UTC 503

CHAPITRES 11 et 12

Programmation Logique

Syntaxe, fonctionnement et programmation récursive

Sébastien Morvan
Sebastien.morvan@lecnam.net

Plan

- Entités Prolog
- Termes atomiques
- Termes composés
- Contraintes
- Clauses de Horn
- Fonctionnement de base
- Programmation récursive
- Ressources
- Exercices

Entités Prolog

Clauses

- Fait : `pere(sebastien, elena).`
- Règle : `grand_pere(X, Y) :- pere(X, Z), pere(Z, Y).`

- Un fait est une règle particulière, i.e. une clause sans queue

Entités Prolog

Questions

- `pere(X, Y).`

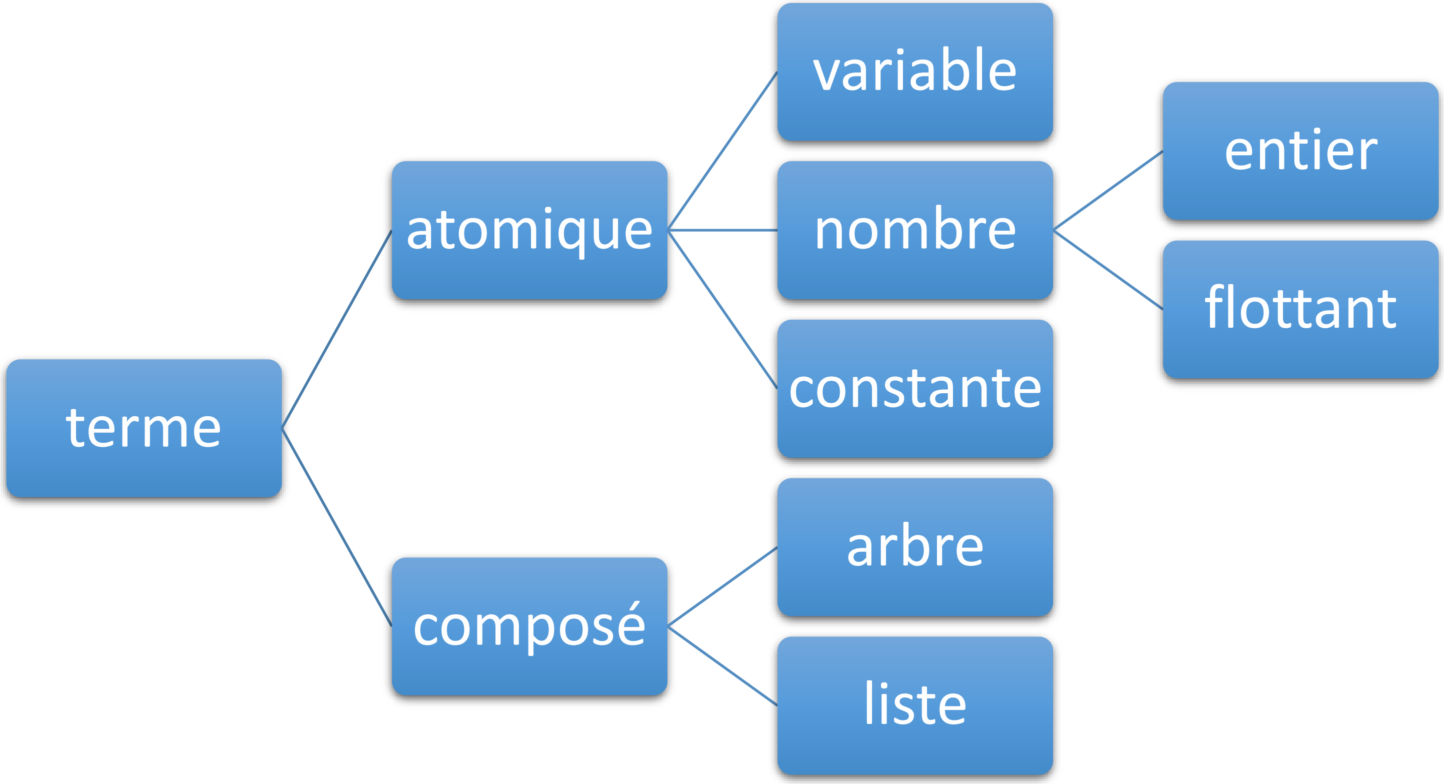
- Une question est une clause sans tête

Entités Prolog

Termes

- Définition :

les objets manipulés par Prolog sont appelés des termes



Termes atomiques

- Constantes symboliques : apparaissent comme des noms écrits en minuscules
  - Ex: « sebastien », « elena », « pere »
- Variables : elles sont en majuscules, précisément la 1ère lettre est en majuscule
  - Ex: « X », « Y », « Head », « Queue », « Name »

Termes atomiques

- Constantes symboliques : apparaissent comme des noms écrits en minuscules
  - Ex: « sebastien », « elena », « pere »
- Variables : elles sont en majuscules, précisément la 1ère lettre est en majuscule
  - Ex: « X », « Y », « Head », « Queue », « Name »

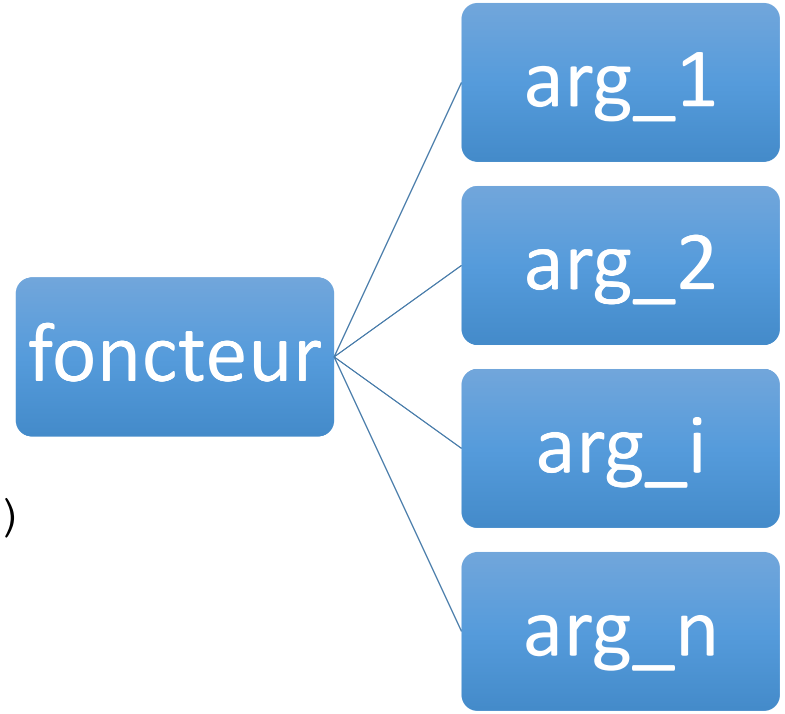
Termes atomiques

- Booléens :
  - 0 = faux
  - 1 = vrai
- Nombres :
  - Entiers : 123
  - Flottants : 12.34

Termes composés

Arbre

- foncteur = 1 identificateur
- arg_i = terme atomique ou composé
- Notation Prolog :
  - `foncteur(arg_1, arg_2, arg_i, arg_n)`
  - Sans espace entre « foncteur » et « (»
- Ex :
  - `logiciel(prolog, auteur(colmerauer, roussel), france).`



Termes composés

Arbre

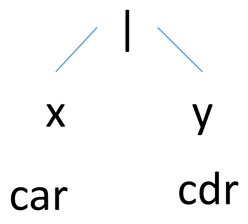
- Attention : auteur(colmerauer, roussel) est uniquement une structure de données arbre passive, et non un prédicat
- Pas d'évaluation directe de auteur
- Evaluation possible via le prédicat englobant
- Ex :

```
?- logiciel(prolog, X, france).
      X = auteur(colmerauer, roussel)
?- logiciel(prolog, auteur(X, Y), france).
      X = colmerauer
      Y = roussel
```

Termes composés

Liste

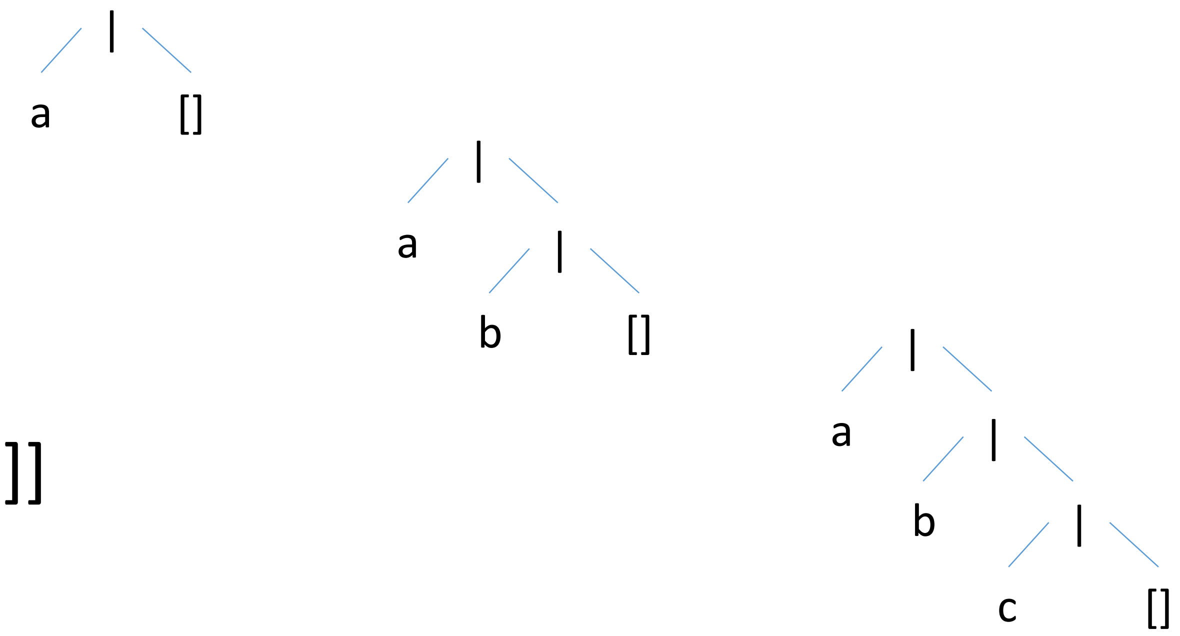
- Une liste est un arbre binaire de la forme
- Notation Prolog : [x|y]
- Avec
  - car : le 1er élément de la liste
  - cdr : la liste privée de son 1er élément



Termes composés

Liste

- Liste vide : []
- Liste à 1 élément : [a|[]]
  - Notation simplifiée : [a]
- Liste à 2 éléments : [a|[b|[]]]
  - Notation simplifiée : [a, b]
- Liste à 3 éléments : [a|[b|[c|[]]]]
  - Notation simplifiée : [a, b, c]

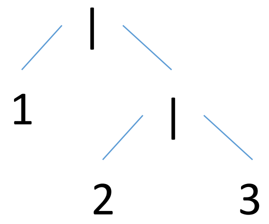


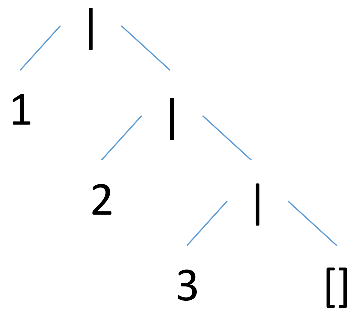
Termes composés

Liste

- Attention :

$$[1, 2|3] \neq [1, 2, 3]$$





Termes composés

Liste

- Exemples :

Liste	car	cdr
[a, b, c, d]	a	[b, c, d]
[a]	a	[]
[[le, chat], assis]	[le, chat]	[assis]

Termes composés

Liste

- Unifications :
  - [chat] $\leftarrow unification \rightarrow$ [X, Y]
    - X = chat et Y = []
  - [X, Y | Z] $\leftarrow unification \rightarrow$ [jean, aime, pêche]
    - X = jean, Y = aime, Z = [pêche]
  - [[le, Y] | Z] $\leftarrow unification \rightarrow$ [[X, lievre], [est, ici]]
    - X = le, Y = lievre, Z = [[est, ici]]

Les contraintes

- Une règle en Prolog peut faire apparaître une contrainte

$$l_0{:}-l_1, l_2, l_i \ldots, l_n, S.$$

- Avec l_i littéral
- Et S un système de contraintes
  - S est un ensemble de propriétés attachées aux variables de la règle devant toujours être satisfaites
- Ex: « X = Y », « X \= Y »

Les contraintes

- Exemple :

```
repas_leger(X, Y, Z) :-
    entree(X, X_cal),
    plat(Y, Y_cal),
    dessert(Z, Z_cal),
    (X_cal + Y_cal + Z_cal < 600),
    Z\=glace.
entree(carotte, 50).
entree(pate, 700).
plat(choucroute, 200).
dessert(glace, 300).
dessert(pomme, 50).
dessert(orange, 80).

?- repas_leger(X, Y, Z).
```

Clauses de Horn

- Les définitions de prédicats prennent la forme d'implications où la conjonction des littéraux du membre droit entraîne le littéral unique du membre gauche, appelé le littéral de tête.

$$c \text{ :- } p_1, p_2, \ldots, p_k. \iff p_1 \land p_2 \land \ldots \land p_k \supset c$$

- On peut les voir comme autant de règles de déduction, dont les littéraux du membre droit sont les prémisses
et le littéral de tête est la conclusion

Clauses de Horn

- Une règle de déduction est formée d'un nombre fini de *prémisses*, et d'une seule *conclusion* (qui sont des formules quelconques).
  On l'écrit traditionnellement ainsi

- ```
  p1, p2, ... , pk      <--- les prémisses
  -----------------
          c             <--- la conclusion
  ```

  $$\frac{p_1,\ p_2,\ \ldots,\ p_k}{c}$$

- Ce type de formule est appelée clause de Horn (par référence à la forme clausale en logique).
  ```

Fonctionnement de base

Paquets de clauses (règles)

L'opération élémentaire du calcul de Prolog est la démonstration d'un énoncé atomique, alias littéral. On parle de *résoudre un littéral*.
Il nous faut donc comprendre comment se passe la résolution d'un littéral.

Cette résolution utilise les règles du système (les *clauses*) dont la conclusion est compatible (*unifiable*) avec le littéral-but donc en particulier qui ont **le même prédicat que le but**.

Fonctionnement de base

Paquets de clauses (règles)

Toutes les clauses relatives au même prédicat sont regroupées en un seul **paquet**, où elles sont rangées dans l'ordre du texte-source.

L'ordre des paquets de clauses n'a pas d'importance,
en revanche l'ordre des clauses dans chaque paquet est significatif !

Fonctionnement de base

Paquets de clauses : le zig-zag

À chaque étape, Prolog doit démontrer une liste de buts : il procède de **gauche à droite** dans cette liste

Pour démontrer chacun de ces buts :

- soit **p** le prédicat du but visé [c-à-d. but = p(....)]
- Prolog essaie les clauses du paquet **p** dans l'ordre (de haut en bas)
- et naturellement, les différents sous-buts qui composent chaque clause sont traités de gauche à droite...

Fonctionnement de base

Paquets de clauses : le zig-zag

Exemple : résolution du littéral **`zig(X)`**, avec :

```
zig(a) :- write('Un '), write('deux '), write('trois'), nl.
zig(b) :- write('Quatre '), write('cinq '), write('six'), nl.
zig(c) :- write('Sept'), nl, write(Fini_pour_zig), nl.
```

write et nl sont des prédicats prédifinis : write écrit sur la console et nl ajoute une « nouvelle ligne »

Fonctionnement de base

Paquets de clauses : le zig-zag

- Résultat

```
Un deux trois
Quatre cinq six
Sept
Fini_pour_zig
```

Fonctionnement de base

Le séquencement standard avec retour-arrière

Le fonctionnement normal de Prolog lui fait chercher *toutes les démonstrations possibles* du but visé.

Ceci le conduit à **revenir en arrière** après un échec, pour essayer la clause suivante du paquet.

Ce mécanisme de retour-arrière (*backtrack*), superposé au « zig-zag » induit un séquencement complexe.

Fonctionnement de base

Le séquencement standard avec retour-arrière

- Exemple : résolution du littéral **zigzag(X, Y)** :

```
zigzag(X, Y) :-
    zig(X), write('X = '), write(X), nl,
    zig(Y), write('Y = '), write(Y), nl.
```

On observe que le littéral le plus à droite zig(Y) a été résolu 9 fois, et zig(X) 3 fois seulement : sorte de boucle intérieure sur zig(Y), extérieure sur zig(X)

Fonctionnement de base

Le séquencement standard avec retour-arrière

- Résultat

```
Un deux trois
X = a
Un deux trois
Y = a
Quatre cinq
Six
Y = b
Sept
Fini_pour_zig
Y = c
```

```
Quatre cinq six
X = b
Un deux trois
Y = a
Quatre cinq six
Y = b
Sept
Fini_pour_zig
Y = c
```

```
Sept
Fini_pour_zig
X = c
Un deux trois
Y =  a
Quatre cinq six
Y = b
Sept
Fini_pour_sig
Y = c
```

Fonctionnement de base

Le séquencement standard avec retour-arrière

- On observe que le littéral le plus à droite `zig(Y)` a été résolu 9 fois, et `zig(X)` 3 fois seulement :
- sorte de boucle intérieure sur `zig(Y)`, extérieure sur `zig(X)`

Fonctionnement de base

Interprétation logique du séquencement de base

Les deux dimensions horizontale et verticale d'un paquet de clauses ont deux interprétations bien différentes.

Fonctionnement de base

Interprétation logique du séquencement de base

Les lignes horizontales, membres droits des clauses, s'interprètent comme des **conjonctions** de littéraux.
À la fin du parcours gauche-droite de chaque ligne, tous les littéraux qui la composent ont été démontrés.

- Mais attention ! la conjonction logique est commutative (clause de Horn),
  pas le parcours gauche-droite de Prolog ...

Fonctionnement de base

Interprétation logique du séquencement de base

La succession verticale des clauses dans le paquet correspond à une **disjonction** entre les clauses :
chacune d'entre elles indique une manière de démontrer son littéral de tête.

- Mais attention ! la disjonction logique est commutative,
  pas le parcours de haut en bas de Prolog ...

Fonctionnement de base

Le coupe-choix : problème

Il arrive souvent que l'exploration de toutes les possibilités conduise à des calculs inutiles voire à des calculs qui bouclent !
L'idée est d'exprimer à un certain point du calcul que **l'on renonce à explorer** des possibilités non encore envisagées.
Par exemple, pour calculer une fonction, on dira que

- lorsqu'on a trouvé un résultat, **c'est le bon !**
- et qu'**il est inutile** (voire nuisible) de chercher plus loin

Fonctionnement de base

Le coupe-choix : solution

On introduit donc le prédicat sans argument « ! » (point d'exclamation), dit en anglais « *cut* » et en français « *coupe-choix* », qui apparaît parmi d'autres littéraux dans les membres droits de clauses

Fonctionnement de base

Le coupe-choix : fonctionnement

- Lors du parcours gauche-droite d'une liste de buts, lorsqu'on rencontre le *cut*, il est aussitôt vérifié (comme un"write") et le calcul se poursuit normalement ;

Fonctionnement de base

Le coupe-choix : fonctionnement

- Mais lorsqu'un retour-arrière se produira, il ne pourra pas remonter pas plus haut que le « **!** ».
  I.e. que la résolution des littéraux situés à gauche du coupe-choix dans la clause courante ne sera pas reprise, et que les clauses suivantes du paquet en cours ne seront pas essayées.

Fonctionnement de base

Le coupe-choix : exemple

Nous pouvons ainsi exprimer que, dans la recherche des parents d'un enfant donné, le père est unique, de sorte que lorsqu'on l'a trouvé il est inutile de chercher plus loin :

```
enfant(E, P, M) :- pere(P, E), !, mere(M, E).
```

Fonctionnement de base

Le coupe-choix : exemple

Lors de la résolution du littéral `enfant(elena, P, M).`, avec ma base de connaissance familiale, on obtiendra comme prévu
P = sebastien et M = nathalie,
mais l'exploration du paquet de clauses « pere » s'arrêtera à la première, tandis que toutes les clauses du paquet « mere » seront successivement essayées…

Fonctionnement de base

Le coupe-choix : exemple

Si on place le coupe-choix à la fin du membre droit, on rétablit la symétrie entre le traitement du père et celui de la mère :

```
enfant(E, P, M) :- pere(P, E), mere(M, E), !.
```

Programmation récursive

- C’est la manière naturelle de programmation en Prolog
- Outils
  - Unification
    - Manipuler des structures de données (SDD)
    - Gestion des appels
  - Résolution
    - Récursivité

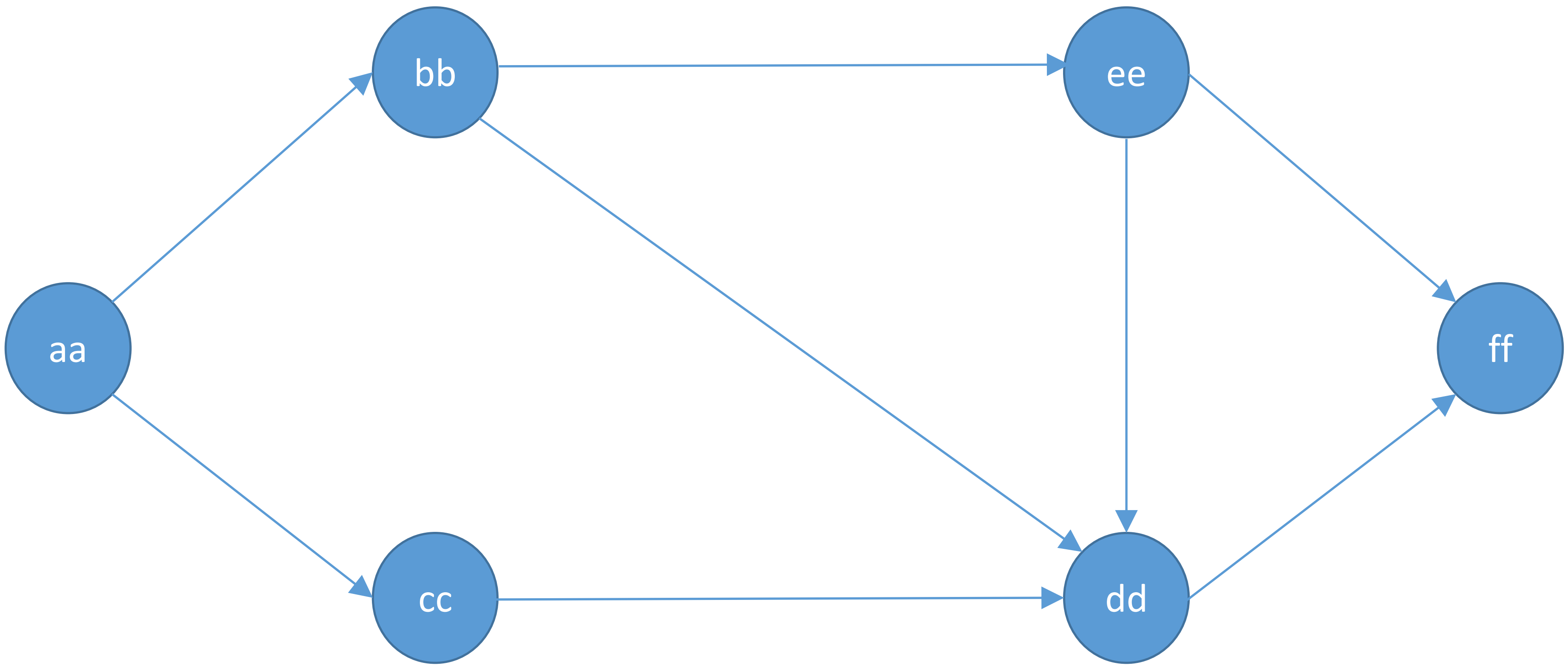
Programmation récursive

Méthode

1. Caractériser les objets manipulés
2. Exprimer le problème sous forme relationnelle
3. Analyser les cas
   - Cas particulier(s)
   - Cas général
4. Décrire des appels de clauses
5. Définir les clauses

Programmation récursive

Exemple 1 : parcours de graphe



Programmation récursive

Exemple 1 : parcours de graphe

Problème : trouver les chemins (points intermédiaires) entre un point de départ et un point d'arrivée

Programmation récursive

Exemple 1 : parcours de graphe

1. Caractériser les objets manipulés

    - Point de départ : D
    - Point d’arrivée : A
    - Liste des points intermédiaires : L

Programmation récursive

Exemple 1 : parcours de graphe

2. Caractériser les objets manipulés

- chemin(D, A, L)
- $D \cdot\!\rightarrow\cdot\!\rightarrow\cdot\!\rightarrow A\ avec\ L = [\cdot,\cdot]$

Programmation récursive

Exemple 1 : parcours de graphe

3. Analyse des cas

- $D \cdot\!\!\rightarrow A \; implique \; L = []$
- $D \cdot\!\!\rightarrow X \cdot\!\!\rightarrow \cdots \cdot\!\!\rightarrow A \; implique \; L = [X|Q]$
  $\text{où } Q \; est \; le \; reste \; de \; la \; liste \; et \; peut \; donc \; \text{être} \; vide$

Programmation récursive

Exemple 1 : parcours de graphe

4. Descriptions des appels de clauses

- `chemin(D, A, [])`
- `chemin(D, A, [X|Q])`

Programmation récursive

Exemple 1 : parcours de graphe

5. Définition des clauses

BC : description du graphe par des faits

```
arc(aa, bb).
arc(aa, cc).
arc(cc, dd).
arc(bb, dd).
arc(bb, ee).
arc(dd, ff).
arc(ee, ff).
```

Programmation récursive

Exemple 1 : parcours de graphe

5. Définition des clauses

- `chemin(D, A, [])` est vrai si `arc(D, A)` est vrai
- `chemin(D, A, [X|Q])` est vrai si `arc(D, X)` est vrai et `chemin(X, A, Q)` est vrai

Programmation récursive

Exemple 1 : parcours de graphe

5. Définition des clauses

Soit les règles suivantes

- `chemin(D, A, []) :- arc(D, A).`
- `chemin(D, A, [X|Q]) :- arc(D, X), chemin(X, A, Q).`

Programmation récursive

Exemple 1 : parcours de graphe

Questions

```
/* démonstration de la prog. réversible : rechercher la liste
des pts int. pour aller de aa à dd */
chemin(aa, dd, L).
   L = [bb]
   L = [cc]
```

Programmation récursive

Exemple 1 : parcours de graphe

Questions

```
/* Rechercher les pts D et A passant uniquement par dd */
chemin(aa, dd, [dd]).
   D = bb, A = ff
   D = cc, A = ff
```

Programmation récursive

Exemple 1 : parcours de graphe

Questions

```
/* Rechercher les pts D et A passant par au moins 2 escales */
chemin(D, A, [E1, E2|Q]).
```

Programmation récursive

Exemple 2 : membre d'une liste

1. SDD

   X l'élément à rechercher, L la liste d'éléments

2. Relations

   membre(X, L) est vraie si X est dans dans L

Programmation récursive

Exemple 2 : membre d'une liste

3. Cas

   Si la liste est vide : échec

   Sinon L = [Y | Q]

   si X = Y alors OK

   sinon X est membre de Q ?

Programmation récursive

Exemple 2 : membre d'une liste

4. Descriptions des appels de clauses

   Comment traduire membre(X, []) = échec ?

   Solution : ne pas l'écrire !

```
membre(X, [Y|Q]) :- X=Y.
membre(X, [Y|Q]) :- membre(X, Q).
```

Programmation récursive

Exemple 2 : membre d'une liste

4. Descriptions des appels de clauses

Astuce de simplification : quand une règle à la forme

```
predicat(…, X, [Y|Q]) :- …, X=Y, ….
```

ou

```
predicat(…, X, …, Y) :- …, X=Y, ….
```

Alors on peut respectivement les simplifier par

```
predicat(X,[X|Q]) :- ….
predicat(X, …, X) :- ….
```

Programmation récursive

Exemple 2 : membre d'une liste

5. Clauses

```
membre(X, [X|Q]).
membre(X, [Y|Q]) :- membre(X, Q).
```

Programmation récursive

Exemple 2 : membre d'une liste

Questions

```
membre(1, [3, 1, 4]).
true

membre(X, [3, 1, 4]).
X = 3
X = 1
X = 4
true
```

Programmation récursive

Exemple 3 : concaténation de 2 listes

1. $\left.\begin{aligned} L1 &= [1,2] \\ L2 &= [3,4] \end{aligned}\right\} \Longrightarrow L3 = [1,2,3,4]$
2. conc(L1, L2, L3) est vraie si L3 = L1 suivi de L2
3. Cas

   Si L1 = [] alors L3 = L2

   Sinon L1=[X|Q] → $\left\{\begin{array}{l} \boxed{\text{X | Q}}\ \boxed{\text{L2}} \\ \boxed{\text{X | Reste}} \end{array}\right.$ L3 = [X|Reste]

Programmation récursive

Exemple 3 : concaténation de 2 listes

4. 5. Clauses

```
conc([], L2, L2).
conc([X|Q], L2, [X|R3]) :- conc(Q, L2, R3).
```

Programmation récursive

Exemple 3 : concaténation de 2 listes

Questions

```
conc([1, 2], [3, 4], L3).
L3 = [1, 2, 3, 4]
```

Programmation récursive

Exemple 3 : concaténation de 2 listes

Ce programme est réversible. Aussi, on peut lui poser les questions suivantes :

```
/* Quelle est la liste de début ? */
conc(L1, [3, 4], [1, 2, 3, 4]).
L1 = [1, 2]
/* Quelle est la liste de fin ? */
conc([1, 2], L2, [1, 2, 3, 4]).
L2 = [3, 4]
```

Programmation récursive

Exemple 3 : concaténation de 2 listes

```
/* Quelles sont toutes les combinaisons de concaténation menant à cette
liste ? */

conc(L1, L2, [1, 2, 3, 4]).

L1 = [],
L2 = [1, 2, 3, 4]

L1 = [1],
L2 = [2, 3, 4]

L1 = [1, 2],
L2 = [3, 4]

L1 = [1, 2, 3],
L2 = [4]

L1 = [1, 2, 3, 4],
L2 = []
```

Ressources

Merci à

- Daniel Rocacher pour son cours dispensé à l'ENSSAT en 2000.
- Nathalie Morvan pour les notes manuscrites
- Jean-François Perrot de l'EPITA pour les compléments
  - https://pages.lip6.fr/Jean-Francois.Perrot/Prolog/Cours1.html

Exercices

FYI

Si en utilisant SWISH, vous avez le message « Singleton variable: [X] », sachez qu'il ne s'agit que d'un avertissement indiquant que X n'est pas utilisé. Si c'est normal, remplacez votre variable X (ou autre) par « _ ».

Si dans une règle de la forme `predicat(…X…) :- …p(A), …p(B)…, X=A+B`

vous pensez mettre la contrainte X doit être égale à A+B, pour Prolog « X=A+B » qui pourrait être par exemple « 3=0+3 » est faux, non unifiable. Aussi, remplacez « X=A+B » par « S is A+B, X=S ».

Exercices

Ex1

- Unifications de liste
  - Dessinez les 2 arbres et expliquez le résultat de l'unification de

$$[[\text{le}, \text{Y}] \mid \text{Z}] \leftarrow unification \rightarrow [[\text{X}, \text{lievre}], [\text{est}, \text{ici}]]$$

Exercices

Ex2

- A propos des clauses sur les chemins, exprimez les buts permettant de répondre aux questions suivantes :
  - Quels sont les chemins menant à ff ?
  - Quels sont les chemins partant de aa ?
  - Quels sont tous les chemins possibles ?
  - Quels sont les chemins dont dd est exactement le 2ème point intermédiaire ?

Exercices

Ex3

- Que donne la question membre(5, L). ?
- Pourquoi ?

Exercices

Ex4

- Voici le prédicat membre modifié :

```
membre(X, [X|_]) :- write("trouvé"), nl.
membre(X, [_|Q]) :- write("continue"), nl, membre(X, Q).
```

- Qu'observez-vous pour le but membre(5, [1, 2, 5, 4, 3] ?
- Considérant l'hypothèse qu'un élément n'est présent qu'une fois dans la liste, est-ce optimisé ?
- Modifiez le prédicat pour que la recherche soit optimale.

Exercices

Ex5

- En vous appuyant sur le prédicat `conc` et une liste résultante donnée, écrivez la question qui permet de déterminer le dernier élément de la liste
- De même, pour le premier élément de la liste
- De même, pour le 2ème élément de la liste

Exercices

Ex6

On peut unifier une variable avec le résultat d'une opération arithmétique grâce à l'opérateur « is ».

Ex : add(X, Y, R) :- R is X + Y.

Sachant cela, écrivez les règles du prédicat « long(L, R) » qui est vrai si R est égal au nombre d'éléments de la liste L.

Exercices

Ex7

On peut unifier une variable avec le résultat d'une opération arithmétique grâce à l'opérateur « is ».

Ex : add(X, Y, R) :- R is X + Y.

Sachant cela, écrivez les règles du prédicat « long(L, R) » qui est vrai si R est égal au nombre d'éléments de la liste L.

Exercices

Ex8

Soit le prédicat gen(X, M) qui va générer des valeurs de X jusqu'à M.

```
gen(X, M) :- between_(0, M, X).
/* between_(I,J,K) is true if K is an integer between I and J inclusive.    */
between_(I,J,I) :- I =< J.
between_(I,J,K) :- I < J, I1 is I+1, between_(I1,J,K).
```

Exercices

Ex8

- Essayez la question `gen(X, 10).`
- A l'aide de ce prédicat, écrivez la règle du prédicat

`combi_mult(X, A, B)` qui est vrai si X = A * B.

La question `combi_mult(10, A, B).` doit générer les résultats suivants :

```
A = 1
B = 10
A = 2
B = 5
A = 5
B = 2
A = 10
B = 1
```