

Chapitre 7 - Programmation logique

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Programmation logique

Paradigme : rappel

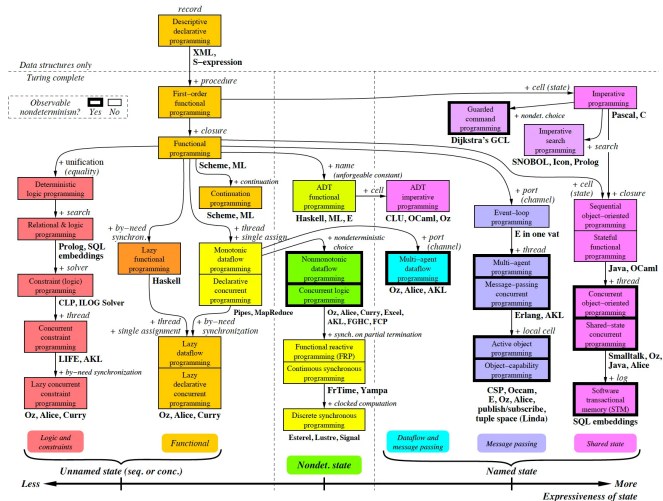
*Un paradigme est — en épistémologie et dans les sciences humaines et sociales — une **représentation du monde**, une manière de voir les choses, un modèle cohérent du monde qui repose sur un fondement défini (matrice disciplinaire, modèle théorique, courant de pensée).*

– Wikipédia

A language that doesn't affect the way you think about programming is not worth knowing.

– Alan Perlis

Exemples de paradigmes (Van Roy, 2009)



Source : "Programming Paradigms for Dummies", P. Van Roy

Programmation logique

Paradigme basé sur la **logique**

- Calcul de **prédicats**
- Cas particulier de la programmation par **contraintes**

Populaire dans les années 80 en intelligence artificielle

Rappel mathématique : relations

Une **relation** (binaire) R entre deux ensembles A et B est un sous-ensemble de $A \times B$

$$R \subseteq A \times B$$

Exemples :

- Relations d'équivalence, e.g., $(=) \subseteq A \times A$
- Relations d'ordre, e.g., $(\leq) \subseteq \mathbb{N} \times \mathbb{N}$
- Autres relations, e.g., $(\neq) \subseteq \mathbb{N} \times \mathbb{N}$, $\{(a, b) \in \mathbb{Z}^2 : a \text{ et } b \text{ sont de même signe}\}$

Rappel mathématique : fonctions

Une **fonction** f est une relation binaire $f \subseteq A \times B$, telle que pour tout $a \in A$, il existe **un unique** $b \in B$ tel que $(a, b) \in f$

- Agit sur des valeurs venant d'un **domaine** A (un ensemble)
- Donne des valeurs dans un **codomaine** B (un ensemble)

Relations n -aires

Pour tout $n \geq 1$, une **relation n -aire** R entre n ensembles $A_1, \dots, A_n$ est un sous-ensemble de $A_1 \times \dots \times A_n$.

Exemples :

- soit R la relation quaternaire ($n = 4$) sur $\mathbb{Z}$ telle que $(a_1, a_2, a_3, a_4) \in R$ si $a_1 + a_2 = a_3 \times a_4$. Par exemple, $(5, 10, 3, 5) \in R$ car $5 + 10 = 3 \times 5$;
- soit R la relation ternaire entre A_1 l'ensemble des systèmes d'exploitation, A_2 l'ensemble des logiciels et A_3 l'ensemble des versions telle que $(s, l, v) \in R$ si le logiciel l en version v existe sur le système d'exploitation s . Par exemple, $(\text{Manjaro Linux } 23.0.4, \text{ SWI-Prolog, } 9.0.4) \in R$.

Les relations n -aires servent ainsi à **représenter** toutes sortes de **connaissances**.

Programmation fonctionnelle vs. logique

Programmation fonctionnelle

Basée sur le concept de **fonction**.

$$f(x) = x \times 3 + 4$$

Programmation logique

Basée sur le concept de **relation** *n*-aire.

$$R(x, y) \Leftarrow y = x \times 3 + 4$$

Programmation fonctionnelle vs. logique

Programmation fonctionnelle

Basée sur le concept de **fonction**. Par exemple, OCaml :

```
let f x = x * 3 + 4
```

Programmation logique

Basée sur le concept de **relation** *n-aire*. Par exemple, Prolog :

```
r(X, Y) :- Y is X*3 + 4.
```

Prolog

Créé en 1972 (Alain Colmerauer et Philippe Roussel à Marseille)

Utilisé en :

- intelligence artificielle
- traitement automatique des langues
- conception de systèmes experts
- planification automatique
- systèmes à base de règles
- preuves de théorèmes

Introduction à Prolog

Implémentation

Il existe de nombreuses implantations de Prolog :

B-Prolog, Ciao, ECLiPSe, GNU Prolog, Poplog Prolog, P#, Quintus Prolog, SICStus, Strawberry, SWI-Prolog, Tau Prolog, tuProlog, WIN-PROLOG, XSB, YAP.

Attention : peu de compatibilité en dehors du standard (ISO/IEC 13211-1:1995)

Nous utiliserons **SWI-Prolog** dans le cadre de ce cours.

- Exécutable : swipl
- Plateforme en ligne : **SWISH**

Références

Pour cette partie du cours :

- Flach, P. [Simply Logical](#) (Chapitres 1 à 4)

Pour aller plus loin que le cours :

- Triska, M. [The Power of Prolog](#)

Hello, World!

Depuis un shell :

```
$ swipl hello.pl
```

```
hello.pl
```

```
:- initialization(main).
```

```
main :-
```

```
    write('Hello, world!'),
```

```
    nl,
```

```
    halt.
```

Charger un fichier

Pour charger un programme Prolog nommé, par exemple, `fichier.pl`, saisir la commande

```
?- [fichier].
```

dans `swipl`

Syntaxe : littéraux

- **Nombres**, exprimés en base dix :
 - Entiers 123
 - Flottants 3.1415
- **Atomes** : `bonjour`, `'bonjour tout le monde'`
 - Commence par une minuscule ou est entouré de guillemets simples (utile pour les atomes sur plusieurs mots)
 - `'bonjour'` et `bonjour` sont le même atome
- **Chaînes de caractères** : `"Cours de Prolog"`
 - Nous les utiliserons assez rarement

Syntaxe : variables logiques

- **Variables nommées** :
 - commencent par une majuscule : X, Foo
 - ou bien par un tiret bas : _Foo, _res
- **Variables anonymes** : _
- Les variables sont **immuables**
- Elles peuvent être instanciées (*ground*) ou non
 - Une variable instanciée est associée à une valeur

Syntaxe : termes

Un **terme** est

- un littéral : nombre, atome, chaîne de caractères
- une variable
- une structure (appelé aussi *terme composé* ou *foncteur*)

Exemples de termes :

- 37
- chat
- aime(chat, croquettes)
- aime(chat, Croquettes)
- adresse(numero(12), rue('Baker Street'), ville('Londres'))

Syntaxe : clause

Une **clause** est un fait ou une règle

- composée de termes
- les termes sont liés avec `:-` ou `,` ou `;`
- une clause se termine par un `.`

Fait (propositions vraies inconditionnelles)

```
chat(mange, croquettes, masse_gramme(50)).
```

Règle (propositions conditionnelles)

```
% Tête                                Corps 1          ET Corps 2
quantite_croquettes(Chat, 45) :- au_regime(Chat), sportif(Chat).
```

Exemples de clauses

```
eligible_assistant_enseignement(Etu) :-  
    (en_maitrise(Etu) ; en_doctorat(Etu)),  
    moyenne(Etu, Moyenne), Moyenne >= 95,  
    formation_pedagogique_recue(Etu).
```

Signifie qu'une personne étudiante peut devenir assistant d'enseignement **SI** (symbole :-) il ou elle

- est à la maîtrise **OU** (symbole ;) au doctorat
- **ET** (symbole ,) possède une moyenne supérieure à 95
- **ET** (symbole ,) a suivi une formation pédagogique.

Exemples de clauses

```
en_maitrise(alice).  
moyenne(alice, 99).  
formation_pedagogique_recue(alice).  
% ?- eligible_assistant_enseignement(alice).  
% Donne true
```

```
en_doctorat(bob).  
moyenne(bob, 95).  
% ?- eligible_assistant_enseignement(bob).  
% Donne false
```

Syntaxe : relation

Une **relation** (ou *prédicat*) est définie par une suite de clauses.

Définition de la relation `fac` tel que `fac(N, M)` vérifie $N! = M$.

% Cas de base.

`fac(0, 1).`

% Cas récursif.

`fac(N, M) :-`

`N >= 1,`

`N1 is N - 1, % `X is EXP` attribue à X le résultat de EXP.`

`fac(N1, M1),`

`M is N * M1.`

L'ordre des clauses est important : on lit de **haut** en bas et de **gauche** à droite.

Syntaxe : programme

Un **programme** est une suite de relations

```
fac(0, 1).  
fac(N, M) :-  
    N >= 1,  
    N1 is N - 1,  
    fac(N1, M1),  
    M is N * M1.
```

Utilisation : requête

Une **requête** peut être faite au REPL Prolog, et sera notée avec `?- :`

```
?- fac(5, M).  
M = 120 .
```

Nombres de Fibonacci

```
fib(0, 1). % Cas de base 1.  
fib(1, 1). % Cas de base 2.  
  
% Cas récursif.  
fib(N, M) :-  
    N >= 2,  
    N1 is N - 1, fib(N1, M1),  
    N2 is N - 2, fib(N2, M2),  
    M is M1 + M2.
```

Représentation de connaissances

Faits

Une relation peut être composée de **faits** :

```
fruit(pomme).
```

```
arbre(pommier, pomme).
```

indique que

- fruit est vrai pour pomme
- arbre est vrai pour la paire pommier et pomme.

Règles

On peut dériver une relation à partir de **règles** :

```
arbre_fruitier(X) :-  
    arbre(X, Y),  
    fruit(Y).
```

- **SI** arbre et vrai pour X et Y,
- **ET** si fruit et vrai pour Y
- **ALORS** arbre_fruitier est vrai pour X

Règles (mathématiques)

La règle :

```
arbre_fruitier(X) :-  
    arbre(X, Y),  
    fruit(Y).
```

est équivalente à :

$$\forall X \forall Y \text{ arbre_fruitier}(X) \iff (\text{arbre}(X, Y) \wedge \text{fruit}(Y))$$

Faits et règles

- **Fait** : vérité inconditionnelle
- **Règle** : vérité conditionnelle

Vocabulaire

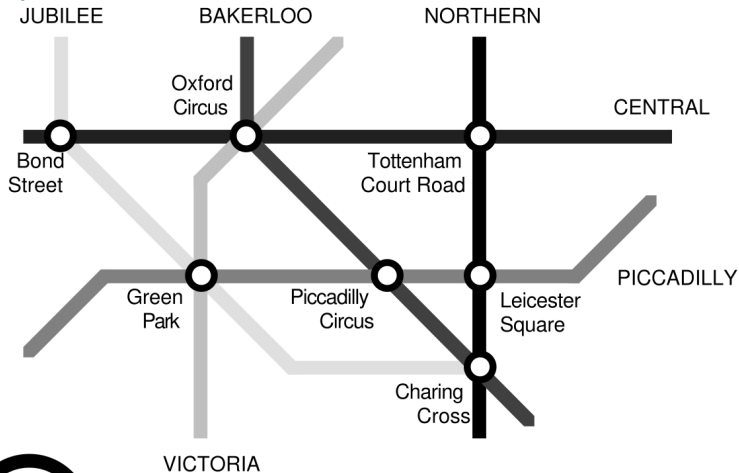
`arbre(pommier, pomme)`

- `arbre` est un symbole de relation
- `arbre/2` est une relation, 2 est son arité
- `pommier` et `pomme` sont des termes, arguments de la relation

`arbre_fruitier(X) :- arbre(X, Y), fruit(Y).`

- `arbre_fruitier(X)` est la *conclusion*
- `arbre(X, Y), fruit(Y)` est la *prémisse*

Exemple



Faits : exemple

```
connected(bond_street, oxford_circus, central).
connected(oxford_circus, tottenham_court_road, central).

connected(bond_street, green_park, jubilee).
connected(green_park, charing_cross, jubilee).

connected(green_park, piccadilly_circus, piccadilly).
connected(piccadilly_circus, leicester_square, piccadilly).

connected(green_park, oxford_circus, victoria).

connected(oxford_circus, piccadilly_circus, bakerloo).
connected(piccadilly_circus, charing_cross, bakerloo).

connected(tottenham_court_road, leicester_square, northern).
connected(leicester_square, charing_cross, northern).
```

Règles : exemple

On souhaite dire que deux stations sont *proches* si elles sont sur la même ligne avec au plus une seule station entre les deux.

```
nearby(X, Y) :-  
    connected(X, Y, _).
```

```
nearby(X, Y) :-  
    connected(X, Z, L),  
    connected(Z, Y, L).
```

Requêtes

```
?- nearby(tottenham_court_road, charing_cross).  
true.
```

```
?- nearby(tottenham_court_road, W).  
W = leicester_square ;  
W = charing_cross.
```

Tous les W qui vérifient la requête sont calculés.

Résolution de requête

On souhaite résoudre :

```
nearby(tottenham_court_road, W).
```

Sachant que :

```
nearby(X, Y) :-  
    connected(X, Y, _).
```

```
nearby(X, Y) :-  
    connected(X, Z, L),  
    connected(Z, Y, L).
```

Résolution de requête

On substitue en utilisant la première règle :

```
nearby(tottenham_court_road, W).
```

```
% Avec X = tottenham_court_road, Y = W  
connected(tottenham_court_road, W, _).
```

On regarde dans notre base de données :

```
connected(tottenham_court_road, leicester_square, northern).
```

Donc, $W = \text{leicester_square}$ et $_ = \text{northern}$.

Résolution de requête

On fait de même avec la seconde règle :

```
nearby(tottenham_court_road, W).
```

```
% Avec X = tottenham_court_road, Y = W  
connected(tottenham_court_road, Z, L),  
connected(Z, W, L).
```

On a deux objectifs à résoudre, on commence par le premier et on trouve

```
connected(tottenham_court_road, leicester_square, northern).
```

Résolution de requête

Donc, `Z = leicester_square` et `L = northern`

Ensuite, on passe au second objectif qui est devenu
`connected(leicester_square, W, northern).`

Et on trouve

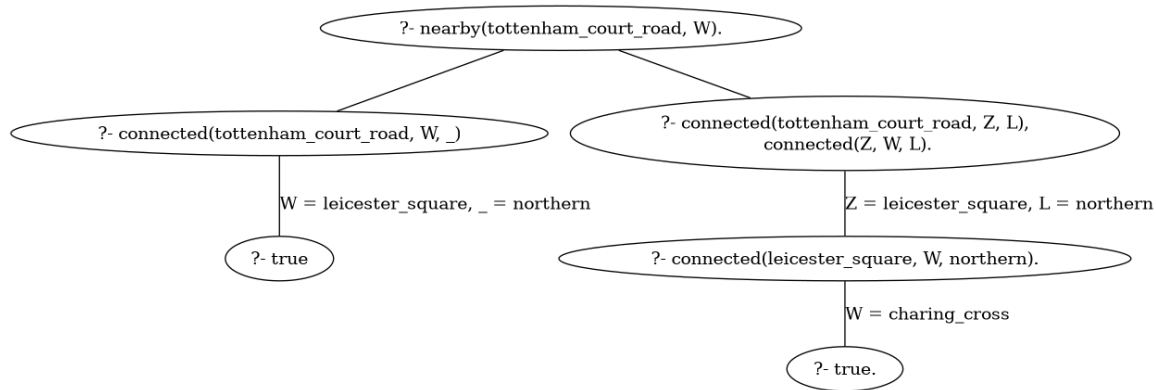
`connected(leicester_square, charing_cross, northern).`

Donc, `W = charing_cross`.

On a alors exploré toutes les solutions possibles.

Arbre de résolution

Résoudre une requête revient à **construire un arbre de résolution**



Construction d'un arbre de résolution

- ① On démarre de la requête
- ② On cherche une règle qui peut s'**unifier** avec le premier élément de la requête
 - la recherche s'effectue dans l'ordre de définition
- ③ Si on unifie avec un **fait** : on a résolu cet élément et on peut l'enlever.
 - si on enlève le dernier élément, la requête devient triviale : true
- ④ Si on unifie avec une **règle** : on continue en remplaçant cet élément par le **corps** de la règle
- ⑤ Après avoir exploré une branche en entier, on a une réponse
 - On peut continuer l'exploration dans d'autres branches (par **backtracking**)

Règle récursive

On peut avoir des règles **récursives** :

```
reachable(X, Y) :-  
    connected(X, Y, _).
```

```
reachable(X, Y) :-  
    connected(X, Z, _),  
    reachable(Z, Y).
```

Règle récursive : exemple

On souhaite montrer

```
reachable(oxford_circus, leicester_square).
```

En utilisant la deuxième clause, avec

- `X = oxford_circus`
- `Y = leicester_square`
- `Z = tottenham_court_road`

on doit montrer que

```
connected(oxford_circus, tottenham_court_road, _),  
reachable(tottenham_court_road, leicester_square).
```

Le `connected(...)` est vrai d'après notre base de données de faits.

Règle récursive : exemple

On doit donc résoudre

```
reachable(tottenham_court_road, leicester_square).
```

En utilisant la première clause avec

- `X = tottenham_court_road`
- `Y = leicester_square`

on doit résoudre

```
connected(tottenham_court_road, leicester_square, _).
```

Ce qui est vrai d'après notre base de données de faits.

Structures de données

Structures de données

Soit le programme Prolog suivant :

```
foo(bar(x)).
```

foo est une relation, bar est une structure.

```
?- foo(X).
```

```
X = bar(x).
```

```
?- bar(X).
```

```
ERROR: Unknown procedure: bar/1
```

Listes

Pour rappel, en OCaml, on pouvait définir

```
type 'a list = Nil | Cons of a * 'a list
```

On peut utiliser la même technique en Prolog :

```
is_a_list(nil).
```

```
is_a_list(cons(_, Y)) :- is_a_list(Y).
```

Attention : le typage de Prolog est dynamique

Listes

Prolog nous donne du sucre syntaxique pour faire définir les listes :

- `[]` est la liste vide
- `[X|Y]` est `cons (::)`

`[a,b,c]` *% correspond à `[a;b;c]` en OCaml*

Ou :

`[a|[b|[c|[]]]]` *% correspond à `a :: b :: c :: []` en OCaml*

Relations sur des listes

```
member(X, [X|_]).  
member(X, [_|R]) :- member(X, R).
```

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

```
?- member(3, X).  
X = [3|_] ;  
X = [_ , 3|_] ;  
X = [_ , _ , 3|_] .
```

Relations sur des listes

OCaml :

```
let rec member (x : 'a) (l : 'a list) : bool =  
  match l with  
  | [] -> false  
  | y :: _ when y = x -> true  
  | _ :: rest -> member x rest
```

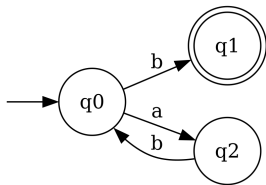
Prolog :

```
member(X, [X|_]).  
member(X, [_|R]) :- member(X, R).
```

Exemple : automate

On peut définir un automate :

```
initial(q0). % état initial de l'automate  
final(q1). % état final ("acceptant") de l'automate  
  
% transitions possibles dans l'automate  
delta(q0, b, q1).  
delta(q0, a, q2).  
delta(q2, b, q0).
```



Exemple : automate

Un automate accepte un chemin Xs (liste de lettres) si :

- en commençant depuis l'état Q
- on peut trouver un chemin (une liste de transitions) jusqu'à un état acceptant

```
accept( $Xs$ ) :-  
    initial( $Q$ ),  
    accept( $Xs$ ,  $Q$ ).
```

On peut définir ce qu'est un chemin acceptant depuis un état Q :

```
accept([],  $Q$ ) :- final( $Q$ ).  
accept([ $X|Xs$ ],  $Q$ ) :-  
    delta( $Q$ ,  $X$ ,  $Q1$ ), % Transition entre Q et Q1 via la lettre X  
    accept( $Xs$ ,  $Q1$ ). % Xs est acceptant depuis Q
```

Attention : on a défini ici deux relations : `accept/1` et `accept/2`

Exemple : automate

On peut vérifier qu'un chemin soit acceptant :

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

Ou générer des chemins :

```
?- accept(Xs).  
Xs = [b] ;  
Xs = [a, b, b] ;  
Xs = [a, b, a, b, b] ;  
Xs = [a, b, a, b, a, b, b] .
```

Bases théoriques

Éléments d'un système logique

There is no universally accepted definition for what logic actually is. (Source)

Il faut trois éléments pour définir un système logique :

- **Syntaxe** : quels termes sont légaux
- **Sémantique** : quel sens donner aux termes
- **Théorie de preuve** : comment dériver de nouveaux termes (règles d'inférences)

Caractéristiques d'un système logique

Deux caractéristiques importantes désirables pour un système logique :

- **Correction** (*soundness*) : ce qu'on peut prouver avec le système est vrai
- **Complétude** (*completeness*) : ce qui est vrai et exprimable dans le système peut être prouvé

Logique du premier ordre (termes et prédicats)

Termes

- Variables : A
- Constantes : a
- $f(t_1, \dots, t_n)$ où f est un symbole de fonction d'arité n et les t_i sont des termes

Exemple

$add(8, mul(X, 3))$ est un terme, où :

- add et mul sont des symboles de fonction d'arité 2
- les nombres 8 et 3 sont des constantes
- X est une variable

Logique du premier ordre (termes et prédicats)

Prédicats

- $P(t_1, \dots, t_n)$ où P est un prédicat d'arité n et les t_i sont des termes

Exemple

$even(add(8, mul(X, 3)))$ est un prédicat d'arité 1.

Logique du premier ordre (formules)

Une **formule** est définie récursivement comme étant

- $P(t_1, \dots, t_n)$ où P est un prédicat d'arité n et les t_i sont des termes
- $\neg F$ (négation)
- $F \wedge F'$ (conjonction)
- $F \vee F'$ (disjonction)
- $F \implies F'$ (implication)
- $\forall X F$ (quantification universelle)
- $\exists X F$ (quantification existentielle)

où F et F' sont des formules et X est une variable.

Exemple

$\forall X \text{ even}(\text{add}(X, 1)) \implies \text{odd}(X)$

Clauses de Horn

Les **clauses de Horn** sont une restriction de la logique de premier ordre aux formules de la forme

$$(P_1 \wedge P_2 \wedge \dots \wedge P_n) \implies P_0$$

- Les P_i sont des prédicats sur des variables universellement quantifiées
- Uniquement des conjonctions à gauche de l'implication
- Un seul prédicat à droite de l'implication

Clauses de Horn et Prolog

Les programmes Prolog sont un ensemble de clauses de Horn

$$(P_1 \wedge P_2 \wedge \dots \wedge P_n) \implies P_0$$

en Prolog :

$P_0 \text{ :- } P_1, P_2, \dots, P_n$

Exemple

```
aime(X, Y) :- ami(X, Y), ami(Y, X), heureux(X), heureux(Y).
```

Clauses de Horn : utilité

On peut résoudre des formules de logique du premier ordre automatiquement, mais on préfère la restriction aux clauses de Horn car :

- Résolution **efficace** (linéaire plutôt qu'exponentiel)
- Même **expressivité** que la logique de premier ordre : on peut traduire la logique de premier ordre en clauses de Horn

(Mais cela peut être NP-complet de traduire une formule de logique de premier ordre en clause de Horn)

Exemple: l'énigme d'Einstein

Énoncé

Il y a cinq maisons de cinq couleurs différentes, alignées le long d'une route. Dans chacune de ces maisons vit une personne de nationalité différente. Chacune de ces personnes boit une boisson différente, fume une marque de cigare différente et a un animal domestique différent.

Question : qui a le poisson rouge ?

Énoncé

Question : qui a le poisson rouge ?

Le Britannique vit dans la maison rouge.

Le Suédois a des chiens.

Le Danois boit du thé.

La maison verte est directement à gauche de la maison blanche.

Le propriétaire de la maison verte boit du café.

La personne qui fume des Pall Mall élève des oiseaux.

Le propriétaire de la maison jaune fume des Dunhill.

La personne qui vit dans la maison du centre boit du lait.

Le Norvégien habite dans la première maison en partant de la gauche.

L'homme qui fume des Blend vit à côté de celui qui a des chats.

L'homme qui a un cheval est le voisin de celui qui fume des Dunhill.

Celui qui fume des Bluemaster boit de la bière.

L'Allemand fume des Prince.

Le Norvégien vit juste à côté de la maison bleue.

L'homme qui fume des Blend a un voisin qui boit de l'eau.

Source

Modélisation en Prolog

On représente une maison par une liste :

```
[couleur, nationalite, boisson, cigare, animal]
```

Et on représente le village par une liste de maisons.

```
gauche(X, Y, L) :- append(_, [X, Y | _], L).
```

```
voisin(X, Y, L) :- gauche(X, Y, L).
```

```
voisin(X, Y, L) :- gauche(Y, X, L).
```

Modélisation en Prolog

```
solve(Houses) :- length(Houses, 5),
    member([rouge,anglais,_,_,_], Houses),
    member([_,suedois,_,_,chien], Houses),
    member([_,danois,the,_,_], Houses),
    gauche([verte,_,_,_,_],[blanche,_,_,_,_], Houses),
    member([verte,_,cafe,_,_], Houses),
    member([_,_,_,pallmall,oiseaux], Houses),
    member([jaune,_,_,dunhill,_], Houses),
    Houses = [_,_,[_,_,lait,_,_,_],_,_],
    Houses = [[_,norvegien,_,_,_,_],_,_,_,_],
    voisin([_,_,_,blend,_],[_,_,_,_,chat], Houses),
    voisin([_,_,_,_,cheval],[_,_,_,dunhill,_], Houses),
    member([_,_,biere,bluemaster,_], Houses),
    member([_,allemand,_,prince,_], Houses),
    voisin([_,norvegien,_,_,_],[bleue,_,_,_,_], Houses),
    voisin([_,_,_,blend,_],[_,_,eau,_,_], Houses),
    member([_,_,_,_,poisson], Houses).
```

Résolution

```
?- [einstein].  
?- solve(H), member([A,B,C,D,poisson], H).  
H = [[jaune, norvegien, eau, dunhill, chat],  
     [bleue, danois, the, blend, cheval],  
     [rouge, anglais, lait, pallmall, oiseaux],  
     [verte, allemand, cafe, prince, poisson],  
     [blanche, suedois, biere, bluemaster|...]],  
A = verte,  
B = allemand,  
C = cafe,  
D = prince.
```

Concepts importants (1)

- Prolog est basé sur le calcul de prédicat
 - fonctions (OCaml) vs. **relations** (Prolog)
 - clauses de Horn
- Un programme est composé de
 - **faits** : vérités inconditionnelles
 - **règles** : vérités conditionnelles
- Une requête se résout
 - en utilisant l'**unification**
 - avec un **arbre de résolution**
 - en recourant au retour sur trace (*backtracking*)

Résolution de programmes

member/2 : définition

```
member(X, [X | _]).  
member(X, [_ | T]) :-  
    member(X, T).
```

Le prédicat `member(X, L)` **met en relation** X avec la liste L **si** X est un élément de la liste L.

member/2 : unification

Soit une requête :

```
?- member(1, [1, 2]).
```

Pour la **résoudre** : on essaye d'**unifier** notre requête avec chaque clause du programme, dans l'ordre de définition.

```
unify(member(X, [X | _]), member(1, [1 | [2]])) % {X: 1}
```

Ici, une unification est possible avec $X = 1$

Comme cette clause n'a pas de prémisses (il s'agit d'un **fait**), la résolution est terminée.

Unification

On peut demander explicitement une unification avec l'opérateur `=/2` :

```
?- member(X, [X | _]) = member(1, [1, 2]).
```

```
X = 1.
```

Le système répond que les termes `member(X, [X | _])` et `member(1, [1, 2])` sont unifiables si on **substitue** X par 1.

Requêtes et validation

Soit une requête :

```
?- member(1, [1, 2]).
```

Le système tente de rendre vraie cette requête en utilisant les clauses disponibles. S'il y parvient, la requête réussit. S'il n'y parvient pas, la requête échoue.

Requêtes et validation

Soit le fait :

```
pere(ike, lea). % Affirme que ike est le père de lea.
```

La requête `?- pere(ike, leo).` échoue car le système ne peut pas établir le fait que ike est le père de leo, faute de preuves dans sa base de connaissances (*hypothèse du monde clos*).

Subtilité importante: si une requête échoue, il est incorrect d'affirmer que sa négation réussit. L'absence de preuve n'équivaut pas nécessairement à la preuve de l'absence.

member/2 : trace

Pour observer la résolution d'un programme, on peut **tracer** son exécution.

Activation du mode de traçage :

```
?- trace.
```

```
true.
```

member/2 : trace

```
[trace] ?- member(1, [1,2]).
```

```
Call: (10) member(1, [1, 2]) ? creep
```

```
Exit: (10) member(1, [1, 2]) ? creep
```

true .

```
[trace] ?- member(10, [1, 2]).
```

```
Call: (10) member(10, [1, 2]) ? creep
```

```
Call: (11) member(10, [2]) ? creep
```

```
Call: (12) member(10, []) ? creep
```

```
Fail: (12) member(10, []) ? creep
```

```
Fail: (11) member(10, [2]) ? creep
```

```
Fail: (10) member(10, [1, 2]) ? creep
```

false.

`member(2, [1,2])` : unification

`?- member(2, [1, 2]).`

On essaye d'unifier notre **requête** avec une clause.

`?- member(X, [X|_]) = member(2, [1,2]).`

false.

`?- member(X, [_|T]) = member(2, [1,2]).`

`X = 2,`

`T = [2].`

`member(2, [1,2])` : résolution

On continue la recherche à partir de la prémisse de la clause :

```
member(X, [_|T]) :-    % conclusion
    member(X, T).      % prémisse
```

Avec $X = 2$ et $T = [2]$, notre requête devient donc :

```
?- member(2, [2]).
```

Que l'on résout par unification avec la première clause.

```
member(2, [1,2]) : trace
```

```
[trace]  ?- member(2, [1,2]).  
    Call: (10) member(2, [1, 2]) ? creep  
    Call: (11) member(2, [2]) ? creep  
    Exit: (11) member(2, [2]) ? creep  
    Exit: (10) member(2, [1, 2]) ? creep  
true .
```

member(3, [1,2]) : résolution

```
?- member(3, [1,2]).
```

```
?- member(X, [X|_]) = member(3, [1|[2]]).  
false.
```

```
?- member(X, [_|T]) = member(3, [1|[2]]).  
X = 3,  
T = [2].
```

On doit donc résoudre :

```
?- member(3, [2]). % prémisse avec X = 3, T = [2]
```

member(3, [1,2]) : résolution

```
?- member(3, [2]).
```

```
?- member(X, [X|_]) = member(3, [2|[]]).  
false.
```

```
?- member(X, [_|T]) = member(3, [2|[]]).  
X = 3,  
T = [].
```

On doit donc résoudre :

```
?- member(3, []). % prémisse avec X = 3, T = []
```

Ce qui réussit avec la première clause ($X = 3$)

`member(3, [1,2])` : résolution

`?- member(X, [X|_]) = member(3, []).`

false.

`?- member(X, [_|T]) = member(3, []).`

false.

On ne sait donc pas résoudre cette requête : on considère le résultat faux.

`member(3, [1,2]) : trace`

```
[trace] ?- member(3, [1,2]).  
  Call: (10) member(3, [1, 2]) ? creep  
  Call: (11) member(3, [2]) ? creep  
  Call: (12) member(3, []) ? creep  
  Fail: (12) member(3, []) ? creep  
  Fail: (11) member(3, [2]) ? creep  
  Fail: (10) member(3, [1, 2]) ? creep  
false.
```

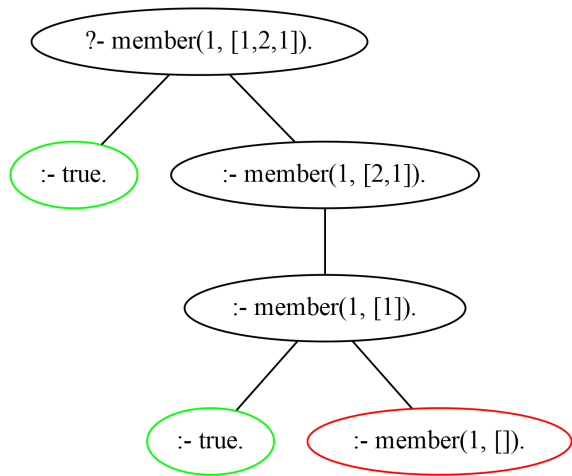
member(1, [1,2,1]) : plusieurs résultats

On peut demander plusieurs résultats à Prolog :

```
[trace] ?- member(1, [1, 2, 1]).  
  Call: (10) member(1, [1, 2, 1]) ? creep  
  Exit: (10) member(1, [1, 2, 1]) ? creep  
true ;  
  Redo: (10) member(1, [1, 2, 1]) ? creep  
  Call: (11) member(1, [2, 1]) ? creep  
  Call: (12) member(1, [1]) ? creep  
  Exit: (12) member(1, [1]) ? creep  
  Exit: (11) member(1, [2, 1]) ? creep  
  Exit: (10) member(1, [1, 2, 1]) ? creep  
true ;  
  Redo: (12) member(1, [1]) ? creep  
  Call: (13) member(1, []) ? creep  
  Fail: (13) member(1, []) ? creep  
  Fail: (12) member(1, [1]) ? creep  
  Fail: (11) member(1, [2, 1]) ? creep  
  Fail: (10) member(1, [1, 2, 1]) ? creep  
false.
```

La ligne **Redo** indique qu'on fait utilisation de retour sur trace (*backtracking*).

member(1, [1,2,1]) : arbre de résolution



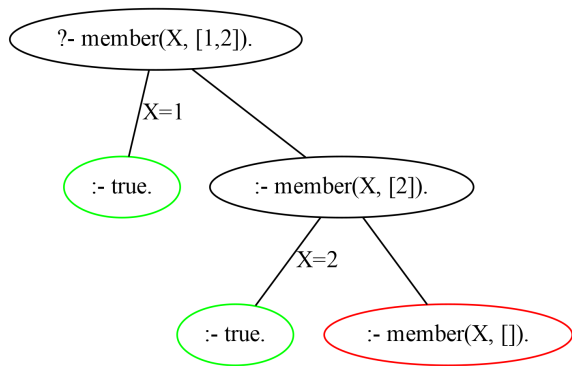
member/2 : requête avec variables logiques

On peut effectuer une requête contenant des variables logiques : « quel élément est membre de la liste [1,2] »

```
[trace] ?- member(X, [1,2]).  
  Call: (10) member(_12902, [1, 2]) ? creep  
  Exit: (10) member(1, [1, 2]) ? creep  
X = 1 ;  
  Redo: (10) member(_12902, [1, 2]) ? creep  
  Call: (11) member(_12902, [2]) ? creep  
  Exit: (11) member(2, [2]) ? creep  
  Exit: (10) member(2, [1, 2]) ? creep  
X = 2 ;  
  Redo: (11) member(_12902, [2]) ? creep  
  Call: (12) member(_12902, []) ? creep  
  Fail: (12) member(_12902, []) ? creep  
  Fail: (11) member(_12902, [2]) ? creep  
  Fail: (10) member(_12902, [1, 2]) ? creep
```

false.

member/2 : arbre de résolution

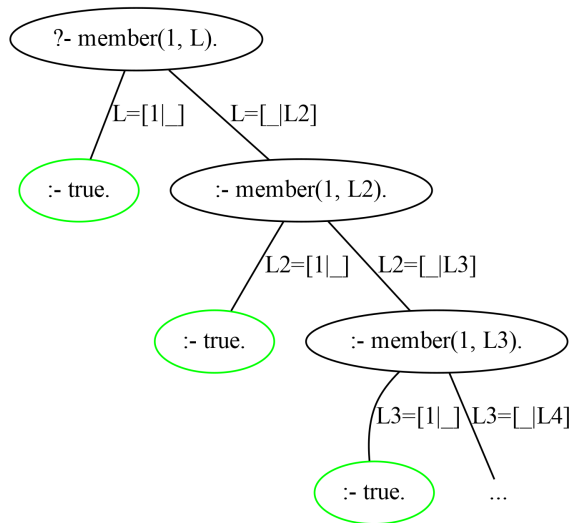


member/2 : requête avec variables logiques

« Quelle liste contient l'élément 1 ? »

```
[trace]  ?- member(1, L).  
    Call: (10) member(1, _26838) ? creep  
    Exit: (10) member(1, [1|_28140]) ? creep  
L = [1|_] ;  
    Redo: (10) member(1, _26838) ? creep  
    Call: (11) member(1, _31294) ? creep  
    Exit: (11) member(1, [1|_32110]) ? creep  
    Exit: (10) member(1, [_31292, 1|_32110]) ? creep  
L = [_ , 1|_] ;  
...
```

member/2 : arbre de résolution



member/2 : requête avec variables logiques

« Quelle liste contient un élément ? »

```
[trace] ?- member(X, L).  
  Call: (10) member(_36122, _36124) ? creep  
  Exit: (10) member(_36122, [_36122|_37456]) ? creep  
L = [X|_] ;  
  Redo: (10) member(_36122, _36124) ? creep  
  Call: (11) member(_36122, _40692) ? creep  
  Exit: (11) member(_36122, [_36122|_41508]) ? creep  
  Exit: (10) member(_36122, [_40690, _36122|_41508]) ? creep  
L = [_ , X|_] ;  
  Redo: (11) member(_36122, _40692) ? creep  
  Call: (12) member(_36122, _45564) ? creep  
  Exit: (12) member(_36122, [_36122|_46380]) ? creep  
  Exit: (11) member(_36122, [_45562, _36122|_46380]) ? creep  
  Exit: (10) member(_36122, [_40690, _45562, _36122|_46380]) ? creep  
L = [_ , _ , X|_] ;  
...
```

head et last

En OCaml :

```
let head = function  
| [] -> failwith "empty list"  
| h :: _ -> h
```

```
let tail = function  
| [] -> failwith "empty list"  
| [x] -> x  
| _ :: r -> last r
```

Comment définir cela en Prolog ?

Fonction vs. relation

Fonction	Relation
$f(a) = b$	$R(a, b)$

head/2 et last/2

OCaml

```
let head = function
| [] -> failwith "empty list"
| h :: _ -> h

let tail = function
| [] -> failwith "empty list"
| [x] -> x
| _ :: r -> last r
```

Prolog

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

last(X, [X]).
last(X, [_ | R]) :- last(X, R).
```

La programmation fonctionnelle et la programmation logique sont deux formes de programmation **déclarative**

head/2 et last/2 : utilisation

```
?- head(X, [1, 2]).
```

```
X = 1.
```

```
?- head(1, L).
```

```
L = [1|_].
```

```
?- head(X, []).
```

```
false.
```

```
?- last(1, L).
```

```
L = [1] ;
```

```
L = [_ , 1] ;
```

```
L = [_ , _ , 1] ;
```

```
L = [_ , _ , _ , 1] .
```

```
?- last(X, [1, 2, 3]).
```

```
X = 3.
```

Unification

On peut demander l'unification directement avec `=/2` :

```
?- X = [1,2].
```

```
X = [1, 2].
```

```
?- [X | _] = [1, 2].
```

```
X = 1.
```

```
?- [X, Y, Z] = [1, Z, Z].
```

```
X = 1,
```

```
Y = Z.
```

```
?- a(b(X), 42) = a(Y, Z).
```

```
Y = b(X),
```

```
Z = 42.
```

append

En OCaml :

```
let rec append l1 l2 =  
  match l1 with  
  | [] -> l2  
  | x :: xs -> x :: (append xs l2)
```

Fonction vs. relation

Fonction	Relation
$f(a) = b$	$R(a, b)$
$f(a, b) = c$	$R(a, b, c)$

append/3

OCaml

```
let rec append l1 l2 =  
  match l1 with  
  | [] -> l2  
  | x :: xs -> x :: (append xs l2)
```

L'appel `append lst1 lst2` **renvoie** la liste obtenue en concaténant `lst1` et `lst2`.

Prolog

```
append([], L, L).  
append([First | Rest], L1, [First | L2]) :- append(Rest, L1, L2).
```

Le prédicat `append(L1, L2, L3)` **met en relation** les trois listes `L1`, `L2` et `L3` si `L3` est la concaténation de `L1` avec `L2`.

append([1, 2], [3], L) : résolution

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

```
append([First | Rest], L1, [First | L2]) :- append(Rest, L1, L2).
```

```
?- append([1, 2], [3], L).
```

Unification :

```
?- append([First | Rest], L1, [First | L2]) = append([1, 2], [3], L).
```

```
First = 1,
```

```
Rest = [2],
```

```
L1 = [3],
```

```
L = [1|L2].
```

`append([1, 2], [3], L) : résolution`

On doit donc résoudre :

`?- append([2], [3], L2)`

append([1, 2], [3], L) : résolution

Le même raisonnement nous permet d'arriver à :

?- append([], [3], L3) % avec $L2 = [2|L3]$

Qui unifie avec append([], L, L), donc :

- $L3 = [3]$
- $L2 = [2|L3] = [2, 3]$
- $L = [1|L2] = [1, 2, 3]$

```
append([1, 2], [3], L) : trace
```

```
?- trace.
```

```
true.
```

```
[trace] ?- append([1, 2], [3], L).
```

```
Call: (10) append([1, 2], [3], _2222) ? creep
```

```
Call: (11) append([2], [3], _23572) ? creep
```

```
Call: (12) append([], [3], _24392) ? creep
```

```
Exit: (12) append([], [3], [3]) ? creep
```

```
Exit: (11) append([2], [3], [2, 3]) ? creep
```

```
Exit: (10) append([1, 2], [3], [1, 2, 3]) ? creep
```

Utiliser append/3 “à l'envers”

« Quelle liste L, à laquelle on ajoute [3] donne [1,2,3] ? »

```
[trace] ?- append(L, [3], [1,2,3]).
```

```
Call: (10) append(_41890, [3], [1, 2, 3]) ? creep
```

```
Call: (11) append(_43264, [3], [2, 3]) ? creep
```

```
Call: (12) append(_44084, [3], [3]) ? creep
```

```
Exit: (12) append([], [3], [3]) ? creep
```

```
Exit: (11) append([2], [3], [2, 3]) ? creep
```

```
Exit: (10) append([1, 2], [3], [1, 2, 3]) ? creep
```

```
L = [1, 2] .
```

append/3 : plusieurs solutions

« Quelles listes L1 et L2 donnent [1,2] quand on les concatène ? »

```
[trace] ?- append(L1, L2, [1,2]).
  Call: (10) append(_82338, _82340, [1, 2]) ? creep
  Exit: (10) append([], [1, 2], [1, 2]) ? creep
L1 = [],
L2 = [1, 2] ;
  Redo: (10) append(_82338, _82340, [1, 2]) ? creep
  Call: (11) append(_87252, _82340, [2]) ? creep
  Exit: (11) append([], [2], [2]) ? creep
  Exit: (10) append([1], [2], [1, 2]) ? creep
L1 = [1],
L2 = [2] ;
  Redo: (11) append(_87252, _82340, [2]) ? creep
  Call: (12) append(_92424, _82340, []) ? creep
  Exit: (12) append([], [], []) ? creep
  Exit: (11) append([2], [], [2]) ? creep
  Exit: (10) append([1, 2], [], [1, 2]) ? creep
L1 = [1, 2],
L2 = [] ;
  Redo: (12) append(_92424, _82340, []) ? creep
  Fail: (12) append(_92424, _82340, []) ? creep
  Fail: (11) append(_87252, _82340, [2]) ? creep
  Fail: (10) append(_82338, _82340, [1, 2]) ? creep
false.
```

prefix/2 et suffix/2

% prefix(P, L) met en relation les listes P et L si P est un préfixe de L
`prefix(P, L) :- append(P, _, L).`

% suffix(S, L) met en relation les listes S et L si S est un suffixe de L.
`suffix(S, L) :- append(_, S, L).`

`?- suffix(X, [4, 5, 6]).`

`X = [4, 5, 6] ;`

`X = [5, 6] ;`

`X = [6] ;`

`X = [] ;`

false.

factor/2

factor(F, L) met en relation les listes F et L si F est un facteur (une sous-liste d'éléments consécutifs) de L.

Idée : F est un facteur de L si F est un préfixe d'un suffixe de L.

```
factor(F, L) :-  
    suffix(S, L),  
    prefix(F, S).
```

```
?- factor(X, [1, 2]).
```

```
X = [] ;
```

```
X = [1] ;
```

```
X = [1, 2] ;
```

```
X = [] ;
```

```
X = [2] ;
```

```
X = [] ;
```

```
false.
```

factor(X, [1,2]) : résolution

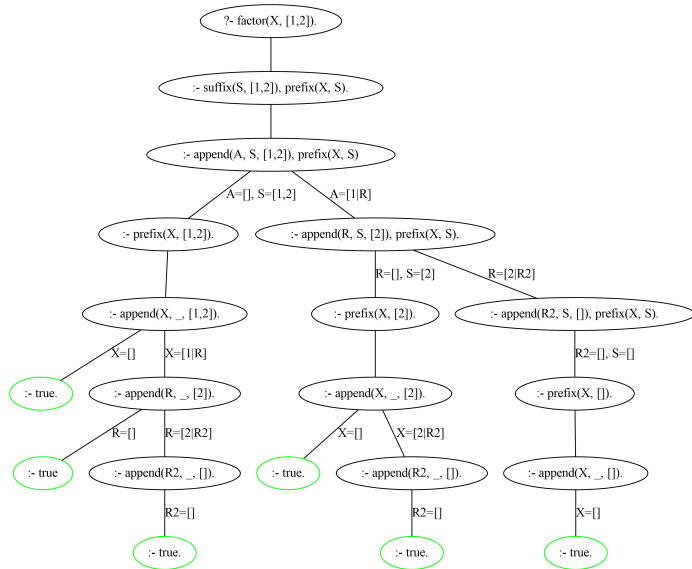
```
append([], L, L).  
append([First | Rest], L1, [First | L2]) :-  
    append(Rest, L1, L2).
```

```
prefix(P, L) :- append(P, _, L).  
suffix(S, L) :- append(_, S, L).
```

```
factor(F, L) :-  
    suffix(S, L),  
    prefix(F, S).
```

```
?- factor(X, [1, 2]).
```

factor(X, [1,2]): résolution



factor/2 — définition alternative

Nous aurions pu définir le prédicat factor par

```
factor(L1, L2) :-  
    prefix(L1, L),  
    suffix(L, L2).
```

Cependant,

```
?- factor(L, [1, 2]).
```

```
L = [] ;
```

```
L = [] ;
```

```
L = [] ;
```

```
L = [1] ;
```

```
L = [2] ;
```

```
L = [1, 2] ;
```

produit une évaluation infinie. Pourquoi ?

sublist/2

sublist(S, L) met en relation les listes S et L si S est une sous-liste de L.

```
sublist([], _).  
sublist([X1 | L1], [X1 | L2]) :- sublist(L1, L2).  
sublist(L1, [_ | L2]) :- sublist(L1, L2).
```

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

```
?- sublist(X, [1, 2, 3]).
```

```
X = [] ;  
X = [1] ;  
X = [1, 2] ;  
X = [1, 2, 3] ;  
X = [1, 2] ;  
X = [1] ;  
X = [1, 3] ;  
X = [1] ;  
X = [] ;  
X = [2] ;  
X = [2, 3] ;  
X = [2] ;  
X = [] ;  
X = [3] ;  
X = [] ;  
false.
```

rev/2 : version naïve

OCaml

rev lst **renvoie** la liste miroir de lst.

```
let rec rev = function
```

```
| [] -> []
```

```
| h :: t -> (rev t) @ [h]
```

Prolog

rev(L1, L2) **met en relation** les listes L1 et L2 si L1 est le miroir de L2.

```
rev([], []).
```

```
rev([H | T], R) :-
```

```
    rev(T, RevT),
```

```
    append(RevT, [H], R).
```

rev/2 : temps d'exécution

```
?- time(rev([1, 2, 3], R)).  
% 11 inferences, 0.000 CPU in 0.000 seconds (78% CPU, 436317 Lips)  
R = [3, 2, 1].
```

rev/2 : trace

```
[trace]  ?- rev([1,2,3], R).  
  Call: (10) rev([1, 2, 3], _23800) ? creep  
  Call: (11) rev([2, 3], _25136) ? creep  
  Call: (12) rev([3], _25948) ? creep  
  Call: (13) rev([], _26760) ? creep  
  Exit: (13) rev([], []) ? creep  
  Call: (13) append([], [3], _25948) ? creep  
  Exit: (13) append([], [3], [3]) ? creep  
  Exit: (12) rev([3], [3]) ? creep  
  Call: (12) append([3], [2], _25136) ? creep  
  Call: (13) append([], [2], _31650) ? creep  
  Exit: (13) append([], [2], [2]) ? creep  
  Exit: (12) append([3], [2], [3, 2]) ? creep  
  Exit: (11) rev([2, 3], [3, 2]) ? creep  
  Call: (11) append([3, 2], [1], _23800) ? creep  
  Call: (12) append([2], [1], _35728) ? creep  
  Call: (13) append([], [1], _36548) ? creep  
  Exit: (13) append([], [1], [1]) ? creep  
  Exit: (12) append([2], [1], [2, 1]) ? creep  
  Exit: (11) append([3, 2], [1], [3, 2, 1]) ? creep  
  Exit: (10) rev([1, 2, 3], [3, 2, 1]) ? creep
```

rev/2 : non terminaison

De manière intéressante, la requête

```
?- rev(L, [])
```

```
L = [] ;
```

produit une résolution infinie. Pourquoi ?

→ Dessiner l'arbre de résolution

rev/2 : version améliorée

OCaml

```
let rev l =  
  let rev_acc acc = function  
    | [] -> acc  
    | h :: t -> rev_acc (h :: acc) t in  
  rev_acc [] l
```

rev/2 : version améliorée

Prolog

`rev_acc(L1, L2, L3)` met en relation les listes `L1`, `L2` et `L3` si `L3` est la concaténation du miroir de `L1` avec `L2`.

```
rev_acc([], A, A).  
rev_acc([H | T], A, R) :-  
    rev_acc(T, [H | A], R).
```

```
rev(L, R) :-  
    rev_acc(L, [], R).
```

rev/2 : nombre d'inférences

```
?- time(rev([1,2,3], R)).  
% 5 inferences, 0.000 CPU in 0.000 seconds (81% CPU, 470633 Lips)  
R = [3, 2, 1].
```

rev/2 : trace

```
[trace] ?- rev([1,2,3], R).  
  Call: (10) rev([1, 2, 3], _18018) ? creep  
  Call: (11) rev_acc([1, 2, 3], [], _18018) ? creep  
  Call: (12) rev_acc([2, 3], [1], _18018) ? creep  
  Call: (13) rev_acc([3], [2, 1], _18018) ? creep  
  Call: (14) rev_acc([], [3, 2, 1], _18018) ? creep  
  Exit: (14) rev_acc([], [3, 2, 1], [3, 2, 1]) ? creep  
  Exit: (13) rev_acc([3], [2, 1], [3, 2, 1]) ? creep  
  Exit: (12) rev_acc([2, 3], [1], [3, 2, 1]) ? creep  
  Exit: (11) rev_acc([1, 2, 3], [], [3, 2, 1]) ? creep  
  Exit: (10) rev([1, 2, 3], [3, 2, 1]) ? creep
```

length/2

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

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

Importance de l'ordre

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

```
?- length(X, 2).  
X = [_, _].
```

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

```
?- length(X, 2).  
ERROR: Arguments are not sufficiently  
instantiated
```

Raisonnement logique

Il existe plusieurs règles de raisonnement logique.

Raisonnement par l'absurde

$$\frac{\neg A \rightarrow \text{faux}}{A}$$

Si $\neg A$ implique une contradiction, **alors** A est vrai

Modus Ponens

$$\frac{A, A \rightarrow B}{B}$$

Si A est vrai **et** A implique B , **alors** B est vrai

Autres règles de raisonnement logique

Modus Tollens

$$\frac{\neg B, A \rightarrow B}{\neg A}$$

Si B est faux **et** A implique B , **alors** A est faux

Tautologie

$$\frac{A}{A} : \text{ Si } A \text{ est vrai, alors } A \text{ est vrai}$$

Contradiction

$$\frac{A, \neg A}{\emptyset} : \text{ Si } A \text{ est vrai et faux, alors une contradiction } (\emptyset) \text{ a lieu}$$

Rappel

$A \rightarrow B$ est équivalent à $\neg A \vee B$

A	B	$A \rightarrow B$	$\neg A \vee B$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

Principe de résolution

Introduit par Robinson (1965), utilisé en programmation logique. C'est une généralisation du *modus ponens*.

Modus ponens :

$$\frac{A, \quad A \rightarrow B}{B}$$

Équivalent à :

$$\frac{A, \quad \neg A \vee B}{B}$$

Principe de résolution : exemple

Modus ponens

$$\frac{A, \neg A \vee B}{B}$$

Règle de résolution

$$\frac{A \vee F_1, \neg A \vee F_2}{F_1 \vee F_2} \quad A \text{ et } \neg A \text{ s'« annulent »}.$$

Cas particuliers

$$\frac{A \vee F, \neg A}{F} \quad \text{et} \quad \frac{A, \neg A \vee F}{F}$$

Preuve par résolution : exemple

On veut prouver ceci :

$$\frac{\begin{array}{c} A \rightarrow (B \rightarrow C) \\ A \rightarrow B \\ A \end{array}}{C}$$

On peut le réécrire sans implication :

$$\frac{\begin{array}{c} \neg A \vee \neg B \vee C \\ \neg A \vee B \\ A \end{array}}{C}$$

Preuve par résolution : exemple

$$\frac{\begin{array}{c} \neg A \vee \neg B \vee C \\ \neg A \vee B \\ A \end{array}}{C}$$

On peut appliquer le principe de résolution pour éliminer les A et $\neg A$

$$\frac{\begin{array}{c} \neg B \vee C \\ B \end{array}}{C}$$

Preuve par résolution : exemple

$$\frac{\neg B \vee C}{B}$$

$$C$$

Principe de résolution avec $\neg B$ et B :

$$\frac{C}{\overline{C}}$$

Ce qui est une tautologie et complète notre preuve.

Résolution de programmes logiques

Programme :

q.

p :- q.

Requête :

?- p.

On veut prouver que

Programme $\rightarrow$ Requête

C'est-à-dire : sur base de notre programme, notre requête est vraie.

Résolution de programmes logiques

Programme $\rightarrow$ Requête est vrai

$\Leftrightarrow \neg(\text{Programme} \rightarrow \text{Requête})$ est faux

$\Leftrightarrow \neg(\neg\text{Programme} \vee \text{Requête})$ est faux

$\Leftrightarrow \text{Programme} \wedge \neg\text{Requête}$ est faux

raisonnement par l'absurde

réécriture de l'implication

négation externe

Si on prouve donc que $\text{Programme} \wedge \neg\text{Requête}$ est **inconsistant**, alors on a prouvé notre requête.

Résolution de programmes logiques

Pour rappel, $p :- q$ signifie

$$p \leftarrow q$$

C'est-à-dire :

$$q \rightarrow p$$

Qu'on réécrit

$$\neg q \vee p$$

Résolution de programmes logiques

Notre programme est donc :

$$\begin{array}{l} q \\ \neg q \vee p \end{array}$$

Et la négation de notre requête est :

$$\neg p$$

Résolution de programmes logiques

$$\frac{\begin{array}{c} q \\ \neg q \vee p \\ \neg p \end{array}}{\emptyset}$$

On a que

- q et $\neg q$ s'annulent par résolution
- p et $\neg p$ s'annulent par résolution

Ce qui prouve que la négation de notre requête rend le tout inconsistant : la requête est donc vraie.

On prouve donc une inconsistance, ce qui s'appelle la preuve par **réfutation**. C'est une preuve par l'absurde qui utilise la résolution.

Résolution : méta-théorie

La résolution par réfutation est

- **correcte** (*sound*) : tout ce qui est prouvé est vrai
- **complète** (*complete*) : tout ce qui est vrai est prouvable

Résolution en Prolog

Prolog utilise la preuve par réfutation au travers de l'algorithme SLD (*Selective Linear Definite clause resolution*).

- SLD supporte l'unification
- il faut décider d'une stratégie d'évaluation
 - Prolog utilise une recherche en profondeur (DFS)

Correction de Prolog

Peut-on prouver quelque chose de faux avec Prolog ?

Quel est le résultat de la requête suivante ?

`?- X = f(X) .`

Correction de Prolog

On ne devrait pas avoir de solution à $X = f(X)$.

Cependant, Prolog résout l'unification :

?- $X = f(X)$.

$X = f(X)$.

Donc, **Prolog n'est pas correct.**

Occurs check

Prolog n'utilise pas d'**occurs check** : un algorithme d'unification correct devrait invalider ce terme car X , la variable unifiée, apparait dans $f(X)$, le terme avec lequel elle est unifiée.

Prolog n'implémente pas l'occurs check par défaut, pour des raisons de performance.

Prolog fournit `unify_with_occurs_check`, qui est correct, mais pas utilisé dans la résolution :

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

Complétion de Prolog

```
% Ordre des clauses inversées  
member(X, [_|R]) :- member(X, R).  
member(X, [X |_]).
```

Quel est le résultat de la requête suivante ?

```
?- member(1, L).
```

Complétion de Prolog

Clairement, il existe plusieurs solutions.

Cependant, la requête ne termine jamais :

```
?- member(1, L).
```

```
^C
```

Pourquoi ? Prolog boucle infiniment dans `member(X, R)` à cause de la stratégie d'exploration DFS.

Donc, **Prolog n'est pas complet** : certains théorèmes vrais ne peuvent pas être prouvés.

Concepts importants (2)

- Prolog utilise la **résolution par réfutation**, avec :
 - **unification**
 - **une stratégie de recherche DFS**
- On peut représenter la recherche de solutions par un **arbre de résolution**
- Prolog n'est :
 - **pas complet** à cause du DFS
 - **pas correct** à cause du manque d'occurs-check
 - pour des raisons d'efficacité

Dessiner des arbres de résolution

Pour dessiner les arbres de résolution, vous pouvez utiliser la bibliothèque [sldnfdraw](#). Référez vous à la documentation pour un exemple d'exécution en local. Un programme d'exemple est donné sur la diapo suivante.

Vous pouvez aussi utiliser l'interface interactive [ici](#)

Dessiner des arbres de résolution

Dans un fichier Prolog, mettre :

```
:- use_module(library(sldnfdraw)).
```

```
:- sldnf.
```

```
:- begin_program.
```

```
% Le programme (i.e., la base de connaissance)
```

```
% composé de faits et règles
```

```
member(X,[X|_T]).
```

```
member(X,[_H|T]):-
```

```
    member(X,T).
```

```
:-end_program.
```

```
:-begin_query.
```

```
% La requête pour laquelle on souhaite obtenir l'arbre de résolution
```

```
member(X,[1,2]).
```

```
:-end_query.
```

Dessiner des arbres de résolution

Ensuite, charger le fichier et exécuter `draw_goal("tree.tex")` :

```
?- [prolog_file].
```

```
?- draw_goal("tree.tex").
```

Vous pouvez ensuite consulter la documentation sur comment compiler le fichier `.tex` pour le visualiser.

Dans le cas de requêtes à profondeur infinie, ou qui prends trop de mémoire on peut limiter la profondeur de recherche Par exemple, ici on limite la profondeur à 2 :

```
?- [prolog_file].
```

```
?- set_depth(2), draw_goal("tree.tex").
```