Prolog

Pascal Brisset

Historique

Résolution [11]

Colmerauer-Roussel[2]

kowalski[9]

<table>
<tr><td>Edinburgh
Compilateur, efficacité</td><td>Marseille
Interpréteur, langage</td></tr>
<tr><td>Warren[13]
wam[a] [14, 1]
Quintus, Sicstus, Sepia
NU-Prolog, SWI, Bin-Prolog, ...</td><td>Van Caneghem[5]
Arbres infinis [4]
Prolog II
Prolog IIR[3]</td></tr>
<tr><td colspan="2">Programmation logique avec contraintes [12]</td></tr>
<tr><td>Chip[6], ECL^{i}PS^e[7]</td><td>Prolog III, Prolog IV</td></tr>
</table>

1 Bases logiques

Un programme Prolog est un ensemble de clauses de Horn positives (définies) :

$$H \vee \neg B_1 \vee ... \vee \neg B_n \equiv B_1 \wedge ... \wedge B_n \rightarrow H$$

H, B_1, ..., B_n sont des atomes. Par exemple avec les symboles de prédicat p_0, q_1, r_2 et les symboles fonctionnels a_0 et f_1 : p, $q(a)$, $r(a, f(a))$, $q(f(f(a)))$, ...

On appelle prédicat (p), l'ensemble des clauses qui le définissent, i.e. l'ensemble des clauses dont le symbole de prédicat de tête est p.

Les prédicats vont jouer le rôle de procédures. Les symboles fonctionnels sont les constructeurs de structure de donnée (cf. ML).

Une clause concrète (un fait si $n = 0$) :

$$\underbrace{H}_{\text{tête}} \texttt{ :- } \underbrace{\underbrace{B_1}_{\text{but}} \texttt{, ... ,} B_n}_{\text{corps}} \texttt{.}$$

Une exécution est la démonstration d'une conjonction de buts (la question) par résolution.

Une étape de résolution correspond à un appel de procédure.

Exemple :

```
conc(nil, Y, Y).
conc(cons(A,X), Y, cons(A,Z)) :- conc(X, Y, Z).

?- conc(cons(1, nil), cons(2, nil), R).
--> R = cons(1, cons(2, nil))
```

à comparer avec une version fonctionnelle (typée) :

```
type 'a list = Nil | Cons of 'a * 'a list;;
let rec conc x y =
  match x with
    Nil -> y
  | Cons (a,x) -> Cons (a, conc x y);;

# conc (Cons (1, Nil)) (Cons (2, Nil));;
--> Cons (1, Cons (2, Nil))
```


1.1 sld-résolution

Définition : Soient deux clauses C et C' telles que $A \in C$, $\neg A' \in C'$ et A et A' sont unifiables par σ le plus grand unificateur. $R = \sigma(C \backslash A \cup C' \backslash \neg A')$ est la résolvante de C et C'.

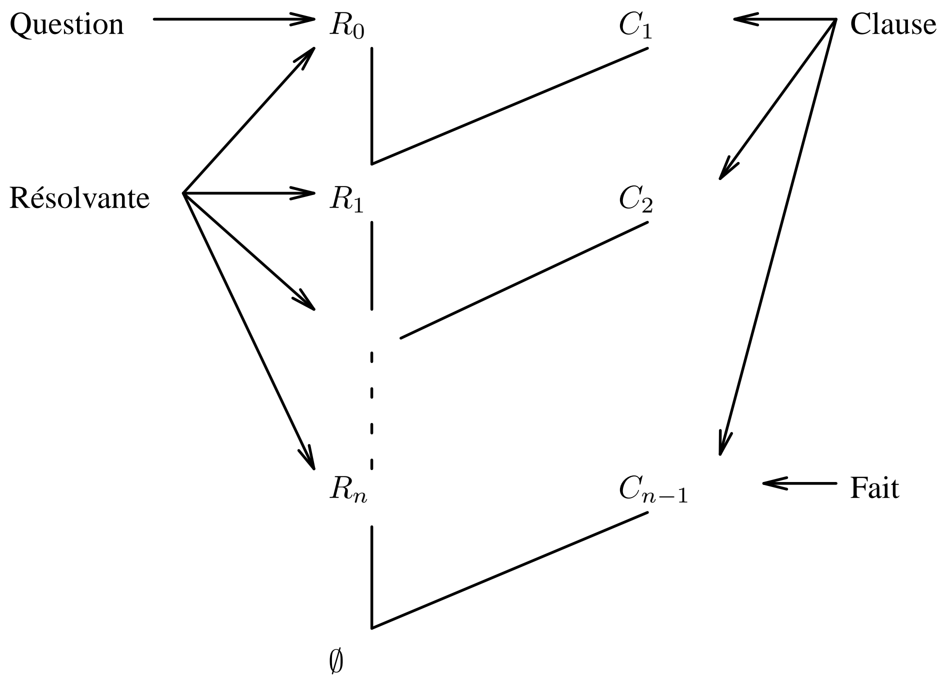
Proposition : $\{C, C'\}$ et $\{C, C', R\}$ sont équivalents

Proposition : Soit E un ensemble de clauses. E est insatisfiable si et seulement si on peut obtenir la clause vide par application répétée du principe de résolution.

Cas particulier de résolution : **S**élection **L**inéaire pour clauses **D**éfinies.

Règle sld : Soient deux clauses C et C' telles que $C = \neg L_1 \vee ... \vee \neg L_q$ et $C' = B_1 \wedge ... \wedge B_p \rightarrow H$ et H et L_1 (ou L_i) sont unifiables par σ le plus grand unificateur. $\neg\sigma L_2 \vee ... \vee \neg\sigma L_q \vee \neg\sigma B_1 \vee ... \vee \neg\sigma B_p$ est la résolvante de C et C'.

Propriété : la SLD-résolution est complète pour les clauses de horn.



1.2 Calcul des séquents intuitionistes

Définition : un séquent est une paire d'ensembles finis de formules: $\Gamma \longrightarrow \Delta$, Γ est l'antécédent, Δ le conséquent.

Dans le cas intuitioniste (i.e. constructif), Δ est un singleton.

Pour Prolog :

- Γ est le programme (ensemble de clauses) ;
- Δ est la question (résolvante).

Système de règles :

$$\frac{}{A, \mathcal{P} \longrightarrow \sigma A}\text{Axiome} \quad \frac{B \rightarrow A, \mathcal{P} \longrightarrow \sigma B}{B \rightarrow A, \mathcal{P} \longrightarrow \sigma A}\text{MP} \quad \frac{\mathcal{P} \longrightarrow A_1 \quad \mathcal{P} \longrightarrow A_2}{\mathcal{P} \longrightarrow A_1 \wedge A_2}\wedge_D$$

La preuve est uniforme (dirigée par le but) : chaque séquent dont le conséquent n'est pas atomique est conclusion d'une règle d'introduction à droite.

Propriété : ce système de règles est complet pour les formules considérées.

Définition générale [10] : un langage de programmation logique est fondé sur une théorie logique pour laquelle la restriction à des preuves uniformes ne perd pas la complétude.

1.3 Contrôle

La stratégie de recherche n'est pas exprimée par la SLD-résolution.

Contrôle ET : c'est toujours le premier littéral de la résolvante qui est sélectionné et remplacé par le corps de la clause choisie.

Contrôle OU : les clauses sont classées par leur ordre d'apparition dans le programme et sont essayées selon cet ordre. L'arbre de recherche est parcouru en profondeur d'abord. Il y a retour-arrière en cas d'échec.

NB : la stratégie de recherche n'est pas complète.

1.4 Syntaxe concrète

program	::= `EOF` | sentence `.` program
sentence	::= clause | directive
directive	::= `:-` goal | decl_op
decl_op	::= `:- op(` precedence `,` associativity `,` name `)`
a precedence	::= integer
associativity	::= `fx` | `fy` | `xfx` | `yfx` | `xfy` | `xf` | `yf`
clause	::= head | head `:-` goals
head	::= atom
goals	::= goals `,` goals | goals `;` goals | atom | `(` goals `)`
atom	::= name | name `(` terms `)`

terms	::= term otherterms
otherterms	::= | `,` subterm(899) terms
term	::= subterm(1200)
subterm(N)	::= term(M) $\{N \geq M\}$
term(N)	::= op(N,`fx`) subterm($N-1$) | op(N,`fy`) subterm(N) |
	::= subterm($N-1$) op(N,`xfx`) subterm($N-1$) |
	::= subterm(N) op(N,`yfx`) subterm($N-1$) |
	::= subterm($N-1$) op(N,`xfy`) subterm(N) |
	::= subterm($N-1$) op(N,`xf`) | subterm(N) op(N,`yf`)
term(0)	::= `(` subterm(1200) `)` | name | name `(` terms `)`
	::= list | string | constant | unknown
op(N,T)	::= name {déclaré opérateur de mode T de précédence N}
list	::= `[ ]` | `[` listexpr `]`
listexpr	::= subterm(999) | subterm(999) `,` listexpr
	::= subterm(999) `|` subterm(999)
constant	::= name | integer | float

name ::= small-letter alpha∗ | symbol-char+ | ' char+ '

integer ::= digit10+ | 0x digit16+ | 0' char

float ::= simple-float | simple-float E exponent

simple-float ::= digit10+ . digit10+

exponent ::= digit10+ | - digit10+ | + digit10+

unknown ::= _alpha∗ | capital-letter alpha∗

variable ::= unknown | name

string ::= "char∗"

comment ::= /* char∗ */ | % char∗ newline

char ::= alpha | symbol-char | solo-char | \ escaped-char

alpha ::= _ | small-letter | capital-letter | digit10

symbol-char ::= {any of} =:#$&*+-./<>?@\^`~

solo-char ::= {any of} ;!()[]{}|'"

escaped-char ::= {any of} ntbrf\'"

2 Prolog par l'exemple

2.1 Base de données

Deux prédicats :

- `pere(X,Y)` exprimant « `X` est le père de `Y` » ;
- `mere(X,Y)` exprimant « `X` est la mère de `Y` » ;

```
pere(jean, paul). pere(paul, martin). pere(paul, marie).
mere(patricia, martin). mere(marie, luc).
```

On aurait aussi pu représenter avec `"jean"`, `"paul"` ... (préférable).

```
?- pere(paul, martin).
 yes
?- pere(jean, marie).
 no
```

Prédicat parent :

$$\begin{aligned} & \forall x \forall y (pere(x,y) \vee mere(x,y)) \rightarrow parent(x,y) \\ \equiv\ & \forall x \forall y (pere(x,y) \rightarrow parent(x,y)) \wedge (mere(x,y) \rightarrow parent(x,y)) \end{aligned}$$

Concrètement :

```
parent(X, Y) :- pere(X,Y) ; mere(X,Y).
```

ou

```
parent(X, Y) :- pere(X,Y).
parent(X, Y) :- mere(X,Y).
```

De même pour grand-père :

```
grand_pere(X, Y) :- pere(X, Z), parent(Z, Y).
```

Développer l'arbre de recherche et un arbre de preuve pour la question `grand_pere(paul, luc).`

Si la question contient des variables, Prolog calcule la substitution correspondant à chaque solution :

```
?- pere(paul, X).
  X = martin ;
  X = marie
```

```
?- grand_pere(X, marie).
?-  -> pere(X, Y), parent(Y, marie).
?-  -> parent(paul, marie) (X = jean)
?-  -> pere(paul, marie)
?-  -> []
  X = jean
```

Prédicat récursif :

```
ancetre(X, X).
ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y).
?- ancetre(X, luc).
```

2.2 Termes composés

Arithmétique de Peano : `z`, `s(z)`, `s(s(z))`, `...`

Successeur : $successeur(x, y)$ ssi « y est le successeur de x dans $\mathbb{N}$ »

```
successeur(X, s(X)).
?- successeur(s(z), s(s(z))).
 yes.
?- successeur(z, U).
 U = s(z)
?- successeur(X, s(z)).
 U = z
```

Addition : $plus(x, y, z)$ ssi « z est la somme de x et y »

```
plus(z, Y, Y).
plus(s(X), Y, s(Z)) :-
  plus(X, Y, Z).
```



```
?- plus(s(z), s(z), Z).
 Z = s(s(z))
?- plus(s(z), Y, s(s(z))).
 Y = s(z)
?- plus(X, s(z), s(s(z))).
 X = s(z)
?- plus(X, Y, s(s(z))).
 X = z, Y = s(s(z)) ;
 X = s(z), Y = s(z) ;
 X = s(s(z)), Y = z
```

Prolog est qualifié de « déclaratif » et « non-déterministe ».

Cependant :

- on n'écrit pas toujours des programmes déclaratifs ;
- la recherche en profondeur d'abord et la sélection gauche-droite perdent la complétude.

2.3 Arithmétique

La syntaxe avec des symboles infixes :

- prédicatifs : =, <, is, ...
- fonctionnels : +, *, ...

Attention, les termes ne sont jamais évalués :

```
?- 2 = (1+1).
 no
```

Le prédicat `is` évalue les expressions arithmétiques : $is(x, y)$ ssi « y est une expression arithmétique et x est unifiable avec l'évaluation de y »

```
?- 2 is (1+1).
 yes
?- X is 2*2.
 X = 4
```

D'où l'écriture de prédicats de calcul :


```
fact(0, 1).
fact(N, F) :-
  N > 0, N1 is N - 1,
  fact(N1, F1), F is F1 * N1.

ack(0, N, N1) :- N1 is N + 1.
ack(M, 0, Ack) :- M > 0, M1 is M - 1,
  ack(M1, 1, Ack).
ack(M, N, Ack) :- M > 0, N > 0, M1 is M - 1, N1 is N - 1,
  ack(M, N1, Ack1), ack(M1, Ack1, Ack).
```

Attention, ce prédicat n'est pas réversible :

```
?- 6 is X * 2.
instantiation fault in *(X, 2, 6)
```

2.4 Listes

Comme en programmation fonctionnelle, la liste est une structure de donnée basique de Prolog et admet une syntaxe ad hoc :

- `[]` la liste vide ;
- `[Head | Tail]` le cons ;
- `[1, 2, "trois"]` une liste (non typée) de 3 éléments ;
- `[1, 2 | Tail]` une liste d'au moins deux éléments.

Le style de programmation ressemble au style fonctionnel mis à part que l'on écrit des relations : à une fonction f telle que $\forall x \exists y \; y = f(x)$, correspond une relation r telle que $\exists x \exists y \; r(x, y)$.

```
% append(L1, L2, L3) L3 est la concaténation de L1 et L2
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Une bonne illustration du non-déterminisme de Prolog :


```
?- append(X, Y, [1,2]).
X = []
Y = [1, 2]
X = [1]
Y = [2]
X = [1, 2]
Y = []
```

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

NB : une relation est une fonction booléenne donc ne pas écrire :

```
% member(X, Xs, R) R = true si X appartient à Xs
member(_, [], false).
member(X, [X | _], true).
member(X, [_ | Xs], R) :- member(X, Xs, R).
```

Deux versions du renversement et calcul de la longueur (non-réversible) :

```
nrev([], []).
nrev([X | Xs], R) :- nrev(Xs, Rs), append(Rs, [X], R).

reverse(Xs, Rs) :- rev(Xs, [], Rs).
rev([], R, R).
rev([X | Xs], As, R) :- rev(Xs, [X | As], R).
```



```
length([], 0).
length([X | Xs], N) :- length(Xs, N1), N is N1 + 1.
```

Recherche dans une liste d'associations (liste de couples clé-valeur) :

```
assoc(Key, [(Key, Value) | _], Value).
assoc(Key, [_ | Assocs], Value) :- assoc(Key, Assocs, Value).
```

Mais ce n'est qu'un cas particulier de `member` :

```
assoc(Key, Assocs, Value) :- member((Key, Value), Assocs).
```

2.5 Arbres

La représentation d'un arbre en Prolog est naturelle puisque les termes manipulés sont des arbres.

Représentation d'expressions arithmétiques : on choisit de coder $1 + 2 * x$ par le terme `plus(int(1), mult(int(2), var("x")))`. D'où par exemple le prédicat d'évaluation d'une expression :

```
% eval(Expression, VarsValues, Value)
eval(int(N), _, N).
eval(var(Var), VarsValues, Val) :-
  assoc(Var, VarsValues, Val).
eval(plus(E1, E2), VarsValues, Val) :-
  eval(E1, VarsValues, Val1), eval(E2, VarsValues, Val2),
  Val is Val1 + Val2.
...
```


Représentation d'arbres n-aires :

- `leaf(Label)` pour une feuille ;
- `node(Label, Sons)` pour un nœud ;
- `[Son1, Son2, ...]` pour les fils d'un nœud.

Ce qui donne pour le calcul de l'ensemble des feuilles d'un arbre :

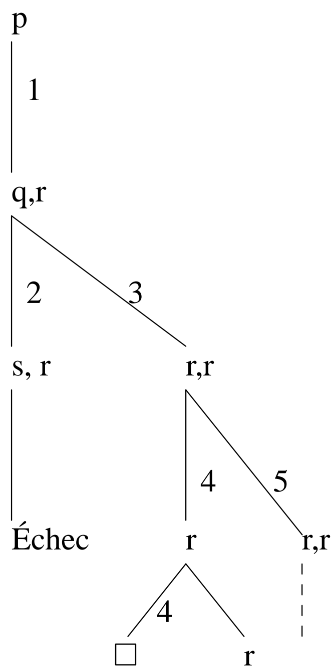
```
leaves(leaf(Label), [Label]).
leaves(node(_Label, Sons), Labels) :-
  leaves_list(Sons, Labels).

leaves_list([], []).
leaves_list([Tree | Trees], Labels) :-
  leaves(Tree, L),
  leaves_list(Trees, Ls),
  append(L, Ls, Labels).
```

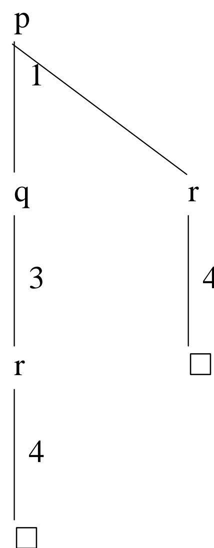
2.6 Contrôle

Les structures de contrôle de Prolog, dues aux choix de stratégie ET et OU, peuvent être rapprochées de celles des langages classiques :

- appel de procédure ;
- conditionnelle : `(test, alors ; sinon)` ;
- séquence :
  - avec un ET : `(p1 , p2)` ;
  - avec un OU : `(p1 ; p2)` (plusieurs solutions) ;
- boucle : par récursivité.



Arbre de recherche



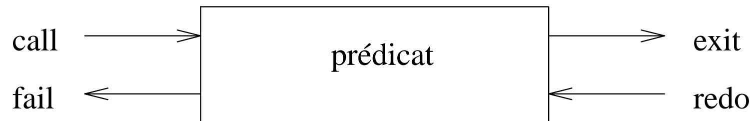
Arbre de preuve

```
p :- q,r.
q :- s.
q :- r
r.
r :- r.
?- p.
```

2.7 Boîtes d'activation

C'est le modèle d'exécution utilisé par l'outil de trace des systèmes Prolog.

À chaque but est associé une boîte munie de portes par lesquelles passe le flux d'exécution.




```
arc(a,b).
arc(b,c).
arc(a,d).
chemin(U,U).
chemin(U,W) :- arc(U,V), chemin(V,W).
```



```
?-  chemin(a, Y).
  (1) 0  CALL   chemin(a, Y) (dbg)?- creep
  (1) 0 *EXIT   chemin(a, a) (dbg)?- creep
Y = a     More? (;)
  (1) 0  REDO   chemin(a, Y) (dbg)?- creep
  (2) 1  CALL   arc(a, V) (dbg)?- creep
  (2) 1 *EXIT   arc(a, b) (dbg)?- creep
  (3) 1  CALL   chemin(b, Y) (dbg)?- creep
  (3) 1 *EXIT   chemin(b, b) (dbg)?- creep
  (1) 0 *EXIT   chemin(a, b) (dbg)?- creep
Y = b     More? (;)
  (1) 0  REDO   chemin(a, Y) (dbg)?- creep
  (3) 1  REDO   chemin(b, Y) (dbg)?- creep
  (4) 2  CALL   arc(b, V) (dbg)?- creep
  (4) 2  EXIT   arc(b, c) (dbg)?- creep
  (5) 2  CALL   chemin(c, Y) (dbg)?- creep
  (5) 2 *EXIT   chemin(c, c) (dbg)?- creep
  (3) 1 *EXIT   chemin(b, c) (dbg)?- creep
  (1) 0 *EXIT   chemin(a, c) (dbg)?- creep
Y = c     More? (;)
```

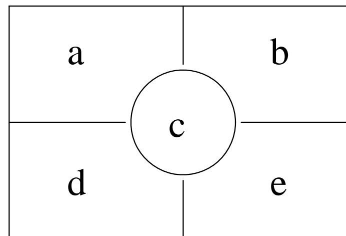
2.8 Générer & tester

Technique de résolution (cf problèmes NP) :

- fabriquer une configuration ;
- vérifier que c'est une solution.

Pour programmer ce paradigme en Prolog, il suffit de décrire ce qu'est une solution.

Exemple : coloriage avec 3 couleurs de la carte suivante :




```
carte(A, B, C, D, E) :-
  couleur(A), couleur(B), couleur(C), couleur(D), couleur(E),
  distinct(A, B), distinct(A, C), distinct(A, D), ...

couleur(jaune).
couleur(vert).
couleur(rouge).

distinct(jaune, vert).
distinct(jaune, rouge).
...
```

ou encore

```
carte(A, B, C, D, E) :-
  distinct(A, B), distinct(A, C), distinct(A, D), ...
```

2.9 Coupure et négation par l'échec

La coupure (`!`, cut) permet de modifier le contrôle OU : utilisée en position de but, elle supprime toutes les alternatives laissées depuis la sélection de la clause :

```
p(1, Y, Z) :- q(Y), !, r(Z).
p(2, 3, 3).
q(1). q(2).
r(1). r(2).
?- p(X, Y, Z).
 X = 1, Y = 1, Z = 1 ;
 X = 1, Y = 1, Z = 2
```

Les points de choix laissés lors de la sélection de la clause et lors de la résolution du but `q` sont supprimés. Ce qui suit la coupure n'est pas altéré.

La coupure permet de programmer une pseudo-négation, la négation par l'échec :

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

qui peut se lire « `distinct(X,Y)` est valide si on ne peut pas montrer l'égalité `X = Y` ».

Prolog autorise le passage de buts en paramètre (ordre supérieur), donc on peut écrire une négation par l'échec générale :

```
not(Goal) :- Goal, !, fail.
not(_Goal).
distinct(X, Y) :- not(X = Y).
```

Attention, la négation n'est correcte que si le but est clos :

```
?- not(X = 1), X = 2.
 no
```

On peut utiliser la coupure (coupure verte) pour améliorer l'efficacité sans changer la sémantique d'un programme :

```
homme(jean). homme(paul). ...
femme(anne). femme(juliette). ...
code_ss(X, 2) :- femme(X), !.
code_ss(X, 1) :- homme(X), !.
```

L'usage des autres coupures (rouges) est délicat. Comparer :

```
mini1(X, Y, X) :- X < Y.
mini1(X, Y, Y) :- X >= Y.
mini2(X, Y, X) :- X < Y, !.
mini2(X, Y, Y).
mini3(X, Y, Z) :- X < Y, !, Z = X.
mini3(X, Y, Y).
?- mini(2, 3, 3).
```


2.10 Listes incomplètes (*en différence*)

L'idée de base est de remplacer le « nil » de fin de liste par une variable ; cette variable étant unifiable avec n'importe quoi, il est possible d'ajouter des éléments en fin de liste.

On représente la séquence d'éléments (eg. 1,2,3) par la liste et la variable en fin de liste : `dl([1,2,3 | Tail], Tail)` ou plus classiquement à l'aide d'un opérateur infixe `[1,2,3|Tail]-Tail`. La liste vide s'écrit `X-X`.

La concaténation pour une liste en différence est immédiate :

```
dappend(A-DA, B-DB, C-DC) :- C = A, DA = B, DB = DC.
```

ou encore

```
dappend(A-DA, DA-DB, A-DB).
```

NB : Une D-liste ne peut être allongée qu'une fois.

```
dl2l(X-DX, X) :- DX = []. l2dl(X, Y-DY) :- append(X, DY, Y).
```

En transformant un algorithme effectuant des concaténations de listes en utilisant des D-listes, il est possible d'améliorer les complexités (analogue à une transformation par continuations pour un langage fonctionnel).

Pour le renversement, on remplace la liste fabriquée par une D-liste :

```
nrev([], []).
nrev([X | Xs], R) :- nrev(Xs, Rs), append(Rs, [X], R).
```

```
rev([], X-X).
rev([X | Xs], R) :- rev(Xs, Rs), dappend(Rs, [X | Tail]-Tail, R).
```

On évalue partiellement la deuxième clause en supprimant le `dappend` :

```
rev([X | Xs], R) :-
  rev(Xs, Rs),
  Rs = A-DA, [X | Tail]-Tail = DA-DB, R = A-DB.
```

puis en simplifiant on reconnaît le renversement non naïf (cf. 2.4) :

```
rev([X | Xs], A-DB) :- rev(Xs, A-[X | DB]).
```


La transformation peut-être appliquée avec succès à de nombreux programmes :

```
leaves(leaf(Label), [Label | Tail]-Tail).
leaves(node(_Label, Sons), Labels) :-
  leaves_list(Sons, Labels).

leaves_list([], X-X).
leaves_list([Tree | Trees], Labels-Labels0) :-
  leaves(Tree, Labels-Labels1),
  leaves_list(Trees, Labels1-Labels0).
```

2.11 Méta-interpréteur[8]

En Prolog, l'expression du contrôle (clauses) et le codage des données (termes) étant identique, il est facile de plonger l'un dans l'autre, i.e. de manipuler un programme Prolog en Prolog (réification). Une utilisation standard de cette facilité est l'écriture d'un interpréteur Prolog.

On représente le programme à l'aide d'un prédicat `clause/1` :

```
clause(append([], Y, Y) :- call(true)).
clause(append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs)).
clause(nrev([], []) :- call(true)).
clause(nrev([X | Xs], R) :- nrev(Xs, Rs), append(Rs, [X], R)).
clause(coucou :- call(print("coucou"))).
...
```

L'ordre des clauses `clause/1` est celui que l'on désire pour nos clauses objets.
On aurait pu choisir (préférable) :

```
clauses([(append([], Y, Y) :- call(true)), (append([X | Xs], Ys, ...
```


La stratégie standard de Prolog se définit alors en quelques lignes (autant que de règles de calcul, cf. 1.2) :

```
solve(call(B)) :- B.
solve((B1, B2)) :- solve(B1), solve(B2).
solve(B) :- clause(B :- C), solve(C).
```

Le contrôle du méta-interpréteur utilise directement le contrôle de l'interpréteur : le ET (`,`) d'un terme devient le ET (`,`) d'une formule ; le OU des clauses méta-interprétées est codé avec le OU des clauses interprétées.

On va expliciter ce contrôle pour pouvoir le modifier. Le contrôle ET est explicité grâce à une continuation de succès, une pile de buts à résoudre (les appels récursifs sont alors terminaux) :

```
solve1([]).
solve1([call(B) | Success]) :- B, solve1(Success).
solve1([(B1, B2) | Success]) :- solve1([B1, B2 | Success]).
solve1([B | Success]) :- clause(B :- C), solve1([C | Success]).
```

Le contrôle OU est explicité grâce à une continuation d'échec, une liste de continuations de succès à résoudre (substitutions de variables non gérées) :

```
solve2([[] | Failure]) :- get(0';), solve2(Failure).
solve2([[call(B) | Success] | Failure]) :-
  B, solve2([Success | Failure]).
solve2([[(B1, B2) | Success] | Failure]) :-
  solve2([[B1, B2 | Success] | Failure]).
solve2([[B | Success] | Failure]) :-
  findall(C, clause(B :- C), Cs),
  add_on_failure2(Cs, Success, Failure, NewFailure),
  solve2(NewFailure).

add_on_failure2([], _, Failure, Failure).
add_on_failure2([C | Cs], Success, Failure, NewFailure) :-
  add_on_failure2(Cs, Success, Failure, Failure0),
  NewFailure = [ [C | Success] | Failure0].
```


Non utilisable à cause de sa complexité exponentielle en espace, une stratégie de recherche en largeur a cependant l'avantage d'être complète. Or, on remarque que « la pile est au parcours en profondeur ce que la file est au parcours en largeur ».

Pour modifier le contrôle du méta-interpréteur, il suffit donc d'abstraire la structure de donnée en remplaçant toutes les occurrences de liste par des appels prédicatifs (à définir ultérieurement).

```
solve4(Failure) :-
  remove_f(Empty, Failure, F0), empty_s(Empty),
  get(0';), solve4(F0).
solve4(Failure) :-
  remove_f(Success, Failure, F0), remove_s(call(B), Success, S0),
  B,
  add_f(S0, F0, F1), solve4(F1).
```

```
solve4(Failure) :-
  remove_f(Success, Failure, F0), remove_s((B1, B2), Success, S0),
  add_s(B2, S0, S1), add_s(B1, S1, S2),
  add_f(S2, F0, F1), solve4(F1).
solve4(Failure) :-
  remove_f(Success, Failure, F0), remove_s(B, Success, S0),
  findall(C, clause(B :- C), Cs),
  add_on_failure4(Cs, S0, F0, NewFailure),
  solve4(NewFailure).

add_on_failure4([], _, Failure, Failure).
add_on_failure4([C | Cs], Success, Failure, NewFailure) :-
  add_on_failure4(Cs, Success, Failure, Failure0),
  add_s(C, Success, S1), add_f(S1, Failure0, NewFailure).
```


Les structures de données abstraites pile et file se définissent comme suit :

```
empty_stack([]).
add_stack(X, Stack, [X | Stack]).
remove_stack(X, [X | Stack], Stack).

empty_fifo([]).
add_fifo(X, Fifo, NewFifo) :- append(Fifo, [X], NewFifo).
remove_fifo(X, [X | Fifo], Fifo).
```

Grâce à l'utilisation de deux files pour les contrôles ET et OU (`add_s` = `add_fifo`, ..., `add_f` = `add_fifo`, ...), les programmes suivants sont exécutables ... sans boucler :

```
p :- call(print(1)), p.
p :- call(print(2)).
q :- call(print(1)), q.
q :- q, call(print(2)), fail.
```

La complétude de la recherche peut être également obtenue par une recherche par approfondissement successifs (iterative deepening). Cette solution a l'avantage d'être de complexité linéaire en espace.

```
solve5(_, Depth, Max) :- Depth > Max, !, fail.
solve5(true, _, _).
solve5(call(B), _, _) :- B.
solve5((B1, B2), Depth, Max) :-
  solve5(B1, Depth, Max), solve5(B2, Depth, Max).
solve5(B, Depth, Max) :-
  D1 is Depth + 1,
  clause(B :- C), solve5(C, D1, Max).

s5(B) :- loop5(B, 1).
loop5(B, Depth) :- solve5(B, 0, Depth).
loop5(B, Depth) :- D1 is Depth + 1, loop5(B, D1).
```


2.12 Coroutines

Du point de vue déclaratif, il n'est pas nécessaire que le ET corresponde à une séquence. Certains systèmes Prolog permettent de modifier ce contrôle en « suspendant » des buts qui sont « réveillés » sous certaines conditions. Cela peut s'apparenter à une évaluation multi-threads. On parle de « coroutining ».

L'usage principal des coroutines concerne l'évaluation de buts dont les arguments ne sont pas suffisamment instanciés :

```
delay(Term, Goal)
```

suspend le but `Goal` qui sera réveillé dès qu'une des variables de `Term` sera instanciée.

C'est le mécanisme de base pour l'exécution en programmation logique avec contraintes.

Suspension de l'évaluation d'une expression non close.

```
:- op(700, xfx, careful_is). % Déclaration d'un opérateur infixe
Val careful_is Expr :-
  nonground(Expr), !, delay(Expr, Val careful_is Expr).
Val careful_is Expr :- Val is Expr.

?-  V careful_is X + Y.
 V = V,  X = X,  Y = Y
 Delayed goals: V careful_is X + Y
?-  V careful_is X + Y, X = 1.
 V = V, Y = Y, X = 1
 Delayed goals: V careful_is 1 + Y
?-  V careful_is X + Y, X = 1, Y = 5.
 V = 6, X = 1, Y = 5
```


Correction de la négation par l'échec.

```
:- op(900, fy, correct_not).
correct_not Goal :-
  nonground(Goal), !, delay(Goal, correct_not Goal).
correct_not Goal :- not Goal.

?-  not X = 1.
 no (more) solution.
?-  correct_not X = 1.
 X = X
 Delayed goals: correct_not X = 1
?-  correct_not X = 1, X = 2.
 X = 2
```

Crible d'Ératosthène : calcul sur des listes infinies.

```
primes :-
  delay(List, print_primes(List)),
  integers(2,List).

integers(N, [N | Ns]) :- N1 is N + 1, integers(N1, Ns).

print_primes([N | List]) :-
  print(N), nl,
  delay(List, sieve(N, List, Newlist)),
  delay(Newlist, print_primes(Newlist)).

sieve(N, [M | List], Newlist) :-
  M mod N =:= 0, !, delay(List, sieve(N, List, Newlist)).
sieve(N, [M | List], [M | Newlist]) :-
  delay(List, sieve(N, List, Newlist)).
```


Par évaluation paresseuse : une liste est représentée soit par elle-même (`[X | Xs]`), soit par un but qui la calcule (`List-Goal`).

```
primes2 :- integers2(2,List), print_primes2(List).

integers2(N, [N | List-integers2(N1, List)]) :- N1 is N + 1.

print_primes2(List-Goal) :-
  Goal, print_primes2(List).
print_primes2([N | List]) :-
  print(T), nl, sieve2(N, List, Newlist), print_primes2(Newlist).

sieve2(N, List-Goal, Newlist) :-
  Goal, sieve2(N, List, Newlist).
sieve2(N, [M | List], Newlist) :-
  M mod N =:= 0, !, sieve2(N, List, Newlist).
sieve2(N, [M | List], [M | Newlist-sieve2(N, List, Newlist)]).
```

2.13 Analyse grammaticale

2.13.1 Traduction directe

Une grammaire non récursive gauche se traduit directement en un programme Prolog.

```
expr  ::= expr1 + expr
      ::= expr1
expr1 ::= expr2 * expr1
      ::= expr2
expr2 ::= 0 | 1 ...
      ::= ( expr )
```

À un non terminal N (e.g. expr), on associe un prédicat `n` (e.g. `expr`) d'arité 1 tel que (`n(L)`) est vrai si `L` est une liste de tokens vérifiant N.

IENAC-S Prolog Prolog par l'exemple

```
expr(L) :-
  expr1(L1), expr(L2), append(L1, [0'+ | L2], L).
expr(L) :-
  expr1(L).
expr1(L) :-
  expr2(L1), expr1(L2), append(L1, [0'* | L2], L).
expr1(L) :-
  expr2(L).
expr2([0'0]).
expr2([0'1]).
...
expr2([0'( | L]) :-
  expr(L1), append(L1, [0')], L).
```

Essais erreurs : fortement non déterministe et inefficace.

2.13.2 Utilisation de listes en différence

`(n(L-DL))` est vrai si `DL` est un suffixe de `L` et si N correspond aux tokens de `L` sauf ceux de `DL`.

```
expr(L-DL) :-
  L = L1, expr1(L1-DL1), DL1 = [0'+ | L2], expr(L2-DL2), DL = DL2.
expr(L-DL) :-
  expr1(L-DL).
expr1(L-DL) :-
  expr2(L-[0'* | L2]), expr1(L2-DL).
expr1(LDL) :-
  expr2(LDL).
expr2([0'0 | DL]-DL).
...
expr2([0'( | L]-DL) :-
  expr(L-[0') | DL]).
```


2.13.3 DCG

La régularité de la dérivation permet de l'automatiser : on va écrire des règles DCG (Definite Clause Grammar) qui vont être traduites automatiquement en Prolog.

Syntaxe :

- `-->` connecteur de règle (le ::=) ;
- `,` opérateur binaire de séquence ;
- `;` opérateur binaire d'alternative (le |) ;
- `[ ]` pour définir une liste de terminaux ;
- { } pour inclure un but Prolog dans une règle.

La grammaire s'utilise avec l'appel `phrase(` Racine `,` Tokens`)`.

```
expr  --> expr1 , [0'+] , expr.
expr  --> expr1.

expr1 --> expr2 , [0'*] , expr1.
expr1 --> expr2.

expr2 --> [C] , {C >= 0'0 , C =< 0'9}.
expr2 --> [0'(] , expr , [0')].
```

On remarque la concision d’écriture.

```
?- phrase(expr, [0'1, 0'*, 0'2]).
 yes
?- phrase(expr, [0'1, 0'*, 0'2, 0'3]).
 no
```


2.13.4 Arbre syntaxique

Construction de l’arbre syntaxique correspondant à l’expression analysée : c’est un argument de chaque non-terminal.

```
expr(plus(E1, E2))  --> expr1(E1), [0'+], expr(E2).
expr(E)             --> expr1(E).
expr1(mult(E1, E2)) --> expr2(E1), [0'*], expr1(E2).
expr1(E)            --> expr2(E).
expr2(integer(I))   --> [C], {C >= 0'0 , C =< 0'9, I is C - 0'0}.
expr2(E)            --> [0'(], expr(E), [0')].
?- phrase(expr(Tree), [0'1, 0'+, 0'3, 0'*, 0'2]).
Tree = plus(integer(1), mult(integer(3), integer(2)))
```

2.13.5 Attributs hérités

L'associativité à gauche du − (moins) ne permet pas d'écrire simplement :

```
expr(minus(E1, E2)) --> expr1(E1), [0'-], expr(E2).
```

La solution est de transformer la grammaire :

expr ::= expr1 rexpr

rexpr ::= ϵ | + expr1 rexpr | − expr1 rexpr

... ::=

et de faire en sorte que rexpr hérite de l'attribut (l'arbre syntaxique) synthétisé par expr1.

```
expr(E)       --> expr1(E1), rexpr(E1, E).
rexpr(E1, E) --> [0'-], expr1(E2), rexpr(minus(E1, E2), E).
rexpr(E, E)  --> [].
```


References

[1] H. Aït-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, 1991.

[2] G. Battani and H. Meloni. Interpréteur du langage de programmation Prolog. Technical report, Groupe d'Intelligence Artificielle, Marseille, France, 1973.

[3] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog interpreters. In S-Å. Tärnlund, editor, 2nd Int. Conf. Logic Programming, pages 343–351. Uppsala University, 1984.

[4] A. Colmerauer. Prolog and infinite trees. In K.L. Clark and S-Å. Tärnlund, editors, Logic Programming, pages 231–251. Academic Press, New-York, 1982.

[5] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Etude et réalisation d'un système Prolog. Technical report, G.I.A. Université Aix-Marseille, May 1979.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In Int. Conf. Fifth Generation Computer Systems, volume 1, pages 693–702, Tokyo, Japan, 1988.

[7] ECL^{i}PS^e user manual (ECRC Common Logic Programming System), 1992.

[8] François Fages. Programmation logique par contraintes. École polytechnique. Ellipses, 1996.

[9] R. Kowalski and M. Van Emden. The semantics of predicate logic as a programming language. JACM, 23(4):733–743, Oct. 1976.

[10] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, 2nd Symp. Logic in Computer Science, pages 98–105, Ithaca, New York, USA, 1987.

[11] J.A. Robinson. A machine-oriented logic based on the resolution principle. JACM, 12(1):23–41, 1965.

[12] Pascal Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, Cambridge, MA, 1989.

[13] D.H.D. Warren. Implementing Prolog — compiling logic programs, vol. 1 and 2. DAI Research Report 39, 40, University of Edinburgh, 1977.

[14] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Stanford, Ca, 1983.