :

  menu(X, Y, Z) :- entree(X), plat(Y), dessert(Z).

  plat(X) :- viande(X).

  plat(X) :- poisson(X).

- Si on définit les entrées par une liste au lieu de faits simples, ce programme échouera. Il faut donc le modifier.
- Pour cela nous allons donner une définition du prédicat *entree (le fait d'être une entrée) basée sur entrees* (*la liste des entrées*).

Elément d'une liste (3)

- On utilise la propriété suivante : X est une entrée, si X est un élément quelconque de la liste des entrées.

  On peut donc écrire :

  entree(X) :- entrees(E), element_de(X, E).

- Et de façon analogue :

  viande(X) :- viandes(V), element_de(X, V).

  poisson(X) :- poissons(P), element_de(X, P).

  dessert(X) :- desserts(D), element_de(X, D).

Accès aux éléments (1)

- Nous recherchons les règles qui vont assurer le fonctionnement du prédicat *element_de(X, L*) qui signifie que X est l'un des éléments de L.
- On sait que toute liste L peut se décomposer simplement en deux parties, la tête et la queue de liste représentées sous la forme [T|Q].

  Cela nous conduit à distinguer deux cas :
  - X est élément de L si X est la tête de [T|Q] : X = T.
  - X est élément de L si X est élément de la queue de [T|Q] : X est élément de Q.

```
element_de(X, [X|_]).

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

Accès aux éléments (3)

```
element_de(X, [X|_]).                    (R1)
element_de(X, [_|Q]) :- element_de(X, Q).  (R2)
```

- On interprète ces deux règles de la façon suivante :
  - R1 : X est élément de toute liste qui commence par X.
  - R2 : X est élément de toute liste, si il est élément de sa queue.
- Il s'agit donc d'un appel récursif. La plupart des problèmes sur les listes ont des solutions mettant en jeu la récursivité.

Récursivité et Arrêt (1)

```
element_de(X, [X|_]).                    (R1)
element_de(X, [_|Q]) :- element_de(X, Q).  (R2)
```

On observe deux types de règles et deux types d'arrêt :

- Une ou plusieurs règles provoquent la récursivité, généralement sur des données assez simples, et assurent le déroulement de l'itération. Dans notre exemple, il s'agit de la règle R2.
- Une ou plusieurs règles stoppent la séquence d'appels récursifs. Appelés en général les cas de base. Dans notre exemple, c'est R1 qui s'en charge.
- Sauf impératif contraire, les règles d'arrêt sont placées en tête.

Récursivité et Arrêt (2)

```
(R1)    element_de(X, [X|_]).
(R2)    element_de(X, [_|Q]) :- element_de(X, Q).
```

- Il apparaît deux types d'arrêt :
  - Un **arrêt explicite**. Par exemple, dans R1, l'identité entre l'élément cherché et la tête de la liste fournie, ce qu'expriment les notations `X` et `[X|_]`
  - Un **arrêt implicite**. En particulier par la rencontre d'un terme impossible à démontrer.
  - Il existe cependant une forme d'erreur pour laquelle ces deux blocages se révèlent insuffisants, c'est la rencontre de listes infinies.

Utilisation du prédicat

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

```
?- element_de(X [artichauts, crevettes, oeufs]).
X = artichauts ;
X = crevettes ;
X = oeufs ;
false.
```

Le prédicat element_de est un prédicat prédéfini en Prolog, il possède le nom de **member**

Parcours d'une liste

- Cas de base :

`parcours([],v).` v = valeur associée à la liste vide

- Cas général :

```
parcours([T|Q],R):- parcours(Q,RI), R is f(RI,T).
```

R est calculé à partir de RI et de T

Exemple : somme des éléments d'une liste

```
somme([], 0).
somme([T|Q], S) :- somme(Q, Sq), S is T + Sq.
```

Exercice

Ecrire le prédicat nbpairs/2 qui prend en premier argument une liste d'entiers et produit en second argument le nombre d’entiers pairs de la liste.

Exemple d'utilisation:

```
?- nbpairs([5,4,1,12,3],N).
N = 2.
```

Solution de l'exercice

```
nbpairs([], 0).
nbpairs([T|Q], N) :- T mod 2 =:= 0, nbpairs(Q, N1), N is N1+1.
nbpairs([T|Q], N) :- T mod 2 =:= 1, nbpairs(Q, N).
```