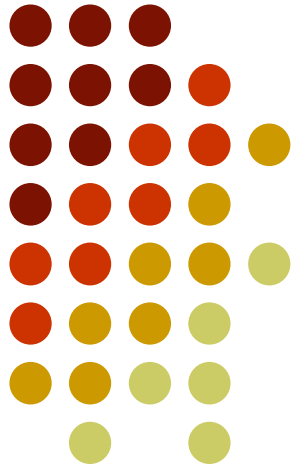
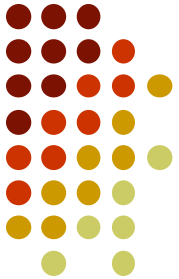
Le langage Prolog

Cours n°5

Grammaire et automates en langage Prolog



Jean-Michel Adam
UFR SHS – L2 MIASHS



Partie 1
-
Prédicats utiles pour la manipulation de textes

Représentation des textes en Prolog

En Prolog, un texte est représenté par :

- Une chaine :

  "bonjour Marcel, comment allez vous"

  qui correspond à la liste d'entiers:

  [98,111,110,106,111,117,114,32,77,97,114,99,101,108,44,32,99,111,109,
  109,101,110,116,32,97,108,108,101,122,32,118,111,117,115]

  les caractères sont représentés par des entiers (leur code UNICODE).

  - Jusqu'à la version 6 de SWI-Prolog les chaines étaient des listes d'entiers:

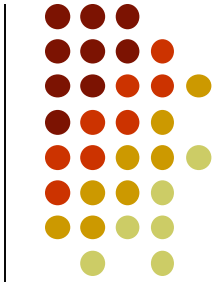
    ```
    ?- A="Hello".
    A = [72, 101, 108, 108, 111].
    ```

  - Depuis la version 7 SWI-Prolog considère les chaines comme des objets atomiques:

    ```
    ?- A="Hello".
    A = "Hello"
    ```

Beaucoup de programmes Prolog sont écrits en considérant les chaines comme des listes de codes

Prédicats pour traiter des textes



En Prolog, un texte est représenté par:

- Une liste d'atomes (mots ou caractères) :

  [bonjour, 'Marcel', comment, allez, vous]

  [b,o,n,j,o,u,r,' ', 'M',a,r,c,e,l,' ',c,o,m,m,e,n,t,' ',a,l,l,e,z,' ',v,o,u,s]

- Une chaine peut facilement être transformée en une liste de caractères avec le prédicat `string_chars` :

  string_chars("bonjour Marcel, comment allez vous" ,L).

  L=[b,o,n,j,o,u,r,' ', 'M',a,r,c,e,l, ',', ' ', c,o,m,m,e,n,t,' ',a,l,l,e,z,' ',v,o,u,s]

- Le prédicat `split_string` permet de découper une chaine en une suite de mots, chaque mot est une chaine, le prédicat peut supprimer les caractères séparateurs (ponctuation) :

```
?- split_string("bonjour Marcel, comment allez vous ?",",? ",",? ",L).
L = ["bonjour", "Marcel", "comment", "allez", "vous"].
```

Prédicats pour traiter des textes

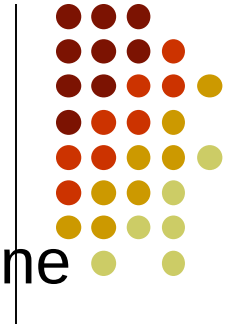
- Le prédicat `tokenize_atom` permet de découper une chaine en une suite de symboles, chaque symbole est un atome :

```
?- tokenize_atom("bonjour Marcel, comment allez vous?",X).
X = [bonjour, 'Marcel', ',', comment, allez, vous, ?].
```

- Le prédicat `atom_string` permet la conversion d'une chaine en atome ou inversement un atome en une chaine. Appliqué à une liste de chaines, on obtient une liste d'atomes :

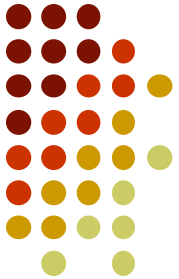
```
?- atom_string(A,"hello").
A = hello.

?- maplist(atom_string,L,["bonjour","Marcel","comment","allez","vous"]).
L = [bonjour, 'Marcel', comment, allez, vous].
```

Prédicats pour traiter des textes



- Le prédicat **`name`** transforme un objet atomique en une liste de codes, et inversement une liste de codes en objet atomique :

```
?- name(hello,L).
L = [104, 101, 108, 108, 111].
?- name("hello",L).
L = [104, 101, 108, 108, 111].
?- name(A,[104, 101, 108, 108, 111]).
A = hello.
?- name(1234,L).
L = [49, 50, 51, 52].
?- name(A,[49, 50, 51, 52]).
A = 1234.
```



Partie 2
-
Grammaires

Définition d'une grammaire (rappel)

- Un langage est en général défini par une grammaire
- Une grammaire définit les règles de construction des phrases du langage (texte).
- Une grammaire G est caractérisée par 4 composants

  $G = < V_T, V_N, S, R >$
- V_T : vocabulaire terminal

  C'est l'ensemble des symboles valables pour le langage (chaînes, termes possibles). Constitué des symboles terminaux du langage.
- V_N : vocabulaire non terminal

  N'a rien à voir avec le vocabulaire terminal, il représente les noms utilisés pour décrire les règles de construction du langage
- $V_N \cap V_T = \varnothing$

Définition d'une grammaire (rappel)

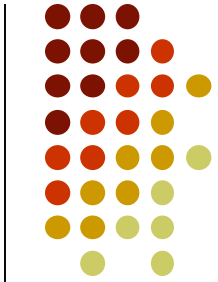
- G = < V_T, V_N, S, R >
- R est l'ensemble des règles de la grammaire
- S est l'axiome (la règle de démarrage de l'analyse)
- Le découpage d'un texte brut en une suite de symboles terminaux lexique du langage est appelé « analyse lexicale »
- La vérification, pour un texte donné, du respect des règles de grammaire du langage est appelée « analyse syntaxique »

Exemple de grammaire simple

S	→	SN SV
SN	→	NP
SV	→	VT SN
NP	→	Marie | Pierre
VT	→	regarde

```
s(L)  :- sn(L1), sv(L2), append(L1, L2, L).
sn(L) :- np(L).
sv(L) :- vt(L1), sn(L2), append(L1, L2, L).
np([pierre]).
np([marie]).
vt([regarde]).
```

Utilisation de la grammaire



```
s(L)  :- sn(L1), sv(L2), append(L1, L2, L).
sn(L) :- np(L).
sv(L) :- vt(L1), sn(L2), append(L1, L2, L).
np([pierre]).
np([marie]).
vt([regarde]).
```

```
?- s([pierre, regarde, marie]).
true

?- s(X).
X = [pierre, regarde, pierre] ;
X = [pierre, regarde, marie] ;
X = [marie, regarde, pierre] ;
X = [marie, regarde, marie].
```

A partir de la grammaire, Prolog peut engendrer toutes
les phrases du langage décrit par la grammaire (si le langage est fini)

Exemple de grammaire simple
écriture classique

S	→	SN SV
SN	→	NP
SV	→	Vt SN
NP	→	Marie | Pierre
Vt	→	regarde

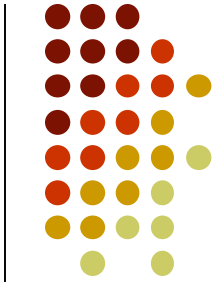
```
s(L1, L2)  :- sn(L1, L3), sv(L3, L2).
sn(L1, L2) :- np(L1, L2).
sv(L1, L2) :- v(L1, L3), sn(L3, L2).
v([regarde|L], L).
np([pierre|L], L).
np([marie|L], L).
```

Chaque règle a 2 arguments :

- la liste des symboles en entrée (liste à analyser)
- la liste des symboles en sortie : après application de la règle

Les symboles terminaux qui ont été énumérés sont supprimés

Utilisation de la grammaire



```
s(L1, L2)   :- sn(L1, L3), sv(L3, L2).
sn(L1, L2) :- np(L1, L2).
sv(L1, L2) :- v(L1, L3), sn(L3, L2).
v([regarde|L], L).
np([pierre|L], L).
np([marie|L], L).
```

```
?- s([pierre, regarde, marie],[]).
true

?- s(X).
X = [pierre, regarde, pierre] ;
X = [pierre, regarde, marie] ;
X = [marie, regarde, pierre] ;
X = [marie, regarde, marie].
```

A partir de la grammaire, Prolog peut engendrer toutes
les phrases du langage décrit par la grammaire (si le langage est fini)

Grammaire DCG (Definite Clause Grammar)

- Prolog offre la possibilité d'utiliser une notation DCG :

  ```
  s(L1, L2) :- sn(L1, L3), sv(L3, L2).
  ```

  peut s'écrire sous la forme :

  ```
  s --> sn, sv.
  ```

- En fait Prolog transforme cette forme en:

  ```
  s(L1, L2) :- sn(L1, L3), sv(L3, L2).
  ```

  avant de l'interpréter.

Grammaire DCG (Definite Clause Grammar)

S → SN SV
SN → NP
SV → Vt SN
NP → Marie | Pierre
Vt → regarde

```
s --> sn, sv.
sn --> np.
sv --> v, sn.
np --> [pierre].
np --> [marie].
v --> [regarde].
```

Grammaire DCG (Definite Clause Grammar)

```
s --> sn, sv.
sn --> np.
sv --> v, sn.
np --> [pierre].
np --> [marie].
v --> [regarde].
```

```
s(L1, L2)  :- sn(L1, L3), sv(L3, L2).
sn(L1, L2) :- np(L1, L2).
sv(L1, L2) :- v(L1, L3), sn(L3, L2).
np([pierre|L], L).
np([marie|L], L).
v([regarde|L], L).
```

```
?- s([pierre, regarde, marie],L).
L = []

?- s([pierre, regarde, marie],[]).
true

?- s(X,[]).
X = [pierre, regarde, pierre] ;
X = [pierre, regarde, marie] ;
X = [marie, regarde, pierre] ;
X = [marie, regarde, marie].
```

A partir de la grammaire, Prolog peut engendrer toutes
les phrases du langage décrit par la grammaire (si le langage est fini)

La notation DCG de Prolog

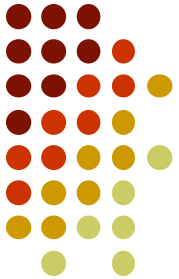
- Les symboles de prédicats et de fonctions, les variables et les constantes obéissent à la syntaxe habituelle de Prolog.
- Les symboles adjacents dans une partie droite de règle sont séparés par une virgule, comme pour les littéraux en partie droite d'une clause
- La flèche est le symbole "-->"
- Les terminaux sont écrits entre "[" et "]"
- La chaîne vide ε est représentée par "[]".
- On peut insérer en partie droite des règles, des buts autres que les symboles de la grammaire ; dans ce cas, ils figurent entre "{" et "}".

Une grammaire DCG où chaque item lexical est introduit par une règle

```
s --> sn, sv.
sn --> np.
sv --> vt, sn.
sv --> vi.
np --> [paul].
np --> [ils].
vi --> [dort].
vi --> [dorment].
vt --> [écoutent].
vt --> [écoute].
```

Des éléments complémentaires (attributs) permettent de prendre en compte des éléments contextuels

Une grammaire DCG où chaque item lexical est introduit par une règle



```
s --> sn(Nombre), sv(Nombre).
sn(Nombre) --> np(Nombre).
sv(Nombre) --> vt(Nombre), sn(_).
sv(Nombre) --> vi(Nombre).
np(sing) --> [paul].
np(plur) --> [ils].
vi(sing) --> [dort].
vi(plur) --> [dorment].
vt(plur) --> [écoutent].
vt(sing) --> [écoute].
```

Des éléments complémentaires (attributs) permettent de prendre en compte des éléments contextuels, comme l'accord en nombre

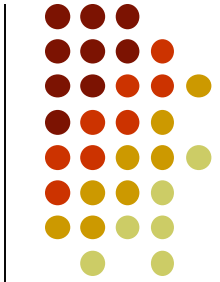
Grammaire DCG où le lexique est donné sous la forme d'une base de faits

```
s --> sn(Nombre), sv(Nombre).
sn(Nombre) --> np(Nombre).
sv(Nombre) --> vt(Nombre), sn(_).
sv(Nombre) --> vi(Nombre).
np(Nombre) --> [Mot], {lexique(Mot, np, Nombre)}.
vi(Nombre) --> [Mot], {lexique(Mot, vi, Nombre)}.
vt(Nombre) --> [Mot], {lexique(Mot, vt, Nombre)}.

lexique(paul, np, sing).
lexique(ils, np, plur).
lexique(dort, vi, sing).
lexique(dorment, vi, plur).
lexique(ecoutent, vt, plur).
lexique(ecoute, vt, sing).
```

Une grammaire DCG où figurent des non terminaux produisant la chaîne vide

```
s --> sn, sv.
sn --> det, n, optrel.
sn --> np.
sv --> vt, sn.
sv --> vi.
optrel --> [].
optrel --> [qui], sv.
det --> [Mot], {lexique(Mot, det)}.
n --> [Mot], {lexique(Mot, n)}.
np --> [Mot], {lexique(Mot, np)}.
vt --> [Mot], {lexique(Mot, vt)}.
vi --> [Mot], {lexique(Mot, vi)}.
```

```
lexique(tout, det).
lexique(un, det).
lexique(etudiant, n).
lexique(programme, n).
lexique(boucle, vi).
lexique(ecrit, vt).
lexique(paul,np).
```



Partie 3
-
Automates

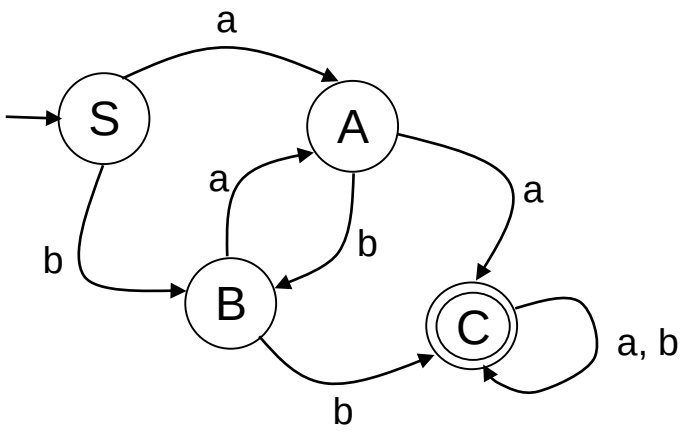
Représentations des Automates

- Un automate correspond à une grammaire de type 3
- On peut donc représenter cette grammaire de la même manière que les autres grammaires
- Si l'automate est déterministe, Prolog ne devrait pas faire de retours arrières

Exemple d'automate déterministe

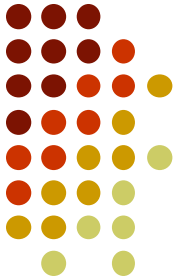
$S \longrightarrow aA$
$S \longrightarrow bB$
$A \longrightarrow aC$
$A \longrightarrow bB$
$B \longrightarrow aA$
$B \longrightarrow bC$
$C \longrightarrow aC$
$C \longrightarrow bC$

```
s --> [a], a.
s --> [b], b.
a --> [a], c.
a --> [b], b.
b --> [b], c.
b --> [a], a.
c --> [b], c.
c --> [a], c.
c --> [].
```



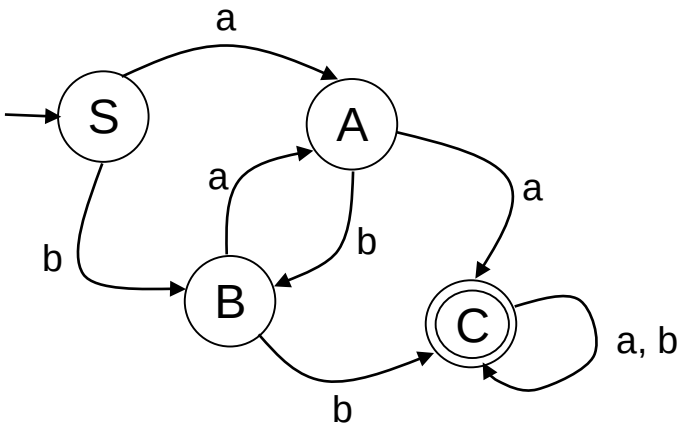
But : s([a, b, a, b, b, a],[]).

Autre programmation possible



$S \longrightarrow aA$
$S \longrightarrow bB$
$A \longrightarrow aC$
$A \longrightarrow bB$
$B \longrightarrow aA$
$B \longrightarrow bC$
$C \longrightarrow aC$
$C \longrightarrow bC$

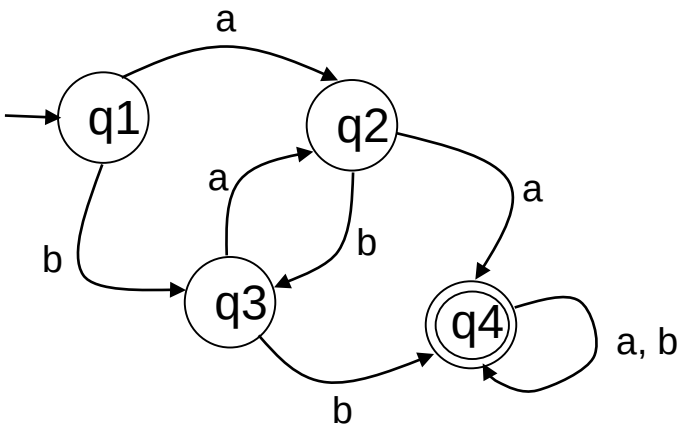
```
s([a|F]) :- a(F).
s([b|F]) :- b(F).
a([a|F]) :- c(F).
a([b|F]) :- b(F).
b([b|F]) :- c(F).
b([a|F]) :- a(F).
c([a|F]) :- c(F).
c([b|F]) :- c(F).
c([]).
```



But : s([a, b, a, b, b, a]).

Autre programmation possible

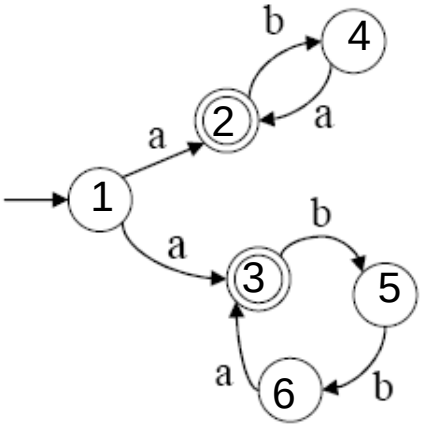
$S \longrightarrow aA$
$S \longrightarrow bB$
$A \longrightarrow aC$
$A \longrightarrow bB$
$B \longrightarrow aA$
$B \longrightarrow bC$
$C \longrightarrow aC$
$C \longrightarrow bC$




```
q1 --> [a], q2.
q1 --> [b], q3.
q2 --> [a], q4.
q2 --> [b], q3.
q3 --> [a], q2.
q3 --> [b], q4.
q4 --> [b], q4.
q4 --> [a], q4.
q4 --> [].
```

But : `q1([a, b, a, b, b, a],X).`
`X=[]`

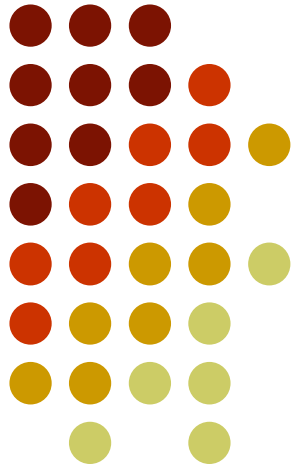
Automate non déterministe




```
q1 --> [a], q2.
q1 --> [a], q3.
q2 --> [b], q4.
q2 --> [].
q3 --> [b], q5.
q3 --> [].
q4 --> [a], q2.
q5 --> [b], q6.
q6 --> [a], q3.
```

Usage des grammaires

Pour la description d’un langage de programmation



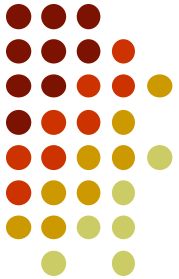
Exemple : grammaire sans retour arrière d'un petit langage de programmation

```
programme

lexique

entier fonction MAX(entier A,B)
/* MAX(A,B) désigne le MAX de A et de B */

   lexique de MAX
     entier M;
   algorithme de MAX
     si A < B alors M := B sinon M := B
     fsi
     renvoyer(M)
ffonction

   entier A,B;
   entier PGCD;

algorithme
      A := 20;
      B := 50;
      tantque A != B faire
           si MAX(A,B) = B
           alors B := B-A
           sinon A := A-B
           fsi
       fait
       PGCD := A

fprogramme
```

Exemple : grammaire sans retour arrière d'un petit langage de programmation

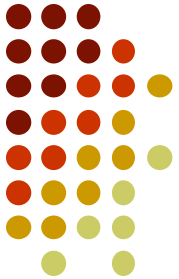
```
programme --> prog, le_lexique, l_algorithme, fprogramme, ff.

le_lexique --> lexique, liste_declarations.

liste_declarations --> declaration, !, liste_declarations.
liste_declarations --> [].

declaration --> type_variable, fin_declaration.

fin_declaration --> identificateur, !, liste_idf, ptvirg.
fin_declaration --> fonction, identificateur, liste_param_formels,
                    corps_fonction, ffonction.

type_variable --> entier ; reel; caractere ; booleen ; chaine.

liste_idf --> virg, !, identificateur, liste_idf.
liste_idf --> [].
```

Exemple : grammaire sans retour arrière d'un petit langage de programmation



```
l_algorithme --> algorithme, liste_instructions.

liste_instructions --> instruction, fliste_instructions.

fliste_instructions --> instruction, fliste_instructions.
fliste_instructions --> [].

instruction --> affectation, !, ptvirg.
instruction --> conditionnelle, !.
instruction --> iteration.

affectation --> identificateur, affect, expression.
affect  --> deuxpts, egal.

conditionnelle --> si, expression, alors, liste_instructions,
                   sinon_opt, fsi.

sinon_opt --> sinon, ! , liste_instructions.
sinon_opt --> [].

iteration --> tantque, !, expression, faire,
              liste_instructions, fait.
```

Exemple : grammaire sans retour arrière d'un petit langage de programmation



```
expression --> expression2, fin_expression.

fin_expression --> op1, !, expression.
fin_expression --> [].

expression2 --> operande, fin_experssion2.

fin_experssion2 --> op2, !, expression2.
fin_experssion2 --> [].

operande --> signe_opt, primaire.

signe_opt       --> op2,!.
signe_opt       --> [].

primaire --> parouvr,!, expression, parferm.
primaire --> constante, !.
primaire --> identificateur, !, liste_param_effectifs_opt.

liste_param_effectifs_opt --> parouvr, !, liste_expr, parferm.
liste_param_effectifs_opt --> [].
```

```
op1 --> [‘<'], !, suit_inf.
op1 --> ['>'], !, suit_sup.
op1 --> [‘='], !.
op1 --> [‘!'], !, [‘='].

suit_inf --> [‘='], !.
suit_inf --> [].
suit_sup --> [‘='], !.
suit_sup --> [].

op2 --> [‘+'],!.
op2 --> [‘-'].
```

Exemple : grammaire sans retour arrière d'un petit langage de programmation

```
liste_expr --> expression, !, fliste_expr.
liste_expr --> [].

fliste_expr --> virg, !, expression, fliste_expr.
fliste_expr --> [].

constante --> constante_bool, !.
constante --> nombre.

constante_bool --> vrai,!.
constante_bool --> faux.
```

Symboles lexicaux

<, >, =, !, +, -,

virg, ptvirg, parouvr, parferm, affect,

vrai, faux, entier, réel, caractère, booléen,

programme, fprogramme, lexique, de, algorithme,

fonction, ffonction, si, alors, sinon, fsi, tantque,
faire, fait, retour,

constante, identificateur

Syntaxe de langages disponibles sur le web

- Langage Java :

http://cui.unige.ch/isi/bnf/JAVAF/

- Langage C :

http://cermics.enpc.fr/polys/info-96/node94.html