Représentation de la connaissance

Introduction à l'intelligence artificielle

Introduction à la programmation logique

PROLOG

Applications de PROLOG à l'IA

P. Gochet et P. Gribomont
LOGIQUE, méthodes pour l'intelligence artificielle
Hermès, Paris, 2000
(Chapitres 10, 11 et 12)

Quelques compléments

Termes, termes atomiques

Prolog et IA : concept fondamental = objet

Objets représentés par des *termes*

Atomes : constantes, variables

Prolog *identifie* les constantes aux objets qu'elles représentent ; pas de signification intrinsèque

`0`, `-3` et `2.567` : nombres (constantes)
`abc`, `+` et `rs232` : symboles (constantes)
`X`, `Ys`, `Truc`, `_ght54d` : variables

Termes composés, termes fondamentaux

foncteur : constructeur de termes composés
arité : entier naturel

`m(a,b,c)`,
`m(f(a,b),a(c,g(2,d)))`
`+(3,*(4,5))`
termes composés *fondamentaux*, sans variable

constante : terme fondamental atomique
foncteur d'arité 0 : constante

Arbres sans signification intrinsèque

On veut `+(3,*(4,5))` égal à `23` ?
Il faudra le "dire" !

`m(a,b,c)` est un arbre ternaire dont la racine est (étiquetée par) `m` et dont les feuilles sont (étiquetées par), de gauche à droite, `a`, `b` et `c`. A une même constante symbolique `f` peuvent correspondre plusieurs foncteurs, que l'on distingue par leurs arités respectives.

Termes, listes, paires

Exemple de terme : `p(gates,bill,45)`

Liste : terme "anonyme", `[gates,bill,43]`

`[gates,bill,43]`
est en fait
`.(gates,.(bill,.(43,[])))`
ou encore
`[gates | [bill | [43 | []]]]`

`.` : foncteur d'arité 2

Liste vide : `[]`

Exemples de termes

Terme fondamental : position à “Hexapion”

```
hp(c(1,1,v),c(1,2,n),c(1,3,n),
   c(2,1,n),c(2,2,b),c(2,3,v),
   c(3,1,b),c(3,2,v),c(3,3,b))
```

b pour pion blanc,
n pour pion noir,
v pour case vide

Terme non fondamental :
`m(f(a,X),f(X,b),Y)`

Instances fondamentales :
`m(f(a,a),f(a,b),c)`
`m(f(a,g(c)),f(g(c),b),g(c))`

Association terme - objet

- Les objets de Prolog sont les termes fondamentaux.
- A tout terme est associé un ensemble d’instances, qui est un ensemble de termes fondamentaux.
- Un terme représente une instance générique, ou une instance particulière, ou l’ensemble des instances associé à ce terme.
- L’ensemble des instances d’un terme fondamental se réduit à ce terme ; en conséquence, un terme fondamental représente un seul objet, lui-même.
- L’ensemble des instances d’une variable est l’ensemble de tous les termes fondamentaux.

Faits

Appartenance d’un n-uplet à une relation.
Le fait

```
r(a,b,c).
```

affirme que le n-uple

```
(a,b,c)
```

appartient à la relation ternaire

```
r
```

$n = 0$: proposition :

```
z.
```

Faits et descriptions

Description d’un graphe

```
node(g,n1).      arc(g,n1,n3).
node(g,n2).      arc(g,n2,n3).
node(g,n3).      arc(g,n1,n5).
node(g,n4).      arc(g,n4,n5).
node(g,n5).      arc(g,n5,n6).
node(g,n6).      arc(g,n2,n4).
                 arc(g,n4,n6).
```

Faits dérivés, règles

```
path(g,[n1,n5,n6]).
```

si X est un nœud,
alors [X] est un chemin
(“clause de base”)

si (X,Y) est un arc et si [Y|Ys] est un chemin,
alors [X,Y|Ys] est un chemin
(“clause inductive, ou récursive”)

Règles et clauses

```
path(G,[X]) :- node(G,X).

path(G,[X,Y|Ys]) :- node(G,X),
                    arc(G,X,Y),
                    path(G,[Y|Ys]).
```

```
A :- B,C,D.
```

si B, C et D sont vrais, alors A est vrai

Fait = règle inconditionnelle

On écrit “A.” au lieu de “A :- .”

Questions . . . et réponses

– Quels sont les nœuds du graphe g ?
– Est-ce que [n1,n5,n6] est un chemin de g ?
– Quels sont les chemins de g dont l’origine est n1 ou n2 et dont l’extrémité est n6 ?

Instance de règle :

```
path(g,[n1,n5|[n6]]) :-
  node(g,n1), arc(g,n1,n5),
  path(g,[n5|[n6]]).
```

ou encore, de manière plus lisible :

```
path(g,[n1,n5,n6]) :-
  node(g,n1), arc(g,n1,n5),
  path(g,[n5,n6]).
```

Réduction, sous-questions

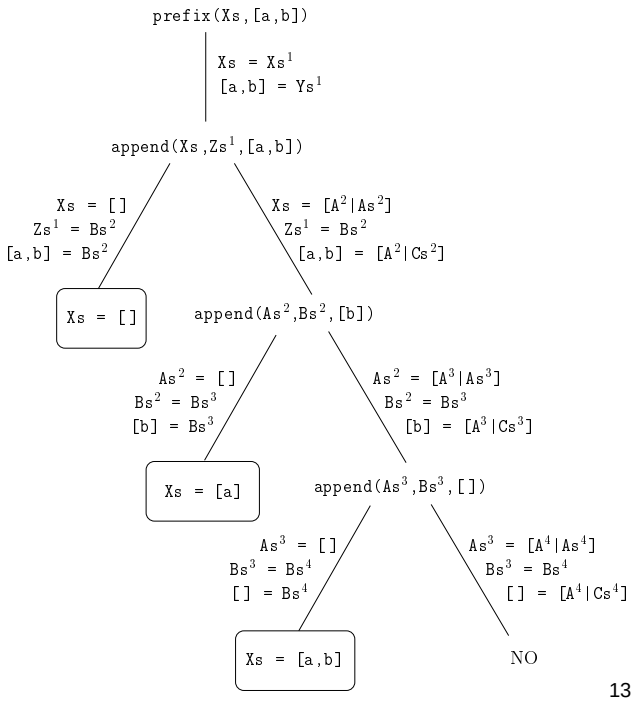
La question principale se réduit à la suite des trois *sous-questions*

```
node(g,n1)
arc(g,n1,n5)
path(g,[n5,n6])
```

Les deux premières sont des faits fondamentaux présents dans la base de connaissance, donc la question principale est maintenant réduite à la sous-question path(g,[n5,n6]).

Programme et arbre de recherche

```
prefix(Xs,Ys) :- append(Xs,Zs,Ys).
append([],Bs,Bs).
append([A|As],Bs,[A|Cs]) :- append(As,Bs,Cs).
```



De l'arbre de recherche à la déduction

1. `append([],[Bs`^{3}`],[Bs`^{3}`])`
   (clause de base – `append`)
2. `append([],[b],[b])` (instantiation, ligne 1)
3. `append([A`^{2}`|As`^{2}`],Bs`^{2}`,[A`^{2}`|Cs`^{2}`])`
   si `append(As`^{2}`,Bs`^{2}`,Cs`^{2}`)`
   (clause inductive – `append`)
4. `append([a],[b],[a,b])`
   si `append([],[b],[b])` (instant., ligne 3)
5. `append([a],[b],[a,b])`
   (déduction, lignes 2 et 4)
6. `prefix(Xs`^{1}`,Ys`^{1}`)`
   si `append(Xs`^{1}`,Zs`^{1}`,Ys`^{1}`)` (clause – `prefix`)
7. `prefix([a],[a,b])`
   si `append([a],[b],[a,b])` (instant., ligne 6)
8. `prefix([a],[a,b])` (déduction, lignes 5 et 7)

Résolution unitaire positive

S_0 : ensemble de clauses de Horn

clause $\simeq$ ensemble disjonctif de littéraux

$\mathbf{F}$: clause vide, inconsistante

Le choix est non déterministe

$\{S = S_0\}$

Tant que $\mathbf{F} \notin S$ faire

choisir p et c tels que

– p clause unitaire positive de S,

– c clause de S contenant $\neg p$;

$r := c \setminus \{\neg p\}$;

$S := (S \setminus \{c\}) \cup \{r\}$.

$\{S = S_f\}$

Résolution unitaire : propriétés

Invariant : $\mathcal{M}(S) = \mathcal{M}(S_0)$

car, pour tout ensemble U de formules,

les ensembles $S_n = U \cup \{p\,,\, \neg p \vee X\}$ et
$S_{n+1} = U \cup \{p\,,\, X\}$

ont exactement les mêmes modèles.

Terminaison : chaque tour de boucle
élimine un littéral négatif.

Efficacité :
quadratique en le nombre de littéraux négatifs.

Terminaison normale :
la garde devient fausse.
Inconsistance car $S_0 \simeq S_f$ et $\mathbf{F} \in S_f$.

Terminaison anormale :
le choix de p et c est impossible.
Consistance :
S_0 et S_f admettent le modèle canonique.

Modèle canonique d'un ensemble de Horn

En cas de terminaison anormale,
l'ensemble S_f admet le modèle C défini par :

$C(p) = \mathbf{V}$ si p est une clause unitaire de S_f ;
$C(p) = \mathbf{F}$ si p n'est pas une clause unitaire de S_f.
NB. On écrit souvent $p \in C$ au lieu de $C(p) = \mathbf{V}$.

C est le *modèle canonique* de S_0 (et de S_f).
Il rend vraies les clauses unitaires de S_f,
et les littéraux négatifs des autres clauses.

Propriétés :

C est le plus petit modèle de S_0 ;
$C(p) = \mathbf{V}$ si et seulement si $S_0 \models p$.

Modèles d'un ensemble de Horn

Soit H un ensemble de Horn sur le lexique Π.
$\mathcal{P}(\Pi)$ est l'ensemble des interprétations.

Soit $\mathcal{T} : \mathcal{P}(\Pi) \to \mathcal{P}(\Pi) : I \mapsto \mathcal{T}(I)$
telle que $p \in \mathcal{T}(I)$ s'il existe dans H une clause
$p \Leftarrow (q_1 \wedge \cdots \wedge q_n)$ avec $q_1, \ldots, q_n \in I$.
(Le cas $n = 0$ est naturellement admis !)

Théorème. Si $p \in \mathcal{T}(I)$, alors $H \cup I \models p$.

Théorème. La fonction $\mathcal{T}$ est croissante.

Théorème. I est modèle de H ssi $\mathcal{T}(I) \subset I$.

Corollaire.
Si $\mathcal{C}$ est un ensemble non vide de modèles de H,
alors $\bigcap C$ est un modèle de H.

Corollaire. Le modèle canonique de H est $\mathcal{T}^*(\emptyset)$.

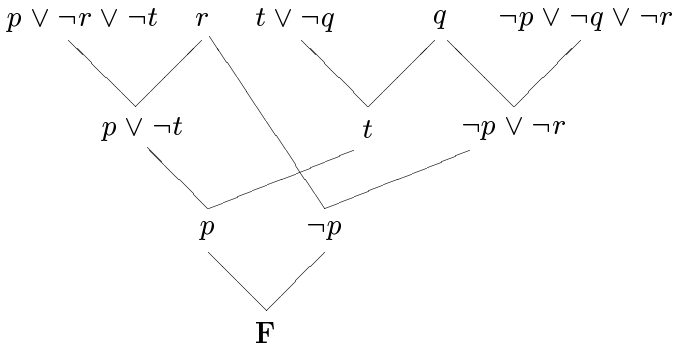
Remarque. Ceci subsiste si Π et H sont infinis.

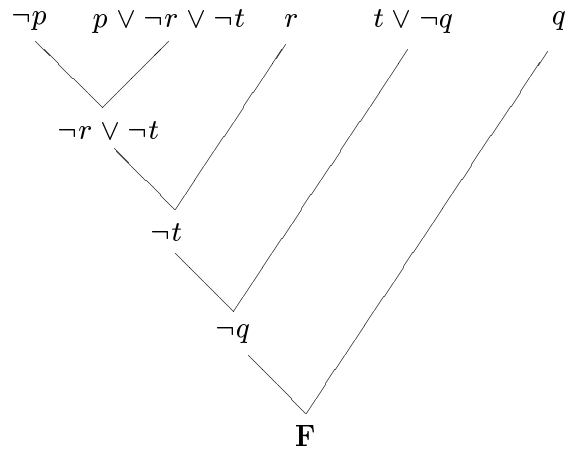
Résolution d'entrée

L'algorithme de résolution unitaire n'accorde aucun rôle particulier à la question p. L'algorithme de résolution d'entrée permet d'exploiter le programme logique de manière plus focalisée. Il utilise une variable G, dont la valeur est toujours une clause de Horn négative ; si $g_1, \ldots, g_n$ est la question, on a $G_0 = \{\neg g_1, \ldots, \neg g_n\} = \neg g_1 \vee \cdots \vee \neg g_n$.

$\{G = G_0\}$
Tant que $G \neq \mathbf{F}$ faire
choisir p et c tels que
– $\neg p \in G$,
– $c \in L$ et $p \in c$;
$G := (G \setminus \{\neg p\}) \cup (c \setminus \{p\})$.

Réfutation unitaire, réfutation d'entrée





Equivalence des deux algorithmes

Théorème. L'ensemble de Horn minimal H admet une réfutation unitaire si et seulement s'il admet une réfutation d'entrée basée sur une clause négative.

Principe de la démonstration.

On montre comment transformer une réfutation unitaire de H en une réfutation d'entrée et réciproquement.
On procède par induction sur le lexique.

Cas de base : $\Pi = \{p\}$.
Cas inductif : réduire Π à $\Pi \setminus \{q\}$, avec $q \in \Pi$.

Le cas de base est évident :
$\{p, \neg p\}$ admet une seule réfutation,
à la fois unitaire et d'entrée (basée sur $\neg p$).

Unitaire $\rightarrow$ entrée

On part d'une réfutation unitaire $\mathcal{R}$ (lexique Π).

Soit q une clause unitaire de H.

On efface de $\mathcal{R}$ la clause q et sa descendance éventuelle ; on efface $\neg q$ des feuilles de $\mathcal{R}$ et de leur descendance éventuelle. On obtient une réfutation unitaire $\mathcal{R}'$ de $H' = \{c \setminus \{\neg q\} : c \in H \setminus \{q\}\}$.

Par hypothèse inductive, il existe une réfutation d'entrée $\mathcal{R}''$ de H', basée sur une clause négative de H'. On ajoute la feuille $\{q\}$ et, dans les autres feuilles, le littéral $\neg q$ si c'est nécessaire, de sorte que toutes les feuilles soient des clauses de H. On propage $\neg q$ vers le bas et on obtient ainsi une dérivation d'entrée du littéral $\neg q$, basée sur une clause négative de H. Une étape supplémentaire, utilisant la clause q, transforme cette dérivation en réfutation d'entrée.

Entrée $\rightarrow$ unitaire

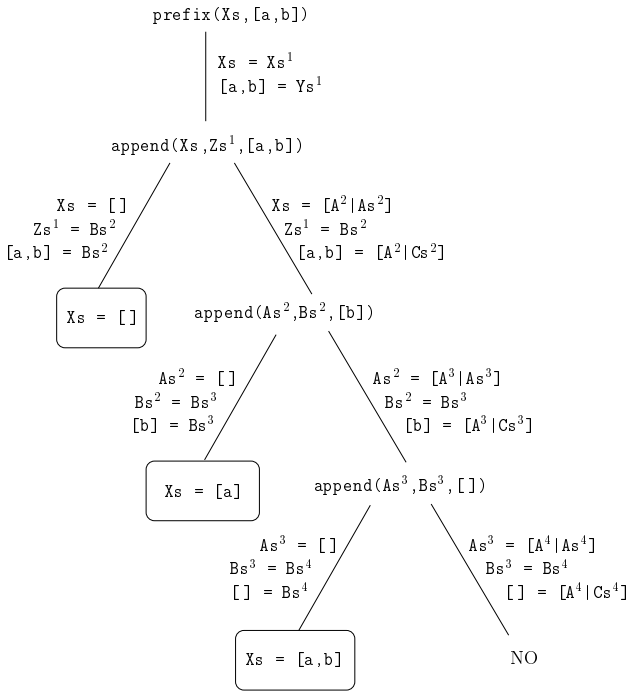
Le même principe s'applique.

Quelques programmes simples

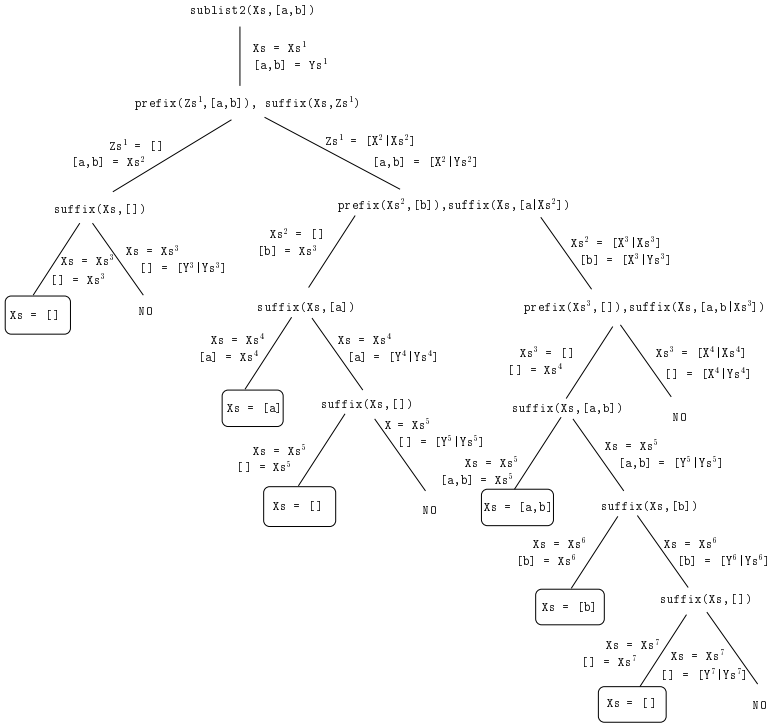
```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :- suffix(Xs,Ys).

prefix([],Xs).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

sublist1(Xs,Ys) :-
   suffix(Xs,Zs), prefix(Zs,Ys).

sublist2(Xs,Ys) :-
   prefix(Zs,Ys), suffix(Xs,Zs).

subset1([],[]).
subset1([X|Xs],[X|Zs]) :- subset1(Xs,Zs).
subset1(Xs,[X|Zs]) :- subset1(Xs,Zs).

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

subset2([],Xs).
subset2([Y|Ys],Xs) :-
   member(Y,Xs), subsetp2(Ys,Xs).
```

Arbre de recherche I

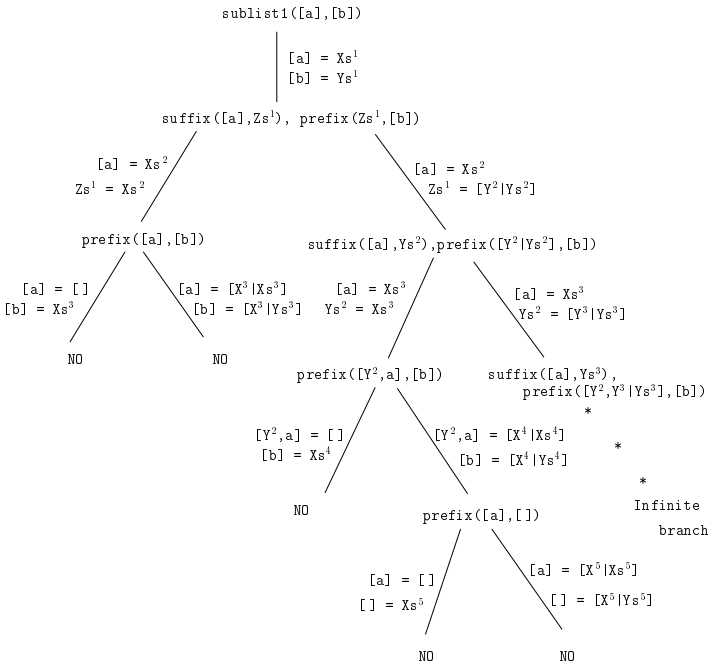
```
prefix(Xs,Ys) :- append(Xs,Zs,Ys).
append([],Bs,Bs).
append([A|As],Bs,[A|Cs]) :- append(As,Bs,Cs).
```



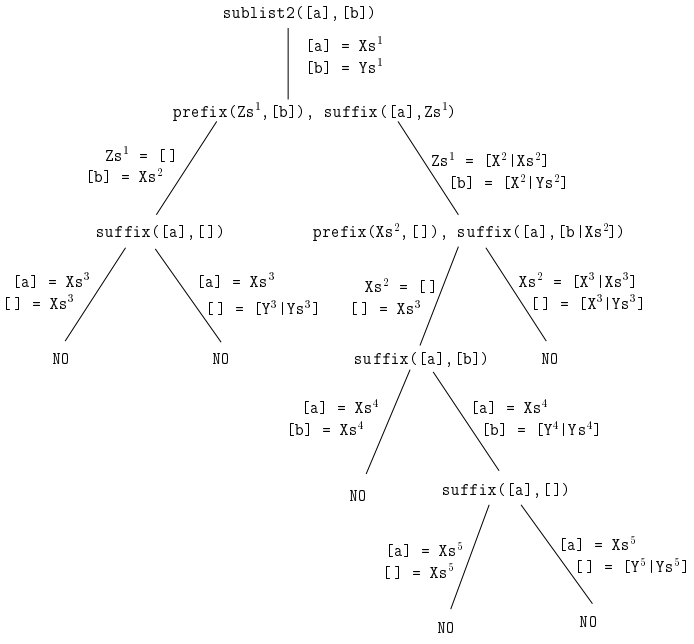
Arbre de recherche II



Arbre de recherche III



Arbre de recherche IV



Exemple d'exécution I

```
?- subset1(Xs,[a,b,c]).
Xs = [a, b, c] ;
Xs = [a, b] ;
Xs = [a, c] ;
Xs = [a] ;
Xs = [b, c] ;
Xs = [b] ;
Xs = [c] ;
Xs = [] ;
No

?- subset1([1,3],[1,2,3,4]).
Yes

?- subset1([1,1,3],[1,2,3,4]).
No

?- subset1([3,1],[1,2,3,4]).
No
```

Exemple d'exécution II

```
?- subset2([1,1,3],[1,2,3,4]).
Yes

?- subset2([3,1],[1,2,3,4]).
Yes

?- subset2(Xs,[1,2,3,4]).
Xs = [] ;
Xs = [1] ;
Xs = [1, 1] ;
Xs = [1, 1, 1] ;
...
```

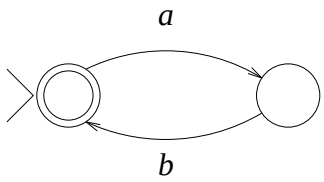

Exemple d'exécution III

```
subset_list([],[[]]).
subset_list([X|Xs],Zss) :-
      subset_list(Xs,Uss),
      put_in_all(X,Uss,Vss),
      append(Vss,Uss,Zss).

put_in_all(A,[],[]).
put_in_all(A,[Bs|Bss],[[A|Bs]|Css]) :-
      put_in_all(A,Bss,Css).
```

L'exemple suivant montre comment ce prédicat et le prédicat auxiliaire `put_in_all` fonctionnent :

```
?- put_in_all(x,[[a,b],[],[c,d,e]],Xss).
Xss = [[x, a, b], [x], [x, c, d, e]]

?- subset_list([a,b,c],Xss).
Xss = [[a, b, c], [a, b], [a, c], [a],
       [b, c], [b], [c], []]
```


Un automate fini




```
/* Automate pour le langage (ab)* */
init(q0).
fin(q0).
delta(q0,a,q1).
delta(q1,b,q0).

/* Interprétation d'un automate */
acc(Xs) :- init(Q), acc(Q,Xs).
acc(Q,[]) :- fin(Q).
acc(Q,[X|Xs]) :- delta(Q,X,Q1),
                 acc(Q1,Xs).
```

Un automate à pile

```
/* Automate à pile pour le langage
   des palindromes sur {a,b} */
init(q0).
fin(q1).
delta(q0,X,S,q0,[X|S]).
delta(q0,X,S,q1,[X|S]).
delta(q0,X,S,q1,S).
delta(q1,X,[X|S],q1,S).

/* Interprétation */
acc(Xs) :- init(Q), acc(Q,Xs,[]).
acc(Q,[],[]) :- fin(Q).
acc(Q,[X|Xs],S) :- delta(Q,X,S,Q1,S1),
                   acc(Q1,Xs,S1).
```

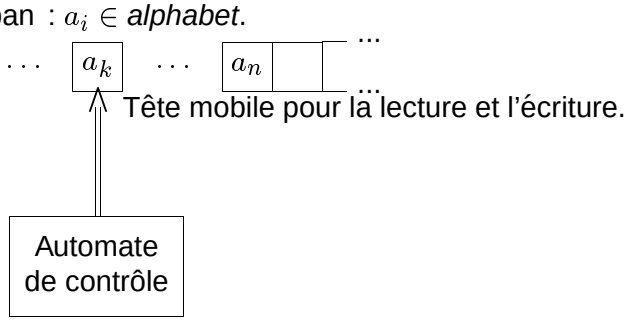
Machine de Turing I

Une pile suffit pour le langage $\{a^n b^n : n \in \mathbb{N}\}$

Deux piles pour le langage $\{a^n b^n c^n : n \in \mathbb{N}\}$

Autre possibilité : Machine de Turing.

Ruban : $a_i \in$ *alphabet*.



Machine de Turing II

```
/* Exemple de machine de Turing codée en Prolog */

init(q0).                 final(q4).

delta(q0,a,q1,x,right).  delta(q0,y,q3,y,right).
delta(q1,a,q1,a,right).  delta(q1,b,q2,y,left).
delta(q1,y,q1,y,right).  delta(q2,a,q2,a,left).
delta(q2,x,q0,x,right).  delta(q2,y,q2,y,left).
delta(q3,y,q3,y,right).  delta(q3,w,q4,w,right).
```

Définition.

`cfg(Q,Ds,Head,Fs)` correspond à la configuration dont l'état de contrôle est `Q` ; le mot contenu sur le ruban est la concaténation de l'inverse de `Ds` avec le symbole `Head` et le mot `Fs` ; le symbole `Head` est sous la tête de lecture.

Machine de Turing III

```
/* Interpréteur Prolog pour machines de Turing */

acc_MT([]) :- init(Q), acc(cfg(Q,[],w,[])).
acc_MT([A|As]) :- init(Q), acc(cfg(Q,[],A,As)).

acc(cfg(Q,Ds,Head,Fs)) :- final(Q).
acc(cfg(Q,Ds,Head,Fs)) :-
   next_cfg(cfg(Q,Ds,Head,Fs),
            cfg(Qn,Dsn,Headn,Fsn)),
   acc(cfg(Qn,Dsn,Headn,Fsn)).

next_cfg(cfg(Q,Ds,Head,Fs),
         cfg(Qn,Dsn,Headn,Fsn)) :-
   delta(Q,Head,Qn,Symb,Act),
   modif(cfg(Q,Ds,Head,Fs), Symb, Act,
         cfg(Qn,Dsn,Headn,Fsn)).

modif(cfg(Q,Ds,Head,[]),Symb,right,
      cfg(Qn,[Symb|Ds],w,[])).

modif(cfg(Q,Ds,Head,[F|Fs]),Symb,right,
      cfg(Qn,[Symb|Ds],F,Fs)).

modif(cfg(Q,[D|Ds],Head,Fs),Symb,left,
      cfg(Qn,Ds,D,[Symb|Fs])).
```

Fonctions récursives I

Primitives

$(x_1, \ldots, x_n) \mapsto 0$: fonction zéro d'arité n ;
$x \mapsto x + 1$: fonction successeur ;
$(x_1, \ldots, x_n) \mapsto x_i$: $i$ème projection d'arité n.

Composition

$$f(x_1, \ldots, x_n) =_{def} h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) .$$

Récursion primitive

$$f(x_1, \ldots, x_n, 0) = h(x_1, \ldots, x_n) ,$$

$$f(x_1, \ldots, x_n, y{+}1) = g(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) .$$

Minimisation

$$f(x_1, \ldots, x_n) = \mu y.[g(x_1, \ldots, x_n, y) = 0] ,$$

Fonctions récursives II

Fonctions constantes nulles,
projections et fonction successeur

```
zero_(X1,...,Xn,o).

pr_j(X1,...,Xn,Xj).

succ_(X,s(X)).
```

Mécanisme de composition

```
f_(X1,...,Xn,Y) :-
      g_1(X1,...,Xn,Y1),
      ...,
      g_m(X1,...,Xn,Ym),
      h_(Y1,...,Ym,Y).
```

Récursion primitive

```
f_(X1,...,Xn,o,Z) :- h_(X1,...,Xn,Z).
f_(X1,...,Xn,s(Y),Z) :-
     f_(X1,...,Xn,Y,U),
     g_(X1,...,Xn,Y,U,Z).
```


Fonctions récursives III

Minimisation

```
f_(X1,...,Xn,Y) :- g_(X1,...,Xn,o,U),
                   r_(X1,...,Xn,o,U,Y).
r_(X1,...,Xn,Y,o,Y).
r_(X1,...,Xn,Y,s(V),Z) :-
     g_(X1,...,Xn,s(Y),U),
     r_(X1,...,Xn,s(Y),U,Z).
```

Le prédicat `r_` à $(n+3)$ arguments est associé à une fonction auxiliaire r d'arité $(n+2)$ qui peut être spécifiée comme suit :

Si $g(x_1,\ldots,x_n,\xi) > 0$ *pour tout* $\xi = 0,1,\ldots,y-1$
et si $g(x_1,\ldots,x_n,y) = u$,
alors $r(x_1,\ldots,x_n,y,u) =$
$\inf\{\zeta : \zeta \geq y \wedge g(x_1,\ldots,x_n,\zeta) = 0\}$.

On voit que la valeur $r(x_1,\ldots,x_n,y,u)$ n'est pas définie si $u > 0$ et $g(x_1,\ldots,x_n,\zeta) > 0$ pour tout $\zeta > y$. Quand c'est le cas, l'évaluation de la question `r_(X1,...,Xn,Y,U,Z)` ne se termine pas.

Un tri naïf

```
naive_sort(Xs,Ys) :- permut(Xs,Ys),
                     ordered(Ys).

permut(Xs,[Z|Zs]) :- select(Z,Xs,Ys),
                     permut(Ys,Zs).
permut([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X=<Y, ordered([Y|Ys]).
```

Inefficace et non réversible

Tri par insertion

```
insert_sort([],[]).
insert_sort([X|Xs],Ys) :-
   insert_sort(Xs,Zs), insert(X,Zs,Ys).

insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :-
   X>Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X|[Y|Ys]]) :- X=<Y.
```

Quadratique et non réversible

Tri rapide

```
quick_sort([],[]).
quick_sort([X|Xs],Ys) :-
    part(Xs,X,Littles,Bigs),
    quick_sort(Littles,Ls),
    quick_sort(Bigs,Bs),
    append(Ls,[X|Bs],Ys).

part([],Y,[],[]).
part([X|Xs],Y,[X|Ls],Bs) :-
    X=<Y, part(Xs,Y,Ls,Bs).
part([X|Xs],Y,Ls,[X|Bs]) :-
    X>Y, part(Xs,Y,Ls,Bs).
```

Efficace, quasi-réversible

```
quick_sort_bis([],[]).
quick_sort_bis([X|Xs],Ys) :-
    append(Ls,[X|Bs],Ys),
    quick_sort_bis(Littles,Ls),
    quick_sort_bis(Bigs,Bs),
    part(Xs,X,Littles,Bigs).
```


Tri par fusion

```
merge_([],Xs,Xs).
merge_([X|Xs],[],[X|Xs]).
merge_([X|Xs],[Y|Ys],[X|Zs]) :-
    X<Y, merge_(Xs,[Y|Ys],Zs).
merge_([X|Xs],[Y|Ys],[X|[Y|Zs]]) :-
    X=Y, merge_(Xs,Ys,Zs).
merge_([X|Xs],[Y|Ys],[Y|Zs]) :-
    X>Y, merge_([X|Xs],Ys,Zs).

merge_sort([],[]).
merge_sort([X],[X]).
merge_sort([X|[X1|Xs]],Ys) :-
    division(Xs,Fs,Ss),
    merge_sort([X|Fs],FFs),
    merge_sort([X1|Ss],SSs),
    merge_(FFs,SSs,Ys).

division([],[],[]).
division([X],[X],[]).
division([X|[X1|Xs]],[X|Ys],[X1|Zs]) :-
    division(Xs,Ys,Zs).
```


Style fonctionnel, style logique I

Une partition d'un ensemble E est une famille de sous-ensembles non vides et deux à deux disjoints de E, dont la réunion est E.

On calcule facilement (avec la récursion)
la liste des partitions d'un ensemble donné.

La seule partition de l'ensemble vide
est l'ensemble vide lui-même.

Si $x \notin E$, on obtient une partition de $\{x\} \cup E$
à partir d'une partition P de E de deux manières :
1) en insérant la partie supplémentaire $\{x\}$ dans P ;
2) ou en insérant x dans une composante de P.

Style fonctionnel, style logique II

Programme SCHEME

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (partitions (cdr e))))
          (append (procede_1 (car e) rec)
                  (procede_2 (car e) rec))))))

(define procede_1
  (lambda (x lp) (insert_in_all (list x) lp)))

(define insert_in_all
  (lambda (x le) (map (lambda (e) (cons x e)) le)))

(define procede_2
  (lambda (x lp)
    (map_append (lambda (p) (split x p)) lp)))

(define map_append
  (lambda (f l)
    (if (null? l)
        '()
        (append (f (car l))
                (map_append f (cdr l))))))

(define split
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons (cons x (car ll)) (cdr ll))
              (map (lambda (ss) (cons (car ll) ss))
                   (split x (cdr ll)))))))
```

Style fonctionnel, style logique III

Equivalent PROLOG

```
c_partitions([],[[]]).
c_partitions([X|Xs],Ps) :-
   c_partitions(Xs,Qs),
   procede_1(X,Qs,P1s),
   procede_2(X,Qs,P2s),
   append(P1s,P2s,Ps).

procede_1(X,Qs,Rs) :-
   insert_in_all([X],Qs,Rs).

insert_in_all(U,[],[]).
insert_in_all(U,[Us|Uss],[[U|Us]|Vss]) :-
   insert_in_all(U,Uss,Vss).

procede_2(X,[],[]).
procede_2(X,[Q|Qs],Ps) :-
   split(X,Q,R1s),
   procede_2(X,Qs,R2s),
   append(R1s,R2s,Ps).

split(X,[],[]).
split(X,[Xs|Xss],[[[X|Xs]|Xss]|Ysss]) :-
   split(X,Xss,Xsss),
   insert_in_all(Xs,Xsss,Ysss).
```


Style fonctionnel, style logique IV

Equivalent PROLOG, style logique

```
partition([],[]).

partition([X|Xs],[[X]|P]) :-
   partition(Xs,P).

partition([X|Xs],Q) :-
   partition(Xs,P),
   append(R1,[R|R2],P),
   append(R1,[[X|R]|R2],Q).

partitions(Xs,Ps) :-
   findall(P,partition(Xs,P),Ps).

partitions_bis(Xs,Ps) :-
   partition(Xs,P), write(P), nl, fail.
```


Style fonctionnel, style logique V

```
?- partitions([1,2,3],Ps).

Ps = [[[1],[2],[3]],[[1],[2,3]],
      [[1,2],[3]],[[2],[1,3]],[[1,2,3]]]

partitions_bis([1,2,3],Ps).
[[1],[2],[3]]
[[1],[2,3]]
[[1,2],[3]]
[[2],[1,3]]
[[1,2,3]]

No
```

Style fonctionnel, style logique VI

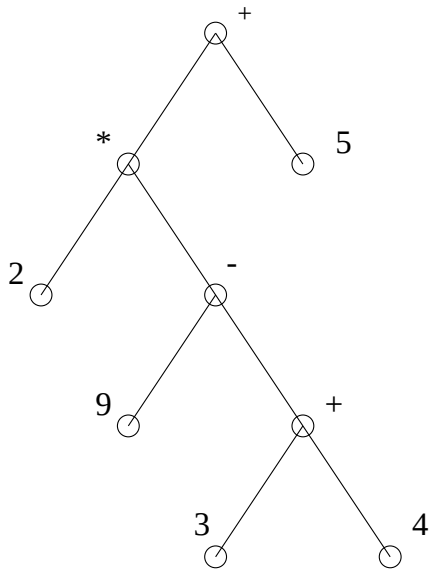
```
bipart([X],[Y|Xs],[X,Y|Xs]) :- X<Y.
bipart([X|Xs],[Y|Ys],[X|Zs]) :-
   bipart(Xs,[Y|Ys],Zs),X<Y.
bipart([X|Xs],[Y|Ys],[Y|Zs]) :-
   bipart([X|Xs],Ys,Zs),X>Y.

part2([X|Xs],[Y|Ys],Zs) :-
   bipart([X|Xs],[Y|Ys],Zs), X<Y.
part2([X|Xs],[Y|Ys],Zs) :-
   bipart([Y|Ys],[X|Xs],Zs), X<Y.

part(Xs,P) :- part1(Xs,P).
part(Xs,P) :- part+(Xs,P).

part1([X|Xs],[[X|Xs]]).

part+(Xs,[Ys,Zs]) :- part2(Ys,Zs,Xs).
part+([X,Y|Zs],[[A|As]|Ass]) :-
   part2([A|As],[U|Us],[X,Y|Zs]),
   part+([U|Us],Ass).

partitions_ter(Xs,Ps) :-
   findall(P,part(Xs,P),Ps).
```

Style fonctionnel, style logique VII

```
?- partitions_ter([1,2,3],Ps).

Ps = [[[1,2,3]],[[1],[2,3]],[[1,2],[3]],
      [[1,3],[2]],[[1],[2],[3]]]

?- partitions_ter([1,2,3,4],Ps).

Ps = [[[1,2,3,4]],[[1],[2,3,4]],[[1,2],[3,4]],
      [[1,2,3],[4]],[[1,3],[2,4]],[[1,3,4],[2]],
      [[1,4],[2,3]],[[1,2,4],[3]],[[1],[2],[3,4]],
      [[1],[2,3],[4]],[[1],[2,4],[3]],
      [[1],[2],[3],[4]],[[1,2],[3],[4]],
      [[1,3],[2],[4]],[[1,4],[2],[3]]]
```


Calcul symbolique I

$$((2 * (9 - (3 + 4))) + 5)$$



Calcul symbolique II

Si on utilise pas `functor`, `args` et `=..`,
on emploiera les notations de listes.

Un $\mathbb{N}$-arbre atomique est un nombre.

Un $\mathbb{N}$-arbre composé est une liste à trois éléments `[X,Y,Z]` où `X` est un opérateur arithmétique et `Y` et `Z` sont les $\mathbb{N}$-arbres représentant les opérandes de gauche et de droite, respectivement.

Le $\mathbb{N}$-arbre

```
[+,[*,2,[-,9,[+,3,4]]],5]
```

correspond à l'expression dont la liste des caractères est

```
[(,(,2,*,(,9,-,(,3,+,4,),),),+,5,)]
```

Calcul symbolique III

Eviter les parenthèses :
notation préfixée ou postfixée.

```
[(,(,2,*,(,9,-,(,3,+,4,),),),+,5,)]
```

devient

```
[+,*,2,-,9,+,3,4,5]
```

ou

```
[2,9,3,4,+,-,*,5,+]
```

Un problème se posant naturellement est celui de la conversion d'un $\mathbb{N}$-arbre quelconque en la liste des symboles correspondant à l'une des trois notations.

Calcul symbolique IV

Analyse de $\mathbb{N}$-arbre, solution naïve

```
tree_to_pref(T,[T]) :- number(T).
tree_to_pref([X,Y,Z],[X|Xs]) :-
    tree_to_pref(Y,Ys),
    tree_to_pref(Z,Zs),
    append(Ys,Zs,Xs).


tree_to_postf(T,[T]) :- number(T).
tree_to_postf([X,Y,Z],Us) :-
    tree_to_postf(Y,Ys),
    tree_to_postf(Z,Zs),
    append(Ys,Zs,Xs),
    append(Xs,[X],Us).
```


Calcul symbolique V

```
tree_to_inf(T,[T]) :- number(T).
tree_to_inf([X,Y,Z],[l|Xs]) :-
    tree_to_inf(Y,Ys),
    tree_to_inf(Z,Zs),
    append([X|Zs],[r],Us),
    append(Ys,Us,Xs).
```

`l` : parenthèse gauche ;
`r` : parenthèse droite.

Calcul symbolique VI

Solution naïve, non réversible

```
?- tree_to_pref
     ([+,[*,2,[-,9,[+,3,4]]],5],L).
L = [+,*,2,-,9,+,3,4,5]

?- tree_to_postf
     ([+,[*,2,[-,9,[+,3,4]]],5],L).
L = [2,9,3,4,+,-,*,5,+]

?- tree_to_inf
     ([+,[*,2,[-,9,[+,3,4]]],5],L).
L = [l,l,2,*,l,9,-,l,3,+,4,4,r,r,r,+,5,r]
```