Intelligence artificielle

ENSIIE

Semestre 4 – 2020/21

Programmation logique [2]

2. Transparents de Catherine Dubois

PROgrammation LOGique

1972 : création de Prolog par A. Colmerauer et P. Roussel à Luminy
conçu pour implanter un système de communication homme-machine, en langage naturel (français)

L'age d'or (1982-1992) : projet de recherche « Ordinateurs de 5ème génération » au Japon.

La Programmation par Contraintes (depuis 1987) : extension de Prolog. Applications industrielles et intérêt académique (conférence internationale CP)

Un grand nombre d'implémentations disponibles. Voir par exemple : `http://fr.wikipedia.org/wiki/Prolog`
La version que nous allons utiliser : GNU prolog et/ou SWI Prolog

GNU-Prolog

- GNU-Prolog est développé à l'INRIA et est libre (`http://gnu-prolog.inria.fr/`)
- GNU-Prolog offre un compilateur en deux parties :
  - compilation en byte-code pour exécution sur la machine virtuelle incluse (Warren Abstract Machine)
  - compilation en code natif (exécution plus rapide)
- GNU-Prolog offre pour l'exécution
  - un interpréteur qui travaille avec le byte-code
  - la possibilité de générer un exécutable pour application stand-alone

Programme Prolog = des faits pour commencer

On exprime des relations entre les données (constantes, variables)

`homme(_)`, `femme(_)`, `humain(_)`, `enfantDe(_,_)` : des relations, encore appelées prédicats Prolog

arité : nombre d'arguments.
On note aussi dans la communauté Prolog `homme/1`, `enfantDe/2` pour indiquer l'arité.

```
homme(jean).
homme(pierre).
femme(marie).
femme(julie).
```

```
enfantDe(pierre, marie).
enfantDe(julie, marie).
enfantDe(marie, jean).
```

On pose des questions

```
?- enfantDe(pierre, marie).

yes

?- enfantDe(pierre, jean).

no

?- enfantDe(jeanne, marie).

no
```

On peut mettre une variable dans une question.

```
?- enfantDe(X, jean).

X = marie
yes
```

Par quelle constante peut-on remplacer `X` pour que l'assertion devienne vraie ?

En fait la question est un prédicat existentiel qu'il faut valider (en proposant une valeur pour la variable existentiel) ou invalider .

La question `enfantDe(X, jean)` signifie
$\exists X.\ enfantDe(X, jean)$

X=marie est appelée une substitution

```
| ?- enfantDe(X,marie).
```

Ici plusieurs substitutions résultats sont possibles

```
X = pierre
X = julie
```

Un peu de vocabulaire

Terme : c'est une donnée manipulée par Prolog

- une variable (ex. `X`) est un terme,
  chaîne alpha-numérique commençant par une majuscule
  Attention : Une variable Prolog s'apparente plus à une variable mathématique qu'à une variable informatique.
- un terme atomique est un terme
  En Prolog : 3 sortes de termes atomiques :
  - les nombres : entiers ou flottants,
  - les identificateurs (parfois appelés atomes) : un identificateur est une chaîne alpha-numérique commençant par une minuscule
    (ex. `marie`, `julie`, `aX12`),
  - les chaînes de caractères (ex. `"Marie"`)

- On peut imbriquer les termes : termes composés
  Les termes composés représentent les objets composés (structurés) de l'univers.
  Syntaxiquement : `foncteur`$(t_1, \ldots, t_n)$ où `foncteur` est une chaîne alpha-numérique commençant par une minuscule, et t_1, ..., t_n sont des termes.
  Le nombre d'arguments n est appelé arité du terme.
  - Le cours de logique a lieu le 25 mai 2020 en salle 101 :
    avec `date/3`, `cours/3` :
    `cours(logique, date(25, 2, 2020), salle(101)).`
  - listes nil : `[]`,
    cons : `[x|l]`
  - entiers de Peano : `zero`, `s/1`

- prédicat ou relation ou atome logique : relation entre des termes.

syntaxiquement de la forme
`symbole-de-prédicat(`t_1**`, ..., `**t_n **`)`** avec t_1, t_2, t_n des termes.

arité d'un prédicat : nb d'arguments.

Par exemple `enfantDe` : prédicat d'arité 2.

- Un fait = un atome logique = clause (élémentaire)

Exemple : `femme(marie).`

Une question peut être composée : conjonction de questions élémentaires (buts)

Une question peut être composée : conjonction de questions élémentaires (buts)

```
| ?- enfantDe(pierre,marie), femme(marie).

yes

| ?- enfantDe(pierre,X), enfantDe(marie,Y).

X = marie Y = jean
yes
```

Plus intéressant quand les buts partagent des variables : 2 fois la même variable dans une question : contrainte d'égalité

```
| ?- enfantDe(Fille, marie), femme(Fille).
```

.... Qui est la fille de marie ?
On cherche une substitution pour Fille qui rend les 2 assertions simultanément vraies
Fille = julie est la seule substitution possible.

Ajoutons des règles

Ajoutons une règle pour généraliser la question concernant la recherche de la mère d'un individu :

Si E est un enfant de M et que M est une femme alors M est la mère de E.

On introduit un nouveau prédicat mere/2 et on écrit la règle suivante :

```
mere(M, E) :- enfantDe(E, M) , femme(M).
```

la virgule se lit « et », le :- se lit « si ».

Elle signifie en logique :

$$\forall E.\ \forall M.\ enfantDe(E, M) \wedge femme(M) \Rightarrow mere(M, E).$$

Dans une règle, les variables sont quantifiées universellement.

On appelle cette règle également une clause Prolog.

Toute règle est de la forme :

```
A0 :- A1 , ... , An.
```

où `A0, A1 , ... , An` sont des atomes logiques.

Une telle règle signifie que la relation `A0` est vraie si les relations `A1` et ... et `An` sont vraies.
`A0` est appelé tête de clause et `A1, ... , An` est appelé corps de clause.

Autres règles :

- Notion de père :

```
pere(P, E) :- enfantDe(E, P), homme(P).
```

Remarque : la variable E n'a rien à voir avec la variable nommée E dans la règle définissant mere

Autres règles :

• Notion de père :

```
pere(P, E) :- enfantDe(E, P), homme(P).
```

Remarque : la variable E n'a rien à voir avec la variable nommée E dans la règle définissant mere

• Un petit-enfant d'une personne est un enfant d'un enfant de cette personne.

```
petitEnfantDe(X,Z) :- enfantDe(X,Y), enfantDe(Y,Z).
```

- Définissons ce qu'est un parent : avec deux règles

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).
```

Ces deux règles définissent le prédicat parent.

Disjonction de 2 clauses :
Rien d'autre que l'application de l'équivalence logique :
$(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \vee (B \Rightarrow C)$

- Définissons ce qu'est un parent : avec deux règles

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).
```

Ces deux règles définissent le prédicat parent.

Disjonction de 2 clauses :
Rien d'autre que l'application de l'équivalence logique :
$(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \vee (B \Rightarrow C)$

- Une règle peut être récursive :

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

Un programme est un ensemble de clauses non vide.

Clause de Prolog = clause de Horn

Une clause de Horn est une clause -disjonctive- (disjonction de littéraux) dont au plus 1 littéral est positif.
Un littéral est une formule atomique (positif) ou la négation d'une formule atomique (négatif)

$$\begin{aligned} & \neg A_1 \vee \cdots \vee \neg A_n \vee A_0 \\ \equiv & (A_1 \wedge \cdots \wedge A_n) \Rightarrow A_0 \\ \equiv & A_0 \ \text{:-} \ A_1, \ldots, A_n \end{aligned}$$

Posons des questions à nouveau

```
| ?- mere(marie, pierre).

yes

| ?- mere(X, julie).

X = marie
yes

| ?- mere(X,pierre), mere(X, julie).

X = marie
yes
```

```
?-  petitEnfantDe(jean, marie).

no

| ?- petitEnfantDe(julie, X).

X = jean
yes

| ?- petitEnfantDe(X, jean).

X = pierre ? ;
X = julie ? ;
no
```

```
| ?- ancetre(X, marie).

X = jean ? ;
no

| ?- ancetre(X, julie).

X = marie ? ;
X = jean ? ;
no
```

Autre exemple : Concaténation de deux listes

On définit le prédicat app/3 tel que app(l1,l2,l3) signifie « l3 est la concaténation des listes l1 et l2 (dans cet ordre) ».

```
app([], L, L).
app([X|R],L, [X|M]) :- app(R,L,M).
```

Remarque : La première clause est un fait mais à la différence des précédents, il comporte une variable (universellement quantifiée).

Pour calculer la concaténation de deux listes données, on posera une question avec une variable en 3ème argument.

```
| ?- app([a, b], [c, d], L).
L= [a, b, c, d]
yes
```

On peut aussi obtenir toutes les listes L1 et L2 telles que leur concaténation donne la liste [a, b, c, d] :

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

L1 = [a]
L2 = [b,c,d]

L1 = [a,b]
L2 = [c,d]

L1 = [a,b,c]
L2 = [d]

L1 = [a,b,c,d]
L2 = []
```

Ou encore ...

```
| ?- app([a,b], L, [a, b, c, d]).
L = [c,d]

| ?- app([a,b], L, [b, c, d]).
no
```

Interprète abstrait : vision logique

- Soit σ une substitution (ensemble de couples variable $\mapsto$ terme)
  Appliquer σ sur la question : remplacer les variables par les termes indiqués dans σ : on obtient une instance de la question
  exemple : soit $\sigma = \{\texttt{X} \mapsto \texttt{marie}\}$, Prolog l'écrit
  `X = marie`
  L'instance de `ancetre(X, julie)` obtenue avec σ est : `ancetre(marie, julie).`
- Instance d'une règle : idem : on remplace les variables de la règle par le terme correspondant dans la substitution.
  C'est la règle logique de l'élimination du $\forall$: règle de l'instanciation : si la formule $\forall X.\phi$ est valide alors toutes les instances de la forme $\phi[x \leftarrow t]$ sont valides.

- Répondre à une question = **trouver** (imaginer) une substitution des variables de la question telle que l'instance obtenue découle des règles et faits du programme.
- Découler du programme ? ? ?

  entrée : question Q instanciée, P programme
  résultat : `yes` si on peut trouver une preuve de Q à partir du programme P (= enchaînement de règles instanciées), `no` sinon

Algorithme

```
R := {Q};
tant que R n'est pas vide faire
    choisir un but G dans R
    et une règle de P
    A :- B1, B2 .. Bk (avec k>=0) telle que
      il existe σ tq σA = σG
    si une telle instance n'existe pas, exit
    R := σ(R - {G}) U {σB1, σB2, .. σBk}
fintq
si R = {} alors resultat := yes
else resultat := no
```

rq : ici un fait est une règle sans corps. $R =$ ensemble des résolvantes.

Déroulons l'algorithme sur la question `petitEnfantDe(X, jean)` avec le programme des relations familiales précédent.

Déroulons l'algorithme sur la question `petitEnfantDe(X, jean)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(X, jean)` }

Déroulons l'algorithme sur la question `petitEnfantDe(X, jean)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(X, jean)`}
  G = `petitEnfantDe(X, jean)`
  $\sigma = \{Z \mapsto$ `jean`$\}$
  Instance de la règle qui définit `petitEnfantDe` :

```
petitEnfantDe(X, jean) :-
   enfantDe(X, Y), enfantDe(Y, jean).
```

Déroulons l'algorithme sur la question `petitEnfantDe(X, jean)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(X, jean)`}
  G = `petitEnfantDe(X, jean)`
  $\sigma = \{Z \mapsto$ `jean`$\}$
  Instance de la règle qui définit `petitEnfantDe` :

```
petitEnfantDe(X, jean) :-
   enfantDe(X, Y), enfantDe(Y, jean).
```

- R = { `enfantDe(X, Y)`, `enfantDe(Y, jean)` }
  G = `enfantDe(Y,jean)`
  $\sigma = \{Y \mapsto$ `marie`$\}$
  On prend comme règle le fait `enfantDe(marie,jean)`.

- R = { `enfantDe(X, marie)` }
  G = `enfantDe(X, marie)`
  $\sigma = \{X \mapsto$ `julie`$\}$
  On prend comme règle le fait `enfantDe(julie,marie)`.

- R = { `enfantDe(X, marie)` }
  G = `enfantDe(X, marie)`
  $\sigma = \{X \mapsto$ `julie`$\}$
  On prend comme règle le fait `enfantDe(julie,marie)`.
- R = $\emptyset$ résultat = `yes`

Déroulons l'algorithme sur la question `petitEnfantDe(marie, Y)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(marie, Y)` }

Déroulons l'algorithme sur la question `petitEnfantDe(marie, Y)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(marie, Y)`}
  G = `petitEnfantDe(marie, Y)`
  $\sigma = \{X \mapsto$ `marie`$\}$
  Instance de la règle qui définit `petitEnfantDe` :

```
petitEnfantDe(marie, Z) :-
   enfantDe(marie, Y), enfantDe(Y, Z).
```

Déroulons l'algorithme sur la question `petitEnfantDe(marie, Y)` avec le programme des relations familiales précédent.

- R = { `petitEnfantDe(marie, Y)` }
  G = `petitEnfantDe(marie, Y)`
  $\sigma = \{X \mapsto$ `marie`$\}$
  Instance de la règle qui définit `petitEnfantDe` :

```
petitEnfantDe(marie, Z) :-
   enfantDe(marie, Y), enfantDe(Y, Z).
```

- R = { `enfantDe(marie, Y)`, `enfantDe(Y, Z)` }
  G = `enfantDe(marie, Y)`
  $\sigma = \{Y \mapsto$ `jean`$\}$
  On prend comme règle le fait `enfantDe(marie,jean).`

- R = { `enfantDe(jean, Z)` }
  il n'y a aucune substitution et aucune règle qui puisse correspondre à `enfantDe(jean, Z)`
  résultat = `no`

Exercice : dérouler l'algorithme sur la question :
`ancetre(X, julie)`

Unification

Comment trouver l'instance du but et les instances des règles ?

- On essaie de faire correspondre le but G avec la tête d'une des règles A
- Pour cela, on peut avoir besoin d'instancier les variables du but et de la règle

On parle d'unifier G et A, c'est-à-dire trouver une substitution σ telle que $\sigma G = \sigma A$

Exemples d'unification

- unifier $f(X, a)$ et $f(b, Y)$, on remplace X par b et Y par a :
  on obtient le terme $f(b, a)$ (unificande)
- unifier $f(X, X)$ et $f(a, b)$:
  a et b sont deux constantes différentes : échec
- unifier $f(X, Y)$ et $h(a, b)$:
  f et h sont deux symboles de fonction différents : échec

- unifier $f(X, Y)$ et $f(Z, X)$:
  on peut remplacer X,Y et Z par $g(f(a, b))$,
  mais on peut aussi remplacer X et Y par Z.
  $\Rightarrow$ pas de solution unique, on préférera la plus générale
  i.e. la moins spécialisée (most general unifier, mgu)

- unifier $f(X, Y)$ et $f(g(X), Y)$:
  pb de circularité : remplacer X par $g(X)$! ! ! ! On
  obtiendrait un unificande infini $f(g(g(g(.....))), Y)$ échec

Unification : filtrage (pattern-matching à la OCaml) dans les deux sens

Unification

- Il existe plusieurs unificateurs si deux termes sont unifiables :
  Exemple : $t_1 = f(X, Y)$ $t_2 = f(Y, X)$
  $\sigma_1 = \{X \mapsto a, Y \mapsto a\}$ est un unificateur pour t_1 et t_2.
  $\sigma_2 = \{X \mapsto f(a, g(b)), Y \mapsto f(a, g(b))\}$ est aussi un unificateur pour t_1 et t_2.
  $\sigma = \{X \mapsto Y\}$ est encore un unificateur pour t_1 et $t_2 =$
  c'est celui-ci que l'on cherche : la plus petite substitution ou la plus générale $=$ mgu $=$ most general unifier
- Tout autre unificateur de t_1 et t_2 se déduit du mgu

- Plus généralement, l'algorithme d'unification permet de résoudre un système d'équations E entre termes en transformant E en un système de la forme $\{X_1 \approx u_1, \ldots X_k \approx u_k\}$, tel que les variables X_i soient deux à deux distinctes et tel que $\{X_1, \ldots, X_k\} \cap \bigcup_{i \in 1..k} Var(u_i) = \emptyset$.
- Décider si deux termes t_1 et t_2 sont unifiables revient à résoudre l'équation $t_1 \approx t_2$.

Unification

Algo présenté sous forme de règles de transformation d'un ensemble d'équations en un autre : système de récriture

Fin de l'algo quand aucune règle ne peut plus s'appliquer ou sur échec.

Si pas échec le résultat est la substitution recherchée : c'est le mgu.

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1, \ldots, t_n \approx u_n\} \cup E$$

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1, \ldots, t_n \approx u_n\} \cup E$$

$$\{f(t_1, \ldots, t_n) \approx g(u_1, \ldots, u_k)\} \cup E \overset{conflit}{\rightarrow} echec$$

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1, \ldots, t_n \approx u_n\} \cup E$$

$$\{f(t_1, \ldots, t_n) \approx g(u_1, \ldots, u_k)\} \cup E \overset{conflit}{\rightarrow} echec$$

$$\{X \approx t\} \cup E \overset{elimination}{\rightarrow} \{X \approx t\} \cup E[x \leftarrow t]$$

si $X \in Vars(E)$ et $X \notin Vars(t)$

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1, \ldots, t_n \approx u_n\} \cup E$$

$$\{f(t_1, \ldots, t_n) \approx g(u_1, \ldots, u_k)\} \cup E \overset{conflit}{\rightarrow} echec$$

$$\{X \approx t\} \cup E \overset{elimination}{\rightarrow} \{X \approx t\} \cup E[x \leftarrow t] \quad \text{si } X \in Vars(E) \text{ et } X \notin Vars(t)$$

$$\{X \approx t\} \cup E \overset{occurrence}{\rightarrow} echec \quad \text{si } X \in Vars(t) \text{ et } X \neq t$$

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1, \ldots, t_n \approx u_n\} \cup E$$

$$\{f(t_1, \ldots, t_n) \approx g(u_1, \ldots, u_k)\} \cup E \overset{conflit}{\rightarrow} echec$$

$$\{X \approx t\} \cup E \overset{elimination}{\rightarrow} \{X \approx t\} \cup E[x \leftarrow t] \quad \text{si } X \in Vars(E) \text{ et } X \notin Vars(t)$$

$$\{X \approx t\} \cup E \overset{occurrence}{\rightarrow} echec \quad \text{si } X \in Vars(t) \text{ et } X \neq t$$

$$\{t \approx t\} \cup E \overset{effacement}{\rightarrow} E$$

Règles de récriture :

Dans les règles, X désigne une variable, t_i (et u_i) un terme quelconque - variable ou non

$$\{f(t_1,\ldots,t_n) \approx f(u_1,\ldots,u_n)\} \cup E \overset{decomposition}{\rightarrow} \{t_1 \approx u_1,\ldots,t_n \approx u_n\} \cup E$$

$$\{f(t_1,\ldots,t_n) \approx g(u_1,\ldots,u_k)\} \cup E \overset{conflit}{\rightarrow} echec$$

$$\{X \approx t\} \cup E \overset{elimination}{\rightarrow} \{X \approx t\} \cup E[x \leftarrow t] \quad \text{si } X \in Vars(E) \text{ et } X \notin Vars(t)$$

$$\{X \approx t\} \cup E \overset{occurrence}{\rightarrow} echec \quad \text{si } X \in Vars(t) \text{ et } X \neq t$$

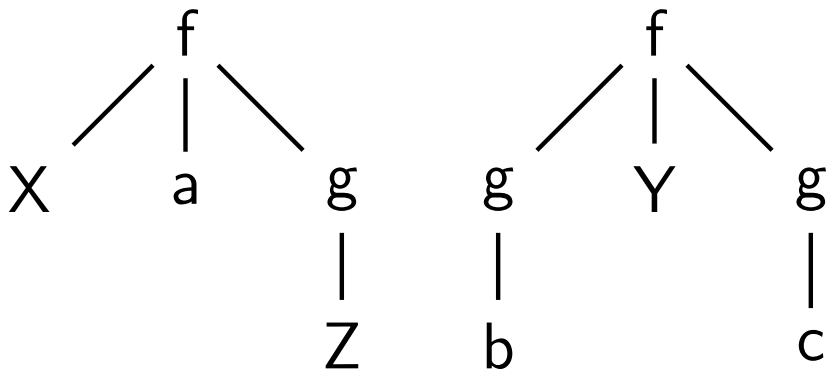
$$\{t \approx t\} \cup E \overset{effacement}{\rightarrow} E$$

$$\{t \approx X\} \cup E \overset{inversion}{\rightarrow} \{X \approx t\} \cup E \quad \text{si } t \text{ non variable}$$

Quelques remarques sur l'algo d'unification :

- L'algorithme (dans cette présentation) est non déterministe : il n'explique pas quelle équation piocher
- L'algorithme termine : sur un échec ou sur une substitution
- Le système de réécriture est confluent.
- La régle d'élimination fait disparaître une variable (il ne reste qu'une seule équation qui parle de la variable éliminée)
- Il existe des algorithmes déterministes et efficaces.

Unification : f(X,a,g(Z)) $\approx$ f(g(b),Y,g(c)) ?



$$\{\mathbf{f(X,a,g(Z))} \approx \mathbf{f(g(b),Y,g(c))}\}) \overset{decomposition}{\rightarrow}$$
$$\{X \approx g(b), \mathbf{a} \approx \mathbf{Y}, g(Z) \approx g(c))\} \overset{inversion}{\rightarrow}$$
$$\{X \approx g(b), Y \approx a, \mathbf{g(Z)} \approx \mathbf{g(c)})\} \overset{decomposition}{\rightarrow}$$
$$\{X \approx g(b), Y \approx a, Z \approx c)\} = \sigma$$

Unification : f(X,Y,X) $\approx$ f(a, X, Y) ?

$$\{\mathbf{f(X,Y,X)} \approx \mathbf{f(a,\ X,\ Y)}\}) \overset{decomposition}{\rightarrow}$$
$$\{\mathbf{X} \approx \mathbf{a}, Y \approx X, X \approx Y\} \overset{elimination}{\rightarrow}$$
$$\{X \approx a, \mathbf{Y} \approx \mathbf{a}, a \approx Y\} \overset{elimination}{\rightarrow}$$
$$\{X \approx a, Y \approx a, \mathbf{a} \approx \mathbf{a}\} \overset{effacement}{\rightarrow}$$
$$\{X \approx a, Y \approx a\}$$

L'unificande est alors f(a,a,a)

Unification : f(X,g(X),a) $\approx$ f(Z, Z, a) ?

$\{$**f(X,g(X),a) $\approx$ f(Z, Z, a)**$\}) \overset{decomposition}{\rightarrow}$
$\{$**X $\approx$ Z**$, g(X) \approx Z, a \approx a\} \overset{elimination}{\rightarrow}$
$\{X \approx Z,$ **g(Z) $\approx$ Z**$, a \approx a\} \overset{inversion}{\rightarrow}$
$\{X \approx Z,$ **Z $\approx$ g(Z)**$, a \approx a\} \overset{occurrence}{\rightarrow}$
échec

Les deux termes ne s'unifient pas.

Unification :

`app([b],[c,d],L)` $\approx$ `app([X|Xs], Ys, [X|Zs])` ?

Unification :

`app([b],[c,d],L)` $\approx$ `app([X|Xs], Ys, [X|Zs])` ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$`app([b],[c,d],L)` $\approx$ `app([X|Xs], Ys, [X|Zs])` $\}$

Unification :

`app([b],[c,d],L)` $\approx$ `app([X|Xs], Ys, [X|Zs])` ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$`app([b],[c,d],L)` $\approx$ `app([X|Xs], Ys, [X|Zs])` $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ `[b]` $\approx$ `[X|Xs]`, `[c,d]` $\approx$ `Ys`, `L` $\approx$ `[X|Zs]` $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{elimination}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{elimination}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, Xs $\approx$ [], [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{elimination}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, Xs $\approx$ [], [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, Xs $\approx$ [], Ys $\approx$ [c,d], L $\approx$ [b|Zs] $\}$

Unification :

app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) ?

Remarque : ici app est un symbole de relation. La règle de décomposition précédente s'étend aux prédicats.

$\{$app([b],[c,d],L) $\approx$ app([X|Xs], Ys, [X|Zs]) $\}$ $\overset{decomposition}{\rightarrow}$

$\{$ [b] $\approx$ [X|Xs], [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{decomposition}{\rightarrow}$

$\{$b $\approx$ X, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [X|Zs] $\}$ $\overset{elimination}{\rightarrow}$

$\{$X $\approx$ b, [] $\approx$ Xs, [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, Xs $\approx$ [], [c,d] $\approx$ Ys, L $\approx$ [b|Zs] $\}$ $\overset{inversion}{\rightarrow}$

$\{$X $\approx$ b, Xs $\approx$ [], Ys $\approx$ [c,d], L $\approx$ [b|Zs] $\} = \sigma$

σ(app([b],[c,d],L))= app([b],[c,d],[b|Zs]) =

σ(app([X|Xs], Ys, [X|Zs])

L'unification est un ingrédient du moteur de résolution de Prolog mais on dispose aussi de prédicats prédéfinis pour utiliser cette opération dans les programmes prolog.

- prédicat =

```
=(?term, ?term)          = is a predefined infix operator

Term1 = Term2 unifies Term1 and Term2. No occurs check is
done, i.e. this predicate does not check if a variable is
unified  with a compound term containing this
variable (this can lead to an infinite loop).
```

Exemples :

```
| ?- X=f(Y), X=a.          no

| ?- X=[a,X].              cannot display cyclic term for X
                           yes
```

- on peut forcer le test d'occurrence : prédicat unify_with_occurs_check

```
unify_with_occurs_check(?term, ?term)

unify_with_occurs_check(Term1, Term2) unifies
Term1 and Term2. The occurs check test is done
(i.e. the unification fails if a variable is
unified with a compound term containing this
variable).
```

Exemple :

```
| ?- unify_with_occurs_check(X, [a, X]).    no
```

- prédicat \=

```
\=(?term, ?term)    \= is a predefined infix operator

Term1 \= Term2 succeeds if Term1 and Term2
are not unifiable (no occurs check is done).
```

Exemple :

```
| ?- X=f(Y), X\=a.          X = f(Y)
                            yes
```

Backtracking

```
R := {Q};
tant que R n'est pas vide faire
    choisir un but G dans R
    et une règle de P
    A :- B1, B2 .. Bk (avec k>=0) telle que
      il existe σ tq σA = σG
    si une telle instance n'existe pas, exit
    R := σ(R - {G}) U {σB1, σB2, .. σBk}
fintq
si R = {} alors resultat := yes
else resultat := no
```

L'algorithme est non-détermiste dans le choix de la règle

En cas d'échec, ou pour chercher d'autres solutions, on doit revenir en arrière pour faire un autre choix $\rightsquigarrow$ **backtrack**

Ordre d'évaluation

Les buts dans `R` sont choisis de gauche à droite

- Pour voir si une règle est applicable, on essaie de satisfaire les prédicats de gauche à droite

Les règles sont choisies dans l'ordre où elles ont été déclarées

- L'ordre des déclarations a un impact sur la recherche de solution.

Quelques règles générales quand on écrit un programme Prolog :

- quand on utilise des définitions récursives, le cas de base doit précéder le cas inductif.
- les faits connus pour un prédicat précèdent généralement les règles définies sur ce prédicat.
- quand on écrit un programme Prolog, il faut tenir compte de l'évaluation de ce programme par Prolog (en particulier du fait que les clauses sont examinées dans l'ordre du programme).

Attention ce ne sont que des heuristiques ...

Prédicats arithmétiques

- Le terme `1 + 2` représente `+(1, 2)` en notation infixe et non le nombre 3
- `1+2` `ne` s'unifie pas avec 3

```
| ?- X=1+3, X=4.

false
```

- Il existe des prédicats spéciaux pour évaluer des expressions arithmétiques
- Ainsi `X` **`is`** `1 + 2` évalue `1 + 2` et donne à `X` la valeur 3 (sorte d'affectation)
- Plus généralement `X` **`is`** `Y` évalue `Y` (pas `X`) et le résultat est unifié avec `X`

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

 ?- 3 is 1+2.
true

 ?- 4 is 1+2.
false

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

 ?- Y = 2, X is Y + 1.
X = 3  Y = 2

 ?- X is Y + 1, Y = 2.
ERROR: Arguments are not sufficiently instantiate
```

Comparaisons arithmétiques

- `X =:= Y` : `X` et `Y` sont évalués.
  Ce prédicat est vrai si les valeurs de `X` et de `Y` sont égales.
  Ne pas confondre avec `X = Y` qui unifie `X` et `Y`
  ni avec `X == Y` qui teste si `X` et `Y` sont instanciés par des termes identiques

```
| ?- X = 3, Y = 2, X + 1 =:= Y * 2.
                                    X = 3 Y = 2

| ?- a == a.
                                    true

| ?- f(X,a) == f(b, a).
                                    false

| ?- f(X,a) = f(b, a).
                                    X = b

| ?- Y = 2, X is Y + 1, X \= Y.
                                    X = 3 Y = 2
```

- `X =\= Y` vrai si les valeurs de `X` et `Y` sont différentes
- `X < Y` si la valeur de `X` est $<$ à la valeur de `Y`
- Idem pour `X > Y`, `X <= Y` et `X >= Y`

Attention ! tous ces prédicats ne peuvent fonctionner que si toutes les variables ont été instanciées

Exemple : factorielle

```
fact(0, 1).
fact(N, P) :-
  N > 0, N1 is N - 1,
  fact(N1, P1), P is P1 * N.

| ?- fact(4,X).
X = 24
```

Exemple : factorielle

```
fact(0, 1).
fact(N, P) :-
  N > 0, N1 is N - 1,
  fact(N1, P1), P is P1 * N.

| ?- fact(4,X).
X = 24

| ?- fact(X, 24).
ERROR: Arguments are not sufficiently instanti
```

Attention, certains prédicats -arithmétiques- ne peuvent être utilisés qu'avec certains *modes*. Ici `fact` ne peut s'utiliser correctement que si le 1er argument est instantié (on dit en *mode input*).

Coupure

- Il existe un prédicat qui permet de contrôler l'espace de recherche des solutions : coupure (cut) souvent notée **!**.
- La coupure réussit toujours.
- On peut placer le cut dans une clause Prolog (règle ou fait) ou dans une requête.
- Attention, ce prédicat affecte la résolution.
  Il permet de couper des branches, dynamiquement, dans l'arbre de résolution.
- Il peut servir à améliorer l'efficacité mais peut aussi supprimer des solutions

$\Longrightarrow$ Attention, usage délicat et à limiter.
souvent incompatible avec les prédicats fonctionnant en mode génération.

Coupure dans une question

Soit la question : `?- C, !, D.` avec C= c1,c2,...,cn et D=d1,d2,...,dk

- Prolog résout d'abord les buts de C, en utilisant le mécanisme de backtracking si besoin,
- Dès qu'une solution est trouvée pour C, la coupure est résolue et coupe tous les choix pendants de la résolution de C,
- D est ensuite résolu,
- Si l'un des buts de D échoue ou si D réussit, le backtracking sur les buts de D est lancé,
- Quand aucun backtracking n'est plus possible sur les buts d1, d2,..., dk, la coupure empêche le backtracking dans les buts de C. On ne peut plus trouver de solutions.

```
poisson(truite).
viande(steak).
```

```
poisson(daurade).
viande(escalope).
```

```
plat(P) :- poisson(P).
plat(V) :- viande(V).
```

plat(P),viande(P) fournit en réponse tous les plats qui sont de la viande.

```
poisson(truite).
viande(steak).

poisson(daurade).
viande(escalope).

plat(P) :- poisson(P).
plat(V) :- viande(V).
```

plat(P),viande(P), ! fournit en réponse le premier plat qui est de la viande

```
poisson(truite).
viande(steak).
```

```
poisson(daurade).
viande(escalope).
```

```
plat(P) :- poisson(P).
plat(V) :- viande(V).
```

plat(P), ! , viande(P) ne fournit aucune réponse

Contrôle des solutions dans un processus de génération

Comme nous l'avons vu ci-dessus, le cut permet d'affiner une question.

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

quels sont les repas comprenant du poisson ?

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

idem

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

Contrôle des solutions dans un processus de génération

Comme nous l'avons vu ci-dessus, le cut permet d'affiner une question.

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

quels sont les repas comprenant du poisson ?

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

idem

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

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

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

Contrôle des solutions dans un processus de génération

Comme nous l'avons vu ci-dessus, le cut permet d'affiner une question.

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

quels sont les repas comprenant du poisson ?

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

idem

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

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

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

construire un repas avec le premier poisson de la base

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

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

quel est le premier repas contenant du poisson ?

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

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

quel est le premier repas contenant du poisson ?

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

idem

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

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

quel est le premier repas contenant du poisson ?

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

idem

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

Même chose que sans cut

De façon générale :

```
?- q(X1,X2,...,Xn)
```

Quel est l'ensemble des toutes substitutions solutions de la question Q ?

```
?- q(X1,X2,...,Xn) ,!
```

Quelle est la première substitution solution de la question Q ? (première, ici, au sens de la sémantique procédurale de Prolog)

Coupure dans une clause

Même sémantique que précédemment avec deux règles supplémentaires

- La coupure coupe les choix relatifs aux alternatives de la clause en cours :

```
(1) C :- C1,!,C2.
(2) C :- C3.
```

Si C1 est résolu, le ! coupe l'alternative restante pour C (2), ainsi bien sûr que tous les autres choix possibles pour la résolution de C1.
Si C1 n'est pas résolu, le ! n'est pas atteint, le backtracking est possible sur (2).

- La coupure n'affecte pas les résolutions englobant la clause le contenant

```
C :- C1,C2.
C1 :- D1.
C1 :- D2.
C2 :- C3,!,C4.
C2 :- D3.
```

la coupure empêche un backtracking sur C3 et la 2ème règle de C2 mais pas sur C1

Maximum de 2 entiers

- Sans cut

```
max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X < Y.
```

Si le 1er test réussit, l'autre échoue forcément : inutile d'y aller (exclusion mutuelle des deux clauses)
Si le 1er test échoue, le 2ème réussit (exhaustivité)

```
max3(X,Y,X) :- X>=Y.
max3(X,Y,Y).
```

Ceci est problématique :

```
max3(2,1,Z).
Z = 2 ? ;      Z = 1
```

- Avec cut :

```
max1(X,Y,X) :- X>=Y, !.
max1(X,Y,Y) :- X < Y.

max2(X,Y,X) :- X>=Y, !.
max2(X,Y,Y) .
```

Mimer un if then else

Il est possible de mimer le "if-then-else" des langages de programmation classiques
Pour cela, on peut définir une expression de la forme :

```
S :- P, !, Q.
S :- R
```

En GNU-prolog, Swi Prolog (et d'autres) :

```
S :- (P -> Q ; R)
```

```
max4(X,Y,Z) :-
    (  X =< Y
    -> Z = Y
    ;  Z = X
     ).
```

Quelques exemples

Appartenance à une liste (à une seule solution)

```
elt(X,[X|_]) :- !.
elt(X,[_|L]) :- elt(X,L).

mem(X,[X|_]) .
mem(X,[_|L]) :- mem(X,L).

?- elt(X, [1,2,3]).
X = 1

?- mem(X, [1,2,3]).
X = 1 ; X = 2 ; X = 3 ; false
```

```
?- elt(2, L).
L = [2 | _16282].

?- mem(2, L).
L = [2 | _16] ; L = [_17, 2 | _18] ;
L = [_19, _20, 2 | _21]

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

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

Ajouter un élément sans duplication

```
ajouter(X, L, L) :- elt(X, L), !.
ajouter(X, L, [X|L]).
```

Négation par l'échec

`fail` : prédicat prédéfini qui produit un échec = c'est le prédicat toujours faux.
Il permet de modéliser une phrase comme Marie aime tous les animaux sauf les souris.

```
souris(jerry).
animal(tom).
animal(X) :- souris(X).
aime(marie,X) :- souris(X) , ! , fail .
aime(marie,X) :- animal(X) .
```

L'ordre des clauses importe et la coupure est indispensable.

```
?- aime(marie, jerry).
false.
```

```
?- aime(marie, tom).
true
```

Définir le prédicat different/2 qui exprime que X et Y ne peuvent s'unifier.

Définir le prédicat different/2 qui exprime que X et Y ne peuvent s'unifier.

```
different(X,X) :- ! , fail .
different(X,Y) .
```

Ceci veut dire que l'on échoue quand on essaie de déduire que deux termes qui peuvent s'unifier sont différents.

```
| ?- different(1,2).
true
| ?- different(1,1).
false
| ?- different(f(X), f(Y)).
false
| ?- different(f(X), f(X)).
false
| ?- different(2, 1+1).
true
```

négation (negation as failure) :
La façon standard de représenter le not est :

```
not(X) :- X, !, fail.
not(X).
```

Attention, cette négation n'est pas une négation logique standard,
En Gnu Prolog et Swi Prolog : `(\+)/1`
`\+(member(Y,T))` réussira si Y n'est pas un élément de T.

```
aime(marie,X) :- animal(X) , \+(souris(X)) .
```

Attention `not p(x)` ne veut pas dire « p(X) faux » mais « on échoue quand on essaie de déduire p(X) ». ces deux propriétés ne sont équivalentes que dans le cas d'un monde fermé.

```
r(a) .
q(b) .
p(X) :- not r(X) .
```

- Question `q(X), p(X)` donne X=b.

  résolution de q(X) donne X = b.
  puis on doit résoudre p(b) ie not r(b).
  La premiere clause définissant not demande de montrer r(b) d'où échec.
- Question `p(X), q(X)` échoue

  On résout p(X) soit not r(X).
  La premiere clause définissant not donne r(X), !, fail.
  r(X) donne X=a on continue et fail échoue. Cette fois pas de retour arrière à cause du ! que l'on a traversé.
  La deuxième clause définissant not n'est pas essayée.
  D'où no.

Pour tout

On veut vérifier que dès que le prédicat P est vrai, on a C qui est vrai aussi.

On peut définir

```
forall(P, C) :- \+ (P, \+ C).
```

Si on a un but `forall(P,C)`, on va essayer de résoudre `P, \+ C` et échouer si c'est possible.
On résoud `P`, puis on essaie de résoudre `C` avec la substitution obtenue. Si ce n'est pas possible, `\+ C` réussit, donc on échoue : la substitution obtenue au final vérifie `P` mais pas `C`.
Si on peut résoudre `C`, alors on backtracke et on cherche une autre solution pour `P`. Une fois toutes les solutions de `P` testées, on sait qu'on peut rendre `C` vrai à chaque fois.

Exemple de pour tout

```
subset(L1, L2) :- forall(mem(X, L1), mem(X, L2)).
```

Pour toutes les substitutions de `X` qui résolvent `mem(X,L1)` on va vérifier que `mem(X,L2)`.

```
?- subset([1,3,6],[6,2,4,1,3,7]).
true.

?- subset([1,3,6],[6,2,4,1,7]).
false.
```

Attention aux instances partielles

`forall` ne fait pas de substitutions

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

(Pour tous les `X` dans `[1,2,3]`, on peut bien trouver une substitution pour `L` telle que `mem(X,L)`.)

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

Boucle infinie : on teste toutes les solutions de `mem(X,L)` :

```
L = [X| Q] ;
L = [Y1, X| Q] ;
L = [Y1, Y2, X| Q] ;
L = [Y1, Y2, Y3, X| Q] ;
...
```