# La Coupure (The Cut)

- Le prédicat cut!/0 offre un moyen de contrôler le backtracking.
- Le backtracking est une caractéristique de Prolog.
- Le cut est un prédicat Prolog, donc on peut l'ajouter aux règles dans le corps.
- Le cut est un but qui réussit toujours.

```
p(X) :- b(X),c(X),!,d(X),e(X).
```

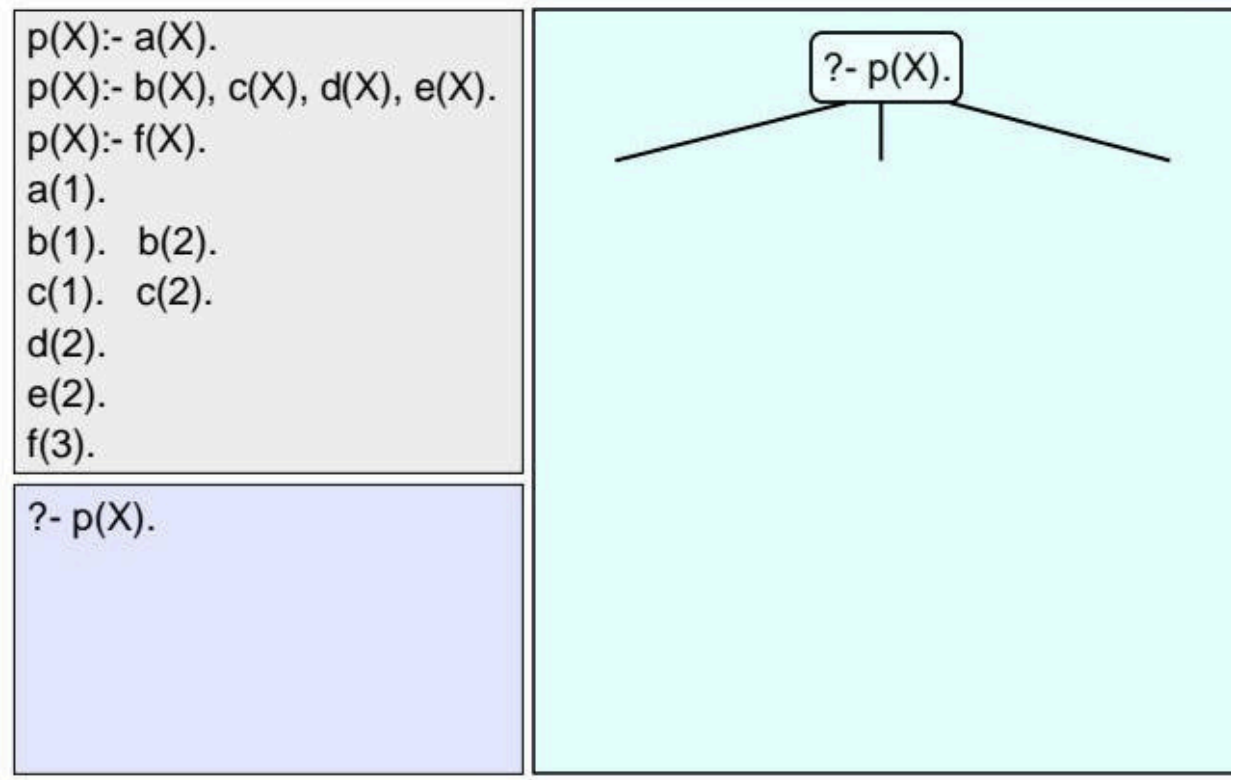
# Expliquer la Coupure

Pour expliquer la coupure, nous allons :

1. Examiner un morceau de code Prolog sans coupure et voir ce qu'il fait en termes de backtracking.
2. Ajouter des coupures et regarder comment les coupures affectent.

# Exemple : Code sans Coupure

```
p(X) :- a(X).
p(X) :- b(X), c(X), d(X), e(X).
p(X) :- f(X).
a(1).
b(1).
b(2).
c(1).
c(2).
d(2).
e(2).
f(3).

?- p(X).
```




L'image ci-dessus illustre un arbre de recherche Prolog sans coupure. On peut voir que Prolog explore toutes les possibilités.

Requête :

```
?- p(X).
```

Résultats :

```
X=1;
X=1;
X=2;
X=3;
no
```

Cet exemple montre que Prolog explore toutes les règles possibles pour satisfaire la requête p(X), même après avoir trouvé une solution. Sans coupure, le système essaie toutes les règles définies pour p(X) afin de trouver toutes les solutions possibles.

# Ajouter une Coupure

Supposons que nous insérons une coupure dans la deuxième clause :

```
p(X) :- b(X), c(X), !, d(X), e(X).
```

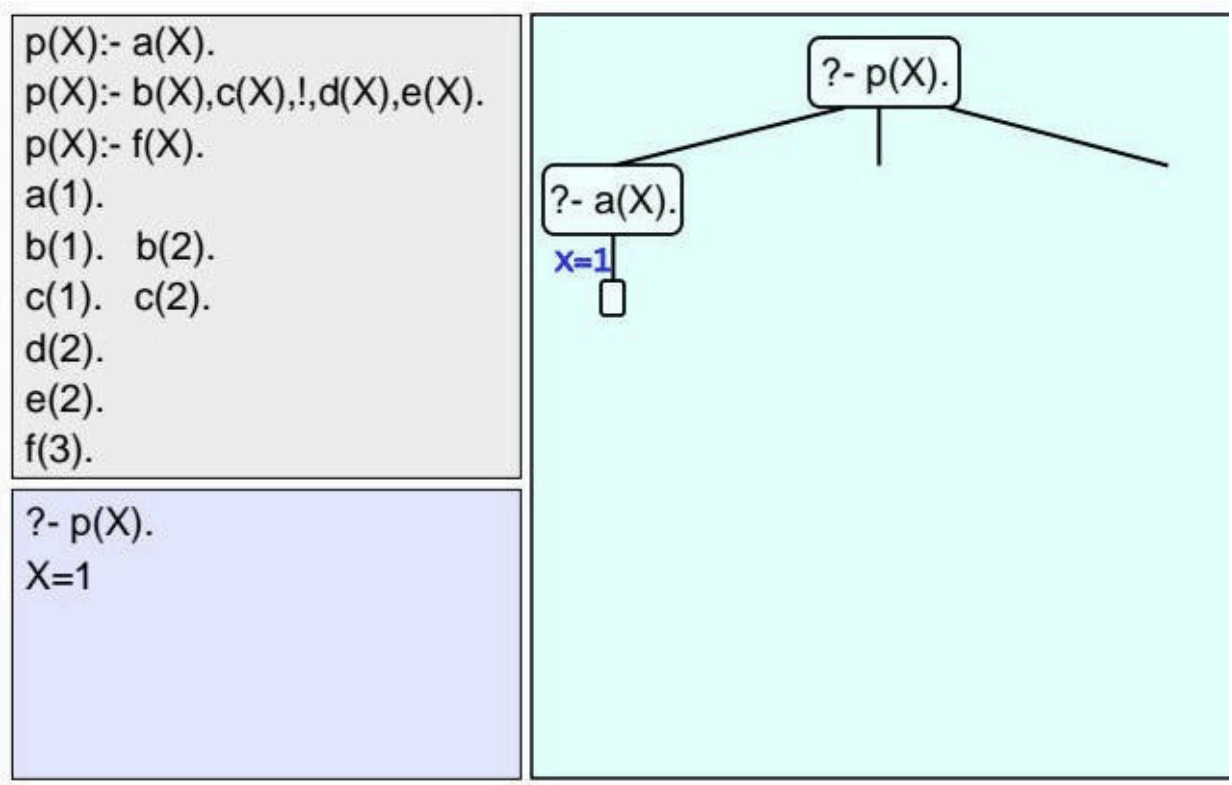
Si nous posons maintenant la même question, nous obtiendrons la réponse suivante :

```
?- p(X).
X=1;
no
```

Après l'ajout de la coupure, l'exploration est limitée.

Voici le code complet avec la coupure :

```
p(X) :- a(X). p(X) :- b(X), c(X), !, d(X),
e(X). p(X) :- f(X). a(1). b(1). b(2). c(1).
c(2). d(2). e(2). f(3).









?- p(X).
```




Dans cet exemple, la coupure est utilisée pour contrôler le backtracking, ce qui peut rendre le code plus efficace, mais peut également modifier le comportement du programme.

## Ce que Fait la Coupure

La coupure ne nous engage qu'aux choix faits depuis que le but parent a été unifié avec le côté gauche de la clause contenant la coupure. Par exemple, dans une règle de la forme :

$q :- p1, \ldots, pn, !, r1, \ldots, r_n.$

Quand on atteint la coupure, elle nous engage :

- à cette clause particulière de q
- aux choix faits par $p1,\ldots,pn$
- PAS aux choix faits par $r1,\ldots,rn$

## Utilisation de la Coupure

Considérons le prédicat `max/3` qui réussit si le troisième argument est le maximum des deux premiers.

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

Exemples d'utilisation :

```
?- max(2, 3, 3).
yes

?- max(7, 3, 7).
yes

?- max(2, 3, 2).
no

?- max(2, 3, 5).
no
```

# Le Prédicat `max/3`

Il y a une inefficacité potentielle. Supposons qu'il soit appelé avec :

```
?- max(3, 4, Y).
```

Il unifiera correctement Y avec 4, mais lorsqu'on lui demandera plus de solutions, il essaiera de satisfaire la deuxième clause. C'est complètement inutile !

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

Quel est le problème ?

# `max/3` avec Coupure

Avec l'aide de la coupure, c'est facile à corriger :

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

Si le X =< Y réussit, la coupure nous engage à ce choix, et la seconde clause de max/3 n'est pas considérée. Si le X =< Y échoue, Prolog passe à la seconde clause.

## Coupures Vertes

> Les coupures qui ne changent pas le sens d'un prédicat sont appelées coupures vertes.

La coupure dans max/3 est un exemple de coupure verte. Le nouveau code donne exactement les mêmes réponses que l'ancienne version, mais il est plus efficace. Pourquoi ne pas supprimer le corps de la deuxième clause ? Après tout, c'est redondant.

## Un Autre max/3 avec Coupure

Quelle est sa qualité ?

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

Exemples :

```
?- max(200, 300, X).
X=300
yes

?- max(400, 300, X).
X=400
yes

?- max(200, 300, 200).
yes
```

Oups...

## max/3 Révisé avec Coupure

Cela fonctionne :

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

L'unification après avoir traversé la coupure.

```
?- max(200, 300, 200).
no
```

# Coupures Rouges

> Les coupures qui changent le sens d'un prédicat sont appelées coupures rouges.

La coupure dans le max/3 révisé en est un exemple.

Les programmes contenant des coupures rouges :

- Ne sont pas entièrement déclaratifs
- Peuvent être difficiles à lire
- Peuvent conduire à des erreurs de programmation subtiles

Si nous supprimons la coupure, nous n'obtenons pas un programme équivalent.

# Prédicat Intégré : fail/0

> Comme son nom l'indique, c'est un but qui échouera immédiatement lorsque Prolog essaiera de le prouver.

Mais rappelez-vous : lorsque Prolog échoue, il essaie de backtracker. Cela peut ne pas sembler très utile.

# Vincent et les Burgers

La combinaison `cut fail` permet de coder les exceptions.

```
enjoys(vincent, X) :- bigKahunaBurger(X), !, fail.
enjoys(vincent, X) :- burger(X).

burger(X) :- bigMac(X).
burger(X) :- bigKahunaBurger(X).
burger(X) :- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

Exemples :

```
?- enjoys(vincent, a).
yes

?- enjoys(vincent, b).
no
```

# Coupe et Échec

La combinaison cut-fail permet de coder des exceptions. Cela est utile lorsque vous voulez qu'une règle échoue intentionnellement sous certaines conditions.

Exemple :

```
enjoys(vincent, X) :-
      bigKahunaBurger(X),
      !,
      fail. % Vincent n'aime pas les Big Kahuna Burgers
enjoys(vincent, X) :-
      burger(X).
```

Dans cet exemple, si X est un `bigKahunaBurger`, la règle échoue. Sinon, elle vérifie si X est un burger.

# La Négation par l'Échec

La combinaison cut-fail peut être utilisée pour implémenter une forme de négation, appelée négation par l'échec.

La négation par l'échec est définie comme `neg(Goal) :- Goal, !, fail. neg(Goal).` Si `Goal` réussit, alors `neg(Goal)` échoue. Si `Goal` échoue, alors `neg(Goal)` réussit.

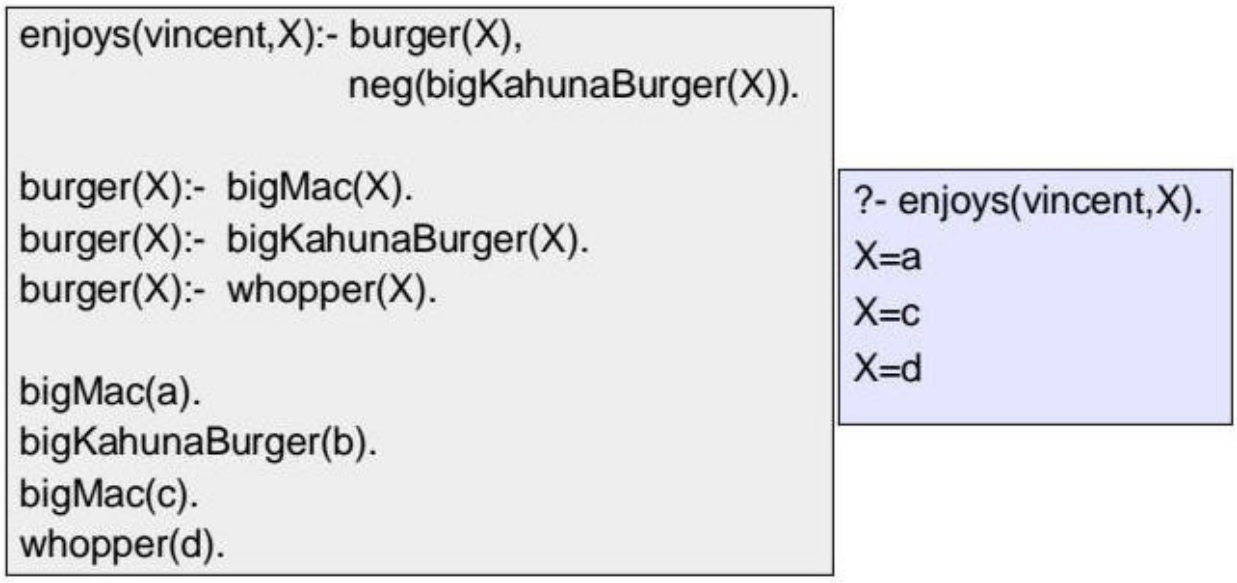
On peut revisiter l'exemple de Vincent et des burgers en utilisant la négation par l'échec :

```
enjoys(vincent, X) :-
    burger(X),
    neg(bigKahunaBurger(X)).
```

Cette règle signifie que Vincent aime les burgers qui ne sont pas des Big Kahuna Burgers.

# Exemple Visuel

Voici un programme Prolog qui illustre les préférences de Vincent en matière de burgers. Vincent aime tous les burgers, sauf les Big Kahuna Burgers.




Le programme définit des règles et des faits concernant les burgers que Vincent aime. La requête `?- enjoys(vincent,X).` renvoie les burgers que Vincent aime, à savoir Big Mac (a), Big Mac (c) et Whopper (d). Il n'aime pas Big Kahuna Burger.

# Prédicat Intégré :

`\+`

Puisque la négation par l'échec est couramment utilisée, Prolog fournit un prédicat intégré \+ pour la négation par l'échec.

Exemple :

```
enjoys(vincent, X) :-
      burger(X),
      \+ bigKahunaBurger(X).
```

Dans Prolog standard, l'opérateur préfixe \+ signifie la négation par l'échec.

## Négation par l'Échec et Logique

La négation par l'échec n'est pas une négation logique. Changer l'ordre des buts peut entraîner un comportement différent.

Exemple :

```
enjoys(vincent, X) :-
      \+ bigKahunaBurger(X),
      burger(X).
```

Cette formulation peut donner un comportement inattendu. Il est crucial de comprendre l'impact de l'ordre des buts lors de l'utilisation de la négation par l'échec.

Références :

SWI-Prolog. (s. d.). https://www.swi-prolog.org/
Learn ProLog Now ! (s. d.). https://lpn.swi-prolog.org/lpnpage.php?pageid=top
PROLOG – Programmation Logique
© Patrick Blackburn, Johan Bos & Kristina Striegnitz
Traduction : Yannick Estève
Cours dispensé à CERI – Avignon Université