Mécanisme Prolog

- Principe général :

1. on prend dans le but (clause ne contenant que des littéraux négatifs) le premier littéral négatif L
3. on parcours la base de faits et règles séquentiellement jusqu'à trouver une clause C dont le littéral positif est identique à L. Si on n'en trouve pas, la résolution s'arrête sur un échec
5. on élimine L du but (le but est géré comme une pile de clauses)
7. on ajoute les littéraux négatifs de C au but en unifiant (bien entendu, si C est un fait, on n'ajoute aucun littéral)
9. si le but est la clause vide, on arrête sur un succès, sinon on repart en 1

Exemple

- Base de faits et de règles :

```
homme('roger').
homme('robert').
femme('gertrude').
femme('germaine').

humain(X) :- homme(X)
humain(X) :- femme(X).
```

- On pose la question : **`humain(H)`**?
  - on trouve **`humain(X) :- homme(X)`**, le but devient **`homme(X)`**? en unifiant par la substitution {X/H}
  - on trouve **`homme('roger')`**, le but devient la clause vide en unifiant par {'roger'/X}
  - résultat en remontant les unifications : **`H = 'roger'`**
- Le mécanisme complet est plus compliqué
  - arbre de résolution

Arbre de recherche (1)

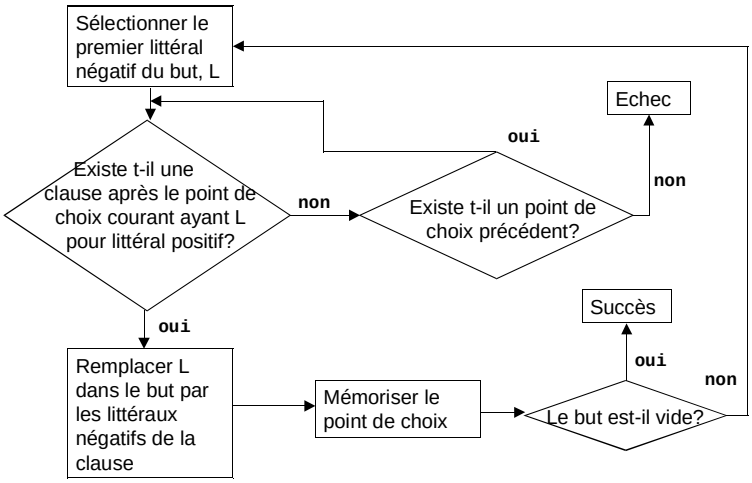
- Base de faits et de règles :

```
femme('gertrude').
femme('germaine').

humain(X) :- homme(X)
humain(X) :- femme(X).
```

- On pose la question : **`humain(H)`**?
  - on trouve **`humain(X) :- homme(X)`**, le but devient **`homme(X)`**?
  - on ne trouve pas de clause contenant homme comme littéral positif => échec
- Il faut pouvoir remonter à la question initiale et utiliser la deuxième clause contenant humain en littéral positif (avec **`femme(X)`**)

Algorithme backtrack Prolog



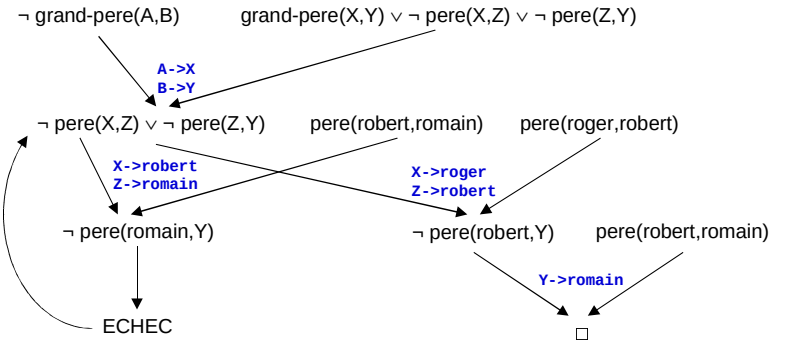
Stratégie de Prolog (1/2)

- Pour résoudre un but, Prolog construit l'arbre de recherche du but et le parcours en profondeur d'abord
  - nœud feuille de succès : c'est une solution, Prolog l'affiche et peut chercher d'autres solutions en remontant au dernier *point de choix*
  - nœud feuille d'échec : remontée dans l'arbre jusqu'à un point de choix possédant des branches non explorées
- Prolog ne trouve pas toujours de solution
  - il faut des faits et règles correspondant au but
  - il faut pouvoir unifier
- *Exemples* :
  - `p(X,X)` et `p(titi,toto)` *ne peuvent être unifiés*
  - `q(X,Y)` et `q(p(T),Y)` *peuvent être unifiés*

Stratégie de Prolog (2/2)

- Si l 'unification échoue : situation d'échec sur la règle considérée pour démontrer la formule.
- Si l'unification réussit : substitution des variables présentes dans la queue de la clause par les valeurs correspondantes des variables de l 'unificateur.
- A la fin, l'ensemble des couples valeur-variable des variables présentes dans la question initiale forme la ***solution*** affichée par Prolog.

Graphe de résolution




```
pere(robert,romain).
pere(roger,robert).

grand_pere(X,Y) :- pere(X,Z),pere(Z,Y).
```

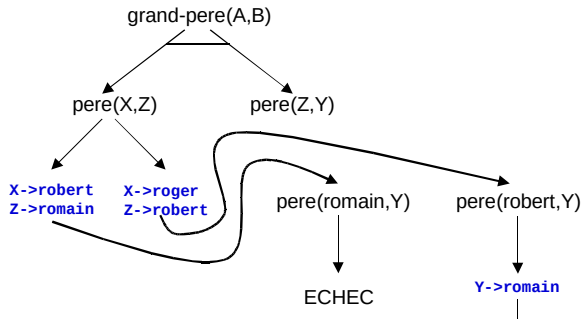
SUCCES

A = X = roger

B = Y = romain

Arbre_Resolution

Graphe ET/OU




```
pere(robert,romain).
pere(roger,robert).

grand_pere(X,Y) :- pere(X,Z),pere(Z,Y).
```

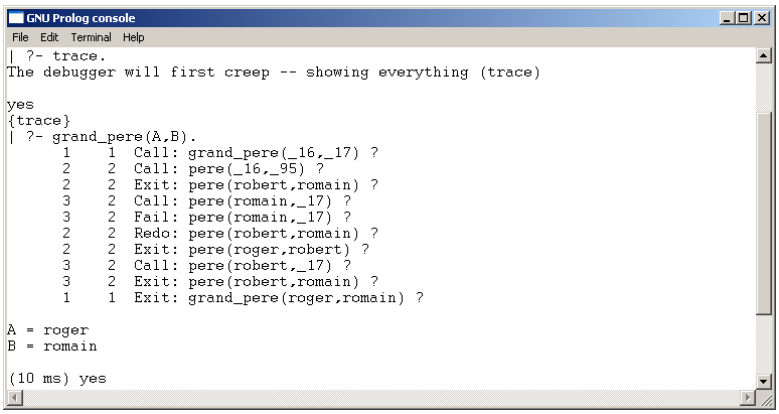
SUCCES

A = X = roger

B = Y = romain

Trace de résolution

- Le prédicat d'arité nulle **`trace`** (resp. **`notrace`**) permet d'afficher (resp. de masquer) la trace de résolution d'un programme Prolog.



Backtrack

- Possibilité de forcer le backtrack en cas de réussite pour trouver une autre solution (; dans l'interpréteur)
- Le résultat peut dépendre de :
  - l'ordre des littéraux dans la queue de clause
  - l'ordre des clauses

- Il peut exister plusieurs solutions à une même requête : non déterminisme de Prolog
- Les prédicats ne sont pas des fonctions : ils "retournent" vrai ou faux, et l'unification permet de trouver des valeurs des variables qui rendent le but vrai

Récursivité en Prolog (1/2)

- Récursivité : un programme est récursif lorsqu'il s'appelle lui même
- En Prolog, les prédicats peuvent "s'appeler" eux-mêmes, c'est-à-dire que le prédicat de tête d'une règle se retrouve dans la queue de la règle
- *Exemple : la factorielle en 2 lignes*


```
factorielle(1,1).
factorielle(N,F) :- N=\=1, X is N-1, factorielle(X,Y), F is Y*N.
```


Récursivité en Prolog (2/2)

- On peut aussi faire de la récursivité croisée entre règles

```
pair(0).
pair(N) :- N=\=0, M is N-1, impair(M).

impair(0) :- fail.
impair(N) :- N=\=0, M is N-1, pair(M).
```

- *Attention à l'ordre des clauses* :
  - sélection d'une clause pour résolution : la première de la liste
  - sélection d'un sous-but : le premier de l'ensemble des buts
  - => la condition d'arrêt de la récursivité doit être en premiére position

Prédicats prédéfinis

- Des dizaines de prédicats sont prédéfinis (built-in predicates) :
  - unification
  - égalité, différence
  - arithmétique
  - analyse des termes
  - gestion des flux d'entrée et de sortie, sockets
  - manipulation de listes
  - méta-programmation (contrôle de la résolution)
  - gestion de l'interpréteur et du système (mémoire, temps d'exécution, ...)
  - ...

- Présentation d'un prédicat :
  - *nom_predicat(mode type_param1, mode type_param2, ...)*
  - <mode> : + (le terme doit être instancié à l'appel), - (le terme doit être une variable qui sera instanciée si le prédicat réussit), ou ?
  - *exemple : **`sort(+list, ?list)`** réussit si la deuxième liste est le résultat du tri de la première liste*

Test sur les types

- Prolog offre la possibilité de tester les types des termes :
  - **`var(?term)`** réussit si le paramètre est non instancié
  - **`nonvar(?term)`** réussit si le paramètre est instancié
  - **`atom(?term)`** réussit si le paramètre est instancié par un atome
  - **`integer(?term)`** réussit si le paramètre est un entier
  - **`float(?term)`** réussit si le paramètre est un réel
  - **`number(?term)`** réussit si le paramètre est un entier ou un réel
  - **`atomic(?term)`** réussit si le paramètre est un nombre ou un atome
  - **`compound(?term)`** réussit si le paramètre est un terme composé (liste non vide ou structure)
  - **`callable(?term)`** réussit si le paramètre est un atome ou un terme composé
  - **`list(?term)`** réussit si le paramètre est une liste
  - ...

Unification de termes

- Prédicat d'unification : =
  - **`X = Y`** (ou **`=(?term, ?term)`**) réussit si X s'unifie avec Y
  - **`X \= Y`** (ou **`\=(?term, ?term)`**) réussit si X ne peut s'unifier avec Y
  - **`unify_with_occurs_check(?term, ?term)`** ne réussit que si aucune variable n'est unifiée avec un terme qui contient cette variable)

- *Exemples :*
  - **`a(B,C) = a(2,3).`** *donne* **`YES {B=2, C=3}`**
  - **`=(a(X,Y,L),a(Y,2,carole)).`** *donne* **`YES {X=2, Y=2, L=carole}`**
  - **`a(X,X,Y) = a(Y,u,v).`** *donne* **`NO`**

- Attention : **=** n'est pas une affectation!

Egalité de termes

- Prédicat d'égalité : ==
  - **`X == Y`** (ou **`==(?term, ?term)`**) réussit si les termes sont identiques (pas simplement unifiables)
  - **`X \== Y`** (ou **`\==(?term,?term)`**) réussit si les termes ne sont pas identiques

- *Exemples :*
  - **`a == a.`** *donne* **`YES`**
  - **`==(A,a).`** *donne* **`NO`**
  - **`2 == 3.`** *donne* **`NO`**
  - **`a(G,7) == a(6,7).`** *donne* **`NO`**

Comparaison de termes

- Les termes Prolog sont totalement ordonnés :
  - les opérateurs @<, @>, @=<, @>= s'appliquent sur tous les termes
  - l'ordre des termes est :
    1) *variables*
    2) *nombres réels*
    3) *nombres entiers*
    4) *atomes dans l'ordre alphabétique*
    5) *termes composés (listes, structures)*

- *Exemples :*
  - `a(F,G) @< ar(T,U).` *donne* **YES**
  - `@<("bla","blabla").` *donne* **YES**
  - `coucou @< couc.` *donne* **NO**
  - `2 @< 5.6.` *donne* **NO**
  - `[a,b] @< [a,b,c].` *donne* **YES**

Arithmétique

- Une expression arithmétique est constituée par
  - des *opérateurs*
  - des *nombres* (constantes ou variables instanciées)

- Les opérateurs sont les opérateurs classiques : **`-, +, *, /, **, //, rem, mod, abs, min, max, sign, sqrt, sin, cos, tan, log, exp, ...`**

- Les opérateurs binaires (sauf **`min`** et **`max`**) peuvent être utilisés en notation préfixée, **`+(2,3)`**, ou infixée, **`2+3`**

- Les opérateurs ne sont pas des prédicats mais des *fonctions*!

Evaluation

- Une expression arithmétique est *évaluée* si elle apparait comme argument d'un prédicat qui porte sur des nombres. L'évaluation est forcée par le prédicat **`is(?var,+evaluable)`**

- *Exemples* :
  - **`R = 2+3.`** *donne* **`YES {R = 2+3}`**
  - **`R is +(2,3).`** *donne* **`YES {R = 5}`**
  - **`is(T,4 mod 3)`** *donne* **`YES {T = 1}`**

- *Attention* : pour qu'une évaluation ait lieu dans une expression où apparaissent des variables, il faut que les variables soient instanciées (par des nombres ou des expressions évaluables) au moment de l'évaluation de l'expression

- *Exemples* :
  - **`R is A + 2.`** *donne une instantiation_error*
  - **`A is 3/4, B is A +1.`** *donne* **`YES {A = 0.75, B = 1.75}`**

Priorité des opérateurs

- La priorité des opérateurs est fixée par un entier

```
1200      xfx     :- -->
1200       fx     :-  ?-
1100      xfy     ;
1050      xfy     ->
1000      xfy     ,
 900       fy     \+
 700      xfx     =  \=
 700      xfx     ==  \==  @<  @=<  @>  @>=
 700      xfx     =..
 700      xfx     is  =:=  =\=  <  =<  >  >=
 500      yfx     +  -  /\  \/
 400      yfx     *  /  //  rem  mod  <<  >>
 200      xfx     **
 200      xfy     ^
 200       fy     -  \
 100      xfx     @
  50      xfx     :
```

Comparaison des nombres

- *Comparaisons de nombres* : <, =<, >= et >
  - utilisés avec une syntaxe de prédicat ou d'opérateurs
  - réussissent si les arguments sont des nombres (évaluation) et s'ils respectent l'ordre indiqué
- *Exemples* :
  - `<(2,5.6).` *donne* **YES**
  - `5.8 >= 5.8.` *donne* **YES**
- *Egalité de nombres* : =:= et =\=
  - utilisés avec une syntaxe de prédicat ou d'opérateurs
  - réussissent si les arguments sont des nombres (évaluation) et s'ils respectent l'égalité ou l'inégalité
- *Exemples* :
  - `2 =:= 3.` *donne* **NO**
  - `3.5 =:= 3.5.` *donne* **YES**

Exemple: calcul du PGCD

- On veut écrire un prédicat qui calcule le PGCD de deux entiers : **`pgcd(+evaluable, +evaluable, -term)`**
- *Algorithme d'Euclide* : calcul du PGCD de x et y
  - si x = y, le pgcd est x
  - si x < y, le pgcd est le pgcd de x et y - x
  - si y < x, le pgcd est le pgcd de y et x

```
pgcd(X,X,X).

pgcd(X,Y,D) :- X < Y, Z is Y-X, pgcd(X,Z,D).

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

Définition d'opérateurs

- Possibilité de définir de nouveaux opérateurs via le prédicat **`op(+integer,+operator_specifier,+atom)`**
  - Le premier argument est la priorité de l'opérateur (entre 0, le plus prioritaire et 1200)
  - Le deuxième argument spécifie le type de l'opérateur
    - **`fx`** : opérateur préfixé non associatif
    - **`fy`** : opérateur préfixé associatif à droite
    - **`xf`** : opérateur postfixé non associatif
    - **`yf`** : opérateur postfixé associatif à gauche
    - **`yfx`** : opérateur infixé associatif à gauche
    - **`xfy`** : opérateur infixé associatif à droite
    - **`xfx`** : opérateur infixé non associatif
  - Le troisième argument spécifie le nom de l'opérateur

- *Exemples* :
  - *Les appels* **`op(200, yfx, ou).`** *et* **`op(100, yfx, et).`** *définissent les opérateurs* et *et* ou