Le langage Prolog

— programmation en logique —

Jacques TISSEAU

ECOLE NATIONALE D'INGÉNIEURS DE BREST
Technopôle Brest-Iroise
CS 73862 - 29238 Brest cedex 3 - France

La première version de ce document a été rédigée avec LaTeX en 1990.
De nombreuses modifications y ont été apportées depuis, en particulier grâce à

- P. De Loor, P.A. Favier et A. Naim qui font partie (ou ont fait partie) de l'équipe pédagogique du module PROGRAMMATION EN LOGIQUE de l'ENIB ;
- F. Mesnard de l'Université de la Réunion qui propose ces notes de cours aux étudiants de Licence (L3 Informatique).

Utiliser la logique comme langage de programmation

Origines

- Travaux des logiciens *(. . . , J. Herbrand, J. Robinson, . . .)*
- Université de Marseille *(A. Colmerauer, Ph. Roussel, . . .)*
- Université d'Edinburgh *(R. Kowalski, D. Warren, . . .)*

Evolutions

- Prolog et apprentissage *(Inductive Logic Programming)*
- Prolog et contraintes *(Constraint Logic Programming)*
- Web sémantique

Remarque (origine de Prolog)

Le nom de Prolog a été créé à Marseille en 1972. Philippe Roussel l'a proposé comme abréviation de « **PRO**grammation en **LOG**ique », pour désigner un outil informatique conçu pour implanter un système de communication homme machine en langage naturel, en l'occurrence le français.
Pour en savoir plus sur les origines de Prolog et de la programmation en logique :

- Cohen J., *A View of the Origins and Development of Prolog*, Communications of the ACM, 31(1):26-36, 1988.
- Colmerauer A., Roussel A., *La naissance de Prolog*, Rapport interne, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, 1992.
- Kowalski R. A., *The early history of logic programming*, Communications of the ACM 31(1):38-43, 1988.
- Robinson J.A., *A machine-oriented logic based on the resolution principle*, Journal of the ACM 12(1):23-41, 1965.

- De la logique à Prolog
- Les termes en Prolog
- La machine Prolog
- Le contrôle de la résolution
- La recherche dans les graphes

Pour en savoir plus :

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990

Aspects syntaxiques

Ce qui peut être dit :

vocabulaire : les mots du langage

grammaire : les phrases du langage

Aspects sémantiques

Ce que l'on peut dire d'une expression logique :

sémantique : signification des phrases du langage

En logique : valeur de vérité (vrai ou faux)

Des règles de déduction décrivent les transformations que l'on peut faire subir aux expressions logiques à partir d'expressions définies comme vraies (axiomes).

Remarque (règles d'inférences)

Règles de déduction (raisonnement déductif) :

Modus Ponens	
$\dfrac{a \quad a \Rightarrow b}{b}$	si a est vrai et si $a \Rightarrow b$ est vrai, alors b est vrai.
Modus Tollens	
$\dfrac{\overline{b} \quad a \Rightarrow b}{\overline{a}}$	si $\overline{b}$ (non b) est vrai et si $a \Rightarrow b$ est vrai, alors $\overline{a}$ (non a) est vrai.

Règle d'abduction (raisonnement hypothétique) :

$\dfrac{b \quad a \Rightarrow b}{a}$	si b est vrai et si $a \Rightarrow b$ est vrai, alors a est vrai.

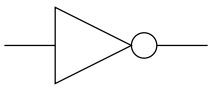
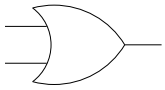
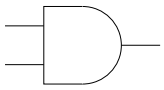
Règle d'induction (raisonnement hypothétique) :

$\dfrac{a \quad b}{a \Rightarrow b}$	si a est vrai et si b est vrai, alors $a \Rightarrow b$ est vrai.

Les mots du langage

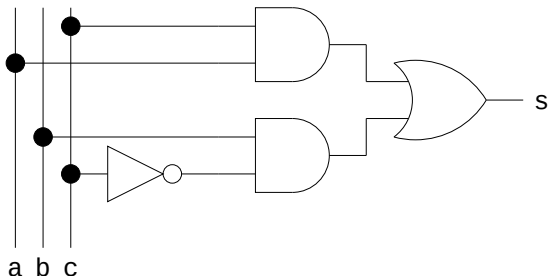
inconnues	x, y, z
constantes fonctionnelles	f, g, h
constantes prédicatives	P, Q, R
connecteurs	$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
quantificateurs	$\forall$, $\exists$
ponctuation	() ,
terme (t_i)	inconnue ou forme fonctionnelle
forme fonctionnelle	$f(t_1, \ldots, t_m)$ $m \geq 0$

Remarque (notations équivalentes)

domaine	négation	disjonction	conjonction
logique	$\neg a$	$a \vee b$	$a \wedge b$
informatique			
C/C++/Java	`!a`	`a || b`	`a && b`
Fortran	`.NOT. a`	`a .OR. b`	`a .AND. b`
Prolog	`\+a`	`a ; b`	`a , b`
Python	`not a`	`a or b`	`a and b`
électronique	$\overline{a}$ 	$a + b$ 	$a \cdot b$ 

TD (circuit logique)

Donner l'expression logique équivalente au circuit ci-dessous.



Les phrases du langage

forme prédicative	$P(t_1, \ldots, t_n)$ $n \geq 0$
formule $(A, B, \ldots)$	forme prédicative
	$(A), \neg A, A \wedge B, A \vee B,$
	$A \Rightarrow B, A \Leftrightarrow B$
	$\forall x\ A, \exists x\ A$

Exemples :

$\forall x\, P(x)$

$\forall x\, (P(x) \Rightarrow Q(x))$

$\forall x\, (\exists y\, (P(x) \Rightarrow Q(x, f(y))))$

$\forall x, y\, (P(x) \Rightarrow ((Q(x, y) \Rightarrow \neg(\exists u\, P(f(u)))) \vee (Q(x, y) \Rightarrow \neg R(y))))$

TD (opérateurs booléens dérivés)

Définir les opérateurs booléens suivants à partir des opérateurs booléens de base ($\neg$, $\wedge$ et $\vee$) :

1. ou exclusif (*xor*, $a \oplus b$)
2. non ou (*nor*, $\neg(a \vee b)$)
3. non et (*nand*, $\neg(a \wedge b)$)
4. implication ($a \Rightarrow b$)
5. équivalence ($a \Leftrightarrow b$)

La signification des phrases

$\mathcal{D}$: domaine de définition des termes A : formule logique

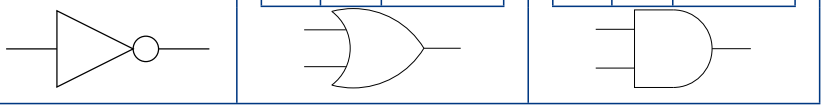
$$A : \mathcal{D} \longrightarrow \begin{array}{l} \{faux; vrai\} \\ \{false; true\} \\ \{0; 1\} \end{array}$$

négation	disjonction	conjonction
$\neg a$	$a \vee b$	$a \wedge b$

a	$\neg a$
0	1
1	0

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1



Remarque (propriétés des opérateurs booléens)

$\forall a, b, c \in \{0; 1\}$:

$a \vee 0 = a$
$a \vee 1 = 1$
$a \vee a = a$
$a \vee \neg a = 1$
$a \vee (a \wedge b) = a$
$(a \vee b) = (b \vee a)$
$\neg\neg a = a$

$\neg(a \vee b) = \neg a \wedge \neg b$
$(a \vee b) \vee c = a \vee (b \vee c)$
$(a \wedge b) \wedge c = a \wedge (b \wedge c)$
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

$a \wedge 1 = a$
$a \wedge 0 = 0$
$a \wedge a = a$
$a \wedge \neg a = 0$
$a \wedge (a \vee b) = a$
$(a \wedge b) = (b \wedge a)$
$\neg(a \wedge b) = \neg a \vee \neg b$

TD (Lois de De Morgan)

Démontrer à l'aide des tables de vérité les lois de De Morgan $\forall a, b \in \{0; 1\}$:

1. $\neg(a \vee b) = \neg a \wedge \neg b$
2. $\neg(a \wedge b) = \neg a \vee \neg b$

Définition

Une forme clausale est une formule logique ne contenant que des disjonctions de littéraux négatifs ou positifs.

Mise en forme clausale

Pour obtenir une forme clausale à partir d'une formule logique quelconque :

1. élimination des $\Rightarrow$: $a \Rightarrow b \longrightarrow \neg a \vee b$
2. déplacement des $\neg$ vers l'intérieur : $\neg(a \vee b) \longrightarrow \neg a \wedge \neg b$; $\neg(a \wedge b) \longrightarrow \neg a \vee \neg b$
3. élimination des $\exists$ ("skolémisation") : $\forall x, \exists y P(y) \longrightarrow P(f(x))$
4. déplacement des $\forall$ vers l'extérieur : $\forall x, P(x) \vee \forall y, Q(y) \longrightarrow \forall x \forall y (P(x) \vee Q(y))$
5. distribution de $\wedge$ sur $\vee$: $a \vee (b \wedge c) \longrightarrow (a \vee b) \wedge (a \vee c)$

Définitions

littéral positif : forme prédicative $(P(t_1, t_2, \ldots, t_n))$

littéral négatif : négation d'une forme prédicative $(\neg P(t_1, t_2, \ldots, t_n))$

$$\forall x((\forall y(P(y) \Rightarrow R(y,x)) \Rightarrow Q(x))$$

1. élimination des $\Rightarrow$: $\forall x(\neg(\forall y(\neg P(y) \vee R(y,x)) \vee Q(x))$
2. déplacement des $\neg$: $\forall x((\exists y(P(y) \wedge \neg R(y,x)) \vee Q(x))$
3. élimination des $\exists$: $\forall x((P(f(x)) \wedge \neg R(f(x),x)) \vee Q(x))$
4. externalisation des $\forall$: $(P(f(x)) \wedge \neg R(f(x),x)) \vee Q(x)$
5. distribution de $\wedge$ sur $\vee$: $(P(f(x)) \vee Q(x)) \wedge (\neg R(f(x),x) \vee Q(x))$

Forme clausale : 2 clauses

1. $P(f(x)) \vee Q(x)$
2. $\neg R(f(x),x) \vee Q(x)$

TD (Mise en forme clausale)

Transformer les formules logiques suivantes en formes clausales.

1. $\forall x(P(x) \Leftrightarrow \exists y(Q(x,y)))$
2. $\forall z(P(z) \Rightarrow (\exists x(P(f(x)) \Rightarrow \forall y(Q(x,y)))))$
3. $\forall x(P(x) \Rightarrow ((\neg\forall y(Q(x,y) \Rightarrow (\exists u P(f(u))))) \wedge (\forall y(Q(x,y) \Rightarrow P(x)))))$

Définition

Une clause de Horn est une clause possédant au plus un littéral positif.

Trois types de clauses de Horn :

Faits : pas de littéral négatif

$$P(x, y, f(x, z))$$

Règles : un littéral positif et au moins un littéral négatif

$$P(x, y, f(x, z)) \vee \neg Q(x, y) \vee \neg R(y, z)$$

Questions : pas de littéral positif

$$\neg Q(x, y) \vee \neg R(y, z)$$

Remarque (Alfred Horn)

Alfred Horn (1918-2001) : mathématicien américain

- Horn A., *On Sentences Which are True of Direct Unions of Algebras*, The Journal of Symbolic Logic, 16(1):14-21, 1951.

Définition

Prolog est un langage de programmation déclarative qui repose sur la logique des prédicats restreinte aux clauses de Horn.

Prolog ≡ **Pro**grammation en **log**ique

De la logique à Prolog

1. Formule logique $\forall x((\exists z,\ P(z,x) \wedge Q(z)) \Rightarrow Q(x))$
2. Forme clausale $Q(x) \vee \neg P(z,x) \vee \neg Q(z)$
3. Clause Prolog `q(X) :- p(Z,X), q(Z).`

Un programme Prolog est un ensemble de clauses de Horn.

Remarque (éléments de la syntaxe Prolog)

opérateur		logique	Prolog
est impliqué par	*si*	$\Leftarrow$	:-
conjonction	*et*	$\wedge$	,
disjonction	*ou*	$\vee$	;

TD (de Horn à Prolog)

Ecrire sous forme de clauses Prolog les clauses de Horn suivantes.

1. $P(x,y)$
2. $P(x,y) \vee \neg Q(x)$
3. $P(x,y) \vee \neg Q(x) \vee \neg Q(y)$
4. $\neg Q(x) \vee \neg Q(y)$

Listing 1 – **interpreteur.pl**

```
1  :- consult('op-interpreteur.pl').
2
3  % modus ponens
4  demontrer(UneConclusion) :-
5      (DesConditions => UneConclusion),
6      demontrer(DesConditions).
7
8  % linéarité de la conjonction
9  demontrer(UneConclusion et UneAutreConclusion) :-
10     demontrer(UneConclusion),
11     demontrer(UneAutreConclusion).
12
13 % clause d'arrêt
14 demontrer(toujours_vrai).
```


Help

?- `help(consult/1).`

`consult(+File)` : Read `File` as a Prolog source file. `File` may be a list of files, in which case all members are consulted in turn... `consult/1` may be abbreviated by just typing a number of file names in a list.

Examples :

```
?- consult(load). % consult load or load.pl
?- [load]. % consult load or load.pl
```

Fichier Prolog (op-interpreteur.pl)

```
% définition d'opérateurs
% « sucre syntaxique » pour interpreteur.pl

% pouvoir écrire "a => b" plutôt que "=>(a,b)"
:- op(900,xfx,=>).

% pouvoir écrire "a et b" plutôt que "et(a,b)"
% pouvoir écrire "a et b et c" plutôt que "et(a,et(b,c))"
:- op(800,xfy,et).
```

Listing 2 – **genealogie.pl**

```
1  :- consult('op-genealogie.pl').
2
3  % Mes parents sont mes ancètres
4  (X parent_de Y) => (X ancetre_de Y).
5
6  % Les parents de mes ancètres sont mes ancètres
7  ((X parent_de Y) et (Y ancetre_de Z)) =>
8      (X ancetre_de Z).
9
10 % Cas particuliers
11 toujours_vrai => (emile parent_de jean).
12 toujours_vrai => (jean parent_de fabien).
```


Help

?- `help(op/3).`

`op(+Precedence, +Type, :Name)` : Declare `Name` to be an operator of type `Type` with precedence `Precedence`.

`Precedence` is an integer between 0 and 1200. `Type` is one of : `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` or `fx`. The `f` indicates the position of the functor, while `x` and `y` indicate the position of the arguments. `y` should be interpreted as "on this position a term with precedence lower or equal to the precedence of the functor should occur". For `x` the precedence of the argument must be strictly lower.

Fichier Prolog (op-genealogie.pl)

```
% définition d'opérateurs
% « sucre syntaxique » pour genealogie.pl

% pouvoir écrire "a parent_de b" plutôt que "parent_de(a,b)"
:- op(100,xfx,parent_de).

% pouvoir écrire "a ancetre_de b" plutôt que "ancetre_de(a,b)"
:- op(100,xfx,ancetre_de).
```

```
% consultation des fichiers sources
?- consult([interpreteur,genealogie]).
% op-interpreteur.pl compiled 0.00 sec, 592 bytes
% interpreteur compiled 0.01 sec, 88,628 bytes
% op-genealogie.pl compiled 0.00 sec, 540 bytes
% genealogie compiled 0.00 sec, 1,596 bytes
true.

% Emile est-il l'ancètre de Fabien ?
?- demontrer(emile ancetre_de fabien).
true.

% Emile est-il l'ancètre de Jacques ?
?- demontrer(emile ancetre_de jacques).
false.
```


www.swi-prolog.org

Remarque (démarrer SWI-Prolog)

```
$ pl
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.64)
Copyright (c) 1990-2008 University of Amsterdam. SWI-Prolog comes
with ABSOLUTELY NO WARRANTY. This is free software, and you are
welcome to redistribute it under certain conditions. Please visit
http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Remarque (quitter SWI-Prolog)

```
?- halt.
$
```

```
% Quels sont les ancètres de Fabien?
?- demontrer(Qui ancetre_de fabien).
Qui = jean ;
Qui = emile ;
false.

% De qui Emile est-il l'ancètre?
?- demontrer(emile ancetre_de Qui).
Qui = jean ;
Qui = fabien ;
false.
```


Help

?- help(help/1).

help(+What) : Show specified part of the manual. What is one of :

<Name>/<Arity>	Give help on specified predicate
<Name>	Give help on named predicate with any arity or C interface function with that name
<Section>	Display specified section. Section numbers are dash-separated numbers : 2-3 refers to section 2.3 of the manual. Section numbers are obtained using apropos/1.

Examples :

?- help(assert).	Give help on predicate assert
?- help(3-4).	Display section 3.4 of the manual
?- help('PL_retry').	Give help on interface function PL_retry()

See also apropos/1, and the SWI-Prolog home page at http://www.swi-prolog.org, which provides a FAQ, an HTML version of manual for online browsing and HTML and PDF versions for downloading.

```
% Qui est l'ancètre de qui ?
?- demontrer(Ascendant ancetre_de Descendant).
Ascendant = emile,
Descendant = jean ;
Ascendant = jean,
Descendant = fabien ;
Ascendant = emile,
Descendant = fabien ;
false.

?- halt.
```


TD (prédicat grand_parent_de)

Définir le prédicat « grand_parent_de(a,b) » vrai si a est le grand-parent de b.
grand_parent_de(a,b) sera utilisé sous la forme d'un opérateur binaire (a grand_parent_de b) grâce à la directive :
:- op(100,xfx,grand_parent_de).

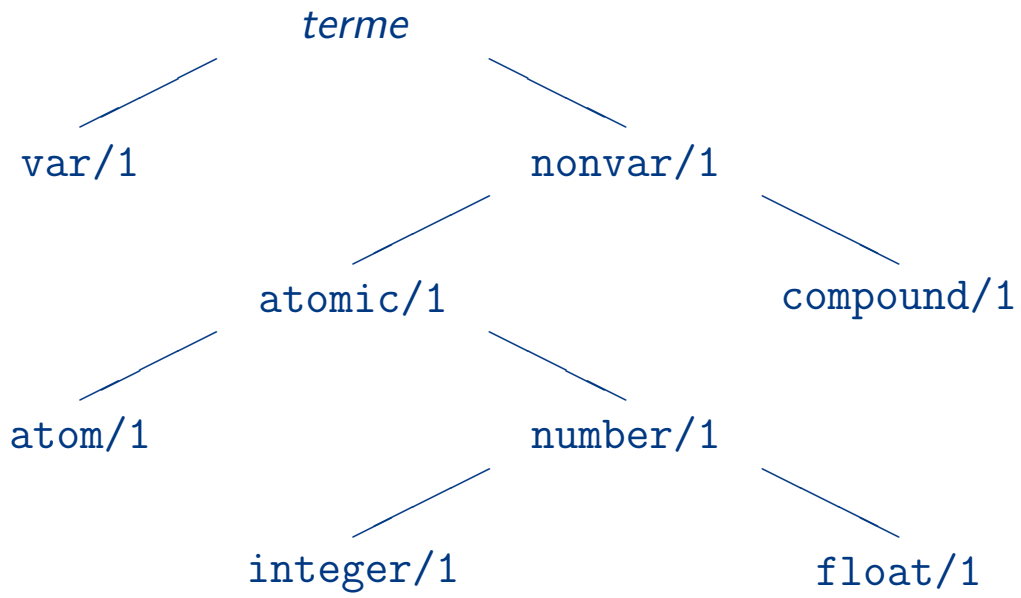
Exemples d'utilisation :

```
?- demontrer(emile grand_parent_de fabien).
true.
?- demontrer(emile grand_parent_de jacques).
false.
?- demontrer(emile grand_parent_de Qui).
Qui = fabien ;
false.
?- demontrer(Qui grand_parent_de fabien).
Qui = emile ;
false.
?- demontrer(GP grand_parent_de PE).
GP = emile,
PE = fabien ;
false.
```

- De la logique à Prolog
- Les termes en Prolog
- La machine Prolog
- Le contrôle de la résolution
- La recherche dans les graphes

Pour en savoir plus :

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990



La norme ISO-Prolog (ISO/IEC 13211-1) référencie les prédicats prédéfinis.

- Deransart P., Ed-Dbali A., Cervoni L.,*Prolog : the standard*, Springer Verlag, 1996

Remarque (classification des termes)

`atom/1`	`atom(T)`	: le terme T est-il un atome ?
`atomic/1`	`atomic(T)`	: le terme T est-il atomique ?
`number/1`	`number(T)`	: le terme T est-il un nombre ?
`float/1`	`float(T)`	: le terme T est-il un réel ?
`integer/1`	`integer(T)`	: le terme T est-il un entier ?
`compound/1`	`compound(T)`	: le terme T est-il composé ?
`nonvar/1`	`nonvar(T)`	: le terme T est-il instancié ?
`var/1`	`var(T)`	: le terme T est-il libre ?

Une variable (ou inconnue) peut remplacer n'importe quel terme Prolog.

variable instanciée à un terme : la variable a pour valeur ce terme.

variable libre : la variable n'est instanciée à aucun terme.

variables liées : des variables libres peuvent être liées entre-elles : dès que l'une d'entre elles sera instanciée à un terme, les autres variables qui lui sont liées le seront aussi.

variable `[_A-Z][_a-zA-Z0-9]*`
`X`
`Nom_compose`
`_variable`
`_192`
`_`

Remarque (Notation of Predicate Descriptions)

SWI-Prolog (www.swi-prolog.org)
We have tried to keep the predicate descriptions clear and concise. First the predicate name is printed in bold face, followed by the arguments in italics. Arguments are preceded by a **mode indicator**.

mode	definition
+	Argument must be fully instantiated to a term that satisfies the required argument type. Think of the argument as *input*. Example : `consult(+File)`
-	Argument must be unbound. Think of the argument as *output*. Example : `recorda(+Key, +Term, -Reference)`
?	Argument must be bound to a partial term of the indicated type. Think of the argument as *either input or output* or *both input and output*. Example : `functor(?Term,?Functor,?Arity)`
:	Argument is a meta-argument. Implies +. Example : `clause(:Head,?Body)`

Atomes : un atome (constante symbolique) permet de représenter un objet quelconque par un symbole.

identificateur `[a-z][a-zA-Z_0-9]*`	**opérateur** `[+-*/^<>=~ :.?#&@]+`	**atome 'quoté'** `'(([^'])|(''))*'`
`atome`	`= :=`	`'ATOME'`
`bonjour`	`:? :? :`	`'ca va?!'`
`c_est_ca`	`--->`	`'c''est ca'`

Nombres : entiers ou réels

TD (algèbre de Boole)

1. Traduire par des faits Prolog les tables de vérité des opérateurs logiques *et* $(a \cdot b)$, *ou* $(a + b)$ et *non* $(\overline{a})$:
   `et(X,Y,Z)` : $Z = X \cdot Y$
   `ou(X,Y,Z)` : $Z = X + Y$
   `non(X,Y)` : $Y = \overline{X}$
2. Vérifier en Prolog les principales propriétés des opérateurs logiques :
   - Commutativité : $X \cdot Y = Y \cdot X$ et $X + Y = Y + X$
   - Associativité :
     $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$ et $X + (Y + Z) = (X + Y) + Z$
   - Distributivité :
     $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$ et $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
   - Idempotence : $X + X = X$ et $X \cdot X = X$
   - Complémentarité : $X + \overline{X} = 1$ et $X \cdot \overline{X} = 0$
   - Théorèmes de De Morgan : $\overline{X \cdot Y} = \overline{X} + \overline{Y}$ et $\overline{X + Y} = \overline{X} \cdot \overline{Y}$

Termes composés

- Un terme composé permet de décrire des données structurées.
- Un terme composé (exemple : `date(25,mai,1988)`) est constitué
  - d'un atome (`date`)
  - d'une suite de termes (arguments : `(25,mai,1988)`) ; le nombre d'arguments (3) est appelé arité
- Le couple atome/arité (`date/3`) est appelé foncteur du terme composé correspondant.

Foncteur	Terme composé
`date/3`	`date(25,mai,1988)`
`'etat-civil'/3`	`'etat-civil'('ac''h',luc,date(1,mars,1965))`
`c/2`	`c(3.4,12.7)`
`c/4`	`c(a,B,c(1.0,3.5),5.2)`
`parallele/2`	`parallele(serie(r1,r2),parallele(r3,c1))`
`list/2`	`list(a,list(b,list(c,'empty list')))`

Terme composé ≡ Structure de données

Exemples de termes composés

une fiche d'état civil

individu

état–civil

Nom	:	Ngaoundéré
Prénom	:	Richard
Nationalité	:	camerounaise
Sexe	:	masculin

Date de naissance :

Jour	:	15
Mois	:	Février
Année	:	1960

adresse

Rue	:	4 rue Leclerc
Ville	:	Brest
Code postal	:	29200

```
% Structures de données
% etat_civil(Nom,Prenom,
              Nationalite,Sexe,Date)
% date(Jour,Mois,Annee)
% adresse(Rue,Ville,Code_postal)

% Base de données
% individu(Etat_civil,Adresse)
individu(
   etat_civil('Ngaoundere',
                          'Richard',
          camerounaise, masculin,
        date(15,'Fevrier',1960)),
   adresse('4 rue Leclerc',
                    'Brest',29200)
).
```


TD (base de données)

On considère une base de données contenant un certain nombre de fiches d'état-civil (définies ci-dessus).
Traduire par des buts Prolog les questions suivantes :

1. Quels sont les individus (nom,prénom) de nationalité française ?
2. Quels sont les individus (nom,prénom) de nationalité française habitant Brest ?
3. Y a-t-il des individus habitant la même adresse ?
4. Quelles sont les femmes (nom,prénom) de moins de 30 ans ?

Exemples de termes composés

Le vocabulaire initial de la théorie des entiers naturels comprend :

- une constante z qui représente l'entier 0 (zéro) : $z \in \mathbb{N}$
- une fonction unaire $s(X)$ qui traduit la notion de successeur d'un entier X : $\forall x \in \mathbb{N}, s(x) \in \mathbb{N}$
- un prédicat unaire *int(X)* (*X est un entier*)

Définition/Génération :

$z \in \mathbb{N}$
$\forall x \in \mathbb{N}, s(x) \in \mathbb{N}$

```
int(z).
int(s(X)) :-
    int(X).
```

Addition/Soustraction :

$\forall x \in \mathbb{N}, x + z = x$
$\forall x, y \in \mathbb{N}, x + s(y) = s(x + y)$

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


TD (les entiers naturels)

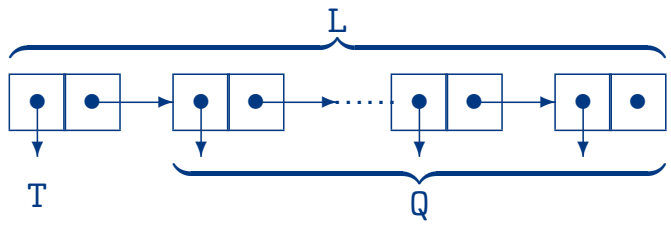
Pour les entiers naturels présentés ci-dessus, définir les opérations suivantes :

1. Multiplication : $\forall x \in \mathbb{N}, x \cdot z = z$ et $\forall x, y \in \mathbb{N}, x \cdot s(y) = x \cdot y + x$
2. Prédécesseur : $\forall x \in \mathbb{N}, p(s(x)) = x$
3. Inférieur ou égal : $\forall x \in \mathbb{N}, x \leq z \Leftrightarrow x = z$ et
   $\forall x, y \in \mathbb{N}, x \leq y \Leftrightarrow x = y$ ou $x \leq p(y)$
4. Strictement inférieur : $\forall x, y \in \mathbb{N}, x < y \Leftrightarrow x \leq y$ et $x \neq y$
5. Quotient et reste : $\forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, x < y \Rightarrow quot(x, y) = 0$ et
   $quot(x + y, y) = s(quot(x, y))$
   $\forall x \in \mathbb{N}, \forall y \in \mathbb{N}^*, x < y \Rightarrow rest(x, y) = x$ et $rest(x + y, y) = rest(x, y)$

Exemples de termes composés

un type de liste

Une liste L est une suite ordonnée de termes.



T	:	tête	(le premier terme)
Q	:	queue	(les autres termes)
l(T,Q)	:	liste	(T : premier terme)
			(Q : queue de liste)

Exemples : 0 *(liste vide)*
l(a,0)
l(a,l(b,l(c,0)))

Définition/génération :

```
list(0).
list(l(_,Q)) :- list(Q).
```

```
?- list(X).
X = 0 ;
X = l(_G2, 0) ;
X = l(_G2, l(_G3, 0)) ;
...
```

Appartenance à une liste :

```
in(T,l(T,_)).
in(X,l(_,Q)) :- in(X,Q).
```

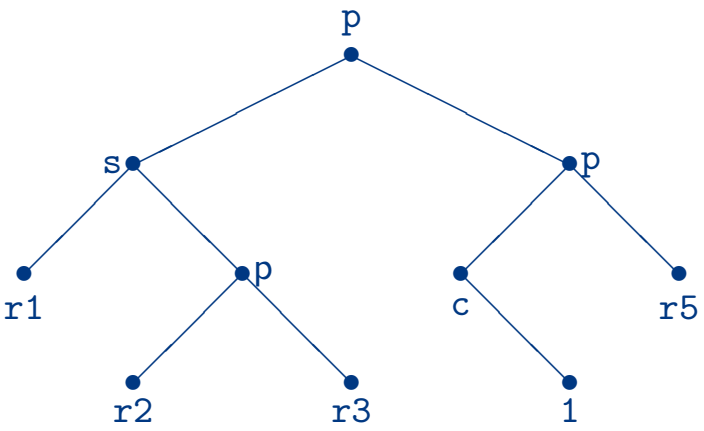
```
?- in(X,l(a,l(b,0))).
X = a ;
X = b ;
false.
```


TD (listes l(T,Q))

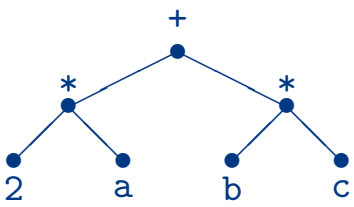
Définir les prédicats suivants :

1. premier(X,L) : X est le premier élément de la liste L.
   ```
   ?- premier(X,l(a,l(b,l(c,0)))).
   X = a.
   ```
2. dernier(X,L) : X est le dernier élément de la liste L.
   ```
   ?- dernier(X,l(a,l(b,l(c,0)))).
   X = c ;
   false.
   ```
3. prefixe(P,L) : P est un préfixe de la liste L.
   ```
   ?- prefixe(P,l(a,l(b,0))).
   P = 0 ;
   P = l(a, 0) ;
   P = l(a, l(b, 0)) ;
   false.
   ```
4. suffixe(S,L) : S est un suffixe de la liste L.
   ```
   ?- suffixe(S,l(a,l(b,0))).
   S = l(a, l(b, 0)) ;
   S = l(b, 0) ;
   S = 0 ;
   false.
   ```

```
p( s(r1,p(r2,r3)) , p(c(1),r5) )
```




```
2 * a + b * c
(2 * a) + (b * c)
```




```
?-
write_canonical(2*a+b*c).
+(*(2, a), *(b, c))
true.
```


TD (représentations arborescentes)

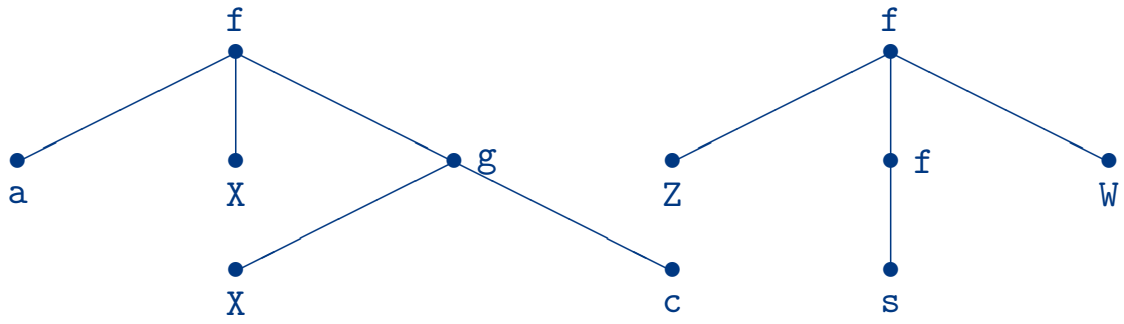
Représenter sous forme d'arbre les termes Prolog suivants :

1. `premier(X,l(a,l(b,l(c,0))))`
2. `3 = 1 + 2`
3. `in(X,l(T,Q)) :- in(X,Q)`
4. `individu(etat_civil('Ngaoundere', 'Richard', camerounaise, masculin, date(15,'Fevrier',1960)), adresse('4 rue Leclerc', 'Brest', 29200))`

Terme1 = Terme2

mise en correspondance d'arbres syntaxiques

`f(a,X,g(X,c))` $\stackrel{?}{=}$ `f(Z,f(s),W)`



$\rightarrow$ `f(a,f(s),g(f(s),c))`

```
?- f(a,X,g(X,c)) = f(Z,f(s),W).
X = f(s),
Z = a,
W = g(f(s), c).
```


TD (unifications)

Que donnent les appels suivants en Prolog ?

1. `?- X = 3.`
2. `?- 3 = 1 + 2.`
3. `?- f(X,f(X)) = f(a,Y).`
4. `?- f(f(f(X)) = f(Y).`
5. `?- X = f(X).`

Problème :

```
?- 3 = 1 + 2.
false.
```

Solution :

```
?- 3 is 1 + 2.
true.

?- X is cos(pi).
X = -1.0.

?- X is random(50).
X = 17.

?- X is 2**3.
X = 8.
```

```
?- current_arithmetic_function(F).
F = exp(_G2) ;
F = acos(_G2) ;
F = asin(_G2) ;
F = cos(_G2) ;
F = sin(_G2) ;
F = atan(_G2, _G3) ;
F = log(_G2) ;
F = atan(_G2) ;
F = tan(_G2) ;
F = sqrt(_G2) ;
F = random(_G2) ;
F = e ;
F = pi ;
...
false.
```


Help

?- `help(is/2).`

`-Number is +Expr` : True if `Number` has successfully been unified with the number `Expr` evaluates to. If `Expr` evaluates to a float that can be represented using an integer (i.e, the value is integer and within the range that can be described by Prolog's integer representation), `Expr` is unified with the integer value.

TD (suites récurrentes)

Définir les prédicats correspondant aux suites récurrentes suivantes :

1. Suite factorielle : $u_0 = 1$ et $u_n = n \cdot u_{n-1}\ \forall n \in \mathbb{N}^*$
2. Suite de Fibonacci : $u_0 = 1$, $u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}\ \forall n \in \mathbb{N}, n > 1$

TD (fonction d'Ackerman)

Définir le prédicat correspondant à la fonction d'Ackerman définie par les relations suivantes :

$$f : \mathbb{N}^2 \rightarrow \mathbb{N} \begin{cases} f(0,n) & = & n+1 \\ f(m,0) & = & f(m-1,1) \text{ si } m > 0 \\ f(m,n) & = & f(m-1, f(m,n-1)) \text{ si } m > 0, n > 0 \end{cases}$$

comparaisons lexicographiques

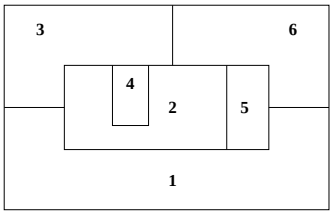
```
T1   ==   T2
T1   \==  T2
T1   @<   T2
T1   @=<  T2
T1   @>=  T2
T1   @>   T2
```

comparaisons numériques

```
N1   =:=  N2
N1   =\=  N2
N1   <    N2
N1   =<   N2
N1   >=   N2
N1   >    N2
```


TD (coloriage d'une carte)

On dispose de 4 couleurs (rouge,jaune,vert,bleu) pour colorier la carte représentée ci-dessous.



Ecrire un programme Prolog qui permette d'associer une couleur (rouge, jaune, vert, bleu) à une région (1, 2, 3, 4, 5, 6) de telle manière que deux régions adjacentes ne soient pas de la même couleur.

accès à la structure interne des termes

`functor(T, A, N)`
`T` est le terme de foncteur `A/N`.

```
?- functor(f(a,f(s),b),A,N).
A = f,
N = 3.

?- functor(T,f,3).
T = f(_G270, _G271, _G272).
```

`arg(N, T, X)`
`X` est le $N^{ième}$ argument du terme `T`.

```
?- arg(X,f(a,f(s),b),N).
X = 1,
N = a ;
X = 2,
N = f(s) ;
X = 3,
N = b.
```


?- `?- help(functor/3).`

`functor( ?Term, ?Functor, ?Arity)` : True if `Term` is a term with functor `Functor` and arity `Arity`. If `Term` is a variable it is unified with a new term holding only variables. If `Term` is an atom or number, `Functor` will be unified with `Term` and arity will be unified with the integer 0 (zero).

?- `?- help(arg/3).`

`arg( ?Arg, +Term, ?Value)` : `Term` should be instantiated to a term, `Arg` to an integer between 1 and the arity of `Term`. `Value` is unified with the `Arg`-th argument of `Term`.

`Arg` may also be unbound. In this case `Value` will be unified with the successive arguments of the term. On successful unification, `Arg` is unified with the argument number. Backtracking yields alternative solutions.

```
1  base(T) :-                % un terme est dit de base si
2      atomic(T).            % c'est un terme atomique
3  base(T) :-                % ou si
4      compound(T),          % c'est un terme composé
5      functor(T,F,N),       % dont les N arguments
6      base(N,T).            % sont des termes de base
7
8  base(N,T) :-              % les N arguments de T sont
9      N > 0,                % des termes de base si
10     arg(N,T,X),           % le Nème argument X
11     base(X),              % est un terme de base
12     N1 is N-1,            % et si les N-1 premiers arguments
13     base(N1,T).           % sont des termes de base
14 base(0,T).
```


Définition

terme de base : Un terme est dit de base si

1. c'est un terme atomique ;
2. c'est un terme composé dont tous les arguments sont des termes de base.

Opérateurs

« sucre syntaxique »

Un opérateur permet une représentation syntaxique simplifiée d'un terme composé, unaire ou binaire.

notation préfixée	`~X`	$\equiv$	`~(X)`
notation postfixée	`X:`	$\equiv$	`:(X)`
notation infixée	`X + Y`	$\equiv$	`+(X,Y)`

Un opérateur est caractérisé par son nom, sa priorité et son associativité.

priorité	`:+(2, *(5, 6))`	$\rightarrow$ `2 + 5 * 6`
associativité	`:+(+(2, 3), 4))`	$\rightarrow$ `2 + 3 + 4`

Aucune opération n'est a priori associée à un opérateur.

TD (priorité et associativité)

Ecrire sous forme fonctionnelle (exemple : `f(x,g(y))`) les expressions suivantes :

1. `1+2`
2. `1*2/3`
3. `1+2*3+4`
4. `1+2+3*4/5*6/7+8-9`

- Chaque opérateur a une priorité comprise entre 1 et 1200.
- La priorité détermine, dans une expression utilisant plusieurs opérateurs, l'ordre de construction du terme composé correspondant.

  `a + b * c` ≡ `a + (b * c)` ≡ `+(a, *(b,c))`
- La priorité d'une expression est celle de son foncteur principal.
- Les parenthèses donnent aux expressions qu'elles englobent une priorité égale à 0.
- La priorité d'un terme atomique est de 0.

position	associativité	notation	exemple
infixée	à droite	xfy	a , b , c
	à gauche	yfx	a + b + c
	non	xfx	x = y
préfixée	oui	fy	\+ \+ x
	non	fx	?- But
postfixée	oui	yf	
	non	xf	

Remarque (opérateurs prédéfinis)

priorité	opérateurs	associativité
1200	:- -->	xfx
1200	:- ?-	fx
1100	; |	xfy
1000	,	xfy
900	\+	fy
900	~	fx
700	= =.. == \== is = := =\= < >	xfx
700	< =< >= > @< @=< @>= @>	xfx
600	:	xfy
500	+ -	yfx
500	?	fx
400	* /	yfx
200	**	xfx
200	+ -	fy

Adapter la syntaxe Prolog aux besoins de l'utilisateur.

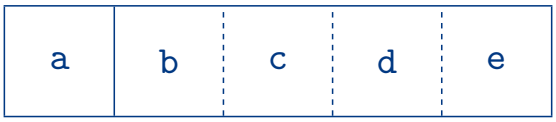
- op(P,A,Op) définit un nouvel opérateur de nom Op, de priorité P et d'associativité A.
- $1 \leq P \leq 1200$
- A est l'un des atomes xfx, xfy, yfx, fx, fy, xf, yf.

```
?- write_canonical(a et b et c et d).
ERROR : Syntax error : Operator expected
ERROR : write_canonical(a
ERROR : ** here **
ERROR : et b et c et d) .
?- op(200,xfy,et).
true.
?- write_canonical(a et b et c et d).
et(a, et(b, et(c, d)))
true.
```

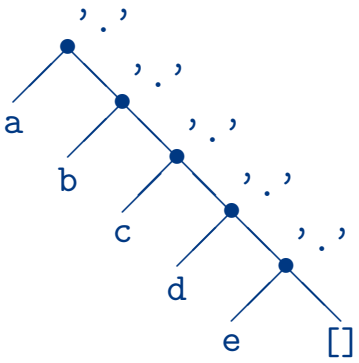

TD (définition d'opérateurs)

Définir les opérateurs Prolog nécessaires pour que les phrases suivantes soient des termes Prolog autorisés :

1. `diane est la secretaire de la mairie de brest`
2. `pierre est le maire de brest`
3. `jean est le gardien de but de brest`



liste vide : []
liste non vide : '.'(Tete,Queue)



Exemples :

```
[]
'.'(a,[])
'.'(a,'.'(b,[]))
'.'(a,'.'(b,'.'(c,[])))
'.'(a,'.'(b,'.'(c,'.'(d,[]))))
'.'(a,'.'(b,'.'(c,'.'(d,'.'(e,[])))))
```


Session Prolog

```
?- apropos(list).
 is_list/1        Type check for a list
 length/2         Length of a list
 sort/2           Sort elements in a list
 merge/3          Merge two sorted lists
 append/3         Concatenate lists
 member/2         Element is member of a list
 select/3         Select element of a list
 last/2           Last element of a list
 reverse/2        Inverse the order of the elements in a list
 permutation/2    Test/generate permutations of a list
 sumlist/2        Add all numbers in a list
 max_list/2       Largest number in a list
 min_list/2       Smallest number in a list
 numlist/3        Create list of integers in interval
 ...
 true
```

`[T1, T2, ... ,Tn|Reste]`

- `T1, T2, ... ,Tn` représente les n ($n > 0$) premiers termes de la liste
- `Reste` représente la liste des éléments restants
- on a l'équivalence :
  `[T1,... ,Tn]` ≡
  `[T1,... ,Tn|[]]`

`'.'(a,'.'(b,'.'(c,[])))`

≡ `[a|[b|[c|[]]]]`
≡ `[a|[b|[c]]]`
≡ `[a|[b,c|[]]]`
≡ `[a|[b,c]]`
≡ `[a,b|[c|[]]]`
≡ `[a,b|[c]]`
≡ `[a,b,c|[]]`
≡ `[a,b,c]`

Session Prolog

```
?- write_canonical([a|[b|[c|[]]]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a|[b|[c]]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a|[b,c|[]]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a|[b,c]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a,b|[c|[]]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a,b|[c]]).
'.'(a, '.'(b, '.'(c, [])))
true.
?- write_canonical([a,b,c]).
'.'(a, '.'(b, '.'(c, [])))
true.
```

```
% in(T,L) : T ∈ L
in(T,[T|_]).
in(T,[_|Q]) :-
    in(T,Q).
```

```
?- in(b,[a,b,c]).
true.
?- in(d,[a,b,c]).
false
?- in(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
false.
```

```
% nth(N,T,L) : T = L_N
nth(1,T,[T|_]).
nth(N,X,[_|Q]) :-
    nth(N1,X,Q),
    N is N1 + 1.
```

```
?- nth(2,X,[a,b,c]).
X = b ;
false.
?- nth(N,X,[a,b,c]).
N = 1,
X = a ;
N = 2,
X = b ;
N = 3,
X = c ;
false.
```


TD (informations sur les listes)

Définir les prédicats suivants :

1. `premier(X,L)` : `X` est le premier élément de la liste `L`.
   ```
   ?- premier(X,[a,b,c]).
   a.
   ```
2. `dernier(X,L)` : `X` est le dernier élément de la liste `L`.
   ```
   ?- dernier(X,[a,b,c]).
   X = c ;
   false.
   ```
3. `longueur(N,L)` : `N` est le nombre d'éléments de la liste `L`.
   ```
   ?- longueur(N,[a,b,c]).
   N = 3 ;
   false.
   ```
4. `occurence(X,N,L)` : `X` apparaît `N` fois dans la liste `L`.
   ```
   ?- occurence(N,X,[a,b,c,a,a]).
   N = 3,
   X = a ;
   N = 1,
   X = b ;
   N = 1,
   X = c ;
   false.
   ```

```
% conc(L1,L2,L3) : L1 + L2 = L3
conc([],L,L).
conc([T|Q],L,[T|QL]) :-
    conc(Q,L,QL).
```

```
?- conc([a,b],[c,d,e],[a,b,c,d,e]).
true.
?- conc([a,b],[c,d,e],L).
L = [a,b,c,d,e] ;
false.
?- conc([a,b],L,[a,b,c,d,e]).
L = [c,d,e] ;
false.
```

```
?- conc(L1,L2,[a,b,c]).
L1 = []      L2 = [a,b,c] ;
L1 = [a]     L2 = [b,c] ;
L1 = [a,b]   L2 = [c] ;
L1 = [a,b,c] L2 = [] ;
false.
```

```
% in(T,L) : T ∈ L
in(X,L) :-
    conc(_,[X|_],L).
```

```
?- in(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
false.
```


TD (manipulations de listes)

1. `supprimer(X,L1,L2)` : toutes les occurences de `X` sont supprimées de `L1` pour donner `L2`.

   ```
   ?- supprimer(X,[a,b,a,a],L).
   X = a,
   L = [b] ;
   X = b,
   L = [a, a, a] ;
   false.
   ```

2. `substituer(X,Y,L1,L2)` : substituer toutes les occurences de `X` par `Y` dans `L1` donne `L2`.

   ```
   ?- substituer(X,z,[a,b,a,a],L).
   X = a,
   L = [z, b, z, z] ;
   X = b,
   L = [a, z, a, a] ;
   false.
   ```

3. `inverser(L1,L2)` : inverser `L1` donne `L2`.

   ```
   ?- inverser([a,b,c],L).
   L = [c, b, a].
   ```

```
permutation([],[]).
permutation(L,[T|Q]) :-
    select(T,L,L1),
    permutation(L1,Q).

select(T,[T|Q],Q).
select(X,[T|Q],[T|L]) :-
    select(X,Q,L).
```

```
?- permutation([1,2,3],L).
L = [1,2,3] ;
L = [1,3,2] ;
L = [2,1,3] ;
L = [2,3,1] ;
L = [3,1,2] ;
L = [3,2,1] ;
false.
```

```
permutation_sort(L1,L2) :-
    % générer
    permutation(L1,L2),
    % tester
    inorder(L2).

% inorder(L) : L est triée
inorder([T]).
inorder([T1,T2|Q]) :-
    T1 < T2,
    inorder([T2|Q]).
```

```
?- permutation_sort([3,1,2],L).
L = [1,2,3] ;
false.
```


Help

```
?- help(=../2).
```

`?Term =.. ?List` : `List` is a list which head is the functor of `Term` and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both.

Examples :

```
?- foo(hello, X) =.. List.
 List = [foo, hello, X]
?- Term =.. [baz, foo(1)]
 Term = baz(foo(1))
```

TD (traitements génériques)

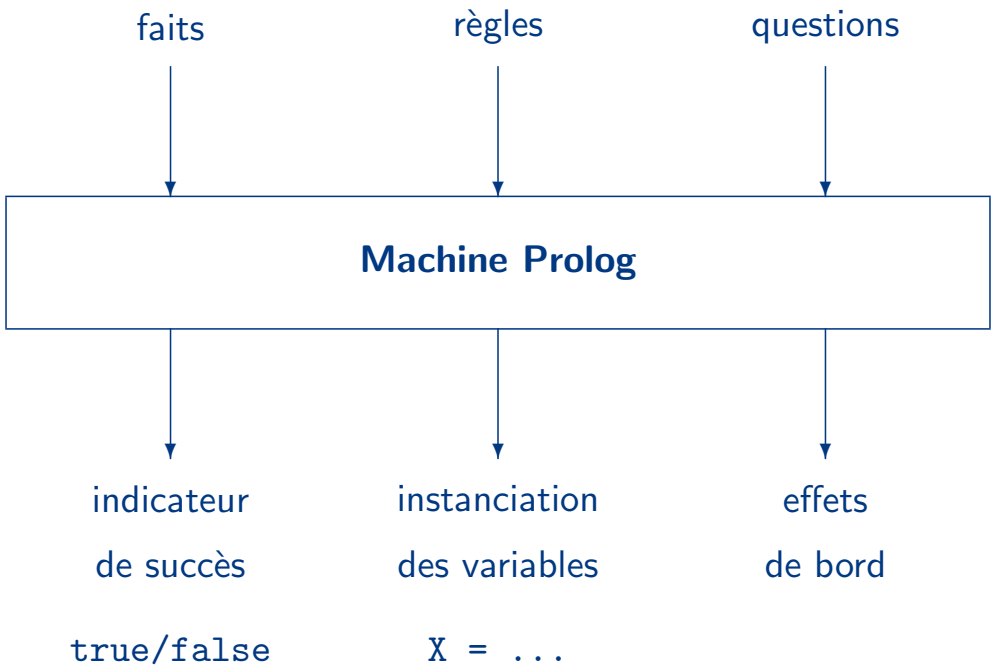
Définir le prédicat `map(T,L1,L2)` vrai si et seulement si `L2` représente `L1` où chaque élément a été traité à l'aide du prédicat de symbole fonctionnel `T`.

```
?- map(abs,[-1,2,-3],L).
L = [1,2,3]
true.
?- map(sin,[1,2,3],L).
L = [0.841471,0.909297,0.14112]
true.
```

- De la logique à Prolog
- Les termes en Prolog
- La machine Prolog
- Le contrôle de la résolution
- La recherche dans les graphes

Pour en savoir plus :

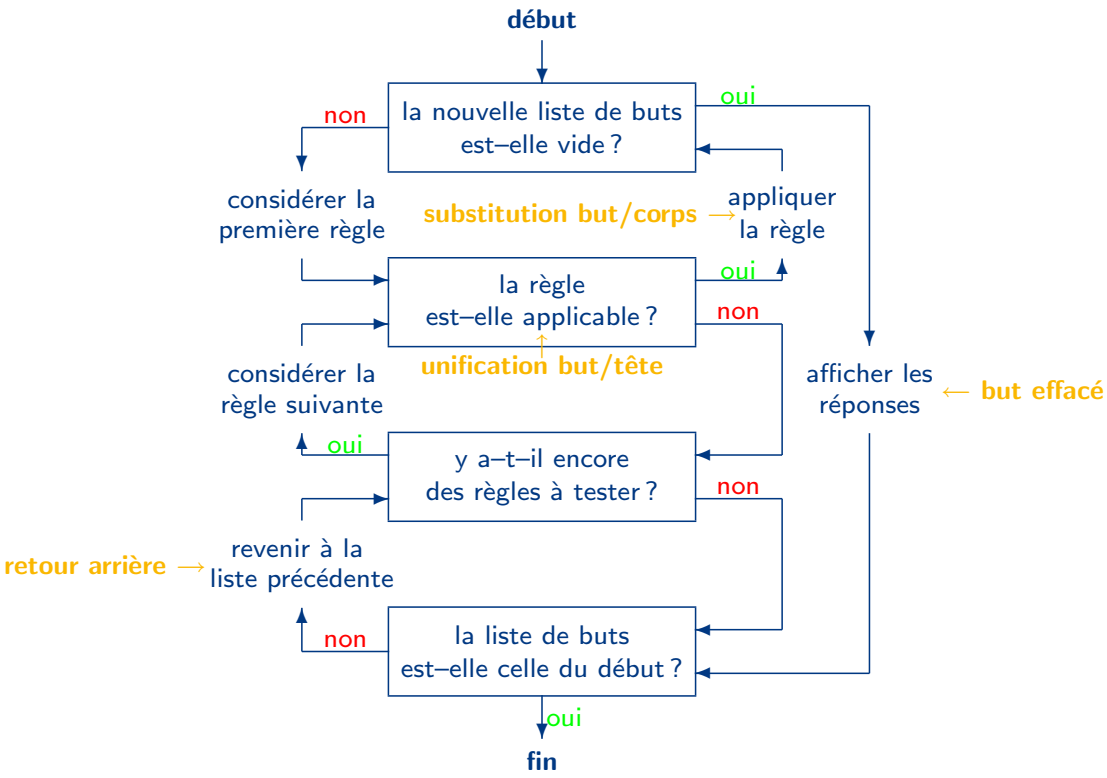
- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990



Pour en savoir plus :

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991

Algorithme Prolog



Effacer un but

$\{x_1, \ldots, x_r\}\ [b_1, b_2, \ldots, b_n]$

- Chercher une règle (dans l'ordre où elles apparaissent dans le programme) dont la tête s'unifie avec le but à effacer :
  - même foncteur (atome/arité)
  - arguments unifiables
- Remplacer le but par le corps de la règle applicable en tenant compte des substitutions de variables effectuées lors de l'unification. Si le corps de la règle est vide, le but est effacé.

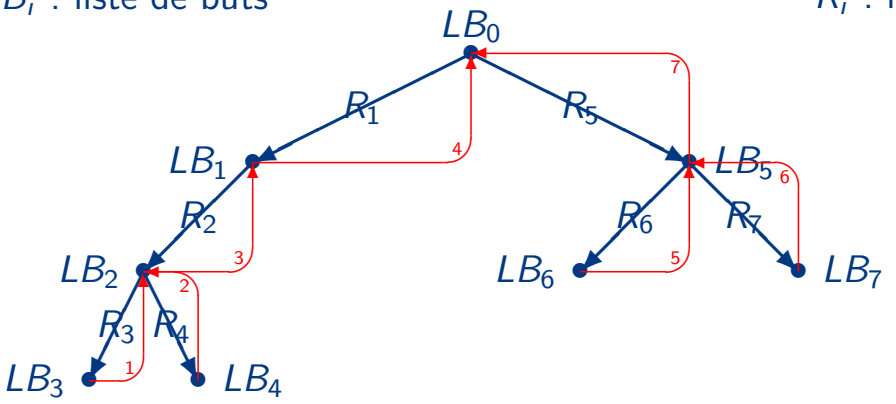
- règle : t :- $q_1, q_2, \ldots, q_m$
  unification : $t = b_1$,
  avec substitution :
  $\{x_i/t_i, x_j/t_j, \ldots\}$
- $\{x_1, \ldots, x_i/t_i, \ldots, x_r\}$
  $[q_1, q_2, \ldots, q_m, b_2, \ldots, b_n]$

$\{x_1/t_1, x_2/t_2, \ldots, x_r/t_r\}\ [\,]$

```
1  u(X,Y) :- var(X), var(Y), X = Y.
2  u(X,Y) :- var(X), nonvar(Y), X = Y.
3  u(X,Y) :- nonvar(X), var(Y), X = Y.
4  u(X,Y) :- atomic(X), atomic(Y), X == Y.
5  u(X,Y) :- compound(X), compound(Y), uTerme(X,Y).
6
7  uTerme(X,Y) :-
8      functor(X,F,N), functor(Y,F,N), uArgs(N,X,Y).
9
10 uArgs(N,X,Y) :-
11     N > 0, uArg(N,X,Y),
12     N1 is N-1, uArgs(N1,X,Y).
13 uArgs(0,X,Y).
14
15 uArg(N,X,Y) :-
16     arg(N,X,ArgX), arg(N,Y,ArgY), u(ArgX,ArgY).
```

LB_i : liste de buts

R_i : règle



succès : liste de buts vide

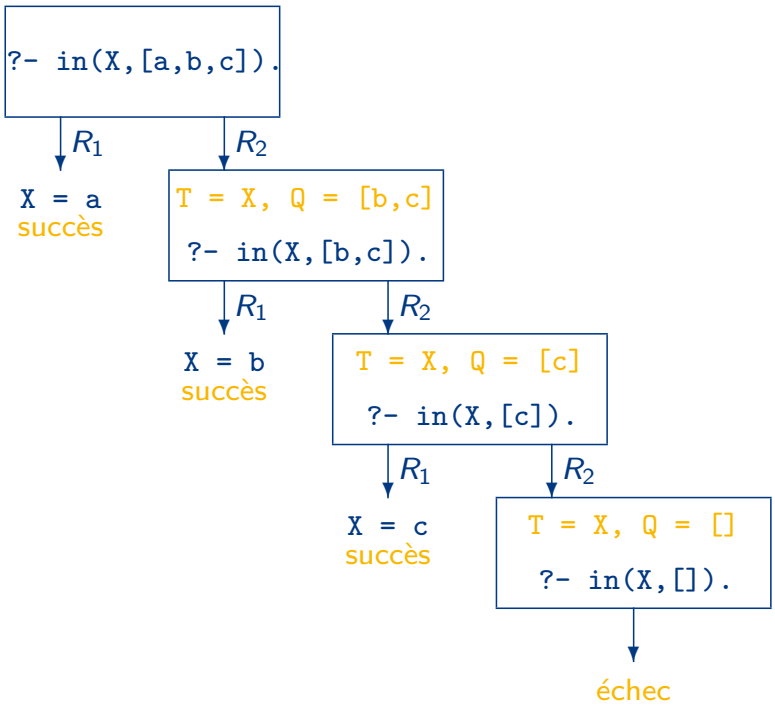
échec : plus de règle à appliquer

retour arrière : 

parcours de l'arbre en profondeur d'abord

```
% in(T,L) : T ∈ L
in(T,[T|_]).          % R1
in(T,[_|Q]) :-        % R2
    in(T,Q).
```

```
?- in(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
false.
```



TD (arbres de résolution)

On considère le prédicat conc/3 de concaténation de listes :

```
conc([],[],L).
conc([T|Q],L,[T|QL]) :- conc(Q,L,QL).
```

Tracer l'arbre de résolution pour chaque appel suivant :

1. `?- conc([a,b],[c,d,e],L).`
2. `?- conc(L,[c,d,e],[a,b,c,d,e]).`
3. `?- conc([a,b],L,[a,b,c,d,e]).`
4. `?- conc(L1,L2,[a,b,c]).`

visualiser la résolution d'un but de manière interactive

call	appel initial du but
exit	sortie avec succès du but
redo	retour arrière sur un but
fail	échec du but initial

```
?- trace.
[trace] ?- ...
```

```
[trace] ?- in(X,[a,b]).↩
Call : (7) in(_G341, [a, b])? ↩
Exit : (7) in(a, [a, b])? ↩
X = a ;
Redo : (7) in(_G341, [a, b])? ↩
Call : (8) in(_G341, [b])? ↩
Exit : (8) in(b, [b])? ↩
Exit : (7) in(b, [a, b])? ↩
X = b ;
Redo : (8) in(_G341, [b])? ↩
Call : (9) in(_G341, [])? ↩
Fail : (9) in(_G341, [])? ↩
false.
```


Remarque (ports de trace)

call : L'entrée par ce port de trace s'effectue avant la première tentative d'unification du but à une tête de clause de la base de clauses.

exit : La sortie par ce port de trace s'effectue lorsque le but a été unifié à une tête de clause et que tous les sous–buts éventuels du corps de la clause ont pu être prouvés.

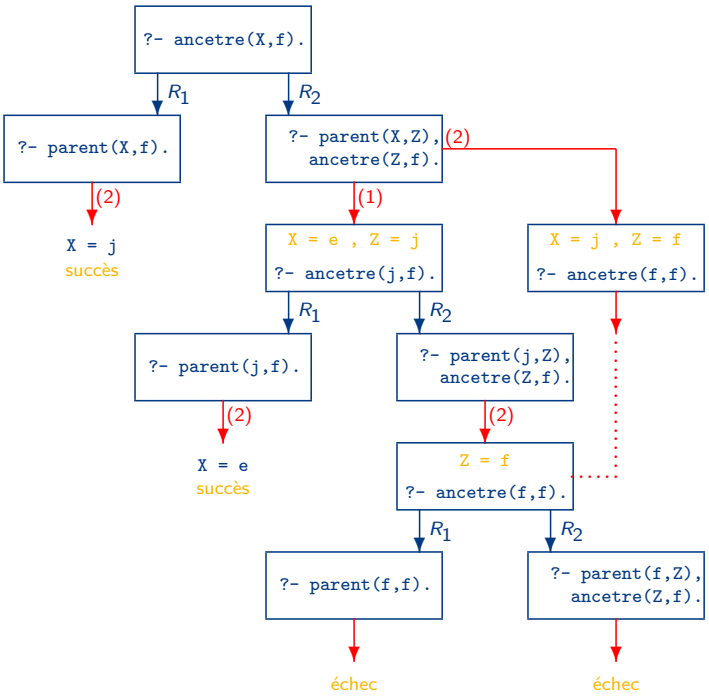
redo : L'entrée par ce port de trace s'effectue lors d'un retour arrière pour unifier le but à la tête de clause suivante dans la base de clauses.

fail : La sortie par ce port de trace s'effectue lorsque le but ne peut pas être unifié à une tête de clause de la base de clauses, ni à aucun prédicat prédéfini.

```
parent(e,j).      % (1)
parent(j,f).      % (2)

% ancetre/2 : v1
ancetre1(X,Y) :-  % R1
    parent(X,Y).
ancetre1(X,Y) :-  % R2
    parent(X,Z),
    ancetre1(Z,Y).

% ancetre/2 : v2
ancetre2(X,Y) :-  % R2
    parent(X,Z),
    ancetre2(Z,Y).
ancetre2(X,Y) :-  % R1
    parent(X,Y).
```



Session Prolog

```
[trace] ?- ancetre1(X,f).
Call : (7) ancetre1(_G557, f) ?
Call : (8) parent(_G557, f) ?
Exit : (8) parent(j, f) ?
Exit : (7) ancetre1(j, f) ?
X = j ;
Redo : (7) ancetre1(_G557, f) ?
Call : (8) parent(_G557, _L197) ?
Exit : (8) parent(e, j) ?
Call : (8) ancetre1(j, f) ?
Call : (9) parent(j, f) ?
Exit : (9) parent(j, f) ?
Exit : (8) ancetre1(j, f) ?
Exit : (7) ancetre1(e, f) ?
X = e ;
Redo : (8) ancetre1(j, f) ?
Call : (9) parent(j, _L225) ?
Exit : (9) parent(j, f) ?
Call : (9) ancetre1(f, f) ?
Call : (10) parent(f, f) ?
Fail : (10) parent(f, f) ?
Redo : (9) ancetre1(f, f) ?
Call : (10) parent(f, _L236) ?
Fail : (10) parent(f, _L236) ?
Redo : (8) parent(_G557, _L197) ?
Exit : (8) parent(j, f) ?
Call : (8) ancetre1(f, f) ?
Call : (9) parent(f, f) ?
Fail : (9) parent(f, f) ?
Redo : (8) ancetre1(f, f) ?
Call : (9) parent(f, _L208) ?
Fail : (9) parent(f, _L208) ?
false.
```

Ordonnancement des clauses

inversion des clauses

```
% ancetre/2 : version 1
ancetre1(X,Y) :-          % R1
    parent(X,Y).
ancetre1(X,Y) :-          % R2
    parent(X,Z),
    ancetre1(Z,Y).
```

```
?- ancetre1(X,f).
X = j ;
X = e ;
false.
```

```
% ancetre/2 : version 2
ancetre2(X,Y) :-          % R2
    parent(X,Z),
    ancetre2(Z,Y).
ancetre2(X,Y) :-          % R1
    parent(X,Y).
```

```
?- ancetre2(X,f).
X = e ;
X = j.
```

L'inversion des clauses ne modifie pas l'arbre de résolution, seul l'ordre des solutions est modifié.

Session Prolog

```
[trace] ?- ancetre2(X,f).
Call : (8) ancetre2(_G338, f) ?
Call : (9) parent(_G338, _L197) ?
Exit : (9) parent(e, j) ?
Call : (9) ancetre2(j, f) ?
Call : (10) parent(j, _L232) ?
Exit : (10) parent(j, f) ?
Call : (10) ancetre2(f, f) ?
Call : (11) parent(f, _L250) ?
Fail : (11) parent(f, _L250) ?
Redo : (10) ancetre2(f, f) ?
Call : (11) parent(f, f) ?
Fail : (11) parent(f, f) ?
Redo : (9) ancetre2(j, f) ?
Call : (10) parent(j, f) ?
Exit : (10) parent(j, f) ?
Exit : (9) ancetre2(j, f) ?
Exit : (8) ancetre2(e, f) ?
X = e ;
Redo : (9) parent(_G338, _L197) ?
Exit : (9) parent(j, f) ?
Call : (9) ancetre2(f, f) ?
Call : (10) parent(f, _L215) ?
Fail : (10) parent(f, _L215) ?
Redo : (9) ancetre2(f, f) ?
Call : (10) parent(f, f) ?
Fail : (10) parent(f, f) ?
Redo : (8) ancetre2(_G338, f) ?
Call : (9) parent(_G338, f) ?
Exit : (9) parent(j, f) ?
Exit : (8) ancetre2(j, f) ?
X = j.
```

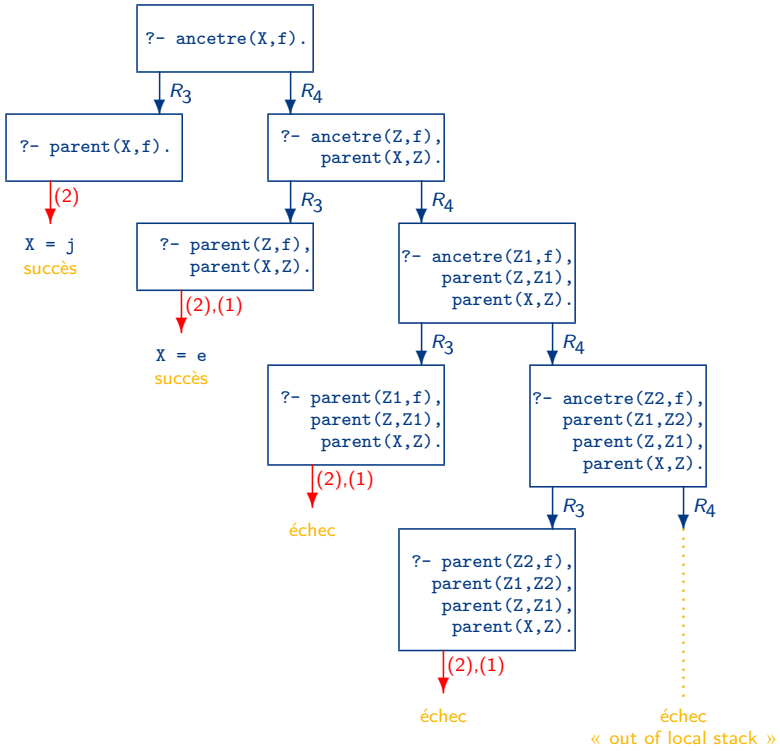
Ordonnancement des clauses

inversion des buts

```
parent(e,j).        % (1)
parent(j,f).        % (2)

% ancetre/2 : v1
ancetre1(X,Y) :-    % R1
    parent(X,Y).
ancetre1(X,Y) :-    % R2
    parent(X,Z),
    ancetre1(Z,Y).

% ancetre/2 : v3
ancetre3(X,Y) :-    % R3
    parent(X,Y).
ancetre3(X,Y) :-    % R4
    ancetre3(Z,Y),
    parent(X,Z).
```



Session Prolog

```
[trace] ?- ancetre3(X,f). Call : (8)
ancetre3(_G338, f) ?
Call : (9) parent(_G338, f) ?
Exit : (9) parent(j, f) ?
Exit : (8) ancetre3(j, f) ?
X = j ;
Redo : (8) ancetre3(_G338, f) ?
Call : (9) ancetre3(_L197, f) ?
Call : (10) parent(_L197, f) ?
Exit : (10) parent(j, f) ?
Exit : (9) ancetre3(j, f) ?
Call : (9) parent(_G338, j) ?
Exit : (9) parent(e, j) ?
Exit : (8) ancetre3(e, f) ?
X = e ;
Redo : (9) parent(_G338, j) ?
Redo : (9) ancetre3(_L197, f) ?
Call : (10) ancetre3(_L208, f) ?
Call : (11) parent(_L208, f) ?
Exit : (11) parent(j, f) ?
Exit : (10) ancetre3(j, f) ?
Call : (10) parent(_L197, j) ?
Exit : (10) parent(e, j) ?
Exit : (9) ancetre3(e, f) ?
Call : (9) parent(_G338, e) ?
Fail : (9) parent(_G338, e) ?
Redo : (10) parent(_L197, j) ?
Redo : (10) ancetre3(_L208, f) ?
Call : (11) ancetre3(_L219, f) ?
Call : (12) parent(_L219, f) ?
Exit : (12) parent(j, f) ?
Exit : (11) ancetre3(j, f) ?
Call : (11) parent(_L208, j) ?
Exit : (11) parent(e, j) ?
Exit : (10) ancetre3(e, f) ?
Call : (10) parent(_L197, e) ?
Fail : (10) parent(_L197, e) ?
Redo : (11) parent(_L208, j) ?
Redo : (11) ancetre3(_L219, f) ?
Call : (12) ancetre3(_L230, f) ?
Call : (13) parent(_L230, f) ?
Exit : (13) parent(j, f) ?
Exit : (12) ancetre3(j, f) ?
Call : (12) parent(_L219, j) ?
Exit : (12) parent(e, j) ?
Exit : (11) ancetre3(e, f) ?
Call : (11) parent(_L208, e) ?
Fail : (11) parent(_L208, e) ?
Redo : (12) parent(_L219, j) ?
Redo : (12) ancetre3(_L230, f) ?
Call : (13) ancetre3(_L241, f) ?
Call : (14) parent(_L241, f) ?
Exit : (14) parent(j, f) ?
Exit : (13) ancetre3(j, f) ?
Call : (13) parent(_L230, j) ?
Exit : (13) parent(e, j) ?
Exit : (12) ancetre3(e, f) ?
Call : (12) parent(_L219, e) ?
Fail : (12) parent(_L219, e) ?
...
ERROR : Out of local stack
```

Ordonnancement des clauses

inversion des buts

```
% ancetre/2 : version 1
ancetre1(X,Y) :-          % R1
    parent(X,Y).
ancetre1(X,Y) :-          % R2
    parent(X,Z),
    ancetre1(Z,Y).
```

```
?- ancetre1(X,f).
X = j ;
X = e ;
false.
```

```
% ancetre/2 : version 3
ancetre3(X,Y) :-          % R3
    parent(X,Y).
ancetre3(X,Y) :-          % R4
    ancetre3(Z,Y),
    parent(X,Z).
```

```
?- ancetre3(X,f).
X = j ;
X = e ;
ERROR : Out of local stack
```

L'inversion des buts dans une clause modifie l'arbre de résolution.

Ordonnancement des clauses

les 4 versions d'ancetre/2

```
ancetre1(X,Y) :-
    parent(X,Y).
ancetre1(X,Y) :-
    parent(X,Z),
    ancetre1(Z,Y).
```

```
?- ancetre1(X,f).
X = j ; X = e ;
no more solution
```

⇑

Permutation des clauses

⇓

```
ancetre2(X,Y) :-
    parent(X,Z),
    ancetre2(Z,Y).
ancetre2(X,Y) :-
    parent(X,Y).
```

```
?- ancetre2(X,f).
X = e ; X = j ;
no more solution
```

⇐ ⇒

Permutation des buts

⇐ ⇒

```
ancetre3(X,Y) :-
    parent(X,Y).
ancetre3(X,Y) :-
    ancetre3(Z,Y),
    parent(X,Z).
```

```
?- ancetre3(X,f).
X = j ; X = e ;
... Out of local stack
```

⇑

Permutation des clauses

⇓

```
ancetre4(X,Y) :-
    ancetre4(Z,Y),
    parent(X,Z).
ancetre4(X,Y) :-
    parent(X,Y).
```

```
?- ancetre4(X,f).
...
... Out of local stack
```


Session Prolog

```
[trace] ?- ancetre4(X,f).
Call : (7) ancetre4(_G335, f) ?
Call : (8) ancetre4(_L197, f) ?
Call : (9) ancetre4(_L215, f) ?
Call : (10) ancetre4(_L233, f) ?
Call : (11) ancetre4(_L251, f) ?
Call : (12) ancetre4(_L269, f) ?
Call : (13) ancetre4(_L287, f) ?
Call : (14) ancetre4(_L305, f) ?
Call : (15) ancetre4(_L323, f) ?
Call : (16) ancetre4(_L341, f) ?
Call : (17) ancetre4(_L359, f) ?
Call : (18) ancetre4(_L377, f) ?
Call : (19) ancetre4(_L395, f) ?
Call : (20) ancetre4(_L413, f) ?
Call : (21) ancetre4(_L431, f) ?
Call : (22) ancetre4(_L449, f) ?
Call : (23) ancetre4(_L467, f) ?
Call : (24) ancetre4(_L485, f) ?
Call : (25) ancetre4(_L503, f) ?
Call : (26) ancetre4(_L521, f) ?
Call : (27) ancetre4(_L539, f) ?
Call : (28) ancetre4(_L557, f) ?
Call : (29) ancetre4(_L575, f) ?
Call : (30) ancetre4(_L593, f) ?
Call : (31) ancetre4(_L611, f) ?
Call : (32) ancetre4(_L629, f) ?
Call : (33) ancetre4(_L647, f) ?
Call : (34) ancetre4(_L665, f) ?
Call : (35) ancetre4(_L683, f) ?
Call : (36) ancetre4(_L701, f) ?
Call : (37) ancetre4(_L719, f) ?
Call : (38) ancetre4(_L737, f) ?
Call : (39) ancetre4(_L755, f) ?
Call : (40) ancetre4(_L773, f) ?
Call : (41) ancetre4(_L791, f) ?
Call : (42) ancetre4(_L809, f) ?
Call : (43) ancetre4(_L827, f) ?
Call : (44) ancetre4(_L845, f) ?
Call : (45) ancetre4(_L863, f) ?
Call : (46) ancetre4(_L881, f) ?
Call : (47) ancetre4(_L899, f) ?
Call : (48) ancetre4(_L917, f) ?
Call : (49) ancetre4(_L935, f) ?
Call : (50) ancetre4(_L953, f) ?
Call : (51) ancetre4(_L971, f) ?
Call : (52) ancetre4(_L989, f) ?
Call : (53) ancetre4(_L1007, f) ?
Call : (54) ancetre4(_L1025, f) ?
Call : (55) ancetre4(_L1043, f) ?
Call : (56) ancetre4(_L1061, f) ?
Call : (57) ancetre4(_L1079, f) ?
Call : (58) ancetre4(_L1097, f) ?
Call : (59) ancetre4(_L1115, f) ?
Call : (60) ancetre4(_L1133, f) ?
Call : (61) ancetre4(_L1151, f) ?
Call : (62) ancetre4(_L1169, f) ?
Call : (63) ancetre4(_L1187, f) ?
...
ERROR : Out of local stack
```

```
% longueur1(N,L)
% processus récursif
longueur1(0,[]).
longueur1(N,[_|Q]) :-
    longueur1(NQ,Q),
    N is NQ + 1.
```

```
[trace] ?- longueur1(N,[a,b,c]).
Call : (7) longueur1(_G344, [a, b, c])?
Call : (8) longueur1(_L198, [b, c])?
Call : (9) longueur1(_L217, [c])?
Call : (10) longueur1(_L236, [])?
Exit : (10) longueur1(0, [])?
Call : (10) _L217 is 0+1?
Exit : (10) 1 is 0+1?
Exit : (9) longueur1(1, [c])?
Call : (9) _L198 is 1+1?
Exit : (9) 2 is 1+1?
Exit : (8) longueur1(2, [b, c])?
Call : (8) _G344 is 2+1?
Exit : (8) 3 is 2+1?
Exit : (7) longueur1(3, [a, b, c])?
N = 3 .
```

```
% longueur2(N,L)
% processus itératif
longueur2(N,L) :-
    longueur2(N,0,L).

longueur2(N,Sum,[_|Q]) :-
    Sum1 is Sum + 1,
    longueur2(N,Sum1,Q).
longueur2(N,N,[]).
```

```
[trace] ?- longueur2(N,[a,b,c]).
Call : (7) longueur2(_G344, [a, b, c])?
Call : (8) longueur2(_G344, 0, [a, b, c])?
Call : (9) _L216 is 0+1?
Exit : (9) 1 is 0+1?
Call : (9) longueur2(_G344, 1, [b, c])?
Call : (10) _L236 is 1+1?
Exit : (10) 2 is 1+1?
Call : (10) longueur2(_G344, 2, [c])?
Call : (11) _L256 is 2+1?
Exit : (11) 3 is 2+1?
Call : (11) longueur2(_G344, 3, [])?
Exit : (11) longueur2(3, 3, [])?
Exit : (10) longueur2(3, 2, [c])?
Exit : (9) longueur2(3, 1, [b, c])?
Exit : (8) longueur2(3, 0, [a, b, c])?
Exit : (7) longueur2(3, [a, b, c])?
N = 3 .
```


TD (récursivité non terminale → terminale)

On considère le prédicat `inverser(L1,L2)` ci-dessous où la liste L2 est le résultat de l'inversion de la liste L1.

```
% inverser1(L1,L2)
% processus récursif
inverser([],[]).
inverser([T|Q],L) :-
    inverser(Q,L1),
    conc(L1,[T],L).
```

Proposer une version récursive terminale de ce prédicat.

- De la logique à Prolog
- Les termes en Prolog
- La machine Prolog
- Le contrôle de la résolution
- La recherche dans les graphes

Pour en savoir plus :

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990

Certains prédicats ont un comportement procédural ; leurs effets ne sont pas effacés par retour arrière.

Coupe–choix : élagage de l'arbre de résolution

Gestion de la mémoire : ajout et/ou retrait de clauses à l'exécution

Entrées/Sorties : écriture ou lecture de termes

`!`	Réussit toujours mais annule tous les points de choix créés depuis l'appel du but parent.	*coupe–choix*
`\+ But`	`But` n'est pas démontrable.	*négation par l'échec*
`Cond -> Act`	if `Cond` then `Act`	*test simple*
`Cond -> Act1 ; Act2`	if `Cond` then `Act1` else `Act2`	*alternative simple*
`call(But)`	Evalue `But`.	*interpréteur*
`fail`	Echoue toujours.	*échec*
`once(But)`	Evalue `But` une seule fois.	*première solution*
`repeat`	Réussit toujours même en cas de retour arrière.	*boucle infinie*
`true`	Réussit toujours.	*succès*

TD

Définir les prédicats suivants sans utiliser de prédicats prédéfinis :

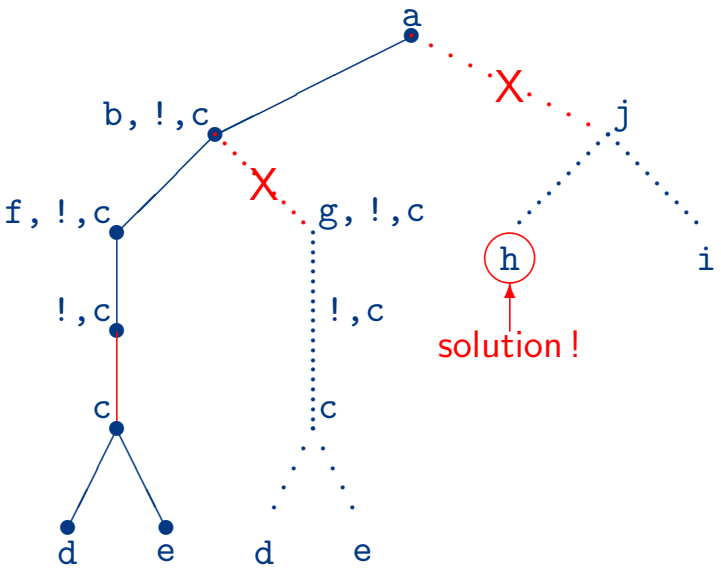
1. `true/0`
2. `fail/0`
3. `repeat/0`

Le coupe-choix (*cut*) a pour but d'élaguer l'arbre de résolution.

Cet élaguage *peut* conduire à

- une plus grande rapidité d'exécution,
- moins de place mémoire utilisée,
- modifier la sémantique du programme.

```
a :- b, !, c.    a :- j.    f.
b :- f.          b :- g.    g.
c :- d.          c :- e.    h.
j :- h.          j :- i.
```



Remarque (coupe-choix « vert » ou « rouge » ?)

- Dans *tous les cas*, l'interprétation procédurale du programme est modifiée : le programme ne s'exécute pas de la même manière avec ou sans coupe–choix.
- Dans *certains cas*, la signification déclarative du programme est conservée (coupe–choix « vert ») : le programme a la même interprétation logique avec ou sans coupe–choix.
- Dans *les autres cas*, la signification déclarative du programme est modifiée (coupe–choix « rouge ») : le programme n'a pas la même interprétation logique avec ou sans coupe–choix.

```
max(X,Y,X) :-
    X > Y,
    !.
max(X,Y,Y) :-
    X =< Y,
    !.
```

```
?- max(3,2,X).
X = 3.
?- max(2,3,X).
X = 3.
?- max(3,2,3).
true.
?- max(3,2,2).
false.
```

```
% alternative simple
:- op(900,fx,si)
:- op(850,xfx,alors)
:- op(800,xfx,sinon).

si Cond alors True sinon False :-
    call(Cond),
    !,
    call(True).
si Cond alors True sinon False :-
    not call(Cond),
    !,
    call(False).

% exemple d'utilisation
max(X,Y,Z) :-
    si X > Y alors Z = X sinon Z = Y.
```


Remarque (coupe-choix « vert »)

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est conservée.

```
max(X,Y,X) :-
    X > Y,
    !.
max(X,Y,Y).
```

```
?- max(3,2,X).
X = 3.
?- max(2,3,X).
X = 3.
?- max(3,2,3).
true.
?- max(3,2,2).
true. % aie !
```

```
sol(1). sol(2). sol(3).
% première solution
psol(X) :- sol(X), !.
% deuxième solution
dsol(X) :- psol(Y), sol(X), X ≡ Y, !.

% tests
?- sol(X).
X = 1 ;
X = 2 ;
X = 3.
?- ps(X).
X = 1.
?- ds(X).
X = 2.
```


Remarque (coupe-choix « rouge »)

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est modifiée.

```
% négation par
l'échec
non(But) :-
    call(But),
    !,
    fail.
non(But).
```

```
?- sol(X).
X = 1 ;
X = 2 ;
X = 3.
?- non(non(sol(2))).
true.
?- non(non(sol(4))).
false.
?- non(non(sol(X))).
true. % ≠ sol(X) !
```

```
[trace] ?- non(non(sol(X))).
Call : (7) non(non(sol(_G335)))?
Call : (8) call(non(sol(_G335)))?
Call : (9) non(sol(_G335))?
Call : (10) call(sol(_G335))?
Call : (11) sol(_G335)?
Exit : (11) sol(1)?
Exit : (10) call(sol(1))?
Call : (10) fail?
Fail : (10) fail?
Fail : (9) non(sol(_G335))?
Fail : (8) call(non(sol(_G335)))?
Redo : (7) non(non(sol(_G335)))?
Exit : (7) non(non(sol(_G335)))?
true.
```


Help

?- `help(\+).`

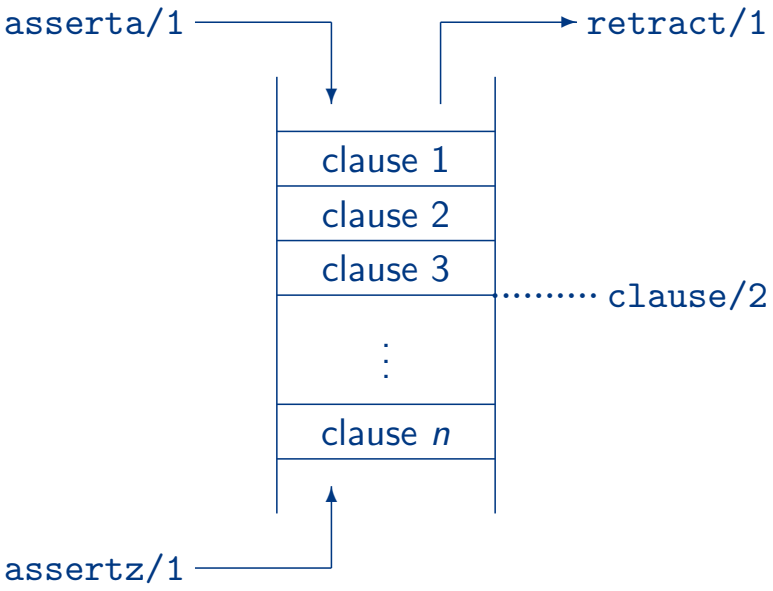
`\+ +Goal` : True if `Goal` cannot be proven.

mnemonic : + refers to provable and the backslash (\) is normally used to indicate negation in Prolog.

TD (->/2 et once/1)

Définir les prédicats suivants à l'aide de `call/1`, `fail/0` et `!/0` :

1. `->/2`
2. `once/1`




```
asserta(Clause)      : empiler une clause
assertz(Clause)      : enfiler une clause
clause(Tete,Corps)   : inspecter une clause
retract(Clause)      : dépiler (défiler) une clause
```


Help

?- `help(clause/2).`

`clause(:Head,?Body)` : True if `Head` can be unified with a clause head and `Body` with the corresponding clause body. Gives alternative clauses on backtracking. For facts `Body` is unified with the atom `true`.

?- `help(assert/1).`

`assert(+Term)` : Assert a fact or clause in the database. `Term` is asserted as the last fact or clause of the corresponding predicate.

?- `help(asserta/1).`

`asserta(+Term)` : Equivalent to `assert/1`, but `Term` is asserted as first clause or fact of the predicate.

?- `help(assertz/1).`

`assertz(+Term)` : Equivalent to `assert/1`.

?- `help(retract/1).`

`retract(+Term)` : When `Term` is an atom or a term it is unified with the first unifying fact or clause in the database. The fact or clause is removed from the database.

```
1  deduire :-
2          C => B,
3          tester(C),
4          affirmer(B),
5          !,
6          deduire.
7  deduire.
8
9  tester(C1 et C2) :- !, tester(C1), tester(C2).
10 tester(C) :- toujours_vrai => C.
11 tester(C) :- affirme => C.
12
13 affirmer(B) :-
14         \+ tester(B),
15         !, assertz(affirme => B),
16         write(B), write(' affirmé'), nl.
```


Session Prolog

```
?- listing('=>'/2).
:- dynamic (=>)/2.

B parent_de A et A parent_de C=>B grand_parent_de C.
A parent_de B=>A ancetre_de B.
B parent_de A et A ancetre_de C=>B ancetre_de C.
toujours_vrai=>emile parent_de jean.
toujours_vrai=>jean parent_de fabien.

true.

?- deduire.
emile grand_parent_de fabien affirmé
emile ancetre_de jean affirmé
jean ancetre_de fabien affirmé
emile ancetre_de fabien affirmé
true.
```

```
1  toutes(T,But,L) :-
2          call(But),
3          assertz(toutes_sol(T)),
4          fail.
5  toutes(T,But,L) :-
6          assertz(toutes_sol('toutes_fin')),
7          fail.
8  toutes(T,But,L) :-
9          recuperer(L).
10
11 recuperer([T|Q]) :-
12         retract(toutes_sol(T)),
13         T \== 'toutes_fin',
14         !,
15         recuperer(Q).
16 recuperer([]).
```


Session Prolog

```
?- listing(foo/3).
foo(1, 2, 3).
foo(1, 2, 4).
foo(2, 3, 5).
foo(2, 3, 6).
foo(3, 3, 7).

true.

?- toutes(C,foo(A,B,C),L).
L = [3, 4, 5, 6, 7].

?- toutes(C,foo(a,b,C),L).
L = [].

?- toutes(A+C,foo(A,B,C),L).
L = [1+3, 1+4, 2+5, 2+6, 3+7].

?- toutes(X,(foo(A,B,C),X is A+C),L).
L = [4, 5, 7, 8, 10].
```

```
?- bagof(C,foo(A,B,C),L).
A = 1,
B = 2,
L = [4, 3] ;
A = 2,
B = 3,
L = [5, 6] ;
A = 3,
B = 3,
L = [7].

?- bagof(C,foo(a,b,C),L).
false.

?- findall(C,foo(a,b,C),L).
[].
```

```
?- findall(C,foo(A,B,C),L).
L = [4, 5, 3, 7, 6].

?- bagof(C,A^foo(A,B,C),L).
B = 2,
L = [4, 3] ;
B = 3,
L = [5, 7, 6].

?- bagof(A+C,B^foo(A,B,C),L).
L = [1+4, 2+5, 1+3, 3+7, 2+6].

?- setof(A+C,B^foo(A,B,C),L).
L = [1+3, 1+4, 2+5, 2+6, 3+7].
```


Help

?- `help(bagof/3).`

`bagof(+Template, :Goal, -Bag)` : Unify Bag with the alternatives of `Template`, if `Goal` has free variables besides the one sharing with `Template` bagof will backtrack over the alternatives of these free variables, unifying Bag with the corresponding alternatives of `Template`. The construct `+Var^Goal` tells bagof not to bind `Var` in `Goal`. `bagof/3` fails if `Goal` has no solutions.

?- `help(setof/3).`

`setof(+Template, +Goal, -Set)` : Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

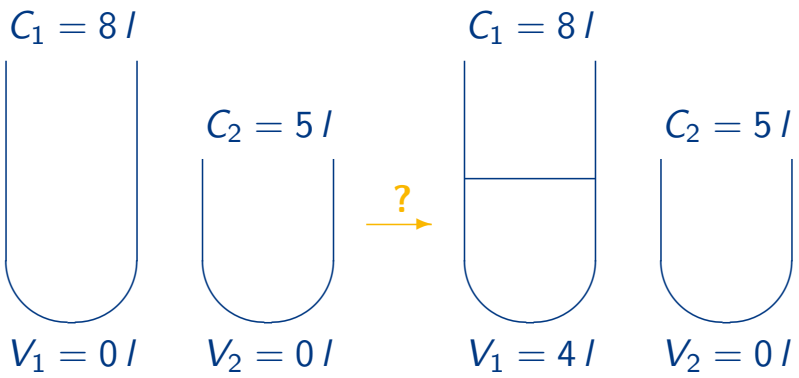
?- `help(findall/3).`

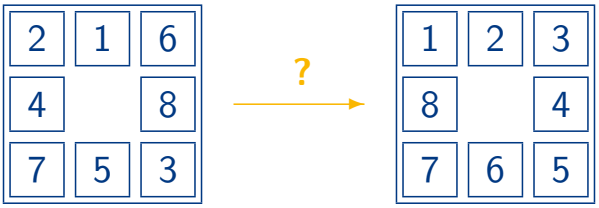
`findall(+Template, :Goal, -Bag)` : Creates a list of the instantiations `Template` gets successively on backtracking over `Goal` and unifies the result with Bag. Succeeds with an empty list if `Goal` has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existential operator (`^`), except that `bagof/3` fails when goal has no solutions.

- De la logique à Prolog
- Les termes en Prolog
- La machine Prolog
- Le contrôle de la résolution
- La recherche dans les graphes

Pour en savoir plus :

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990





Remarque (transvasements)

états :

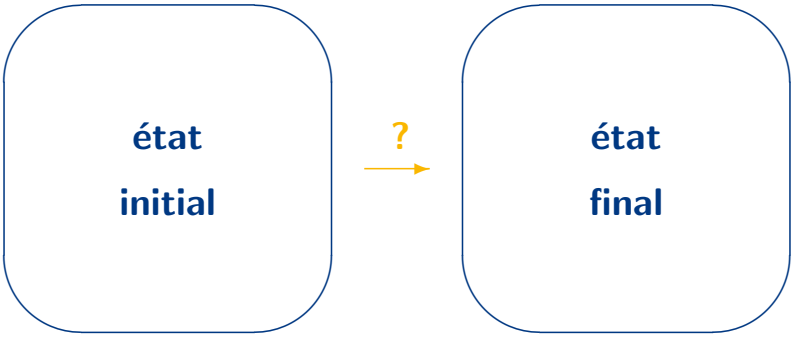
- état initial : `(0,0)`
- état final : `(4,0)`

opérations élémentaires :

- remplir complètement un récipient non plein ;
- vider complètement un récipient non vide ;
- transvaser un récipient dans l'autre ; on s'arrête dès que
  - le récipient source est vide ou
  - le récipient destination est plein.

TD (jeu du taquin)

Dans le cas du jeu du taquin, définir les états initial et final ainsi que les opérations élémentaires permises.

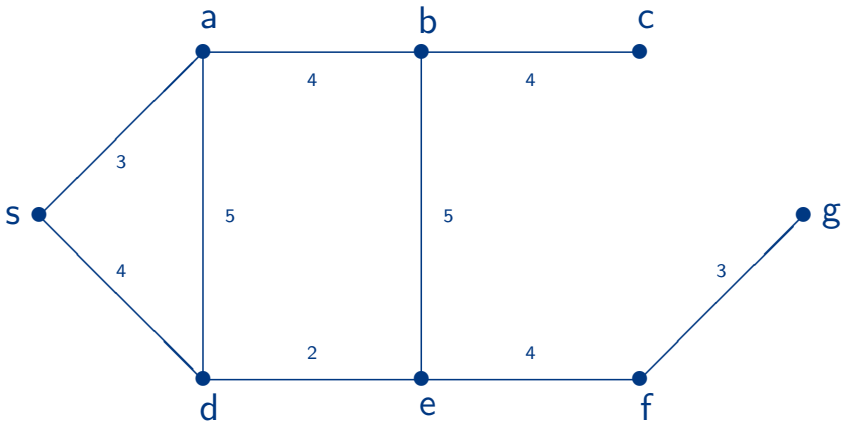


Problématique générale

Soit un système dans un état initial donné. On veut le faire passer dans un état final donné connaissant les opérations élémentaires que l'on peut appliquer sur le système.
Quelle(s) suite(s) d'opérations élémentaires doit-on appliquer au système pour le faire passer d'un état initial donné à un état final donné sans passer plus d'une fois par un même état ?

Remarque (transvasements)

état initial	(0,0)	
	↓	remplir 2
	(0,5)	↓
	↓	transvaser 2 dans 1
	(5,0)	↓
	↓	remplir 2
	(5,5)	↓
	↓	transvaser 2 dans 1
	(8,2)	↓
	↓	vider 1
	(0,2)	↓
	↓	transvaser 2 dans 1
	(2,0)	↓
	↓	remplir 2
	(2,5)	↓
	↓	transvaser 2 dans 1
	(7,0)	↓
	↓	remplir 2
	(7,5)	↓
	↓	transvaser 2 dans 1
	(8,4)	↓
	↓	vider 1
	(0,4)	↓
	↓	transvaser 2 dans 1
état final	(4,0)	




```
route(s,a,3). route(s,d,4). route(a,b,4). route(b,c,4). route(a,d,5).
route(b,e,5). route(d,e,2). route(e,f,4). route(f,g,3).
```

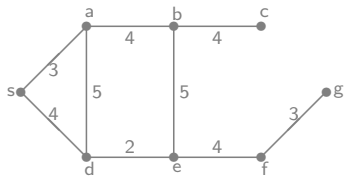
Comment aller de s en g ?

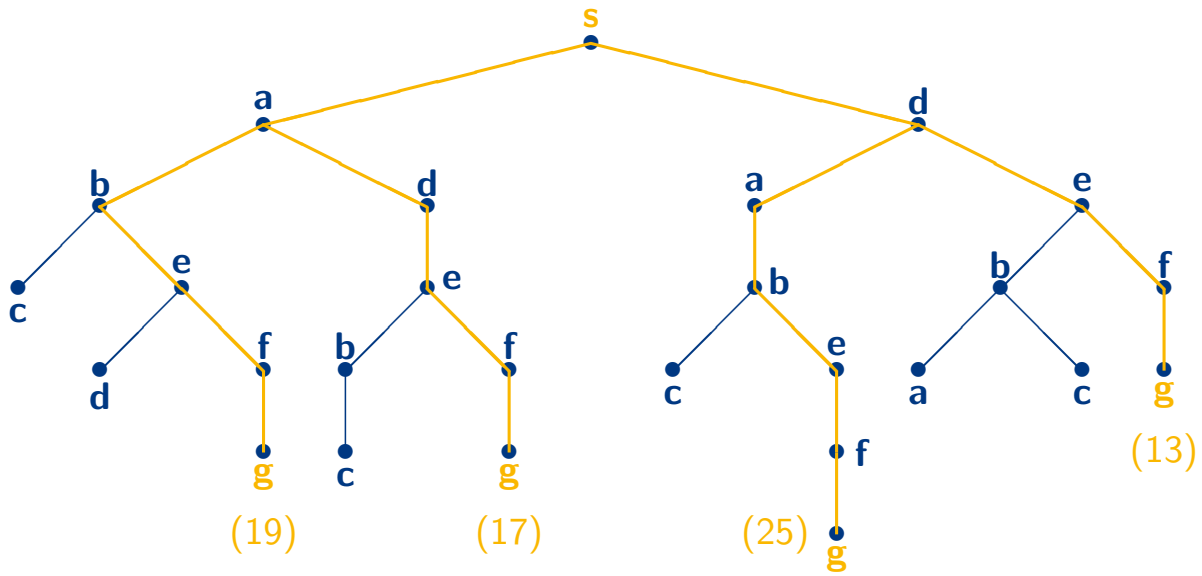
TD (réseau routier)

Dans le cas du réseau routier précédent, combien y a-t-il de chemins qui permettent d'aller de la ville s à la ville g sans passer deux fois par la même ville ?
Pour chaque chemin, on précisera le nombre de kilomètres parcourus.

Du graphe à l'arbre

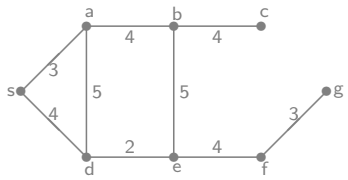
pour aller de s en g

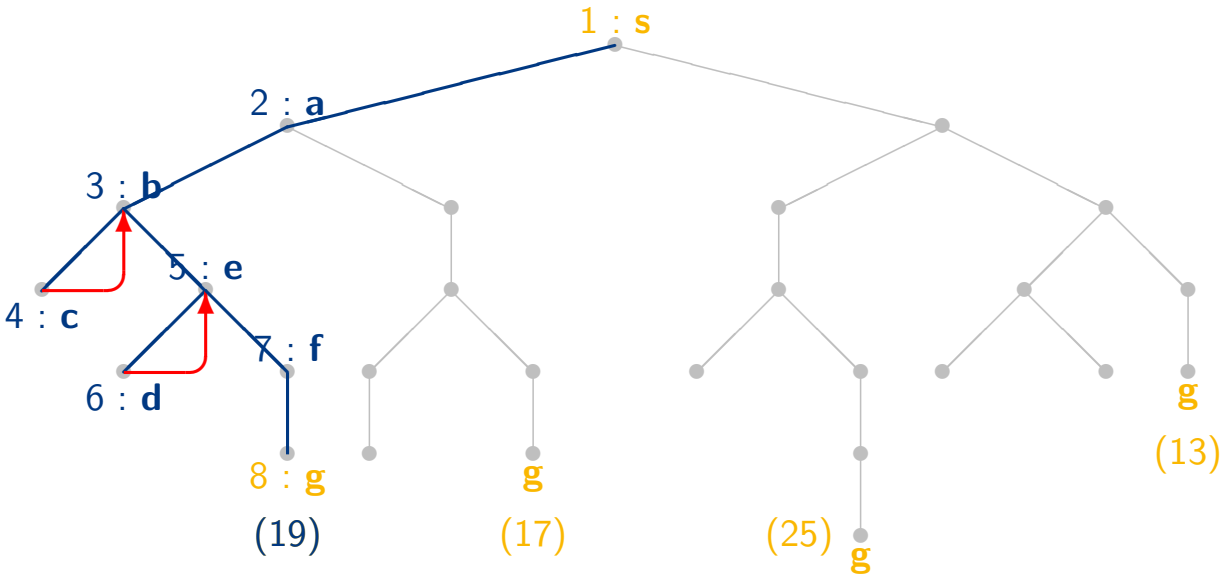




Recherche en profondeur

principe





Recherche en profondeur

version « Prolog »

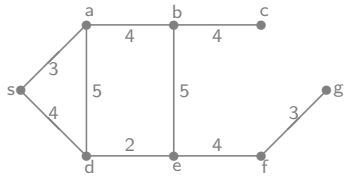
```
1 profondeur(D,A,Chemin,Km) :-
2         profondeur(D,A,[D],0,Chemin,Km).
3
4 profondeur(A,A,Chemin,Km,Chemin,Km).
5 profondeur(D,A,Passe,KP,Chemin,Km) :-
6         suivant(D,S,KDS),
7         \+ in(S,Passe),
8         KP1 is KP + KDS,
9         profondeur(S,A,[S|Passe],KP1,Chemin,Km).
```

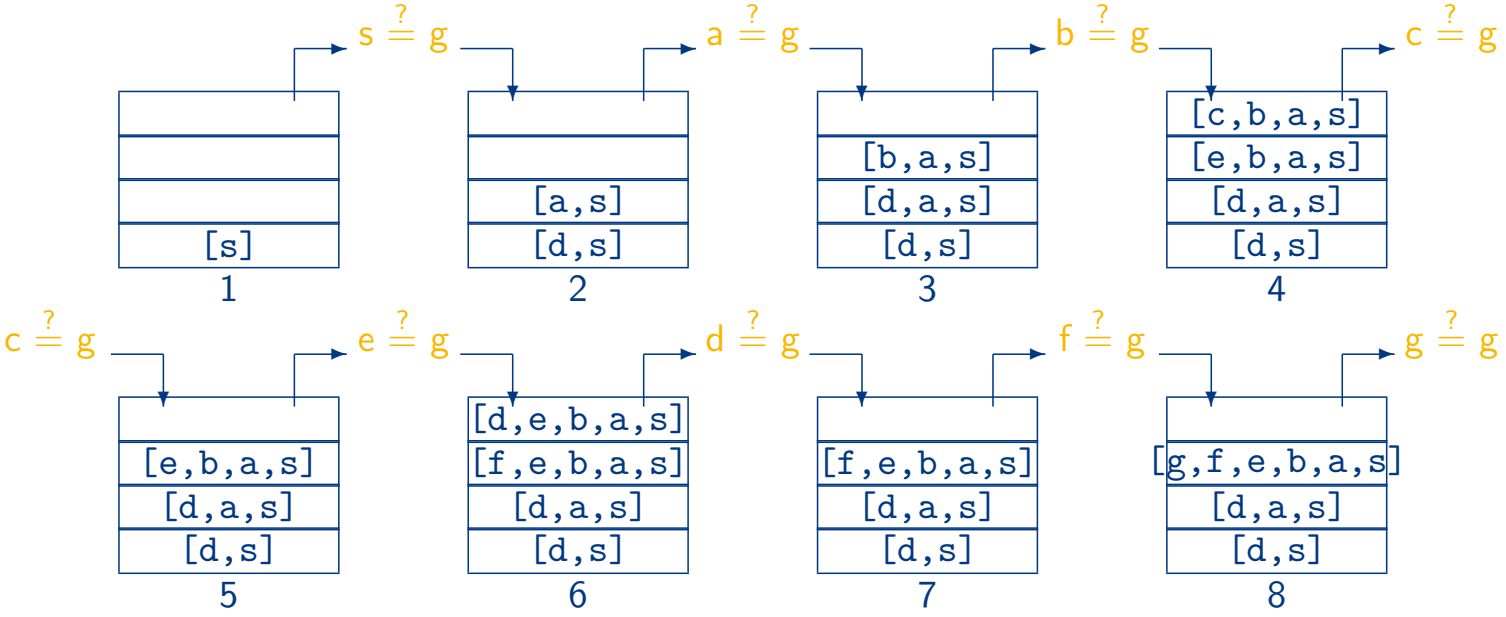

Session Prolog

```
?- listing(suivant/3).
suivant(B, A, C) :-
    ( route(A, B, C)
    ; route(B, A, C)
    ).

true.

?- profondeur(s,g,C,K).
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
C = [g, f, e, d, s],
K = 13 ;
false.
```





TD (transvasements)

Dans le cas du problème de transvasements

1. Définir les prédicats relatifs aux opérations élémentaires (vidage, remplissage, transvasement). On priviligérera les transvasements aux autres opérations en leur attribuant un coût de 1 contre 2 aux autres opérations (vidage, remplissage).
2. Définir le prédicat `suivant/3`.

   Exemple :
```
?- suivant((3,4),(V1,V2),N).
        V1 = 8, V2 = 4, N = 2 ; % remplir 1
        V1 = 3, V2 = 5, N = 2 ; % remplir 2
        V1 = 0, V2 = 4, N = 2 ; % vider 1
        V1 = 3, V2 = 0, N = 2 ; % vider 2
        V1 = 2, V2 = 5, N = 1 ; % transvaser 1 → 2
        V1 = 7, V2 = 0, N = 1 ; % transvaser 2 → 1
        false.
```

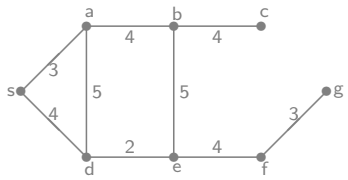
```
1  profondeur(D,A,Chemin,Cout) :-
2          profondeur1([0-[D]],A,Chemin,Cout).
3
4  profondeur1([K-[A|Passe]|_],A,[A|Passe],K).
5  profondeur1([K-[D|Passe]|Reste],A,Chemin,Cout) :-
6          suivants(K-[D|Passe],Suivants),
7          conc(Suivants,Reste,Pile1), %------- empiler
8          profondeur1(Pile1,A,Chemin,Cout).
9
10 suivants(K-[D|Passe],Suivants) :-
11         findall(K1-[S,D|Passe],
12                 (suivant(D,S,KDS),
13                  \+ member(S,[D|Passe]),
14                  K1 is K + KDS
15                 ),
16                 Suivants).
```

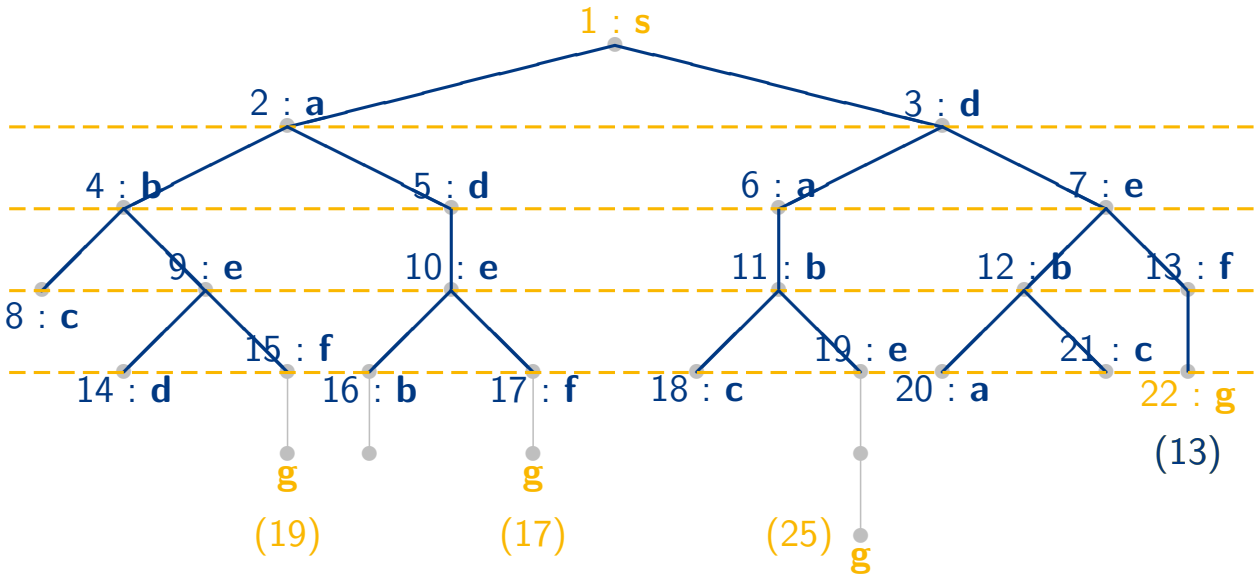

Session Prolog

```
?- profondeur(s,g,C,K).
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
C = [g, f, e, d, s],
K = 13 ;
false.
```

TD (transvasements)

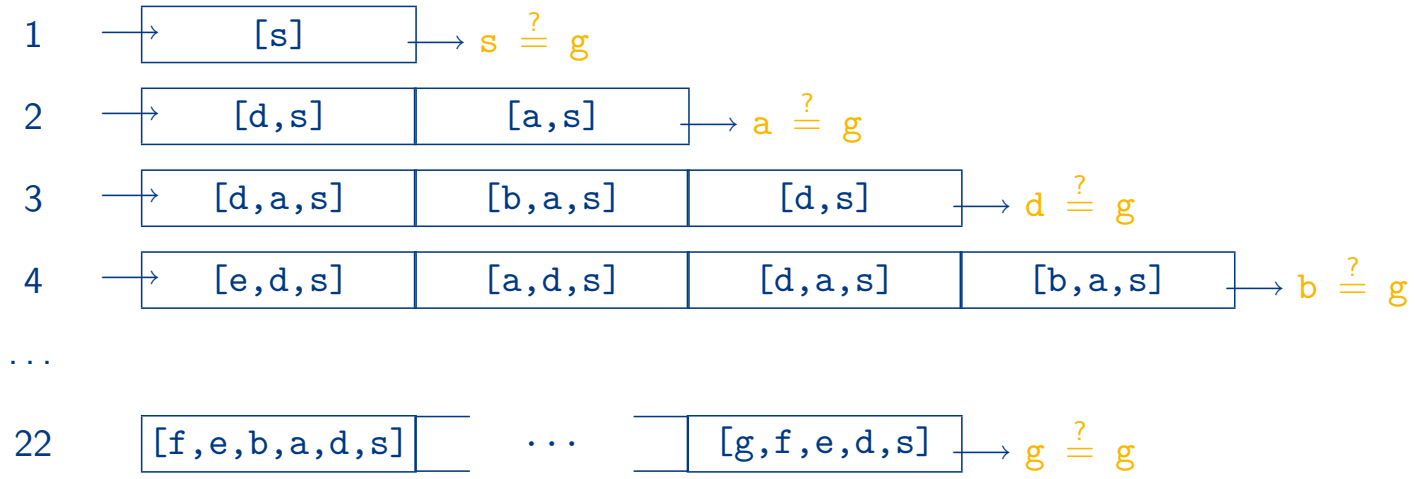
Appliquer la méthode de recherche en profondeur au problème de transvasements.


a
b
c
s
g
d
e
f
4
4
3
5
5
4
3
2
4


1 : s
2 : a
3 : d
4 : b
5 : d
6 : a
7 : e
8 : c
9 : e
10 : e
11 : b
12 : b
13 : f
14 : d
15 : f
16 : b
17 : f
18 : c
19 : e
20 : a
21 : c
22 : g
(13)
g
(19)
g
(17)
(25)
g

Recherche en largeur

utiliser une file



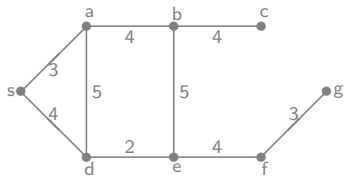
```
1  largeur(D,A,Chemin,Cout) :-
2          largeur1([0-[D]],A,Chemin,Cout).
3
4  largeur1([K-[A|Passe]|_],A,[A|Passe],K).
5  largeur1([K-[D|Passe]|Reste],A,Chemin,Cout) :-
6          suivants(K-[D|Passe],Suivants),
7          conc(Reste,Suivants,File1), %------- enfiler
8          largeur1(File1,A,Chemin,Cout).
9
10 suivants(K-[D|Passe],Suivants) :-
11         findall(K1-[S,D|Passe],
12                 (suivant(D,S,KDS),
13                  \+ member(S,[D|Passe]),
14                  K1 is K + KDS
15                 ),
16                 Suivants).
```

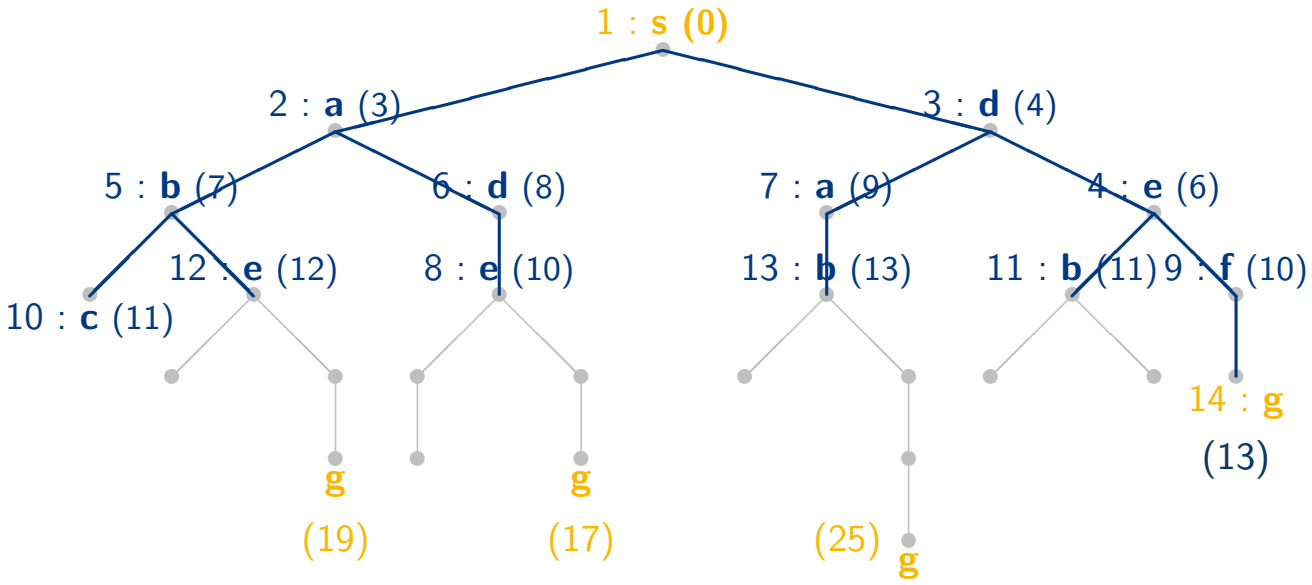

Session Prolog

```
?- largeur(s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.
```

TD (transvasements)

Appliquer la méthode de recherche en largeur au problème de transvasements.


a
b
c
s
d
e
f
g
4
4
3
5
5
4
3
2
4


1 : s (0)
2 : a (3)
3 : d (4)
5 : b (7)
6 : d (8)
7 : a (9)
4 : e (6)
12 : e (12)
8 : e (10)
13 : b (13)
11 : b (11)
9 : f (10)
10 : c (11)
14 : g
(13)
g
(19)
g
(17)
(25)
g

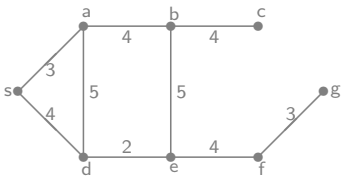
```
1  meilleur(D,A,Chemin,Cout) :-
2          meilleur1([0-[D]],A,Chemin,Cout).
3
4  meilleur1([K-[A|Passe]|_],A,[A|Passe],K).
5  meilleur1([K-[D|Passe]|Reste],A,Chemin,Cout) :-
6          suivants(K-[D|Passe],Suivants),
7          conc(Reste,Suivants,File1), %------- enfiler
8          sort(File1,FileTriee), %------------ trier
9          meilleur1(FileTriee,A,Chemin,Cout).
```


Session Prolog

```
?- meilleur(s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.
```

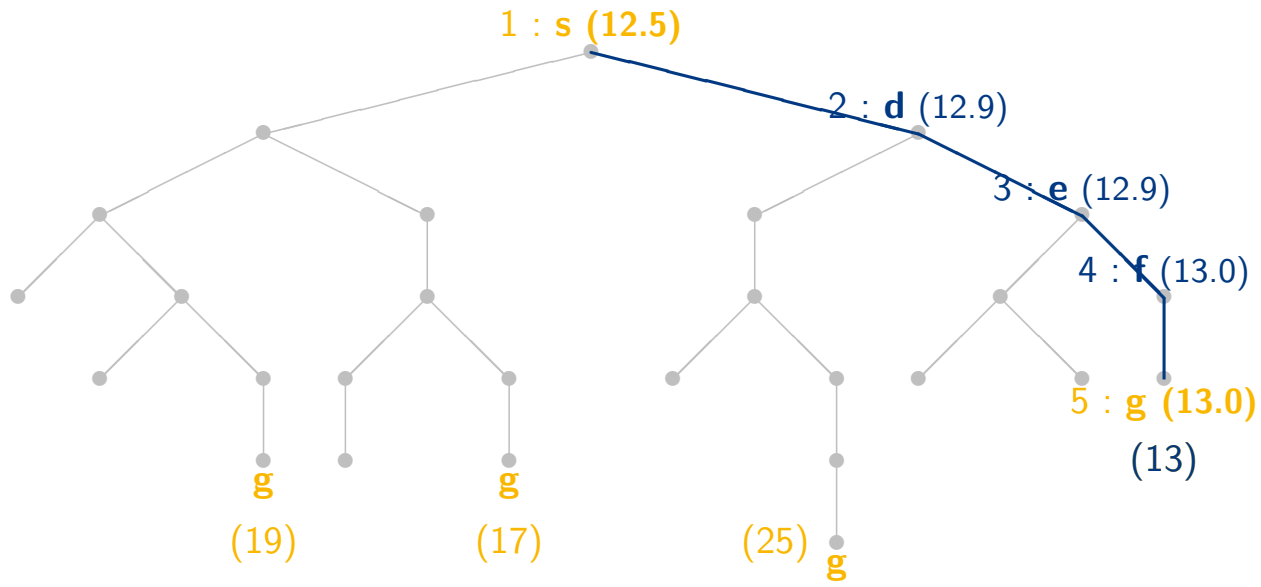
TD (transvasements)

Appliquer la méthode de recherche du meilleur chemin au problème de transvasements.



$$d_h(D \xrightarrow{V} A) = d_{eff}(D \to V) + estim(V \to A)$$

```
estim(a,g,10.4). estim(b,g,6.7). estim(c,g,4.0).
estim(d,g, 8.9). estim(e,g,6.9). estim(f,g,3.0).
estim(s,g,12.5). estim(g,g,0.0).
```



Définition

heuristique : méthode de résolution de problèmes, non fondée sur un modèle formel et qui n'aboutit pas nécessairement à une solution.

Une heuristique se fonde sur des règles empiriques pratiques, simples et rapides, facilitant l'exploration de l'arbre de recherche. Elle permet de réduire la complexité en examinant d'abord les cas qui ont le plus de chances de donner une réponse. Le choix d'une telle heuristique suppose de connaître déjà certaines propriétés du problème.

Remarque (distance heuristique)

$$d_h(D \xrightarrow{V} A) = d_{eff}(D \to V) + estim(V \to A)$$

$d_h(D \xrightarrow{V} A)$: distance heuristique pour aller de D en A en passant par V
$d_{eff}(D \to V)$: distance effective parcourue pour aller de D en V
$estim(V \to A)$: distance estimée pour aller de V en A

Recherche heuristique

utiliser une file (ou une pile) triée selon une heuristique

```
1  heuristique(D,A,Chemin,Cout) :-
2          estim(D,A,EDA),
3          heuristique1([EDA-0-[D]],A,Chemin,Cout).
4
5  heuristique1([E-K-[A|Passe]|_],A,[A|Passe],K).
6  heuristique1([E-K-[D|Passe]|Reste],A,Chemin,Cout) :-
7          suivants1(E-K-[D|Passe],A,Suivants),
8          conc(Reste,Suivants,File1), %------- enfiler
9          sort(File1,FileTriee), %------------ trier
10         heuristique1(FileTriee,A,Chemin,Cout).
```


Session Prolog

```
?- listing(suivants1/3).
suivants1(_-F-[A|B], D, J) :-
    findall(H-E-[C, A|B],
            (suivant(A, C, G),
             \+member(C, [A|B]),
             estim(C, D, I),
             E is F+G,
             H is E+I),
            J).

true.

?- heuristique(s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.
```

```
1  chemin(Methode,I,F,Chemin,Cout) :-
2          initialiser(Methode,I,F,PilFil),
3          rechercher(Methode,F,PilFil,Chemin,Cout).
4
5  initialiser(heuristique,I,F,[E-0-[I]]) :- !,
6          estim(I,F,E).
7  initialiser(_,I,_,[0-[I]]).
8
9  rechercher(profondeur,F,PilFil,Chemin,Cout) :-
10         profondeur1(PilFil,F,Chemin,Cout).
11 rechercher(largeur,F,PilFil,Chemin,Cout) :-
12         largeur1(PilFil,F,Chemin,Cout).
13 rechercher(meilleur,F,PilFil,Chemin,Cout) :-
14         meilleur1(PilFil,F,Chemin,Cout).
15 rechercher(heuristique,F,PilFil,Chemin,Cout) :-
16         heuristique1(PilFil,F,Chemin,Cout).
```


Session Prolog

```
?- chemin(profondeur,s,g,C,K).
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
C = [g, f, e, d, s],
K = 13 ;
false.

?- chemin(largeur,s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.
```

```
?- chemin(meilleur,s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.

?- chemin(heuristique,s,g,C,K).
C = [g, f, e, d, s],
K = 13 ;
C = [g, f, e, d, a, s],
K = 17 ;
C = [g, f, e, b, a, s],
K = 19 ;
C = [g, f, e, b, a, d, s],
K = 25 ;
false.
```

- Ait Kaci H., *Warren' s abstract machine : a tutorial reconstruction*, MIT Press, 1991
- Blackburn P., Bos J., Striegnitz K., *Prolog, tout de suite !*, (free online version : http://www.learnprolognow.org), Cahiers de Logique et d'Epistémologie, College Publications, 2007
- Bratko I., *Prolog programming for artificial intelligence*, Addison Wesley, 2000
- Clocksin F.W., Mellish C.S., *Programming in Prolog : using the ISO standard*, Springer, 2003
- Coello H., Cotta J.C., *Prolog by example. How to learn, teach and use it*, Springer, 1988
- Deransart P., Ed-Dbali A., Cervoni L., *Prolog : the standard*, Springer, 1996
- O'Keefe R.A., *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E., *L'art de Prolog*, Masson, 1990

Implémentations « domaine public »

- Ciao Prolog : www.clip.dia.fi.upm.es/Software/Ciao (université de Madrid)
- GNU Prolog : www.gprolog.org (INRIA)
- SWI Prolog : www.swi-prolog.org (université d'Amsterdam)
- YAP Prolog : www.dcc.fc.up.pt/~vsc/Yap (université de Porto)