Chapitre 2
Le langage Prolog

- Les langages déclaratifs
- La programmation logique et langages relationnels
- Prolog : champs d'application
- Le langage : syntaxe, modèle d'exécution, preuves
- Quelques références
- Prolog, une présentation détaillée

Langages déclaratifs

- La plupart des langages de programmation sont conçus pour permettre la spécification de programmes en tant que séquences ordonnées d'instructions à exécuter:
  - ☛ Les langages procéduraux / impératifs

- Mais il existe des langages où la notion d'ordre d'exécution est exclue, et où ce souci est délégué à l'implantation du langage (p.ex. SQL, Trilogy):
  - ☛ Les langages déclaratifs (relationnels, fonctionnels)
  - ☛ Langages dont l'éxécution des instructions dépend des données (langages fonctionnels "purs")
- ☛ Les langages de programmation purement déclaratifs sont rares.
- En général on distingue la sémantique abstraite du mode opératoire (la stratégie d'évaluation)

Exemple du langage SQL

- SQL (*Structured Query Language*): langage d'interrogation et de mise à jour des bases de données.

```
SELECT Numetu, Nometu
FROM ETUDIANT
WHERE Dnaiss>='01-01-1980' AND Dnaiss<='12-31-1980'
ORDER BY Nometu, Dnaiss ASC;
```

- La base de données cache entièrement le mécanisme opérationnel sous-jacent.

Exemple du langage Trilogy

- Langage de spécification de contraintes (*CLP*: *Constraint Logic Programming*):

```
all  x::I & x>-100 & x<100 & x*x<=10
```

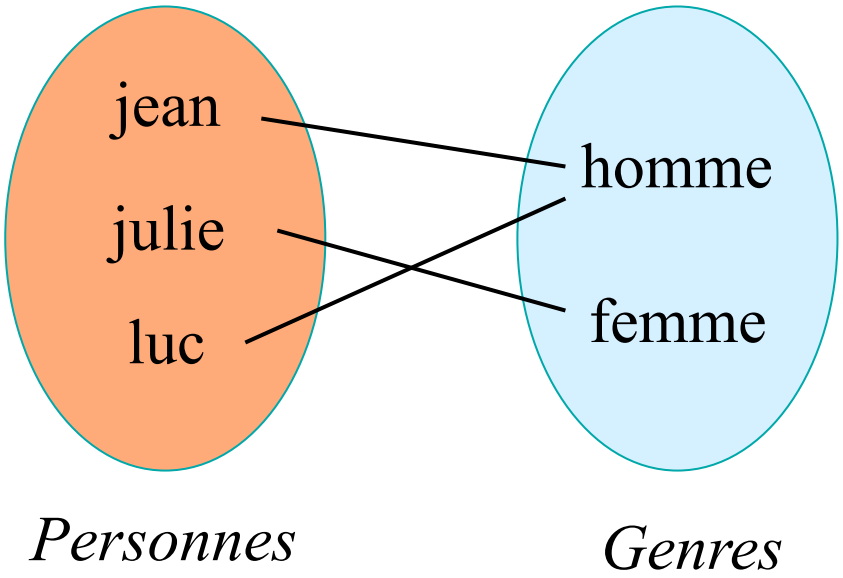
- Il s'agit d'un langage de spécifications logiques pures (Prolog est hybride à ce point de vue).

La programmation logique

- La programmation logique peut être vue comme un sous-ensemble de l'approche relationnelle
- La programmation logique est également basée sur un sous-ensemble de la logique des prédicats (clauses de Horn)
- Dans la programmation logique, formalisée par Kowalski dès 1972, le rôle de l'ordinateur consiste en une série contrôlée d'inférences logiques. Elle remonte au calcul des prédicats introduit par Frege en 1879, au principe d'unification dû à Herbrand en 1929 et au principe de résolution formulé par Robinson en 1963.
- Prolog (=PROgrammation en LOGique) en est le principal représentant

La notion de “relation”

- Définition:
  - une **relation binaire** sur les ensembles *X1* et *X2* est un ensemble de couples de la forme <*x1*, *x2*>, où *x1* est élément de *X1* et *x2* est élément de *X2*



La notion de “relation”

- Généralisation:
  - une **relation *N*-aire** sur les ensembles *X1*...*XN* est un ensemble de tuples de la forme <*x1*, *x2*, ..., *xN*>, où *xi* est élément de *Xi*
- Définition:
  - une **fonction** de *X1* (domaine) vers *X2* (l'image) est une relation **binaire** qui met en relation **au plus un élément** de *X2* avec chaque élément de *X1*
- Les langages fonctionnels manipulent des fonctions; les langages relationnels, des relations N-aires

Exemples de relations

- Exprimée comme fonction:
  - `genre(Personne)` retourne des valeurs de `Genres`
  - P.ex: `genre(julie)` retourne `femme`
- Exprimée comme relation:
  - `genre(Personne,Genre)` est un ***prédicat***
  - P.ex: `genre(jean,homme)` est *vrai*
  - Mais aussi: `genre(Qui,homme)` retourne *vrai* et `Qui=jean`; `Qui=luc`
- Exemple de relation ternaire:
  - `concat(X1,X2,X3)`, la concaténation des chaînes de caractères `X1` et `X2` dans `X3`

Champs d'application de Prolog

- Prolog est conçu pour manipuler des symboles et des relations.
- Quelques domaines d'application:
  - analyse de langues informatiques ou naturelles;
  - bases de données et systèmes experts;
  - logique mathématique, preuve de théorèmes, résolution symbolique d'équations;
  - travaux d'architecture, conception, plans de site, logistique;
  - analyse biochimique et conception de médicaments.

Applications industrielles de Prolog

- Quelques applications industrielles, rapportées par des vendeurs de compilateurs/interpréteurs Prolog:
  - www.visual-prolog.com/vipexamples/applications.htm
  - www.amzi.com/customers/index.htm www.amzi.com/customers/education_government.htm
  - www.lpa.co.uk/ind_inf.htm
  - www.als.com/cust_stories.html
  - www.swi-prolog.org/

Syntaxe de Prolog (1)

- Programme = ensemble de relations + **requête** vérifiant qu'une relation existe

- 2 façons de spécifier des relations:
  - **fait**: indique un tuple particulier dans la relation
  - **règle**: indique comment déduire qu'un tuple fait partie d'une relation

Syntaxe de Prolog (2)

Syntaxe d'**Edinburgh**: quasi-standard

- *<variable>* ::= identificateur débutant avec majuscule
- *<atome>* ::= identificateur débutant avec minuscule
- *<terme>* ::= *<nombre>*
  <terme> ::= *<variable>*
  <terme> ::= *<atome>*
  <terme> ::= *<atome>* (*<liste_termes>*)
- *<liste_termes>* ::= *<terme>* { , *<terme>* }
- *<fait>* ::= *<terme>* .
- *<règle>* ::= *<terme>* :- *<liste_termes>* .
- *<requête>* ::= *<liste_termes>* .

Les atomes et les faits

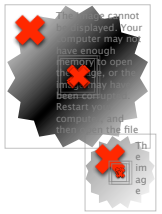
- Les <*atome*> désignent des symboles (donc sont des données) et sont aussi utilisés pour nommer les relations:
  - *<atome>* `( <liste_termes> )`
- on dit que l'atome est ici le **foncteur** du terme et entre parenthèses on a ses **arguments**
  - Exemple: `genre(luc,homme)`
- Une relation peut s'exprimer en énumérant les tuples qu'elle contient comme des **faits**
  - Exemple: soit *X1*=`{luc,julie,jean}` et *X2*=`{homme,femme}`. La relation `genre` s'exprime avec trois faits:

```
genre(luc,homme).
genre(julie,femme).
genre(jean,homme).
```

Les règles

- Les règles permettent de définir des relations à partir d'autres relations:
  - *<terme>* `:-` *`<liste_termes>`* `.`
- c'est ce qu'on appèle une **clause de Horn**, qui est divisée en sa **tête** et ses **conditions**
  - Exemple 1: `male(X) :- genre(X,homme).`
    Traduction logique: pour tout `X tel que la relation genre(X,homme) existe, la relation male(X) existe.`
  - Exemple 2: `hermaphrodite(X) :- genre(X,homme), genre(X,femme).`
    Traduction logique: pour tout `X tel que la relation genre(X,homme) existe et genre(X,femme) existe, la relation hermaphrodite(X) existe.`

Exercice



- Définir la relation “mariage possible entre *X* et Y” telle que `mp(X,Y) si X et Y sont de sexe opposé.`

```
mp(X,Y) :- genre(X,homme), genre(Y,femme).
mp(X,Y) :- genre(X,femme), genre(Y,homme).
```

Résultats tabulés de la **requête existentielle** `mp(X,Y).` :

X	Y
luc	julie
jean	julie
julie	luc
julie	jean

Modèle d'exécution de Prolog (1)

- Selon Kowalski (1979):
  - *algorithme = logique + contrôle*
- Où *logique* désigne les faits et règles (le *quoi*)
- et *contrôle* est l'application de ces faits et règles selon un certain ordre (le *comment*)

- Un mécanisme de résolution mécanique (dit de *Robinson*) est employé pour examiner successivement et systématiquement toutes les solutions possibles.
- Il y a un **ET logique** sous-entendu entre chaque condition d'une clause, et un **OU inclusif** entre chaque clause.

Modèle d'exécution de Prolog (2)

- Prolog tente de trouver, à l'aide de la procédure suivante, une **preuve** que la requête soumise est une relation qui existe:

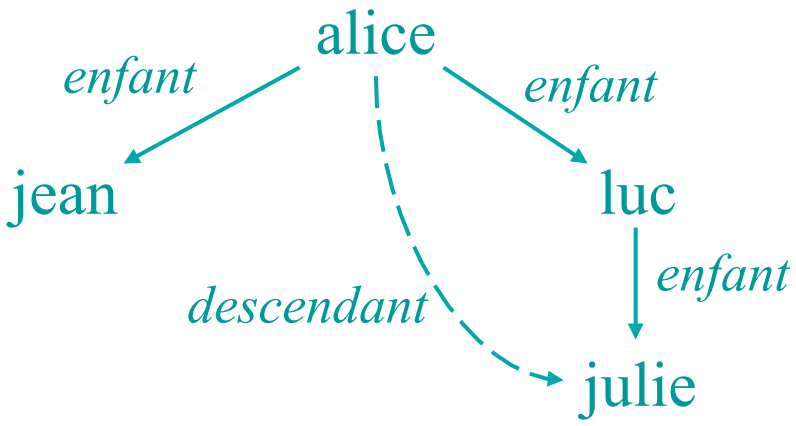
  prouver (*requête*):

  1. Pour chaque **fait** et **tête de règle** qui concorde avec la *requête*:
     - Si un **fait**: on a trouvé une preuve
     - Si une **tête de règle**:
       - Pour chaque *condition* de cette règle: **prouver(*condition*)**
       - Si succès pour chaque condition: on a trouvé une preuve
  2. Si aucune concordance: Echec (=Retour arrière)

- Les faits et règles sont examinés dans leur **ordre d'apparition** dans le programme (stratégie linéaire définie, résolution SLD; d'autres stratégies sont envisageables)
- Si une **sous-preuve** échoue, la recherche revient sur ses pas (**retour-arrière, backtracking**) pour tenter la prochaine alternative de preuve

Arbres de preuves

- À toute preuve complète correspond un **arbre de preuve** qui indique chaque sous-preuve effectuée pour prouver la requête.
- Exemple: représentation du graphe de filiation

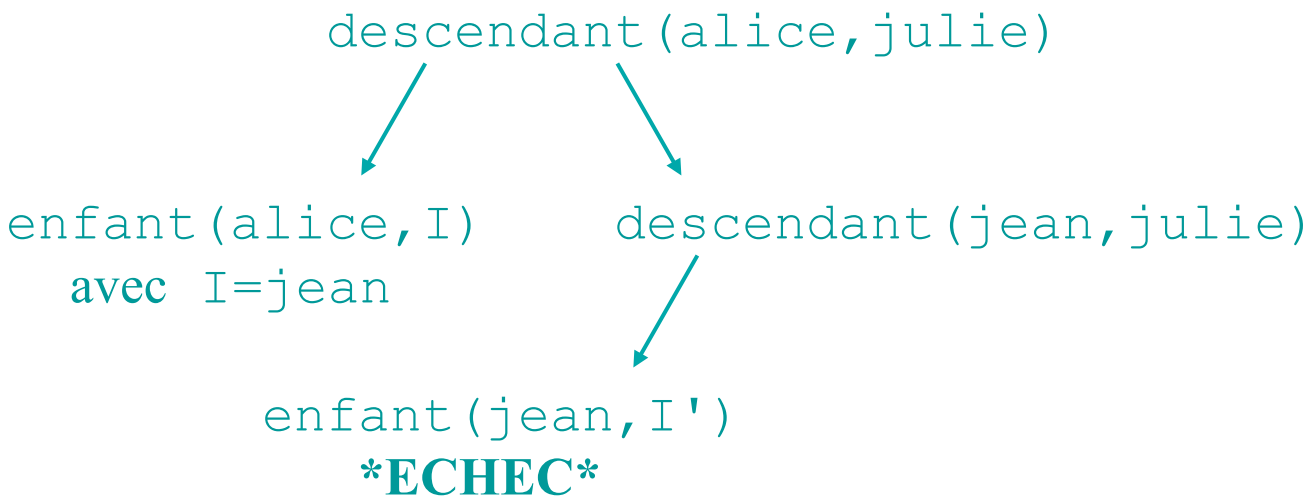



```
enfant(alice,jean).
enfant(alice,luc).
enfant(luc,julie).

descendant(P,P).
descendant(X,Y) :-
  enfant(X,I),
  descendant(I,Y).
```

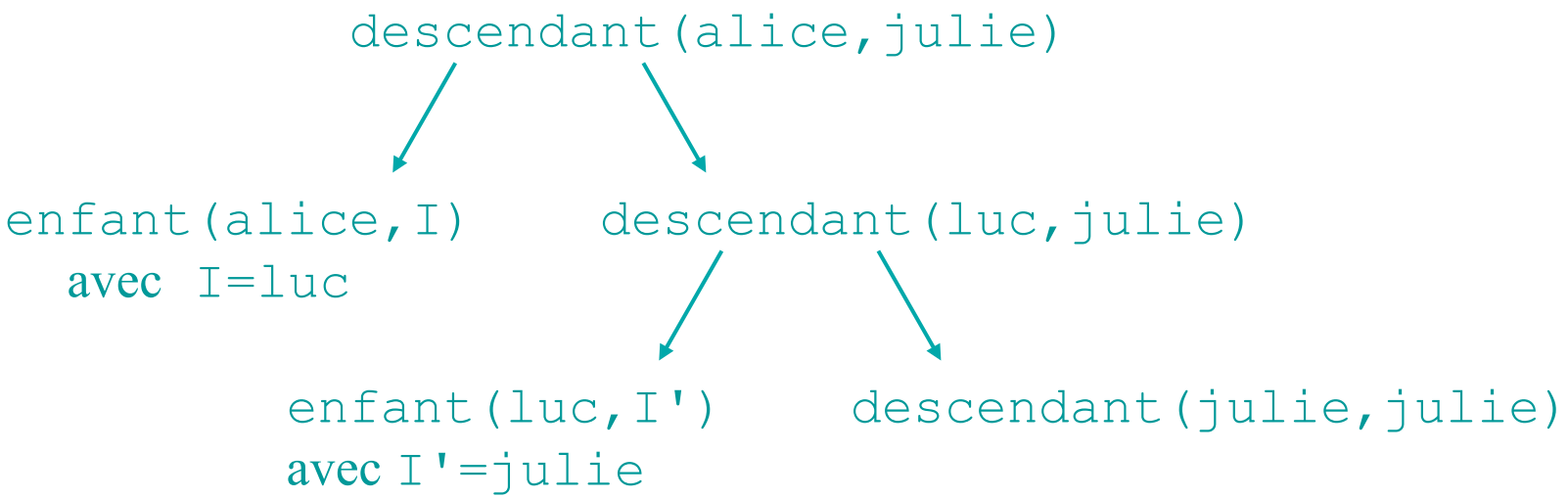
Exemple d'arbre de preuves (1)

- Requête: `descendant(alice,julie).`
- Construction de l'arbre de preuve, 1ère partie:



Exemple d'arbre de preuves (2)

- 2ème partie de l'arbre de preuve, après retour-arrière:



- La preuve a donc réussi, avec `I=luc` et `I'=julie`

Non-déterminisme

- Le retour-arrière illustre le **non-déterminisme** de Prolog
- Définition:
  - Le **non-déterminisme** est la propriété qui permet à deux exécutions/évaluations successives de fournir des résultats différents
- Le non-déterminisme se situe au niveau du choix de la tête de clause (pour les faits/règles ayant plusieurs clauses) et des liaisons de variables correspondantes.
- Chaque emplacement où il y a non-déterminisme est considéré comme un **point de choix** vers lequel on reviendra en cas d'échec d'une sous-preuve.

Variables en Prolog

- Les variables réprésentent des “inconnues” au sens mathématique
  - elles ne peuvent prendre qu’une seule valeur dans une branche d’exécution (en fait possiblement une succession d’unification)
  - une variable est soit *libre*, soit *liée* à un terme
  - *Un terme* est soit *clos,* soit *libre (cf. termes)*
  - une variable liée redevient libre par backtracking a son dernier point de choix
- Syntaxiquement une variable est soit nommée par un identificateur débutant avec une majuscule ou un ‘underline’. ‘_’ est une variable anonyme.
- Différent des langages impératifs:
  - pas d’opération d’affectation => écriture différente des algorithmes.

Structures de données en Prolog

- Prolog possède une structure uniforme, formée de termes, depuis laquelle toutes les données et les programmes sont construits. Un terme est un arbre dont:
  - les **noeuds** sont des constantes symboliques
  - les **feuilles** sont soit des constantes, soit des variables
    - P.ex: `enfant(alice,X).` Ou `nbrEnfant(julie,2).`
- PROLOG possède une opération fondamentale sur les termes: **l'unification**. Deux termes sont unifiables si on peut trouver des valeurs de variables (substitution) qui les rendent égaux.

  ?- `Y = julie.` => la variable `Y` prend la valeur `julie`

  ?- `enfant(alice,X)= enfant(alice,jean).` => la variable `X` prend la valeur `jean`

  ?- `enfant(alice,paul)= enfant(alice,jean).` => il n'y a pas d'unification

- Cf. algorithme de preuve: «concorder» = être unifiable

Usage de Prolog

- C'est un langage qui dérange les habitudes mais qui est relativement simple.
- Le programmeur PROLOG doit répondre aux questions suivantes par rapport au problème à résoudre:
  - Quels faits et quelles relations formelles existent-ils pour ce problème?
  - Quelles relations doivent-elles être vérifiées pour qu'on ait une solution?
- Prolog fut choisi par les Japonais, en 1979, pour être le langage des ordinateurs dits de la *cinquième génération*: ce fut un échec monumental !

Synthèse

- Relativement facile si premier langage, sinon non !
- Avantages du Prolog
  - Séparation de la logique et du contrôle (focus sur la structure logique du problème, plutôt que sur le contrôle de l'exécution, d'où meilleure productivité du programmeur)
  - Avantageux pour le prototypage, la programmation exploratoire
  - Support transparent pour le parallélisme
- Désavantages
  - Implémentation opérationnelle pas fidèle à la sémantique déclarative (contrôle de l'ordre de résolution, modèle du monde clos, problèmes avec la négation)
  - Gestion difficile de projet importants, multi-langages
  - Efficacité réduite en-dehors des domaines d'application prévus

Quelques références

- Clocksin, W.E., and Mellish, C.S., *Programming in Prolog*, 2nd ed., Springer Verlag, New York, 1984.
- Colmerauer inventeur du langage
- **Ivan Bratko, PROLOG Programming for Artificial Intelligence Addison-Wesley, August 2000.**
- **SWI-Prolog, un compilateur Prolog gratuit: http://www.swi-prolog.org**
- La FAQ du newsgroup comp.lang.prolog: http://www.faqs.org/faqs/prolog/

Présentation plus détaillée du langage Prolog

- Termes et structures de données
  - Manipulation de termes
- Unification
- Base de données/faits
- Résolution et contrôle d'exécution
  - Vue procédurale
  - cut, failure
  - Négation
- Conseils et astuces
- Built-in predicates
  - Input-output
  - Meta-programming

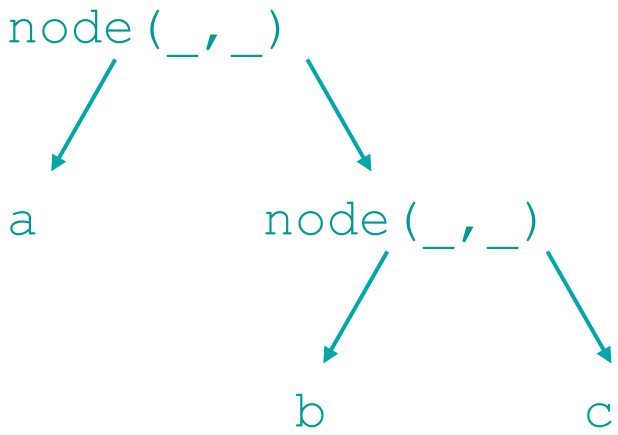
Types de données et Termes

- Tout peut être décrit par des termes !
  - Ensembles, tables
  - Listes, entiers
    - Éventuellement il y a des relations d'équivalence entre termes !
- Mais, Prolog intègre des types de données ad-hoc:
  - Listes
  - Integer
  - Atoms
  - Chaines de caractères

- => simplification de l'écriture, et augmentation de la performance, mais moins d'homogénéité
- Des prédicats prédéfinis existent pour ces structures de données

Termes

- Exemple: arbre binaire
  - node(a,node(b,c))



  - node(node(node(X,Y),3),node(node(a,b),c)).

Termes

- Il n'y a pas de typage dans les termes => n'importe quoi peut-être écrit sans restriction de compatibilité.
- Les nœuds de l'arbres sont appelé des foncteurs
  - node(_,_)
- La taille est à prioris non-limitée (sauf par les contraintes de mémoire)
- Le nombre de paramètre d'un terme est appelé son arité
  - node/2 correspond aux foncteurs de taille 2: node(_,_)
  - append/3 correspond aux foncteurs de taille 3: append(_,_,_)

Types de données ad-hoc: Listes

- La liste vide
  - []
- Concaténation d'un élément H à une liste L
  - [H | L]
- Exemples:
  - [1,2,3]
  - [jean,paul,lea]
  - [node(X,Y),node(a,node(b,c)),Y]

Unification de listes

- ?- [a,b] = [X,Y].
  - X = a
  - Y = b
- ?- [H | L] = [1,2,3] .
  - H = 1
  - L = [2, 3]
- ?- [node(X,Y),node(a,node(b,c)),Y] = [node(a,b) | L] .
  - X = a
  - Y = b
  - L = [node(a, node(b, c)), b]

Quelques prédicats sur les listes

- member(X,L) réussi si X est dans L (prédéfinis dans SWI)
  - member(X,[X|L]).
  - member(X,[Y|L]):-member(X,L).
- Interprétation:
  - member(a,[a,b,c,d]).
    - Yes
- ?- member(X,[a,b,c,d]).
  - X = a ;
  - X = b ;
  - X = c ;
  - X = d ;
  - No

- ?- member(a,[a,b,c,a]).
  - Yes
  - Question: est-ce que le prédicat est déterministe (une seule réponse !!)
- ?- member(Y,[a,b,c,a]),Y=a. Question logicalement équivalente !!
  - Y = a ;
  - Y = a ;
  - No
  - En fait opérationnellement NON !!
- ?-

Entiers

- Prolog défini des constantes telles que 1,2,3 …
- Des opérateurs sont également définis +,-,*,//

- Les entiers (et les flottants) sont employé d'une manière un peu différente que les autres structures de données car des expressions peuvent-être évaluées.

- Actions des opérations:
- Vue naïve:
  - ?- X = 1+ 2.
    - X = 1+2
- = est le symbole de l'unification, pour activer l'application d'opérateurs il faut utiliser le prédicat is.
  - ?- X is 1 + 2.
    - X = 3

Expressions vues comme des termes

- En prolog un certain nombres d'opérateurs sont prédéfinis. Même les expressions de bases comme les clauses et les connecteurs logiques sont des opérateurs !

- ?- +(1,2) = X.
- X = 1+2

- ?- :-(a,b) = X.
- X = a:-b

- ?- ','(a,b) = X.
- X = a, b

Manipulation de termes

- functor(A,f,3) : construit une structure de 3 éléments
- arg(2,A,1): place un élément ‘1’ à la 2ème position des paramètres
- ?- functor(A,f,3),arg(2,A,1).
  - A = f(_G229, 1, _G231)

- Prédicat réversibles:
- ?- functor(f(g(2),2,3),A,N).
  - A = f
  - N = 3
- ?- arg(1,f(g(2),3,4),X).
  - X = g(2) ;

- Autre prédicat:
- f(g(2),3,4) =.. [f,g(2),3,4]
- Intérêt: méta-programmation, construction de tableaux à accès directs, …

Unification

- Il s'agit de trouver une substitution aux variables pour rendre égaux deux termes (En prolog):
  - node(x,b) = node(A,X).
    - A = x
    - X = b ;
  - node(node(node(X,Y),3),node(node(a,b),c)) = T
    - X = _G157
    - Y = _G158
    - T = node(node(node(_G157, _G158), 3), node(node(a, b), c))
- Mais si on ajoute une précisions sur X, donc aussi une solution !
  - node(node(node(X,Y),3),node(node(a,b),c))= T, X = d.
    - X = d
    - Y = _G158
    - T = node(node(node(d, _G158), 3), node(node(a, b), c))
- Naturellement la solution la plus générale est calculée et elle est unique ! Mgu = most general unifier

Unification: particularités et problèmes

- enfant(alice,X)= X.
- X = enfant(alice, enfant(alice, enfant(alice, enfant(alice, enfant(alice, enfant(alice, enfant(alice, enfant(alice, enfant(..., ...)))))))))

- [X,Y,X] = [a,b,c]. Echec !

- [X,Y,X] = [a,b,T].
  - X = a
  - Y = b
  - T = a

Type d'un terme

- var(X) X est une variable
- nonvar(X) X n'est pas une variable
- atom(X) X est un atome
- integer(X) X est un entier
- float(X) X est un flotant
- atomic(X) X est un atome ou un nombre
- compound(X) X est une structure

Comparaison de termes

- X == Y X et Y sont identiques
- X \== Y X et Y sont pas identiques
- X =:= Y X et Y sont arithmétiquement identiques
- X == Y X et Y sont non arithmétiquement identiques
- X @< Y X précède Y
- ? - X @< Y.
  - X = _G157
  - Y = _G158
- ?- a(X) @< Y.
- No
- ?- a(X) @< a(Y).
  - X = _G157
  - Y = _G159
- ?- a(X) @< a(b(Y)).
  - X = _G157
  - Y = _G159

L’ unification en général

- Il s’agit de trouver la solution au système d’équations
- U ≡ V ⇔ ∃ s une substitution, t.q. sU = sV

- Une substitution est une application des variables dans les termes:
  - X = node(Z,3)
  - Y = b
- sU est l’application de la substitution au terme U fournissant un terme instancié

{X = node(Z,3);Y = b }(node(node(node(X,Y),3),node(node(a,b),c)))
= node(node(node(node(Z,3),b),3),node(node(a,b),c))

L’ unification en général

- Combien de solution (les plus générales)?
  - Théorie unitaire => 1
    - Unification de termes libres comme en Prolog
  - Théorie finitaire => un nombre fini
    - Unification d’ensembles:
      - {X,Y} ≡ {a,b} solution ?
  - Théorie infinitaire => une infinité
    - Unification de fonctions surjectives !
      - Odd(X) ≡ true solution ?

Unification, implémentations des différentes variantes

- Unitaire et finitaire => possibilités d'implémentation (sémantique opérationelle)

- Infinitaire ?

Atomes et chaînes de caractères

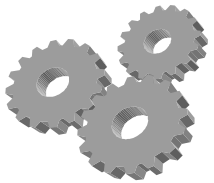
‘alphaomega’ est un atome

?- atom_concat(‘alpha’,’omega’, X)

X=‘alphaomega’

- ?- append([a],[b],C).
  - C = [a, b] ;
- ?- append("a","b",C).
  - C = [97, 98] ;
- ?- append("aa","bb",C).
  - C = [97, 97, 98, 98] ;

Exercice:



Ecrire la fonction ‘append’ (pour éviter les conflits dans l'interpréteur, utiliser le nom ‘my_append’)

Base de donnée/ base de fait

- Prolog permet l'accès aux faits et clauses permettant une gestion de type 'base de donnée'.

- Actions possibles:
  - Ajouter/Supprimer des faits et clauses

  - Construire des collections de faits (ensembles ou multi-ensembles)

Ajouter/Supprimer des faits et clauses

- La base de faits peut être enrichie
  - assert : ajouter une clause
    - assert(pere(jean, lise)).
  - retract: retirer une clause
- Mais l'ordre des faits est opérationnellement importante:
    - assert(pere(jean,luc)).
    - ?- pere(X,Y).
      - X = jean
      - Y = lise ;
      - X = jean
      - Y = luc ;
  - asserta/assertz insertion en début respectivement en fin
  - retract , le fait qui 'match' (s'appareille)

Gestion de la base avec compilateur

- L'implémentation des faits est différentes si les faits peuvent être précompilés ou non.

- Les faits peuvent être statiques ou dynamiques
  - Les faits statiques doivent être adjacent dans le fichier pour pouvoir les compiler efficacement sous la forme d'une procédure.
  - Clause :- dynamic nompred/n. (clause de contrôle exécutable à la compilation) n = arité du prédicat

Ordre d'assertion et retract

- ?- pere(jean,X).
  - X = lise ;
  - X = luc ;
  - No
- ?- retract(pere(jean,X)).
  - X = lise
  - Yes
- ?- pere(jean,X).
  - X = luc ;
  - No
- ?- assert(pere(jean,lise)).
  - Yes
- ?- pere(jean,X).
  - X = luc ;
  - X = lise ;
- No

Exemple d'un graphe: exercice

- Representation d'un graphe labellé:
  - edge(a,e1,b). edge(b,e2,c). edge(a,e2,d). edge(d,e4,b).
  - edge(d,e5,b).

- Fermeture transitive: ?

Réponse: fermeture d'un couple

- transclosure(X,Z):-
  edge(X,E,Y),edge(Y,F,Z),assert(edge(X,seq(E,F),Z)).

- transclosure(X,Y).
  - X = a
  - Y = c ;

  - X = a
  - Y = b ;

  - X = d
  - Y = c ;

  - Comment faire pour tous les couples ! (la récursion sur la base de fait n'est pas possible car ce n'est pas une structure représentable par une variable!)

Contenu de la base de fait

- ?- listing(edge/3).
- :- dynamic edge/3.

- edge(a, e1, b).
- edge(b, e2, c).
- edge(a, e2, d).
- edge(d, e4, b).
- edge(a, seq(e1, e2), c).
- edge(a, seq(e2, e4), b).
- edge(d, seq(e4, e2), c).

- Yes

Prédicat constructif de ‘quantification universelle’

- Construction d’ensembles de valeurs:

- bagof(X,P,L)
  - Construire la liste L d’objets X tel que P est satisfait

  - bagof(Edge,edge(Edge,Label,Edge2),L).

Suite de listes ! Mais pas d'ensemble

- ?- bagof(Node,edge(Node,Label,Node2),L).
  - Node = _G157
  - Label = e1
  - Node2 = b
  - L = [a] ;

  - Node = _G157
  - Label = e2
  - Node2 = c
  - L = [b] ;

  - Node = _G157
  - Label = e2
  - Node2 = d
  - L = [a] ;

  - Node = _G157
  - Label = e4
  - Node2 = b
  - L = [d] ;

Valeurs des nœuds pas prise en compte

- bagof(Node,Node^ Node2^edge(Node,Label, Node2),L).
  - Node = _G157
  - Node2 = _G159
  - Label = e1
  - L = [a] ;
  - Node = _G157
  - Node2 = _G159
  - Label = e2
  - L = [b, a] ;
  - Node = _G157
  - Node2 = _G159
  - Label = e4
  - L = [d] ;

Valeurs des nœuds et arc pas prise en compte

- bagof(Node,Node^ Node2^Label ^ edge(Node,Label, Node2),L).
  - Node = _G157
  - Node2 = _G159
  - Label = _G158
  - L = [a, b, a, d] ;
- ?- setof(Node,Node^ Node2^Label ^ edge(Node,Label, Node2),L).
  - Node = _G157
  - Node2 = _G159
  - Label = _G158
  - L = [a, b, d] ;
- ?- setof(Node2,Node^ Node2^Label ^ edge(Node,Label, Node2),L).
  - Node2 = _G159
  - Node = _G157
  - Label = _G158
  - L = [b, c, d] ;

Findall

- ?- findall(Node,edge(Node,Label, Node2),L).

- Node = _G157
- Label = _G158
- Node2 = _G159
- L = [a, b, a, d] ;

- No

Appliquer successivement un prédicat sur la base de fait

- 1ère méthode

- ?- findall((X,Y),transclosure(X,Y),L).
  - X = _G157
  - Y = _G158
  - L = [(a, c), (a, b), (d, c)] ;
- No
- ?- setof(Label,Node^ Node2^Label ^ edge(Node,Label, Node2),L).
  - Label = _G158
  - Node = _G157
  - Node2 = _G159
  - L = [e1, e2, e4, seq(e1, e2), seq(e2, e4), seq(e4, e2)] ;
- No

Est-ce que l'on a un point fixe ?

- Definition d'un point fixe pour la fermeture: Closure(X) = X ?

- essai:
- Nouvel arc (introduit un cycle):
  - assert(edge(c,e3,a)).
- ?- findall((X,Y),transclosure(X,Y),L).

  - X = _G157
  - Y = _G158
  - L = [(a, c), (b, a), (a, b), (d, c), (d, a), (c, b), (c, d), (c, c), (..., ...)] ;
- No

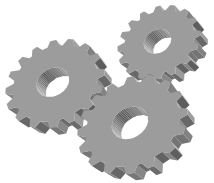
Pas de point fixe !

- L'ens des clauses à parcourir n'est pas mise a jour dynamiquement !
- ?- listing(edge).
- :- dynamic edge/3.
  - edge(a, e1, b).
  - edge(b, e2, c).
  - edge(a, e2, d).
  - edge(d, e4, b).
  - edge(c, e3, a).
  - edge(a, seq(e1, e2), c).
  - edge(b, seq(e2, e3), a).
  - edge(a, seq(e2, e4), b).
  - edge(d, seq(e4, e2), c).
  - edge(d, seq(e4, seq(e2, e3)), a).
  - edge(c, seq(e3, e1), b).
  - edge(c, seq(e3, e2), d).
  - edge(c, seq(e3, seq(e1, e2)), c).
  - edge(c, seq(e3, seq(e2, e4)), b).

Répétition

- ?- repeat,transclosure(X,Y),fail.
  - A cause des cycles du graphe ne termine pas !!
- ?- retract(edge(_,seq(_,_),_)),fail.
  - Supprime tous les faits ayant un label incluant une séquence

Exercice



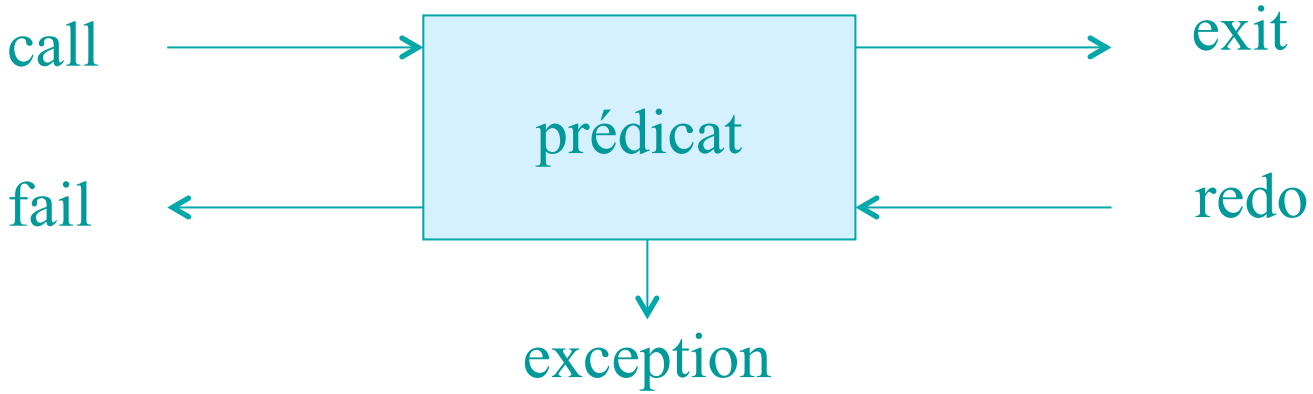
- Créer un predicat permettant de générer une valeur différente à chaque appel ?

Modèle procédural et contrôle d'exécution

- L'exécution d'un programme prolog peut être contrôlé par l'ordre des clauses et des buts.
- De manière additionnelle, le contrôle du 'backtracking' se fait au moyen du prédicat 'cut' (!)
- Autres commandes prédéfinies:
  - fail, échec permanent
  - ; la disjonction, le 'ou'
  - 'trace', met dans un mode d'évaluation ou les étapes d'exécutions sont présentées pas à pas.

Modèles d'exécutions

- L. Byrd's box model (voir affichage de 'trace'):



  - Enter/call: accès au prédicat
  - redo: reprend l'exécution au cas suivant
  - fail: signifie un échec
  - exit : signifie qu'une solution à été trouvée et que l'évaluation continue
  - Pour plus de détail: http://pauillac.inria.fr/~diaz/gnu-prolog/manual/index.html
- Machine abstraite pour compilation efficace et correcte (Warren abstract machine:sémantique opérationnelle)
  - Pour plus de détail: http://www.vanx.org/archive/wam/wam.html

Exemple:

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

- [debug] ?- member(a,[c,b,a]).
  - T Call: (7) lists:member(a, [c, b, a])
  - T Call: (8) lists:member(a, [b, a])
  - T Call: (9) lists:member(a, [a])
  - T Exit: (9) lists:member(a, [a])
  - T Exit: (8) lists:member(a, [b, a])
  - T Exit: (7) lists:member(a, [c, b, a])
- Yes

- [debug] ?- member(X,[c,b,a]).
  - T Call: (7) lists:member(_G292, [c, b, a])
  - T Exit: (7) lists:member(c, [c, b, a])
- X = c ;
  - T Redo: (7) lists:member(_G292, [c, b, a])
  - T Call: (8) lists:member(_G292, [b, a])
  - T Exit: (8) lists:member(b, [b, a])
  - T Exit: (7) lists:member(b, [c, b, a])

- X = b ;
  - T Redo: (8) lists:member(_G292, [b, a])
  - T Call: (9) lists:member(_G292, [a])
  - T Exit: (9) lists:member(a, [a])
  - T Exit: (8) lists:member(a, [b, a])
  - T Exit: (7) lists:member(a, [c, b, a])
- X = a ;
  - T Redo: (9) lists:member(_G292, [a])
  - T Call: (10) lists:member(_G292, [])
  - T Fail: (10) lists:member(_G292, [])
  - T Fail: (9) lists:member(_G292, [a])
  - T Fail: (8) lists:member(_G292, [b, a])
  - T Fail: (7) lists:member(_G292, [c, b, a])
- No

Modèle opérationnel de Prolog

- Soit G1,G2,…,Gm les buts à exécuter
  - Si la liste des buts est vide terminer en *success* (exit)
  - Si la liste de but est non vide continuer avec l'opération 'RECHERCHER'
- RECHERCHER:
  - Parcourir du début à la fin la liste de clause jusqu'à trouver la clause C, tel que tête de C *match* G1, sinon terminer avec *failure*.
  - Si il y a une telle clause H :-B1,…,Bn , alors renommer les variables pour que G1 et C n'aient pas de variables en commun.
  - La clause C' est donc H' :-B1',…,Bn '

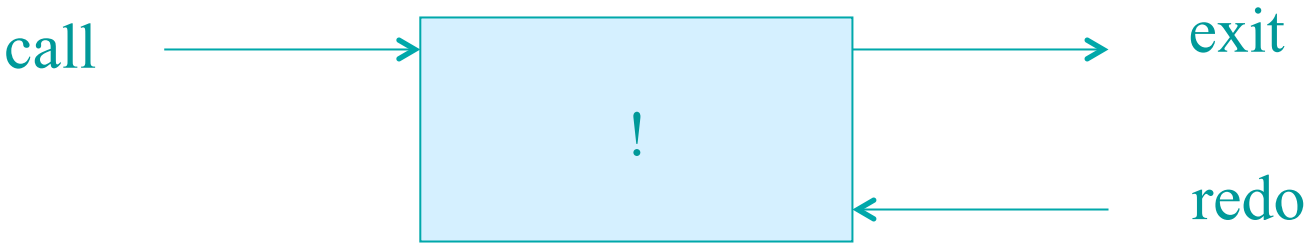
Modèle opérationnel de Prolog (suite)

- Faire correspondre (match) G1 et H' ce qui génère la substitution S
- Substituer G1 par le nouveau but: B1',…,Bn ',G2,…,Gm
- Appliquer la substitution ce qui donne
- B1'',…,Bn '',G2',…,Gm'

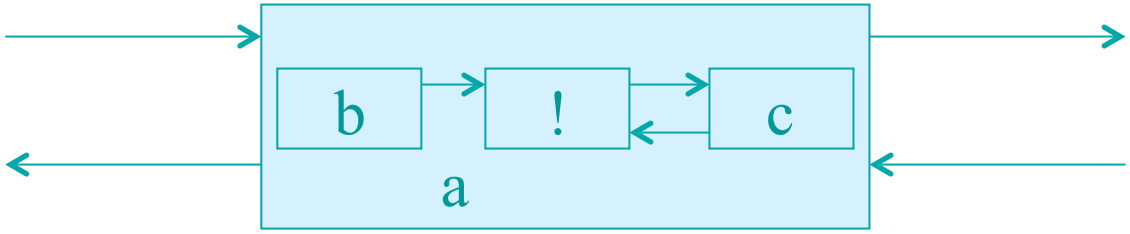
- Executer récursivement la nouvelle liste de but
  - Si termine avec succès terminer la liste originale avec succès
  - Si l'évaluation termine sans succès revenir sur le choix de C, et utiliser une clause apparaissant immédiatement après.

Cut

- Prédicat n'autorisant pas la remise en cause des prédicats apparaissant avant ce prédicat.



- a:-b,!,c.



Le cut, calculer le maximum

- max(X,Y,Max)

- Deux règles exclusives !
  - max(X,Y,X):-X>=Y.
  - max(X,Y,Y):-X<Y.

- Plus économiquement (‘sinon’):
  - max(X,Y,X):-X>=Y,!.
  - max(X,Y,Y).

- Évite de reconsidérer la seconde règle si la 1ère réussis !

Problèmes !

- Le cut rend les procédures logiquement éronées !,
- Par exemple:
  - max(3,1,1) est vrai !!
  - Solution:
    - max(X,Y,Max):-X>=Y,!,Max=X ; Max=Y.
- Les prédicats ne sont éventuellement plus inversibles

Exercice:

- Construire le prédicat member(X,L), tel qu'il ne fournisse qu'une solution à:

- Member(X,[1,2,3,1]).

Négation par échec

- different(X,Y)

- different(X,X):-!,fail.
- different(X,Y).

- Ou:

- different(X,Y):-
  - X=Y,!,fail
  - ;
  - true.

not

- A la place de cut (utiliser de préférence)

- not(P):-
  - P,!,fail
  - ;
  - true.

  - Exemple not(X=Y) a la place de different(X,Y)

Problème avec cut

- Le cut permet d'améliorer la performance des algorithmes, mais perd la correspondance entre forme déclarative et procédurale
- Exemple prolog pur:
  - p:-a,b
  - p:-c
  - Logiquement: p <=> (a&b)+c
- Exemple avec cut:
  - p:-a,!,b
  - p:-c
  - Logiquement: p <=> (a&b)+(-a&c)
- Exemple avec cut, mais clause dans l'autre ordre:
  - p:-c
  - p:-a,!,b
  - Logiquement: p <=> c + (a&b)

Problème avec not

- Le not conduit à des déduction fausse par exemple:
  - ?- dynamic(human/1).
  - ?- not(human(mary)).
    - Yes
  - L'hypothèse du *monde clos* fait déduire cette réponse car mary n'est pas dans la relation human ! (tout ce qui est connus est dans un fait !)
  - person(jean). person(luce).
  - employe (jean). chomeur(X):-not(employe(X)).
  - ?- person(X),chomeur(X).
    - X = luce ;
    - No
  - ?- chomeur(X),person(X). (inversion de exist à forall)
    - No

Entrée-sorties

- Les clauses d'entrées sorties suivent partiellement le mode opératoire des autres clauses. Ces clauses agissent sur le terminal d'une manière similaire à ce que fait assert/retract sur la base de fait.

- read(X)
- A chaque appel prend la valeur entrée suivante

- write(X)
- Génère à chaque appel une nouvelle valeur.

Conclusions

- Prolog implémente les principes de la logique prédicative.
- Prolog possède une sémantique opérationelle et une sémantique déclarative.
- Prolog est très utile pour le prototypage d'applications.
- Prolog est également très pratique pour prototyper des langages.
  - parsing en implémentant les règles syntaxiques
  - Interprétation en exécutant les règles de la sémantique opérationelle
  - Compilation en générant du prolog