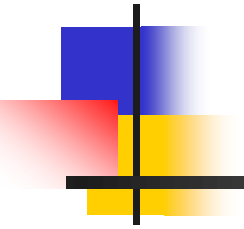
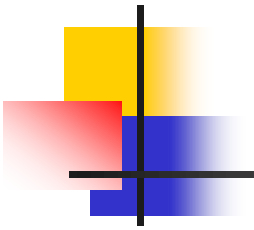


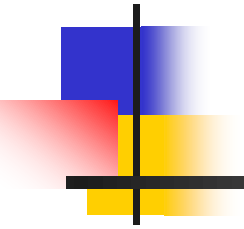
Logique et programmation

INFO 631

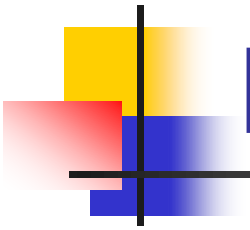


Contenu du cours

- Logique des propositions
  - Langage, Sémantique
  - Aspects déductifs
  - Formes normales, principe de résolution
- Logique des prédicats du premier ordre
  - Quantificateurs, Variables
  - Modèles, Preuves
  - Normalisation, Unification, Résolution
- Programmation logique
  - Programmer en logique
  - Clauses de Horn
  - Bases de PROLOG
- Logique floue
  - Interface numérique/symbolique
  - Extension des opérateurs logiques
  - Raisonnement



Programmation logique

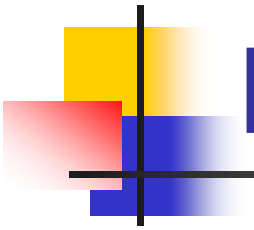


Programmer en logique

Programmation = logique + contrôle

Programmation logique	Programmation impérative
Formule	Procédure
Ensemble de formules	Programme
Question	Appel de procédure
Preuve	Exécution
Substitution (unification)	Passage de paramètres

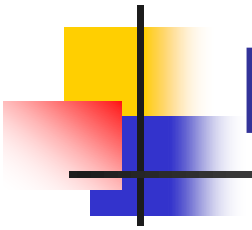
Contrôle : orienter et modifier le déroulement de la preuve



Programmer en logique

Principales caractéristiques des langages de programmation logique

- Langages logiques : le langage est un sous-ensemble de la logique et une exécution est une preuve.
- Langages symboliques : les données manipulées sont principalement des symboles.
- Langages déclaratifs : le « que faire » plutôt que le « comment faire ».
- Langages relationnels : description de l'état du « monde » en termes de données et de prédicats (relations) entre ces données.



Programmer en logique

Langage naturel

Il y a un vol direct entre

Paris et Hongkong,

Paris et Tokyo,

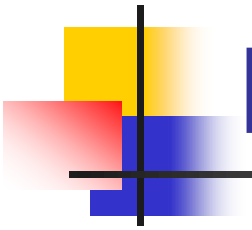
Hongkong et Sydney,

Sydney et Melbourne.

S'il y a un vol direct entre deux villes, ces deux villes sont connectées.

S'il y a un vol direct entre les villes x et z puis une connection entre z et y, alors les villes x et y sont connectées.

Est-ce que Paris et Melbourne sont connectés ?



Programmer en logique

Traduction en logique en premier ordre

Constantes : Paris, Hongkong, Tokyo, Sydney et Melbourne

Prédicats : direct et connection d'arité 2

Programmer en logique

Traduction en langage PROLOG (fichier texte Vol.pl)

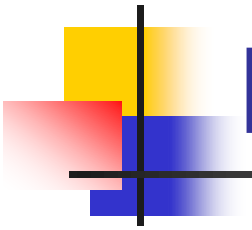
```
% Faits
direct(paris, hongkong).
direct(paris, tokyo).
direct(hongkong, sydney).
direct(sydney, melbourne).

% Règles
connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

Fichier texte
Vol.pl

Exécution dans l'environnement SWI-Prolog

```
?- consult('E://INFO631/Prolog/Vol.pl').
?- connection(paris, melbourne).
Yes
```

Chargement
Question ou but
Réponse obtenue



Programmer en logique

Mise sous forme clausale

Négation de la formule à déduire

Ensemble de clauses dont on cherche à prouver l'inconsistance

Programmer en logique

Résolution « input linéaire »

S = { direct(Paris, Hongkong), direct(Paris,Tokyo),
direct(Hongkong, Sydney), direct(Sydney, Melbourne),
$\neg$direct(x, y) $\vee$ connection(x, y),
$\neg$direct(x, z) $\vee$ $\neg$connection(z, y) $\vee$ connection(x, y),
$\neg$connection(Paris, Melbourne)}

Programmer en logique

- Le système d'exécution des programmes PROLOG est un démonstrateur de théorèmes fondé sur la résolution et l'unification appliquées aux clauses de Horn.
- Soit C un programme PROLOG constitué d'un ensemble de clauses (faits et règles). Le système répond à une question Q en trouvant une substitution σ (unification) qui rend inconsistant l'ensemble $\sigma[C \cup (\neg Q)]$ (réfutation/résolution).
- La stratégie de résolution de PROLOG est une stratégie dite « input linéaire ».

Choisir un couple de clauses C_1 et C_2 de S à résoudre

Dernière clause produite
- au départ la formule à déduire niée -

Donnée du problème
- élément initial de S -

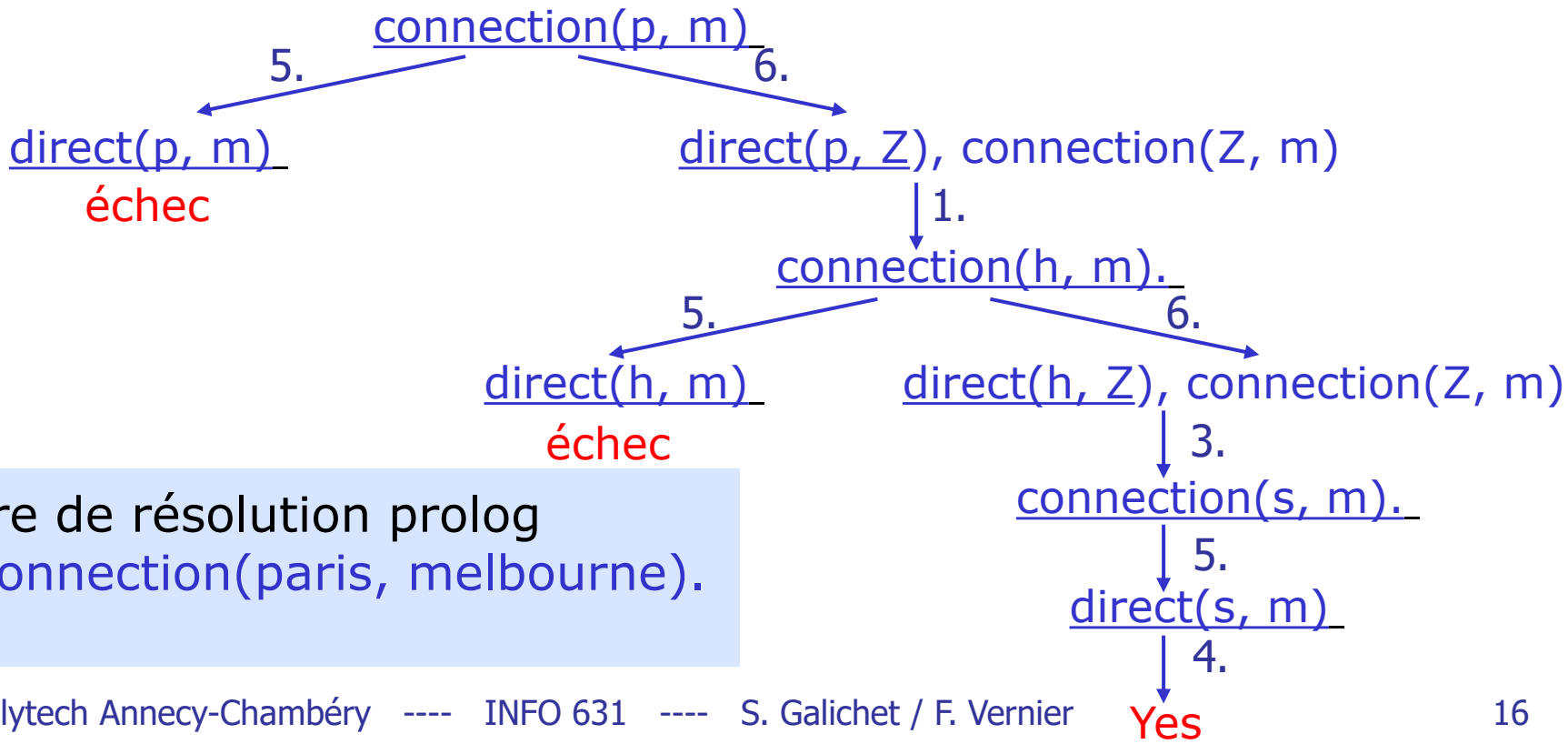
Programmer en logique

- La clause C_1 est la clause à résoudre (but à réduire) et C_2 la clause de réduction).
- La clause à résoudre est traitée linéairement dans l'ordre d'apparition des littéraux (de gauche à droite).
- Lorsque plusieurs clauses de réduction concernent la même clause à résoudre, elles sont essayées les unes après les autres dans l'ordre d'écriture des clauses.
- Représentation sous forme d'arbre.

Programmer en logique

```
% Faits
direct(paris, hongkong).
direct(paris, tokyo).
direct(hongkong, sydney).
direct(sydney, melbourne).
```

```
% Règles
connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```



Arbre de résolution prolog

?- connection(paris, melbourne).

Yes

Programmer en logique

```
direct(paris, hongkong).
direct(paris, tokyo).
direct(hongkong, sydney).
direct(sydney, melbourne).
```

Faits

```
connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

Règles

```
?- connection(hongkong, X).
X = sydney ;
X = melbourne ;
No
```

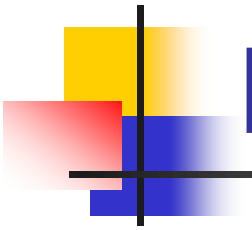
Question ou but
Réponses obtenues

Pas d'autre solution

```
?- connection(X,melbourne).
X = sydney ;
X = paris ;
X = hongkong ;
No
```

Question ou but
Réponses obtenues

Pas d'autre solution



Programmer en logique

- Dans l'idéal, le programmeur devrait pouvoir ignorer le fonctionnement du mécanisme d'exécution pour se concentrer sur l'écriture des faits et règles (programmation déclarative).
- Malheureusement, les langages actuels de programmation logique, en particulier PROLOG, sont encore loin de cet idéal et une bonne programmation nécessite une bonne compréhension du système d'exécution de façon à optimiser le code (ordre des règles, ordre des buts, terminaison).

Syntaxe PROLOG

En PROLOG, les objets traités sont des termes, objets purement syntaxiques appartenant à trois catégories :

- Les constantes individuelles ou atome
- Les variables
- Les fonctions ou termes fonctionnels

- Un atome (constante) peut s'écrire de trois façons :
  - ✓ Comme un identificateur dont le premier symbole est une lettre minuscule : marcel, henri_IV
  - ✓ Comme un nombre : 123, 12.3
  - ✓ Comme une suite quelconque de caractères entre apostrophes : 'Le cours de PROLOG', 'Que dites-vous ?'

- Une variable est un identificateur débutant par une lettre majuscule.
  - ✓ Exemples : X, Nom, No_place
  - ✓ Quand il n'est fait qu'une seule référence à une variable, il est possible d'utiliser une variable anonyme.

    Notation : _

Syntaxe PROLOG

- Un terme fonctionnel est un nom de fonction suivi d'une liste de termes entre parenthèses, séparés par des virgules. Un nom de fonction est un atome non numérique.

  ✓ Exemples :
  auteur('Malade imaginaire')
  f(3)
  cube(X)
  'Que dites-vous ?'(marcel)
  edition(gallimard,1974)

Les connaissances sur les objets et les relations entre objets sont représentées par les prédicats. Ils ont la même forme que les fonctions (au niveau syntaxique, rien ne permet de les distinguer).

Les prédicats sont les constituants de base des faits, règles et questions.

Les faits

- Affirmation de l 'existence d 'une relation entre certains objets ou d'une propriété d'un objet.
- Formule vraie a priori.
- Constitue une partie des données d'un problème.

- Un fait est un prédicat suivi d'un point.

```
pair(4).
direct(paris, hongkong).
livre(hernani, 'victor hugo', edition(gallimard,1974)).
```

Terme fonctionnel
Vu comme un objet composite
Ce n'est pas une fonction appliquée à des arguments et désignant le résultat

Les questions

- Une question élémentaire s'écrit comme un prédicat suivi d'un point. Elle est entrée derrière le prompt (?-).

```
?- connection(hongkong, X).
X = sydney ;
X = melbourne ;
No
```

Autre solution ?

- Une question peut être composée à l'aide d'une conjonction ou d'une disjonction.

```
?- connection(X, hongkong), connection(X, tokyo).
X = paris ;
No
```

et (∧)

```
?- direct(X, hongkong); direct(X, melbourne).
X = paris ;
X = sydney ;
No
```

ou (∨)

Les règles

- Enoncent la dépendance d'une relation entre objets par rapport à d'autres relations.
  - En langage naturel

    Si un vol direct relie x à y alors les villes x et y sont connectées
  - En logique du premier ordre

    $Direct(x, y) \rightarrow Connection(x, y)$
  - En PROLOG

    connection(X, Y) :- direct(X, Y).

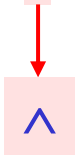
    ← Séparateur

connection(X, Y) est Vrai si direct(X, Y) est Vrai

Les règles

- La forme générale d'une règle est la suivante :

  H :- P_1, P_2, …, P_n

  où H, P_1, P_2, …, P_n sont des prédicats.

- Le prédicat H est la *tête* de la règle, la suite de prédicats P_1, P_2, …., P_n est le *corps* de la règle.

- Elle s'interprète en « Si P_1 et P_2 et … et P_n alors H ». D'un point de vue logique elle est analogue à :

  $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow H$

  ou encore à la *clause de Horn* :

  $\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee H$

  Exemple

```
connection(X, Y) :- direct(X, Z), connection(Z, Y).
grandpere(X, Y) :- pere(X, Z), mere(Z, Y).
```

Les règles

- Toutes les variables qui sont utilisées dans une règle sont vues comme quantifiées universellement.

  La règle grandpere(X, Y) :- pere(X, Z), mere(Z, Y).

  s'interprète comme la formule logique

  $\forall x \, \forall y \, \forall z \, (\, (\text{pere}(x, z) \wedge \text{mere}(z, y)) \rightarrow \text{grandpere}(x, y) \,)$

- La portée d'une variable est l'énoncé (fait, règle, question) dans laquelle elle figure.

- La portée de tous les autres noms (constantes individuelles, fonctionnelles ou prédicatives) est le programme.

Les règles

- Tout prédicat défini avec une règle récursive doit avoir au moins une définition non récursive (condition d'arrêt évitant que le programme boucle indéfiniment).

```
connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).

ancetre(X, Y) :- mere(X, Y).
ancetre(X, Y) :- mere(X, Z), ancetre(Z, Y).
```

- Résumé : Lecture d'une règle, d'un fait et d'une question

Règle : H est vrai si P_1 et P_2 et … et P_n sont vrais
Fait : H est vrai
Question : P_1 et P_2 et … et P_n sont-ils vrais ?

Prédicat d'unification =

```
?- a(B,C)=a(2,3).
B = 2
C = 3 ;
No
```

```
?- a(X,X)=a(Y,2).
X=2
Y=2
Yes
```

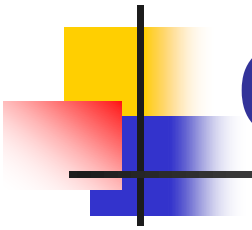
```
?- a(X,Y,L)=a(Y,2,carole).
X = 2
Y = 2
L = carole ;
No
```

```
?- a(X,X,Y)=a(Y,u,v).
No
```

```
?- a(X,s(X))=a(s(Y),Y).
X = s(s(s(s(s(s(s(s(s(...)))))))))
Y = s(s(s(s(s(s(s(s(s(...))))))))
Yes
```

Pas de test d'occurrence

Création d'un terme infini



Calcul numérique

En Prolog, le calcul numérique repose sur les considérations suivantes :

- Il existe des atomes numériques, qui se distinguent des autres atomes par leur écriture (par exemple 7, 12.86).
- Certains opérateurs et certaines fonctions sont définis comme exécutables (exemples : +, -, *, /, sin, …).
- Un prédicat prédéfini, noté is, transforme un terme exécutable en l'atome numérique représentant le résultat du calcul et unifie ce résultat avec un autre terme (normalement une variable).

Calcul numérique

- Prédicat prédéfini is pour l'évaluation des expressions arithmétiques

  X is Y

  Y doit être unifiable à un terme qui peut être interprété comme une expression arithmétique.

  Ce terme est évalué.

  Le résultat est alors unifié avec X.

- Exemples

  Echecs : X is marcel 6 is 2+3 W is sin(Y)

  Succès : X is 3+5 5 is 2+3

Calcul numérique

?- X is 2/3. Division réelle

X = 0.666667

Yes

?- X is 5//3. Division entière

X = 1

Yes

?- Z is X*X.

ERROR: Arguments are not sufficiently instantiated

?- X is sin(1), Y is cos(1), Z is X*X + Y*Y.

X = 0.841471

Y = 0.540302

Z = 1

Yes

?- 5 = 2+3. L'évaluation n'est pas faite lors d'une unification

No

Calcul numérique

Opérateurs de comparaison

X =:= Y X est numériquement égal à Y

X =\= Y X est numériquement différent de Y

X < Y

X =< Y

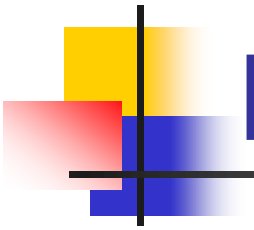
X > Y

X >= Y

Il y a évaluation des expressions droite et gauche puis comparaison.

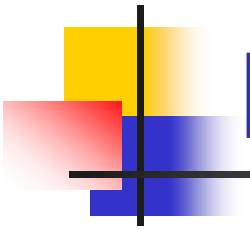
Exemples

```
?- 3+2 =:= 6-1.
Yes
?- 3+2 = 6-1.
No
```



La négation

- Si *F* est une formule, sa négation est notée *not(F).*
- Prolog pratique la **négation par l'échec**, c'est-à-dire que pour démontrer *not(F)* Prolog va tenter de démontrer *F*. Si la démonstration de *F* échoue, Prolog considère que *not(F)* est démontrée.
- Pour Prolog, *not(F)* signifie que la formule F n'est pas démontrable, et non que c'est une formule fausse. C'est ce que l'on appelle l'**hypothèse du monde clos**.
- La négation par l'échec ne doit être utilisée que sur des prédicats dont les arguments sont instanciés et à des fins de vérification.



La négation

Danger ….

Considérez le programme:

```
p(a).
```

Et interrogez Prolog avec :

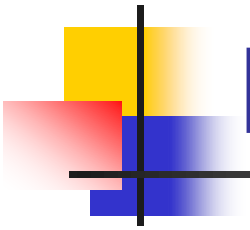
```
?- X = b, not(p(X)).
X = b
Yes
```

Prolog répond positivement car p(b) n'est pas démontrable.

Par contre, à l'interrogation suivante :

```
?- not(p(X)), X=b.
No
```

Prolog répond négativement car p(X) étant démontrable avec X=a, not(p(X)) est considéré comme faux.



Listes

- Les listes sont des séquences d'éléments symboliques qui peuvent être eux-mêmes des listes.
- Définition des listes
  - L'atome [] est une liste et représente la liste vide.
  - Une liste non vide est une suite de termes séparés par des virgules et placés entre crochets.
  - La notation [T|Q] représente la liste dont la tête est l'élément T et la queue la liste Q.
  - La notation [T,H|Q] permet de spécifier une tête de liste à deux éléments
  - Exemples : [] [a] [a, b, c]
    [[a,b], c, [d,e]]
    [a|[a,b,c]]
    [[a,b]|[c,[d,e]]] [a,b|[c, d]]

Listes

- Le prédicat appartenir/2 réussit si son premier argument représente un élément de la liste spécifiée par le second argument.

- Ecriture de appartenir(X, Y)

  X appartient à la liste Y

  1. si X est la tête de la liste Y
  2. si X appartient à la queue de la liste Y

```
?- appartenir(b,[a,b,c]).
Yes
```

```
?- appartenir(d,[a,b,c]).
No
```

```
?- appartenir(Z,[a,b,c]).
Z = a ;
Z = b ;
Z = c ;
No
```

Listes

- Le prédicat appartenir/2 réussit si son premier argument représente un élément de la liste spécifiée par le second argument.

- Ecriture de appartenir(X, Y)

```
appartenir(X, [X|_]).
appartenir(X, [_|Y]) :- appartenir(X, Y).
```

```
?- appartenir(b,[a,b,c]).
Yes
```

```
?- appartenir(d,[a,b,c]).
No
```

```
?- appartenir(Z,[a,b,c]).
Z = a ;
Z = b ;
Z = c ;
No
```

Listes

- Spécification : prédicat concatener/3

  concatener(List1, List2, List3)

  Réussit si List3 est unifiable avec la concaténation des listes List1 et List2.

- Ecriture du prédicat

```
concatener([], L, L).
concatener([T|Q1], L, [T|Q2]) :- concatener(Q1, L, Q2).
```

```
?- concatener([a,b],[c,d],L3).
L3 = [a, b, c, d] ;
No
?- concatener([a],L2,[a,b,c]).
L2 = [b, c] ;
No
```

```
?- concatener(L1,L2,[a,b]).
L1 = []
L2 = [a,b] ;
L1 = [a]
L2 = [b] ;
L1 = [a,b]
L2 = [] ;
No
```

Listes

- Prédicat inverse/2 : inverse(List1, List2)

  réussit si les deux arguments représentent des listes dont les éléments sont en ordre inverse

```
inverse([], []).
inverse([T|Q],LI) :- inverse(Q, QI), concatener(QI,[T],LI).
```

```
?- inverse([a,b,c],L).
L = [c, b, a] ;
No
```

```
?- inverse(L,[a,b,c]).
L = [c, b, a] ;
...
Bouclage infini
```

Listes

- Prédicat inserer/3 : inserer(Elemt, List1, List2)

  réussit si l'insertion de l'élément Elemt dans la liste List1 est unifiable avec la liste List2.

```
?- inserer(k,[a,b,c],L).
L = [k, a, b, c] ;
L = [a, k, b, c] ;
L = [a, b, k, c] ;
L = [a, b, c, k] ;
No

?- inserer(X,[a,b],[b,c,a]).
No
```

```
?- inserer(X,L,[a,b,c]).
X = a
L = [b, c] ;
X = b
L = [a, c] ;
X = c
L = [a, b] ;
No
```

Listes

- Le prédicat permuter/2 réussit si ses deux arguments sont des listes composées des mêmes éléments.

  permuter(List1, List2)

```
?- permuter([a,b,c],Z).
Z = [a, b, c] ;
Z = [b, a, c] ;
Z = [b, c, a] ;
Z = [a, c, b] ;
Z = [c, a, b] ;
Z = [c, b, a] ;
No
```

Exécution des programmes

Arbre de résolution PROLOG

- Racine de l'arbre : question.
- Nœuds : points de choix (formules à démontrer).
- Arc : Passage d'un nœud vers son fils en considérant l'une des règles et en effectuant une unification et un pas de démonstration.
- Nœuds de gauche à droite dans l'ordre de déclaration des règles.
- Nœuds d'échec : aucune règle ne permet de démontrer la première formule du nœud.
- Nœuds de succès : ne contient plus aucune formule, tout a été démontré et les éléments de solution sont trouvés en remontant vers la racine de l'arbre.

Exécution des programmes

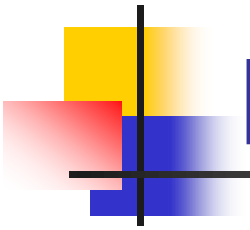
- Pour résoudre une question, Prolog construit l'arbre de recherche de la question.
- Parcours en profondeur d'abord
  - Nœud de succès : c'est une solution, Prolog l'affiche et cherche d'autres solutions.
  - Nœud d'échec : remontée dans l'arbre jusqu'à un point de choix possédant des branches non explorées.
  - On parle de **backtrack**. Si un tel nœud de choix n'existe pas, la démonstration est terminée, il n'y a pas d'autre solution.
- Possibilité de branche infinie et donc de recherche sans terminaison …
- Attention à l'ordre des prédicats dans le corps des règles et à l'ordre des clauses.

Mise au point des programmes

Trace des programmes : méthode des boîtes de Byrd

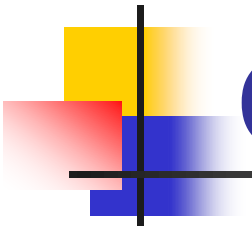
Call : premier appel du but (activation initiale)
Exit : sortie du but avec succès
Fail : sortie du but après échec
Redo : nouvelle tentative de résolution du but

- Les outils d'aide à la mise au point indiquent quand les événements call, exit, redo et fail se produisent lors de l 'exécution d'un programme.
- Pour distinguer quels événements surviennent pour quels buts, un nombre entier est associé à chaque boîte but (niveau d'appel).



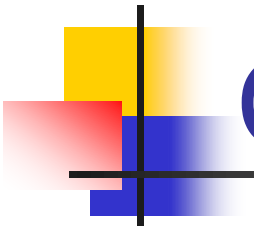
Mise au point des programmes

- trace/0 : la satisfaction du prédicat trace a pour effet de mettre en route une trace exhaustive.
- trace/1 : l'argument permet de spécifier le prédicat auquel sera attaché un point de trace.
- trace/2 : le deuxième argument permet de spécifier l'événement (call, …) à tracer.
- notrace/0 : l'effet du but notrace est d'arrêter la trace exhaustive (cependant les traces dues aux points de trace continuent).
- debugging/0 : le prédicat debugging permet de visualiser les points de trace actifs.
- nodebug : le prédicat nodebug provoque le retrait de tous les points de trace.



Coupure (cut ou coupe-choix)

- Notation : **!**
- Introduit un contrôle du programmeur sur l'exécution de ses programmes en élaguant des branches de l'arbre de recherche.
- La coupure est utile dans les cas suivants :
  - Le programmeur sait qu'il n'y a qu'une solution et la coupure interrompra la recherche dès que cette solution est trouvée.
  - Le programmeur n'a besoin que d'une solution.
  - Seule la première solution est correcte et il faut éliminer les autres.



Coupure

- Une seule solution est possible

  Présence de clauses exclusives

  Exemple :

```
humain(X) :- homme(X).
humain(X) :- femme(X).

humain(X) :- homme(X),!.
humain(X) :- femme(X),!.
```

- Une seule solution est utile

  Pour prouver que *b* est un élément d'une liste, on peut s'arrêter dès que la première occurrence de *b* a été trouvée.

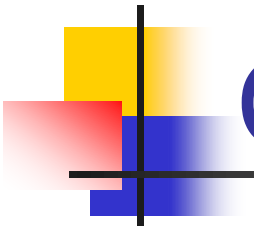
Coupure

- Une seule solution est correcte

  La recherche d'autres solutions conduit à un arbre de recherche infini.

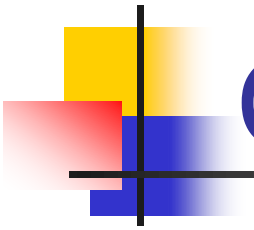
  Exemple : Pour calculer la racine carrée entière d'un nombre, on utilise un générateur de nombres entiers *entier(k)*. L'utilisation du coupe-choix est alors indispensable après le test K*K>N car la génération des entiers n'a pas de fin.

```
entier(0).
entier(N) :- entier(N1), N is N1+1.
racine(N, R) :- entier(K), K*K >N, !, R is K-1.
```



Coupure

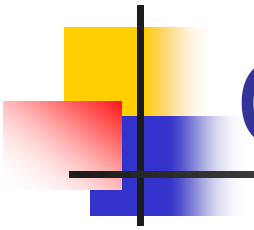
- La coupure réussit toujours mais une fois seulement.
- Aucune autre unification ne sera recherchée pour le but qui a été mis en unification avec la tête d'une règle dans le corps de laquelle se trouve une coupure qui a réussi.
- La coupure va donc empêcher tout retour en arrière dans la partie de la règle, tête comprise, qui précède cette coupure.
- Si retour en arrière il y a, le but qui avait réussi pour la tête de règle contenant la coupure est sauté et on examine à nouveau le but précédent.



Coupure

```
p1(X,Y):-p2(X),!,p3(Y).
p1(X,Y):-p4(Y).
p2(a).
p2(b).
p3(b).
p4(a).
p5(a).
p5(b).
```

```
?- p1(a,a).
No

?- p5(X), p1(X,X).
X = b
Yes
```



Coupure

En résumé, les utilités du coupe-choix sont :

- éliminer les points de choix menant à des échecs certains.
- supprimer certains tests d'exclusion mutuelle dans les clauses.
- permettre de n'obtenir que la première solution de la démonstration.
- assurer la terminaison de certains programmes.
- contrôler et diriger la démonstration.

Listes

- Le prédicat appartenir/2 réussit si son premier argument représente un élément de la liste spécifiée par le second argument.

```
appartenir(X, [X|_]).
appartenir(X, [_|Y]) :- appartenir(X, Y).
```

```
?- appartenir(Z,[a,b,c]).
Z = a ;
Z = b ;
Z = c ;
No
```

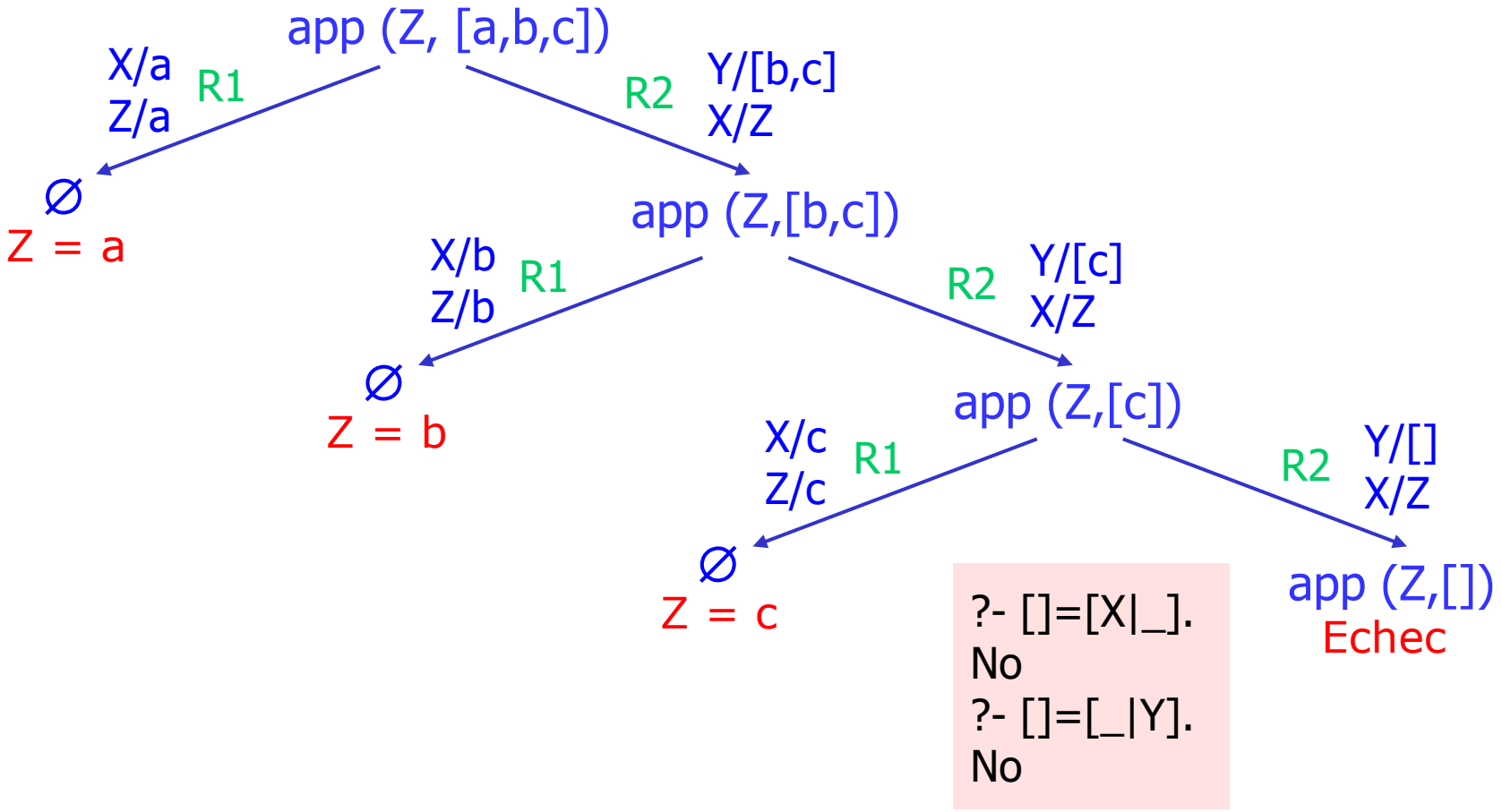
```
?- appartenir(a,L).
L = [a|_G306] ;
L = [_G305, a|_G309] ;
L = [_G305, _G308, a|_G312] ;

…. bouclage à l'infini
```

ERROR: Out of global stack

Listes

R1 appartenir(X, [X|_]).
R2 appartenir(X, [_|Y]) :- appartenir(X, Y).

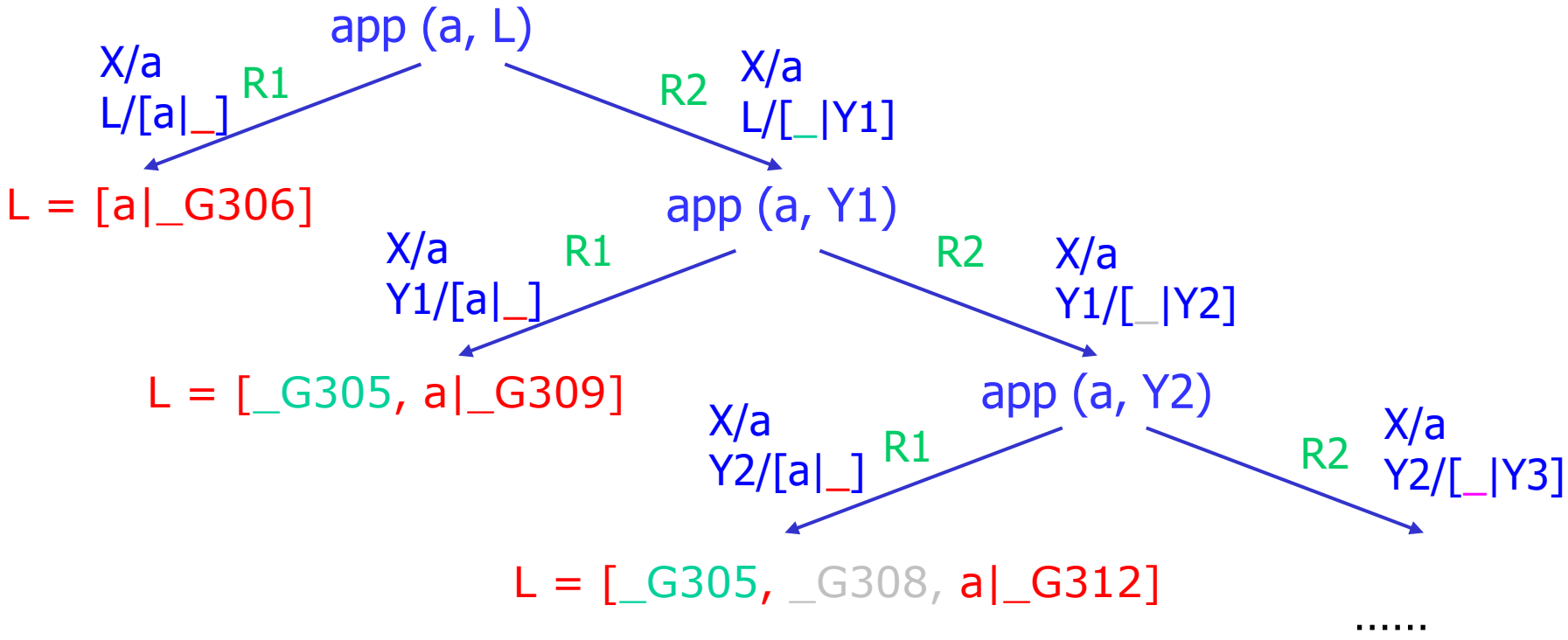
- Arbre de résolution PROLOG pour ?- appartenir(Z, [a,b,c])



Listes

R1 appartenir(X, [X|_]).
R2 appartenir(X, [_|Y]) :- appartenir(X, Y).

- Arbre de résolution PROLOG pour ?- appartenir(a, L)



ERROR:
Out of global stack

Listes

- Le bouclage à l'infini peut être éliminé par introduction d'une coupure dans la définition du prédicat appartenir.

```
appartenir_bis(X, [X|_]) := !.
appartenir_bis(X, [_|Y]) :- appartenir_bis(X, Y).
```

```
?- appartenir_bis(Z,[a,b,c]).
Z = a ;
No
```

```
?- appartenir_bis(a,L).
L = [a|_G306] ;
No
```

- Les prédicats appartenir et appartenir_bis correspondent à la réécriture des prédicats prédéfinis member/2 et memberchk/2.

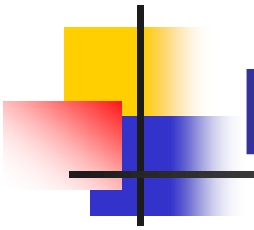
Listes

- Pour une description claire et concise des prédicats définis, les arguments sont précédés du symbole +, – ou ?.
  - \+ L'argument est une entrée du prédicat (doit être complètement instantié)
  - – L'argument est une sortie du prédicat.
  - ? L'argument est soit une entrée, soit une sortie du prédicat.

member(*?Elem, ?List*)
Succeeds when *Elem* can be unified with one of the members of *List*. The predicate can be used with any instantiation pattern.

memberchk(*?Elem, +List*)
Equivalent to `member/2`, but leaves no choice point.

length(*?List, ?Int*)
Succeeds if *Int* represents the number of elements of list *List*. Can be used to create a list holding only variables.



Listes

- Spécification : prédicat concatener/3

  concatener (?List1, ?List2, ?List3)

  Réussit si List3 est unifiable avec la concaténation des listes List1 et List2.

  Equivalent au prédicat prédéfini append/3

- Spécification : prédicat inverse/2

  inverse (+List1, -List2)

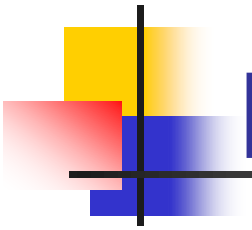
  Equivalent au prédicat prédéfini reverse/2

- Spécification : prédicat inserer/3

  inserer (?Elemt, ?List1, ?List2)

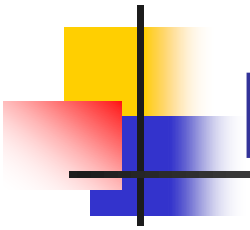
- Spécification : prédicat permuter/2

  permuter(+List1, ?List2)



Les entrées-sorties

- Prolog standard fournit un petit nombre de prédicats d'entrée-sortie, qui sont assez rudimentaires (les transferts y sont prévus terme par terme ou caractère par caractère). Chaque caractère est identifié à son code ASCII.
- Prolog a un flux courant pour l'entrée et la sortie. Par défaut, ce sont le clavier et l'écran.
- Les prédicats d'entrée-sortie ne réussissent qu'une seule fois; ils sont ignorés par les retours en arrière.



Les entrées-sorties

Ecriture à l'écran, mode caractère

put/1 put(+Char)

Affiche le caractère Char à l'écran. Char est soit une variable ou constante à valeur entière qui représente le code ASCII du caractère, soit un atome d'un caractère.

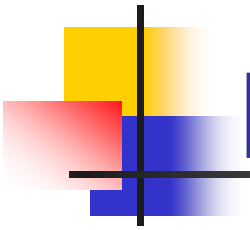
Exemple : Affichage des caractères A ou a.

```
put(65).
put('A').
A is 65, put(A).
put(a).
```

nl/0 Affiche le caractère new line

tab/1 tab(+N)

Affiche N caractères espace



Les entrées-sorties

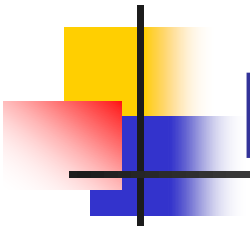
Ecriture à l'écran, mode terme

write/1 write(+Term)

Affiche le terme Term à l'écran.
Aucune évaluation n'est effectuée.

Exemples

write(sin(1)).	sin(1)
X = sin(1)^2 + cos(1)^2, write(X).	sin(1)^2+cos(1)^2
X is sin(1)^2 + cos(1)^2, write(X).	1
write('la maison').	la maison
write(append([a],[b,c],X)).	append([a],[b,c],_G338)
append([a],[b,c],X), write(X).	[a, b, c]
write(f(f(f(3)))).	f(f(f(3)))
write(f(f(f(3))).	ERROR: Syntax error: Unexpected end of clause



Les entrées-sorties

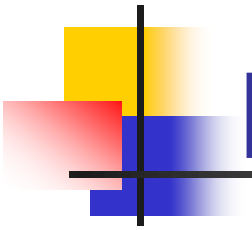
Lecture au clavier, mode caractère

get/1 get(-Char)

Lit au clavier le premier caractère non blanc entré et unifie son code ASCII avec Char.

```
?- get(X).
|: a
X = 97
Yes

?- get(X), get(Y).
|: a   b
X = 97
Y = 98
Yes
```

|: prompt PROLOG spécifique au mode lecture



Les entrées-sorties

Lecture au clavier, mode terme

read/1 read(-Term)

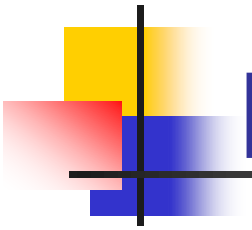
Lit un terme au clavier et l'unifie avec son argument. Le terme lu doit être obligatoirement suivi d'un point.

Exemples

```
?- read(X).
|: a(2,1).
X = a(2, 1)
Yes
```

```
?- read(X)
|: a.
X = a ;
No
```

```
?- read(f(X)).
|: f(p).
X = p ;
No
```



Les entrées-sorties

Les fichiers

- Ouverture d'un fichier

  open/3 open(+Filename, +Mode, -Stream)

  modes d'ouverture : write, append ou read

- Fermeture d'un fichier

  close/1 close(+Stream)

- Accès au fichier

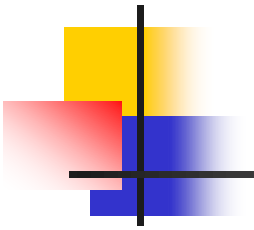
  Des prédicats de lecture / écriture d'arité 2 permettent de spécifier le flux à utiliser.

  put(+Stream, +Char), write(+Stream, +Term)

  get(+Stream, -Char), read(+Stream, -Term)

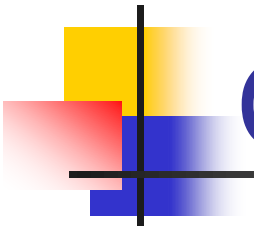
  Exemple :

```
ecrire(T) :- open('a.dat', write, Flux),
             write(Flux, T), nl(Flux),
             close(Flux).
```



Prédicats de contrôle

- Le prédicat fail/0 est un prédicat qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure.
- Le prédicat true/0 est un prédicat toujours démontrable.
- Le prédicat repeat/0 est un prédicat prédéfini qui est toujours démontrable mais laisse systématiquement un point de choix derrière lui. Il a une infinité de solutions.
- L'utilisation conjointe de repeat/0, fail/0 et du coupe-choix !/0 permet de réaliser des boucles.



Classes d'opérateurs

- Les opérateurs sont des noms de prédicat ou de fonction (unaire ou binaire) qui peuvent s'écrire avant, après ou entre les arguments.
- Chaque opérateur possède les caractéristiques suivantes :
  - Une précédence qui est un entier représentant l'inverse de la priorité (plus grande est la précédence, plus faible est la priorité)
  - Une position qui peut être infixée, préfixée ou postfixée.
  - Un type d'associativité.

    gauche : X op Y op Z est lu comme (X op Y) op Z
    droite : X op Y op Z est lu comme X op (Y op Z)
    aucune : Les parenthèses sont obligatoires.
    La syntaxe X op Y op Z est interdite

Classes d'opérateurs

- La position et l'associativité sont notées ensemble :
  - Opérateurs préfixés : fx, fy
  - Opérateurs postfixés : xf, yf
  - Opérateurs infixés : xfx, xfy, yfx

xfx	xfy	yfx
Pas Asso.	Asso. droite	Asso. gauche

- Dans la notation
  - f représente l'opérateur
  - x ou y représentent les opérandes
  - x veut dire « cet opérande doit être de précédence strictement inférieure à celle de l'opérateur »
  - y veut dire « cet opérande doit être de précédence inférieure ou égale à celle de l'opérateur.

Classes d'opérateurs

Accès aux caractéristiques des opérateurs définis

current_op/3 current_op(?Prec, ?Type, ?Nom)

```
?- current_op(P, T, N).
P = 700
T = xfx
N = >= ;
P = 1200
T = fx
N = :- ;   ....
```

Déclaration d'un opérateur

op/3 op(+Prec, +Type, ?Nom)

```
?- op(700, xfx, inferieur).
Yes
?- 3 inferieur 5        (avec inferieur(X,Y) :- X < Y.)
Yes
```