Systèmes Experts 8
Outils en Prolog

Richard.Moot@labri.fr
http://www.labri.fr/perso/moot/SE/

Le projet

- La date limite pour la soumission du totalité (rapport en pdf et programmes) de votre projet de programmation est le dimanche 13 décembre à 23:59.
- Faites attention que votre programme compile sans erreurs et vérifiez bien son bon comportement.

L'examen

- L'examen est le vendredi 18 décembre de 14:00 à 15:30
- Vous avez le droit d'amener les papiers qui vous semblent utiles.
- Vous devez pouvoir répondre au tout le sujets traités dans le cours et le TD.

Prolog
Rappel : Meta-programmes

- call(Terme) nous permet de traiter un terme comme un prédicat (partie du programme) et d'essayer de faire la preuve qui en correspond.
- Terme doit correspondre à un prédicat du programme au moment de son execution.
- Notamment, Terme ne peut pas être une variable libre.

Prolog
Rappel : Meta-programmes

- les prédicats assert(Terme) et retract(Terme) nous permettent de changer le programme lors de son execution.
- NB : si terme est un prédicat qui apparaît déjà dans votre programme (c'est à dire qu'il y a des faits ou des clauses qui ne sont pas créés par assert(Terme) vous avez besoin de le déclarer avant le premier clause du prédicat.

Prolog
Rappel : Meta-programmes

```
:- dynamic parent/2.

parent(jean, marie).
...

:- dynamic homme/1.

homme(jean).
...
```

directif au compilateur, sous forme d'un clause sans tête

Prolog
Meta-programmes

- alors le prédicat

  assert(parent(jean,marie))
  nous ajoute le prédicat

  parent(jean,marie)
  à notre programme. Si parent(jean,marie) était faux (ou ne pas démontrable) avant, il sera vrai après l'assert.

Prolog
Meta-programmes

- si l'ordre des clauses est important, on peut utiliser deux variants d'assert :
  - asserta(Clause) ajoute Clause au début des clauses pour le prédicat,
  - assertz(Clause) ajoute Clause à la fin.

Prolog
Rappel : Meta-programmes

```
:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Prolog
Rappel : Meta-programmes

```
?- asserta(parent(jean,marie)).

:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Prolog
Rappel : Meta-programmes

```
?- asserta(parent(jean,marie)).

:- dynamic parent/2.

parent(jean,marie).
parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Prolog
Rappel : Meta-programmes

```
?- asserta(parent(jean,marie)).

:- dynamic parent/2.

parent(jean,marie).
parent(jean,marie).
parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Attention : Prolog ajoute un clause même s'il existe déjà !

Prolog
Rappel : Meta-programmes

```
:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Prolog
Rappel : Meta-programmes

```
?- assertz(parent(jean,marie)).

:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
```

Prolog
Rappel : Meta-programmes

```
?- assertz(parent(jean,marie)).

:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
parent(jean,marie).
```

Prolog
Rappel : Meta-programmes

```
?- retract(parent(leto,paul)).

:- dynamic parent/2.

parent(leto, paul).
parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
parent(jean,marie).
```

Prolog
Rappel : Meta-programmes

?- retract(parent(leto,paul)).

:- dynamic parent/2.

parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
parent(jean,marie).

Prolog
Rappel : Meta-programmes

?- retract(parent(jessica,X)).

:- dynamic parent/2.

parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
parent(jean,marie).

Prolog
Rappel : Meta-programmes

?- retract(parent(jessica,X)). X = paul

:- dynamic parent/2.

parent(leto, alia).
parent(jessica, alia).
parent(jean,marie).

Prolog
Meta-programmes

- inversement, le prédicat **retractall(parent(_,_))** supprime tout les clauses du forme parent(X,Y).
- après il n'y aura aucun fait du forme parent(X,Y) dans la base de données (le fait que parent soit dynamique reste par contre).

Prolog
Rappel : Meta-programmes

```
?- retractall(parent(_,_)).

:- dynamic parent/2.

parent(leto, alia).
parent(jessica, paul)
parent(jessica, alia).
parent(jean,marie).
```


Prolog
Rappel : Meta-programmes

```
?- retractall(parent(_,_)).

:- dynamic parent/2.
```


Prolog
Meta-programmes

- si vous changez la base de données souvent, il peut être utile de voir les données qui sont stockées.
- les prédicat listing(Pred) écrit toutes les clauses pour Pred.
- attention : si vous posez deux question à Prolog, vous devez assurer que le données stockées pour la réponse au premier question n'empêchent pas une réponse correcte au deuxième

Prolog
Meta-programmes

- assert et retract sont utiles pour enregistrer des données de façon permanente.
- alors ils peuvent servir pour avoir une base de données dynamique (où les données peuvent changer)
- mais aussi pour faire la tabulation : pour enregistrer les résultats de calcul intermédiaire et ainsi d'éviter de recalculer de résultats.

Prolog
Meta-programmes

- les prédicats findall, bagof et setof (partie de SWI Prolog et du standard ISO Prolog)
- on parlera ici de setof, qui dans le contexte de système experts est le plus utile.

Prolog
Meta-programmes

```
?- setof(P, parent(P,E), ListeDeParents).
```

Prolog
Meta-programmes

```
?- setof(P, parent(P,E), ListeDeParents).

E = paul
ListeDeParents = [jessica,leto]
;
E = ghanima
ListeDeParent = [chani,paul]
```

Prolog
Meta-programmes

```
?- setof(P, E^parent(P,E), ListeDeParents).
```

quantificateur existentiel :
n'a du sens en Prolog qu'en
setof et bagof
On indique explicitement que les valeurs
pour E ne nous intéressent pas

Prolog
Meta-programmes

```
?- setof(P, E^parent(P,E), ListeDeParents).

ListeDeParents = [chani,jessica,leto,paul]
```

Prolog
Meta-programmes

- Remarque 1 : ça n'a presque jamais du sens d'avoir des variables libres dans le prédicat que vous donnez comme deuxième argument de setof, faites attention à bien utiliser "X^" pour chaque variable libre X du prédicat.
- Remarque 2 : setof échoue s'il n'y a pas de solutions (et ne réussit pas avec l'ensemble vide). Normalement ce comportement est souhaitable; sinon, un petit peu de programmation suffit pour avoir le liste vide.

Prolog
Meta-programmes

- Alors, la fin d'une interaction avec un système expert pourrait avoir le forme suivant :

```
boucle(X) :-
      setof(Y, Z^solution(X,Y,Z), Ensemble),
      !,
      imprimer_solutions(Ensemble).
boucle(X) :-
      ...     % pas de solution, continuer
```

Outils en Prolog

- bibliothèques pour
  - le traitement des listes,
  - les ensembles
  - les entrées de l'utilisateur
- automates finis pour un système expert

Outils en Prolog

:- ensure_loaded(library(lists)).

Directif pour le compilateur, comme pour les déclarations ":- dynamic" ; ":- ensure_loaded" charge le fichier donnée comme argument si celui-ci n'est pas encore chargé.

library(lists) indique que le programme "lists.pl" est dans le bibliothèque standard :- ensure_loaded(fichier) suffit pour charger "fichier.pl"

Outils en Prolog library(lists)

- library(lists) contient plusieurs prédicats de base : on a déjà vu la plupart de ces prédicats pendant le cours et le reste n'est pas difficile à programmer non plus.
- pour éviter de passer trop de temps à la programmation des prédicats relativement triviaux, vous pouvez vous servir de ce bibliothèque.

Outils en Prolog library(lists)

:- ensure_loaded(library(lists)).

append([a,b,c], [1,2,3], [a,b,c,1,2,3]).

concatenation des deux premiers listes; soit le premier soit le dernier argument doit être un liste propre.

member(2, [1,2,3]).

le premier élement est un membre du liste; liste doit être un liste propre

Outils en Prolog library(lists)

:- ensure_loaded(library(lists)).

memberchk(2, [1,2,2,3]).

élement est strictement égal à un des membres de liste; réussit une seule fois, même si élement apparaît plusieurs fois (contrairement au member)

select(2, [1,2,3], [1,3]).

enleve un élement du liste et donne le reste du liste comme résultat.

Outils en Prolog ordset

- ordset.pl (fichier à télécharger sur la page web du cours; une version moins complète fait partie du bibliothèque standard de Prolog) contient des prédicats pour manipuler des ensembles en Prolog.
- ici, un ensemble est representé comme un liste ordonné.

Outils en Prolog ordset

- pour transformer un liste en ensemble ordonné, vous pouvez utiliser le prédicat sort(Liste, EnsembleOrdonnee)
  qui trie les éléments de Liste et élimine les doublons en EnsembleOrdonnee
- alors sort([5,5,4,4,3,2,2,2,1], Ensemble) donne Ensemble = [1,2,3,4,5]
- Important : aucun de ces prédicats a un sens si les arguments ne sont pas d'ensembles !

Outils en Prolog ordset

- ord_union(Ensemble1, Ensemble2, Total)
  ce prédicat fait l'union de deux ensembles, c'est-à-dire que Total contiendrai toutes les éléments de Ensemble1 et toutes les éléments de Ensemble 2. Total est lui-même un ensemble ordonnée : alors, il ne contient pas de doublures et les éléments sont ordonnées.

Outils en Prolog ordset

- ord_intersect(Ensemble1, Ensemble2, Inter)
  ce prédicat fait l'intersection de deux ensembles, c'est-à-dire que Inter contiendrai les éléments qui sont dans Ensemble1 et en Ensemble2.
- ord_intersect(Ensemble1, Ensemble2)
  est vrai si Ensemble1 et Ensemble2 ont une intersection non-vide (et alors partagent au moins un élément).

Outils en Prolog ordset

- ord_member(Element, Ensemble)
  vrai si Element fait partie de Ensemble
- ord_insert(Ensemble0, Element, Ensemble)
  vrai si Ensemble est Ensemble0 avec Element ajouté (s'il faisait pas partie de Ensemble0)
- ord_select(Element, Ensemble0, Ensemble)
  vrai si Ensemble0 contient Element et si cet élément n'appartient plus à Ensemble.

Outils en Prolog ordset

```
?- member(1, [3,7,9]).


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

Outils en Prolog ordset

```
?- ord_member(1, [3,7,9]).


ord_member(X, [X|_]).
ord_member(X, [Y|Ys]) :-
        X @> Y,
        ord_member(X, Ys).
```

Outils en Prolog ordset

```
?- ord_member(1, [3,7,9]).


ord_member(X, [X|_]).
ord_member(X, [Y|Ys]) :-
        X @> Y,
        ord_member(X, Ys).
```

X @> Y est comme X > Y mais donne un résultat pour tout les termes, pas juste pour des nombres.

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [0,3,4,6], Inter).

% si un des deux ensembles est vide,
% l'intersection est vide aussi.

ord_intersect([], _, []) :-
        !.
ord_intersect(_, [], []) :-
        !.
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [0,3,4,6], Inter).

% si les deux ensembles ont le meme element
% comme premier element, cet element est
% dans l'intersection

ord_intersect([X|Xs], [Y|Ys], [X|Zs]) :-
        X == Y,
        !,
        ord_intersect(Xs, Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [0,3,4,6], Inter).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [0,3,4,6], Inter).

% si Y est plus petit que X, on jette Y car il ne
% peut pas être égal à un des éléments de [X|Xs]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        Y @< X,
        !,
        ord_intersect([X|Xs], Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [3,4,6], Inter).

% si Y est plus petit que X, on jette Y car il ne
% peut pas être égal à un des éléments de [X|Xs]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        Y @< X,
        !,
        ord_intersect([X|Xs], Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [3,4,6], Inter).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([2,3,4,5], [3,4,6], Inter).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([3,4,5], [3,4,6], Inter).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([3,4,5], [3,4,6], [3|Zs]).

% si les deux ensembles ont le meme element
% comme premier element, cet element est
% dans l'intersection

ord_intersect([X|Xs], [Y|Ys], [X|Zs]) :-
        X == Y,
        !,
        ord_intersect(Xs, Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([4,5], [4,6], Zs).

% si les deux ensembles ont le meme element
% comme premier element, cet element est
% dans l'intersection

ord_intersect([X|Xs], [Y|Ys], [X|Zs]) :-
        X == Y,
        !,
        ord_intersect(Xs, Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([4,5], [4,6], [4|Zs]).

% si les deux ensembles ont le meme element
% comme premier element, cet element est
% dans l'intersection

ord_intersect([X|Xs], [Y|Ys], [X|Zs]) :-
        X == Y,
        !,
        ord_intersect(Xs, Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([5], [6], Zs).

% si les deux ensembles ont le meme element
% comme premier element, cet element est
% dans l'intersection

ord_intersect([X|Xs], [Y|Ys], [X|Zs]) :-
        X == Y,
        !,
        ord_intersect(Xs, Ys, Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([5], [6], Zs).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([], [6], Zs).

% si X est plus petit que Y, on jette X car il ne
% peut pas être égal à un des éléments de [Y|Ys]

ord_intersect([X|Xs], [Y|Ys], Zs) :-
        X @< Y,
        !,
        ord_intersect(Xs, [Y|Ys], Zs).
```

Outils en Prolog
ordset

```
?- ord_intersect([], [6], []).

% si un des deux ensemble est vide,
% l'intersection est vide aussi.

ord_intersect([], _, []) :-
        !.
ord_intersect(_, [], []) :-
        !.
```

Outils en Prolog
ordset

```
?- ord_intersect([1,2,3,4,5], [0,3,4,6], [3,4]).


ord_intersect([], _, []) :-
        !.
ord_intersect(_, [], []) :-
        !.
```

Outils en Prolog tokenize

- tokenize.pl (fichier à télécharger sur la page web du cours) contient des prédicats pour transformer des characters qui ont été lus en un liste de Termes.
- ceci ce combine avec le prédicat read_line_to_codes de la bibliothèque standard.

Outils en Prolog tokenize

```
:- ensure_loaded(library(readutil)).
:- use_module(tokenize, [tokenize_string/2]).
:- ensure_loaded(ordset).

read_line(ListeDeMots) :-
        read_line_to_codes(user_input, Codes),
        tokenize(Codes, ListeDeMots).
```

Outils en Prolog tokenize

- quand on a lu une ligne, on peut utiliser le prédicat sous-liste (du dernier cours) pour voir si la ligne contient une séquence de mots.
- on peut aussi utiliser les prédicats du bibliothèque ordsets pour voir si la séquence contient un mot d'un certain ensemble ou tous les mots d'un certain ensemble.

Rappel: Listes

```
% = sous_liste(Partie, Totale)
%
% vrai si Totale contient Partie
```

C'est à dire :

```
sous_liste([a,b,c,d], [a,b,c,d,e,f,g]).
sous_liste([b,c,d], [a,b,c,d,e,f,g]).
sous_liste([f], [a,b,c,d,e,f,g]).
sous_liste([], [a,b,c,d,e,f,g]).
```

sont tous vrais.

Rappel: Listes

```
% = sous_liste(Partie, Totale)
%
% vrai si Totale contient Partie

sous_liste(Partie, Totale) :-
        append(Partie, _Apres, Liste),
        append(_Avant, Liste, Totale).
```

Rappel: Listes


```
% = sous_liste(Partie, Totale)
%
% vrai si Totale contient Partie

sous_liste(Partie, Totale) :-
        append(Partie, _Apres, Liste),
        append(_Avant, Liste, Totale).
```

Interaction

```
% application de sous_liste dans un systeme
% expert.


boucle(Etat, Entree) :-
        sous_liste([comprends,pas], Entree),
        !,
        explique(Etat).
```

Interaction

```
% application de membre dans un systeme
% expert.


boucle(Etat, Entree) :-
        membre(pourquoi, Entree),
        !,
        explique(Etat).
```

Interaction

```
% application de ord_intersect dans un systeme
% expert.

boucle(Etat, []) :-
        !,
        writeln('À bientôt').
boucle(Etat, Entree) :-
        ord_intersect([arret,stop,marre], Entree),
        !,
        writeln('À bientôt').
```

Interaction

```
% application de ord_intersect dans un systeme
% expert.
```

Trouvez l'erreur !

```
boucle(Etat, []) :-
        !,
        writeln('À bientôt').
boucle(Etat, Entree) :-
        ord_intersect([arret,stop,marre], Entree),
        !,
        writeln('À bientôt').
```

Interaction

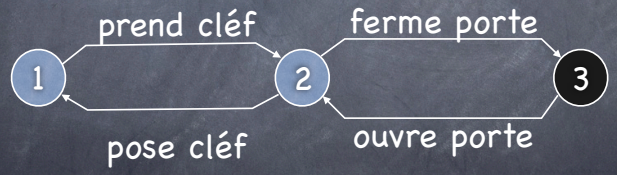
```
% application de ord_intersect dans un systeme
% expert.

boucle(Etat, []) :-
        !,
        writeln('À bientôt').
boucle(Etat, Entree) :-
        sort(Entree, Ensemble),
        ord_intersect([arret,marre,stop], Ensemble),
        !,
        writeln('À bientôt').
```

Automates Finis

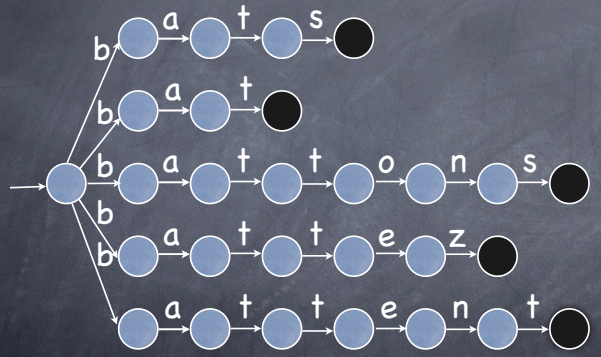
- Des automates finis sont un sujet bien étudié en théorie des languages formelles, avec beaucoup de bons propriétés.
- Ils sont équivalents aux expressions régulières ("grep" en UNIX/Linux, mais aussi en Perl)
- Il y a des applications en language naturel (surtout en morphologie)

Automates Finis

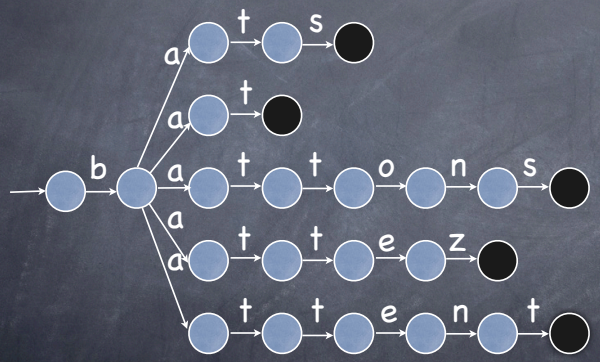
1: cléf dans le cabinet, porte ouverte
2: cléf sur moi, porte ouverte
3: cléf sur moi, porte fermé



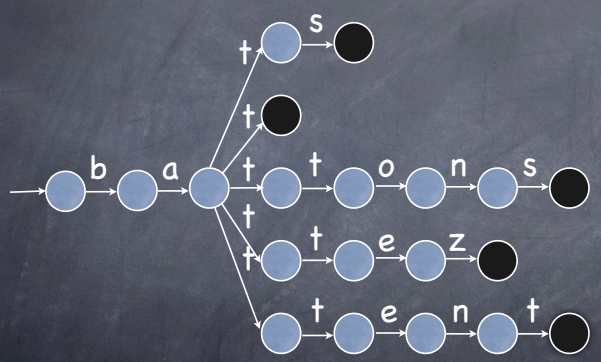
Automates Finis



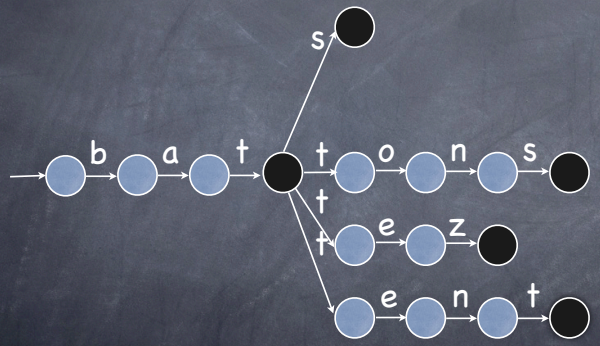
Automates Finis



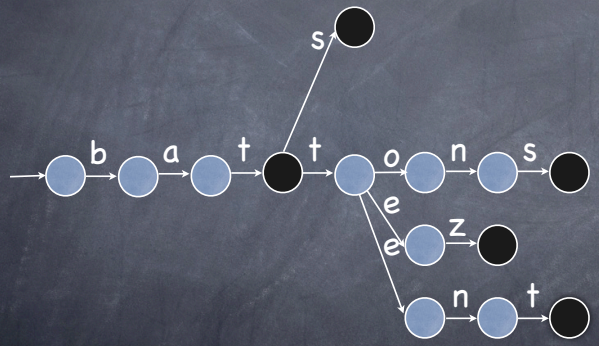
Automates Finis



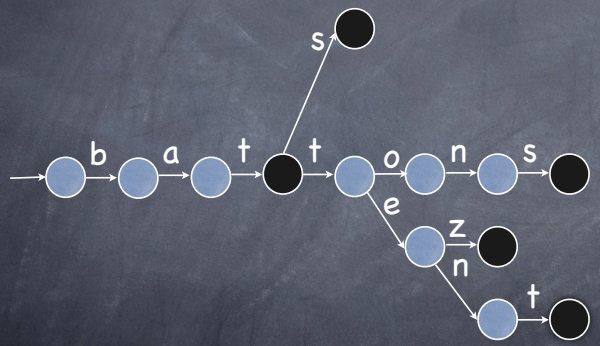
Automates Finis



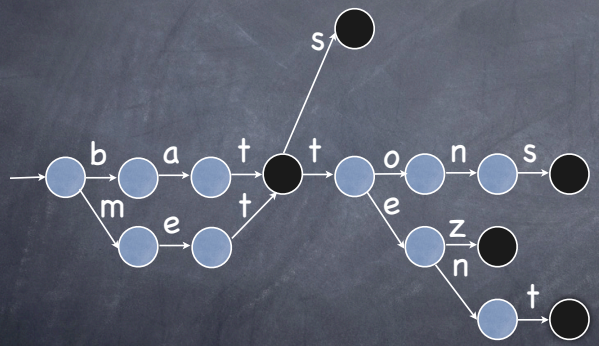
Automates Finis



Automates Finis



Automates Finis

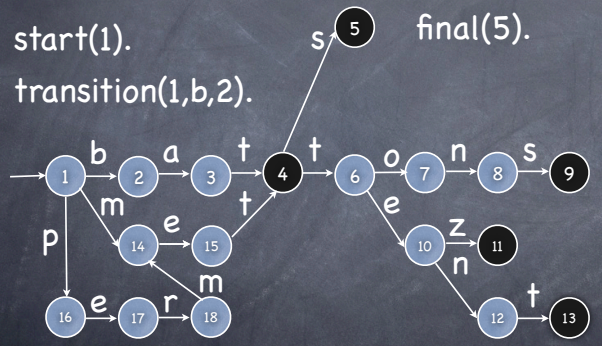


Automates Finis

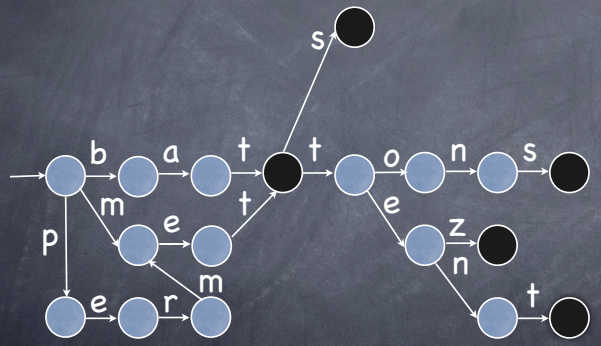
start(1).

transition(1,b,2).

final(5).



Automates Finis



Automates Finis en Prolog

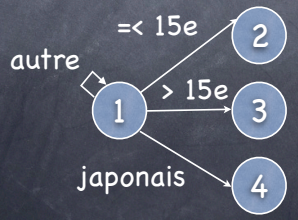
```
start(1).

final(4).
final(5).
...

transition(1,b,2).
transition(1,m,14).
...
```

Automates Finis en Prolog

```
fsa(Liste) :-
        start(Etat).

fsa([], Etat) :-
        final(Etat).
fsa([S|Ss], Etat0) :-
        transition(Etat0, S, Etat),
        fsa(Ss, Etat).
```

Automates Finis

1: on veux traiter la réponse sur une question sur le budget du restaurant
2: petit budget
3: grand budget
4: changement du sujet



Automates Finis

```
fsa(1, Reponse) :-
        member(X, Reponse),
        integer(X),
        X =< 15,
        !,
        fsa(2, Reponse).
```

Automates Finis

```
fsa(1, Reponse) :-
        member(X, Reponse),
        integer(X),
        X > 15,
        !,
        fsa(3, Reponse).
```

Automates Finis

```
fsa(1, Reponse) :-
        sort(Reponse, Ensemble)
        ord_intersect([chinois,japonais], Ensemble),
        !,
        fsa(4, Reponse).
```

Automates Finis

```
fsa(1, _Reponse) :-
        writeln('Je n\'ai pas compris votre
réponse.'),
        writeln('Veuillez indiquer, en chiffres, le
montant maximal par couvert que vous
souhaitez dépenser.'),
        read_line(Reponse),
        fsa(1, Reponse).
```

Automates Finis

f ermier

l oup

c hèvre

ch o u

Automates Finis

f ermier

l oup

c hèvre

ch o u

Automates Finis

- Le fermier a une seule place dans son bateau.
- Si le chèvre et le chou sont ensemble au même côté du fleuve sans que le fermier soit présent, le chèvre mange le chou.
- Si le loup et le chèvre sont seules sans le fermier, le loup mange le chèvre.

Automates Finis

- Comment peut le fermier faire pour tout transporter à l'autre coté du fleuve ?
- Comment peut-on faire en Prolog ?
- On peut décrire le système comme automate fini
- Mais avec des états structurés pour nous rendre la vie plus facile.

Automates Finis

f ermier

l oup

c hèvre

ch o u

e(g,g,g,g)

Automates Finis

start(e(g,g,g,g)).

position du fermier
(g)auche ou (d)roite

end(e(d,d,d,d)).

Automates Finis

start(e(g,g,g,g)).

position du chou
(g)auche ou (d)roite

end(e(d,d,d,d)).

Automates Finis

start(e(g,g,g,g)).

position du chèvre
(g)auche ou (d)roite

end(e(d,d,d,d)).

Automates Finis

start(e(g,g,g,g)).

position du loup
(g)auche ou (d)roite

end(e(d,d,d,d)).

Automates Finis

start(e(g,g,g,g)).

end(e(d,d,d,d)).

inverse(g, d).
inverse(d, g).

Automates Finis

```
% le fermier traverse seul.

transition(e(Loup,Chevre,Chou,Fermier0),
           e(Loup,Chevre,Chou,Fermier)) :-
      inverse(Fermier0, Fermier),
      Chevre \== Chou,
      Loup \== Chevre.
```

Automates Finis

```
% le fermier traverse avec le loup.

transition(e(Loup0,Chevre,Chou,Fermier0),
           e(Loup,Chevre,Chou,Fermier)) :-
      Fermier0 == Loup0,
      inverse(Fermier0, Fermier),
      inverse(Loup0, Loup),
      Chevre \== Chou.
```

Automates Finis

```
% le fermier traverse avec le chevre.

transition(e(Loup,Chevre0,Chou,Fermier0),
           e(Loup,Chevre,Chou,Fermier)) :-
      Fermier0 == Chevre0,
      inverse(Fermier0, Fermier),
      inverse(Chevre0, Chevre).
```

Automates Finis

```
% le fermier traverse avec le chou.

transition(e(Loup,Chevre,Chou0,Fermier0),
           e(Loup,Chevre,Chou,Fermier)) :-
      Fermier0 == Chou0,
      inverse(Fermier0, Fermier),
      inverse(Chou0, Chevre),
      Loup \== Chevre.
```

Automates Finis

```
fsa :-
      start(Etat),
      fsa(Etat, [Etat]).

fsa(Etat, Chemin) :-
      final(Etat),
      !,
      write(Chemin).
```

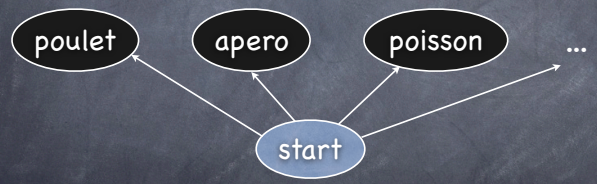
Automates Finis

```
fsa(Etat0, Chemin) :-
        transition(Etat0, Etat),
        \+ member(Etat, Chemin),
        fsa(Etat, Chemin).
```

Le mini Système Expert

- Le système expert "accords.pl" est vraiment trivial, mais nous permet de voir un petit peu comment programmer un système expert en Prolog avec les outils qu'on a déjà vu.
- De point de vue d'un automate fini, le système n'est pas compliqué non plus.

Le mini Système Expert



Le mini Système Expert

```
etat_final(poulet).
etat_final(agneau).
...

process(Etat, Mots) :-
        etat_final(Etat),
        !,
        ecrire_accords(Etat).
```

Le mini Système Expert

```
accord(poulet, [hermitage, blanc]).
accord(agneau, [pauillac, rouge]).
...

ecrire_accords(Etat) :-
        setof(Vin, accord(Etat,Vin), Ensemble),
        write_list(Ensemble).
```

Le mini Système Expert

```
regle_mot(mouton, agneau).
regle_mot(mouton, mouton).
...

process(_Etat, Ligne) :-
        regle_mot(Mot, Etat),
        member(Mot, Ligne),
        !,
        process(Etat, Ligne).
```

Le mini Système Expert

```
regle_mot(mouton, agneau).
regle_mot(mouton, mouton).
...

process(_Etat, Ligne) :-
        regle_mot(Mot, Etat),
        member(Mot, Ligne),
        !,
        process(Etat, Ligne).
```

Pour un système plus sophistiqué, utiliser aussi l'état d'entrée

Le mini Système Expert

```
regle_sequence([foie,gras], foie_gras).
regle_sequence([coq,au,vin], coq_au_vin).
...

process(_Etat, Ligne) :-
        regle_sequence(Sequence, Etat),
        sous_liste(Sequence, Ligne),
        !,
        process(Etat, Ligne).
```

Le mini Système Expert

```
regle_mots([poisson,sushi,...], poisson).
regle_mots([bavette,tournedos], viande_rouge).
...

process(_Etat, Ligne) :-
        regle_mots(Mots, Etat),
        sort(Mots, Ensemble1),
        sort(Ligne, Ensemble2),
        ord_intersect(Ensemble1, Ensemble2),
        !,
        process(Etat, Ligne).
```

Le mini Système Expert

```
accord(poulet, [hermitage, blanc]).
accord(agneau, [pauillac, rouge]).
...

ecrire_plats(Etat, MotsEns) :-
        setof(Vin-Etat,
                (accord(Etat, Vin),
                 sort(Vin, VinEns),
                 ord_intersect(VinEns,MotsEns,VinEns))
                Ensemble),
        write_list(Ensemble).
```

Le mini Système Expert

- Comment étendre un tel système expert ?
  - ajouter des connaissances du monde (contenu du cave, rélations entre apellations) ,
  - utiliser l'état pour choisir les règles,
  - utiliser un état en forme de terme complexe.

Le mini Système Expert

- Comment étendre un tel système expert ?
  - ajouter des connaissances du monde (contenu du cave, rélations entre apellations, cépages, etc.) ,
  - utiliser l'état pour choisir les règles (par exemple, l'état "poulet" peut avoir des états qui lui succedent, comme "poulet basquaise" et "poulet entier" avec des accords différents).