

Université de SOUK AHRAS

Faculté des sciences et technologie

Département de Mathématiques et Informatique

Programmation Logique

Enseignant: Arous Mokdad

Niveau: 3^{eme} Année Informatique

2016/2017



◆ La négation par échec en Prolog

1. Négation en Prolog

- ◆ La coupure élague l'arbre de résolution et par la suite le temps de calcul est réduit.
- ◆ En pratique, cette utilisation est même plus importante pour sauvegarder de l'espace mémoire.
- ◆ Pour un calcul est déterministe, il est moins nécessaire de garder de l'information pour traiter d'éventuels retours en arrière.
- ◆ L'ajout de la coupure n'altère pas la sémantique déclarative des programmes.
- ◆ On peut aussi se rendre compte du fonctionnement de la coupure en l'utilisant directement dans une question.

1. Négation en Prolog

- ◆ La coupure peut aussi être utilisée pour exprimer de l'information négative.
- ◆ C'est la construction de base pour implanter une forme limitée de la négation en Prolog qu'on appelle la **négation par échec**.
- ◆ Le prédicat suivant, **non/1**, est la définition standard de la négation.
- ◆ Le but **non(But)** sera vrai si **But** échoue c'est-à-dire non démontrable.
- ◆ **non(P) :- si P est vrai alors renvoyer faux ...** **Comment coder une telle propriété?**

1. Négation en Prolog

- ◆ On utilise le prédicat **fail** qui est toujours faux.
- ◆ Si **P** est vrai alors faux, sinon vrai.

1. Négation en Prolog

- ◆ On utilise le prédicat **fail** qui est toujours faux.

non(P) :- P , fail .

non(P).

- ◆ Si **P** est vrai alors faux, sinon vrai.
- ◆ Le problème ici, c'est que si **P** est vrai, **non(P)** vaut, simultanément **vrai** et **faux**.
- ◆ On doit donc utiliser

1. Négation en Prolog

- ◆ On utilise le prédicat **fail** qui est toujours faux.

```
non(P) :- P , fail .  
non(P).
```

- ◆ Le problème ici, c'est que si **P** est vrai, **non(P)** vaut, simultanément **vrai** et **faux**.
- ◆ On doit donc utiliser la coupure :

```
non(P) :- P , ! , fail.  
non(P).
```

1. Négation en Prolog

```
non(P) :— P, !, fail.  
non(P).
```

- ◆ On effectue ce que l'on appelle une «**négation par l'échec**».
- ◆ La coupure oblige la première règle à échouer et interdit le *backtracking* :
- ◆ Pour démontrer **non(P)**, Prolog essaie de démontrer **P**,
- ◆ S'il réussit, un coupe-choix élimine les points de choix éventuellement créés durant cette démonstration puis échoue.

1. Négation en Prolog

- ◆ Pour exprimer la négation en Prolog, on utilise le prédicat prédéfini **not/1** défini par l'échec comme suit:

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

- ◆ La conjonction de la coupure (!) et de **fail** est appelée une combinaison ***coupure-échec***.

1. Négation en Prolog

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

- ◆ Dans le cas de la question :
?- non(But).
- ◆ La première clause s'applique:
 - Si **But** réussit, la construction coupure-échec est rencontrée. Le calcul se termine alors sur la première clause et le but demandé échoue.
 - Si **But** échoue, alors la seconde clause du programme est utilisée et elle réussit.

1. Négation en Prolog

- ◆ La construction coupure-échec est utilisée pour implanter d'autres prédicats qui comprennent la négation.

Exemple:

- ◆ Le prédicat **différent/2**, qui vérifie l'inégalité entre 2 termes, peut être programmé simplement en utilisant l'unification et la coupure-échec :

1. Négation en Prolog

- ♦ La construction coupure-échec est utilisée pour implanter d'autres prédicats qui comprennent la négation.

Exemple:

- ♦ Le prédicat différent/2, qui vérifie l'inégalité entre 2 termes, peut être programmé simplement en utilisant l'unification et la coupure-échec :

```
différent(X, Y) :- X = Y, ! , fail.  
différent(X, Y).
```

1. Négation en Prolog

Expliquez les résultats suivants :

?- X is 1+1.

?- not(X is 1+1).

?- not(not(X is 1+1)).

1. Négation en Prolog

Expliquez les résultats suivants :

?- X is 1+1.

X = 2.

?- not(X is 1+1).

false.

?- not(not(X is 1+1)).

true.

1. Négation en Prolog

- ◆ La négation par l'échec ne doit être utilisée que sur des prédicats dont les arguments sont déterminés et à des fins de vérification.
- ◆ Son utilisation ne détermine jamais la valeur d'une variable.
- ◆ Elle est utilisée pour vérifier qu'une formule n'est pas vraie.

1. Négation en Prolog

- ◆ Considérez le programme suivant :

`p(a).`

- ◆ Et interrogez Prolog avec :

`?- X = b , not(p(X)).`

1. Négation en Prolog

- ◆ Considérez le programme suivant :

$p(a).$

- ◆ Et interrogez Prolog avec :

$?- X = b, \text{not}(p(X)).$

True

$\% \{X = b\}$

- ◆ Prolog répond positivement car $p(b)$ n'est pas démontrable.

1. Négation en Prolog

- ◆ Par contre, interrogez le avec :

?- not(p(X)) , X=b.

1. Négation en Prolog

- ◆ Par contre, interrogez le avec :

?- not(p(X)) , X=b.

false

- ◆ Prolog répond négativement car $p(X)$ étant démontrable avec $\{X = a\}$, not $p(X)$ est considéré comme faux
- ◆ => Incohérence qui peut être corrigée si on applique la règle vue précédemment :
n'utiliser la négation que sur des prédicats dont les arguments sont déterminés.

1. Négation en Prolog

Exemple :

?- not(X=1) , X=2.

?- X=2, not(X=1).

1. Négation en Prolog

Exemple :

?- not(X=1) , X=2.

False

?- X=2 , not(X=1).

X = 2

- ◆ Il faut voir que Prolog tente de rendre les expressions vraies.
- ◆ Alors dans le premier cas, il lie **X** à **1**, pour rendre la première partie de la première clause vraie, le *cut* interdit un *backtrack* sur **X**, et c'est l'échec.

2. Hypothèse du monde Clos

- ◆ Pour Prolog, $\text{not}(F)$ signifie que la formule F **n'est pas démontrable**, et non que c'est une formule **fausse**.
- ◆ C'est ce que l'on appelle **l'hypothèse du monde clos**.
- ◆ Dans l'hypothèse du monde fermé ou clos, seul les faits positifs sont enregistrés dans la base de données;
- ◆ les **faits non enregistrés** dans une extension de prédicat **sont supposés faux**.
- ◆ Dans l'hypothèse du monde ouvert, un fait non enregistré est inconnu.