PROLOG
- Applications -

1) **Analyse Syntaxique — DCG**

- Transformation de grammaires algébriques en Prolog
- Les DCGs
  - Définitions — Exemples
  - Ecriture en Prolog

2) **Méta-programmation**

- Evaluateurs simples
- Adaptation de la résolution (Contrôle de la profondeur de la recherche, Recherche en largeur d'abord)
- Compilation

APPLICATIONS
— ANALYSE SYNTAXIQUE & DCG —

- **Prolog** a été conçu pour
  l'analyse du langage naturel

- Le ***formalisme des clauses de horn***
  est très proche de celui des ***grammaires "context free"***

ANALYSE SYNTAXIQUE & DCG
— TRANSFORMATION DE GRAMMAIRES ALGEBRIQUES EN PROGRAMMES PROLOG —

Une Grammaire algébrique simple	**Programme prolog correspondant**
phrase → *groupe_nominal, groupe_verbal*	**phrase(S-S0)** :– groupe_nominal(**S-S1**), groupe_verbal(**S1-S0**).
groupe_nominal → *prep_article, groupe_nominal2*	**groupe_nominal(S-S0)** :– prep_article(**S-S1**), groupe_nominal2(**S1-S0**).
groupe_nominal → *groupe_nominal2*	groupe_nominal(**S**) :– groupe_nominal2(**S**) .
groupe_nominal2 → *adjectif, groupe_nominal2*	**groupe_nominal2(S-S0)** :– adjectif(**S-S1**), groupe_nominal2(**S1-S0**).
groupe_nominal2 → *nom*	groupe_nominal2(**S**) :– nom(**S**).
groupe_verbal → *verbe*	**groupe_verbal(S)** :– verbe(**S**).

groupe_verbal → *verbe, groupe_nominal*	groupe_verbal(**S-S0**) :– verbe**(S-S1),** groupe_nominal**(S1-S0).**
prep_article → *[le]*	**prep_article**([le|S]-S).
prep_article → *[un]*	prep_article([un|S]-S).
nom → *[os]*	**nom**([os|S]-S).
nom → *[plat]*	nom([plat|S]-S).
verbe → *[contient]*	**verbe**([contient|S]-S).
adjectif → *[beau]*	**adjectif**([beau|S]-S).

ANALYSE SYNTAXIQUE & DCG
— TRANSFORMATION DE GRAMMAIRES ALGEBRIQUES EN PROGRAMMES PROLOG — (suite)

- **SCHEMA DE TRADUCTION**

  Symbole non terminal → Clause unaire

  Vocabulaire → Faits

  Les ***listes de différence*** représentent la partie connue de la phrase à analyser.

  Exemple :

```
le beau plat   contient   un    os
↑              ↑                ↑
S              S1               S0
```

- **AMELIORATION POSSIBLE :** calcul simultané de l'arbre de syntaxe abstraite

ANALYSE SYNTAXIQUE
— LES DCGs —

♦ **DEFINITION**

- **Extension** des grammaires algébriques
- **Exécutables** (Classe de programmes Prolog)

♦ **EXTENSIONS**

- Ajout d'arguments pour définir la structure de
  l'arbre d'analyse
- Spécification de propriétés (e.g., accord en nombre)

Implantation aisée en Prolog grâce à l'unification

ANALYSE SYNTAXIQUE
— LES DCGs : EXEMPLES —

Une DCG calculant l'arbre d'analyse

phrase(ph(N,V))	→	groupe_nominal(N) , groupe_verbal(V)
groupe_nominal(gn(PA,N))	→	prep_article(PA), groupe_nominal2(N)
groupe_nominal(gn(N))	→	groupe_nominal2(N)
groupe_nominal2(gn2(A,N))	→	adjectif(A), groupe_nominal2(N)
groupe_nominal2(gn2(N))	→	nom(N)
groupe_verbal(gv(V))	→	verbe(V)
groupe_verbal (gv(V,N))	→	verbe(V), groupe_nominal(N)

prep_article(pa(le))	→	[le]
prep_article(pa(un))	→	[un]
nom(nom(os))	→	[os]
nom(nom(plat))	→	[plat]
verbe(verbe(contient))	→	[contient]
adjectif(adjectif(beau))	→	[beau]

ANALYSE SYNTAXIQUE
— LES DCGs : EXEMPLES —

Programme Prolog calculant l'arbre d'analyse

```
phrase(ph(N,V),S-S0)                :- groupe_nominal(N,S-S1), groupe_verbal(V,S1-S0).

groupe_nominal(gn(PA,N),S-S0) :- prep_article(PA,S-S1), groupe_nominal2(N,S1-S0).
groupe_nominal(gn(N),S)             :- groupe_nominal2(N,S) .

groupe_nominal2(gn2(A,N),S-S0) :- adjectif(A,S-S1),  groupe_nominal2(N,S1-S0).
groupe_nominal2(gn2(N),S)           :- nom(N,S).

groupe_verbal(gv(V),S)              :- verbe(V,S).
groupe_verbal(gv(V,N),S-S0)         :- verbe(V,S-S1),  groupe_nominal(N,S1-S0).
prep_article(pa(le),[le|S]-S).      prep_article(pa(un),[un|S]-S).
nom(nom(os),[os|S]-S).              nom(nom(plat),[plat|S]-S).
verbe(verbe(contient),[contient|S]-S).   adjectif(adjectif(beau),[beau|S]-S).
```

ANALYSE SYNTAXIQUE
— LES DCGs : EXEMPLES —

Une DCG avec accord du sujet et de l'objet

phrase(ph(N,V),Nb) → groupe_nominal(N,Nb) , groupe_verbal(V,Nb)
groupe_nominal(gn(PA,N),Nb) → prep_article(PA,Nb), groupe_nominal2(N,Nb)
groupe_nominal(gn(N),Nb) → groupe_nominal2(N,Nb)
groupe_nominal2(gn2(A,N),Nb)→ adjectif(A,Nb), groupe_nominal2(N,Nb)
groupe_nominal2(gn2(N),Nb) → nom(N,Nb)
groupe_verbal(gv(V),Nb) → verbe(V,Nb)
groupe_verbal (gv(V,N),Nb) → verbe(V,Nb) , groupe_nominal(N,Nb)
prep_article(pa(le),sing) → [le] prep_article(pa(des),plur) → [des]
prep_article(pa(un),sing) → [un]
nom(nom(os),Nb) → [os] nom(nom(plat),sing) → [plat]
nom(nom(plats),plur) → [plats]
verbe(verbe(contient),sing) → [contient]
verbe(verbe(contiennnent),plur) → [contiennnent]
adjectif(adjectif(beau),sing) → [beau]
adjectif(adjectif(beaux),plur) → [beaux]

Definite clause grammars

- Notations-

- A convenient notation for expressing grammar rules
- An extension of the well-known context-free grammars.
- A grammar rule in Prolog takes the general form

***head --> body*.**

DCG notation implements a single implicit accumulator

Definite clause grammars extend context-free grammars

- ***A non-terminal symbol*** : any Prolog term (other than a variable or number).
- ***A terminal symbol*** : any Prolog term.

  To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list **[]** or **“”** .
- ***Extra conditions***, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in **{}** brackets.
- ***The left-hand side*** of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
- ***Disjunction*** and not-provable may be stated explicitly in the right-hand side of a grammar rule, using the **;** and **\+** as in a Prolog clause.
- ***The cut symbol*** may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in **{}** brackets.

DCG : A simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}. % C is the character code of some digit.
```

The query ?- expr(Z, "-2+3*5+1", []).

will compute ***Z*=14.**

Grammar rules : "syntactic sugar" for ordinary Prolog clauses

- Grammar rules : a convenient "syntactic sugar" for ordinary Prolog clauses.
- Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion as output for further analysis.
- The arguments required for the input and output strings are not written explicitly in a grammar rule

A rule such as

p(X) --> q(X).

translates into

p(X, S0, S) :- q(X, S0, S). *% S0 : input string*
% S : output string

If there is more than one non-terminal on the right-hand side, as in

p(X, Y) -->
q(X),
r(X, Y),
s(Y).

translates into

p(X, Y, S0, S) :-
q(X, S0, S1),
r(X, Y, S1, S2),
r(Y, S2, S).

Grammar rules : "syntactic sugar" for ordinary Prolog clauses (continued)

Terminals are translated using the built-in predicate

`'C'(S1, X, S2)`

(point S1 is connected by terminal X to point S2")

defined by the single clause

'C'([X|S], X, S).

Example :

p(X) --> [go,to], q(X), [stop].

is translated by

```
p(X, S0, S) :-
        'C'(S0, go, S1),
        'C'(S1, to, S2),
        q(X, S2, S3),
         'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls translate as themselves, e.g.

p(X) --> [X], {integer(X), X>0}, q(X).

translates to

```
p(X, S0, S) :-
        'C'(S0, X, S1),
        integer(X),
        X>0,
        q(X, S1, S).
```

DCG : a general programming technique

Example 1:

Grammar rule:

Tree ::= node(Tree,Tree) | leaf(Data)

DCG :

```
tree(leaf(Data)) --> [Data].

tree(node(L,R)) --> tree(L), tree(R).
```

Prolog Program:

```
tree(leaf(A), B, C) :-
        'C'(B, A, C).

tree(node(A,B), C, D) :-
        tree(A, C, E),
        tree(B, E, D).
```

Question:

?- T= node(node(leaf(a),leaf(b)),leaf(c)), tree(T,A,B).

A = [a,b,c|B]

Remark: tree/3 generates a list (‘’reverse’’ of a parsing)

DCG : a general programming technique (continued)

Example 2:

Grammar rule:

List ::= cons(E,List) | []

DCG :

```
literal([])--> [].

literal([X|Xs])--> [X], literal(Xs).
```

Prolog Program:

```
literal([], A, B) :-
    B=A.

literal([A|B], C, D) :-
        'C'(C, A, E),
        literal(B, E, D).
```

Questions:

?- literal(A, [a,b,c], D).

```
A = [],          D = [a,b,c] ? ;
A = [a],         D = [b,c] ? ;
...
A = [a,b,c],     D = [] ?
no
```

?- literal([a,b], C, [c,d,e]).

```
C = [a,b,c,d,e] ? ;
```

DCG : a general programming technique (continued)

Example 3:

DCG :

```
len([]) --> [0].
len([_|L]) --> [1],len(L).
```

Prolog Program:

```
len([], A, B) :-
'C'(A, 0, B).                    % We must redefine 'C' :   'C'(In, N, Out):-  Out is In+N.
len([_|A], B, C) :-
        'C'(B, 1, D),
        len(A, D, C).
```

Question:

?- len([a,b],C,D).

C = [1,1,0|D] ?

APPLICATIONS
— META-PROGRAMMATION —

ECRITURE D'EVALUATEURS SIMPLES

- Méta-interprète Prolog Pur

```
solve(true).
solve((A,B)) :-
        solve(A),
        solve(B).
solve(A):-
        clause(A,B),
        solve(B).
```

- Méta-interprète Prolog Pur (opérationnel!)

```
solve(true).
solve((A,B)) :-
       solve(A),
       solve(B).
solve(A):-
       A \== true, not(A=(A1,A2)),
       clause(A,B),
       solve(B).

% Jeu d'essai
g(X,Y,Z) :-
       g1(X),    g2(Y),       g3(Z).
g(a,a,a).
g1(b). g2(c). g3(d). g3(e).
```

% Requête : `solve(g(X,Y,Z)).`

APPLICATIONS
— ECRITURE D'EVALUATEURS SIMPLES (suite) —

- Méta-interprète Prolog Pur — Affichage de la preuve

```
:- op(400,xfy,<-).

solve(true,true).
solve((A,B),(PreuveA,PreuveB)) :-
    solve(A,PreuveA),
    solve(B,PreuveB).
solve(A,(A '<-'Preuve)):-
    A \== true,
    not(functor(A,',',_)),
    clause(A,B),
    solve(B,Preuve).
```

% Jeu d'essai

```
g(X,Y,Z) :-
    g1(X), g2(Y),     g3(Z).
g(a,a,a).
```

g1(b). g2(c). g3(d). g3(e).

% Requête

```
?-solve(g(X,Y,Z),Preuve).
X = b   Y = c    Z = d
Preuve = g(b,c,d)<-(g1(b)<-true),g2(c)<-
```

true),g3(d)<-true

CONTROLE DE LA PROFONDEUR DE LA RECHERCHE

% Représentation des règles: `règle(signature,postconditions):-préconditions`
% où postcondition représente le nouvel état

```
bk(T):-
       etat_initial(E),
       solve([E],T).
solve([Ec|E],[]):-
       etat_succes(Ec).

solve(E,_):-
       profondeur_max(M),
       longueur(E,L),
       L > M, !, fail.

solve([Ec|E],[R|Chemin]):-
       apply(Liste_regles,R,Ec,REc),
       hors(REc,E),
       solve([REc,Ec|E],Chemin).

apply([[R,REc]|L],R,Ec,REc).
apply([_|Liste],R,Ec,REc):-
        apply(Liste,R,Ec,REc).
```

APPLICATIONS
— CONTROLE DE LA PROFONDEUR DE LA RECHERCHE (suite) —

```
applicables(E,L):-
        /* Sous traite à Prolog la vérification des Préconditions */ assert(etat(E)),
        once(exec([],L)),
        retract(etat(E)).

exec(Liste,L):-
       call(regle(R,REc)),
       \+ member([R,REc],Liste),
       exec([[R,REc]|Liste],L).
exec(L,L).

/* utilitaires */

enum(X,Y,X).
enum(X,Y,Z):- Z > Y,Z1 is Z-1, enum(X,Y,Z1).

longueur([],0).
longueur([X|R],L1):-
       longueur(R,L),L1 is L+1.
once(P):- P,!.
```

APPLICATIONS
— META PROGRAMMATION (suite) —

CHAINAGE AVANT "NAIF"

% Définition des opérateurs utilisés

```
:-op(900,xfy,[et,ou]).
:-op(900,fy,[vrai,nom,non]).
:-op(995,fx,[si,alors,sinon,coef]).
vrai X et Y :-  vrai X, vrai Y.
vrai X ou Y :-  vrai X; vrai Y.
vrai X :- clause(X,true).
vrai non X :- not clause(X,true).
```

```
% Chainage avant: interprête

chainage_avant:-
     cond_terminale.            % Solution trouvée?
chainage_avant:-
   clause(regle(X),
   (si Cond, alors Action)),
   vrai Cond,                   % Pré-conditions satisfaites ?
   not deja_fait(Action),
   exec(Action),                % Exécuter les actions
   chainage_avant.
chainage_avant.                 % Condition terminale =  Saturation
```

APPLICATIONS
— CHAINAGE AVANT "NAIF"(suite) —

% Exécution et contrôle d'exécution des actions élémentaires

```
deja_fait(ajouter(X)):- vrai X.
deja_fait(ecrire(X)):-
   vrai ecriture(X).

exec(ajouter(X)):-
   assert(X).
exec(ecrire(X)):-
  nl,write(X),
  assert(ecriture(X)).
```

% Exemple

```
cond_terminale:-but(X).   %but = fin

regle(nom r1):-
   si p1(X) et p2(X),
   alors ecrire(X).
regle(nom r2):-
   si p1(X) et p3(X) et non p2(X),
        alors ajouter(but(0)).
regle(nom r3):-
   si p2(a) et non p1(b),
     alors ajouter(but(1)).

p1(a).  p1(c).    p2(a).    p3(a).  p3(e).
```