Rappels cours précédent *(VIDEO-PPT 3.1)*

- Une clause Prolog a une interprétation logique : c'est une clause de Horn.
- D'un point de vue logique, montrer que `Q` est vrai revient à montrer que la négation `¬Q` conduit à une contradiction en appliquant la règle de résolution.
- La SLD-résolution est l'algorithme utilisé par Prolog pour mettre en œuvre ce processus. Cet algorithme introduit des spécificités car il tient compte de l'ordre d'écriture des clauses et de l'ordre des littéraux dans les clauses (évaluation de gauche à droite). Par la suite, ces spécificités devront être prises en compte lors de l'écriture de programmes Prolog.
- Les listes non vides se définissent sur des arbres [T | Q] constituées d'un élément de tête T et de Q qui est la liste privée du 1er élément
- Les arbres se mettent en œuvre avec des termes composés et foncteurs ou comme des listes de listes

4.2. Récursivité et réversibilité

En Prolog la récursivité est une manière naturelle de manipuler des données et des programmes.

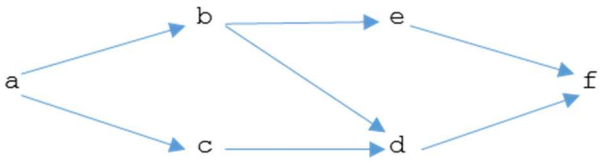
Éléments de méthode :

1. Caractériser les objets à manipuler / en relation.
2. Exprimer le problème sous forme relationnelle.
3. Analyse de cas :
   - cas général ;
   - cas particulier(s) (cas de base).
4. Pour chaque cas, description de la tête de clause.
5. Définition des clauses.
6. Est-ce que tous les cas possibles ont été traités ?

Chaque règle doit avoir un sens logique et donc être compréhensible indépendamment les unes des autres.

➢ *Exemple* : parcours d'un graphe

Trouver les chemins (listes des points intermédiaires) entre un point de départ et un point d'arrivée du graphe suivant :

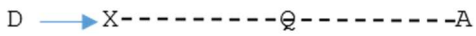


En Prolog, la description de ce graphe peut être exprimée par la base de faits suivante :

```
arc(a, b).
arc(a, c).
arc(b, e).
arc(b, d).
arc(c, d).
```

```
arc(d, f).
arc(e, f).
```

1. Le problème met en relation 3 objets :
   - Un point de départ : `D`
   - Un point d'arrivée : `A`
   - Une liste de points intermédiaires : `L`
2. `chemin(D, A, L)` est vrai si `L` est la liste des points entre `D` et `A`
3. Analyse, 2 cas :
   - `L` est vide = `[]` : `D` et `A` sont en liaison directe
   - `L` est non vide = `[X|Q]` : `L` contient au moins un élément `X`, c'est le premier point sur le chemin de `D` à `A`.
4. A chaque cas correspond une tête de clause :
   - `chemin(D, A, [])`
   - `chemin(D, A, [X|Q]`
5. Définition des règles :
   - `chemin(D, A, [])`est vrai si `arc(D, A)`
   - `chemin(D, A, [X|Q])`est vrai si `arc(D, X)` et `chemin(X, A, Q)`



D'où le programme :

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

Des questions :

```
|?- chemin(a, d, L).
      L = [b] ?a
      L= [c]
|?- chemin(a, f, L).
      L = [b,e] ? a
      L = [b,d]
      L = [c,d]
|?- chemin(D, A, [d]).
      A = f , D = b ? a
      A = f , D = c

|?- chemin(D, A, [I]).
      A = f , D = b , I = b ? a
      A = d , D = a , I = b
      A = d , D = a , I = c
      A = f , D = b , I = d
      A = f , D = c , I = d
```

Ces exemples illustrent la notion de réversibilité des arguments dans une question Prolog. En général Prolog n'impose pas de statut aux paramètres d'un prédicat (la notion de paramètre entrée / sortie n'a pas de sens en logique). Cette caractéristique confère une puissance parfois insoupçonnée au programme Prolog.

Sur l'exemple, le prédicat `chemin` a été construit de manière implicite en pensant « si j'ai D et A, quel est le résultat L ». Cependant, du fait de la réversibilité, il peut être utilisé de multiples façons différentes.

Néanmoins, compte tenu du mode de recherche de Prolog (en profondeur d'abord et de gauche à droite), certains arguments peuvent avoir un sens E/S imposé. C'est le problème de l'arrêt de la récursivité qui guide ces choix.

➢ *Autre exemple :* le prédicat membre

Le prédicat `membre(X, L)` est vrai si `X` est un élément de la liste `L`.

- Si `L` est vide : échec quel que soit `X`
- Si `L` est non vide
  - Soit `X` est le premier élément de `L = [X|Q]`
  - Soit `X` est membre de `Q` avec `L = [Y|Q]`

Le problème est ici d'exprimer que `membre (X, [])` est faux. Il faut se rappeler que la base de connaissances n'exprime que des propositions vraies (hypothèse du monde fermé). Tout ce qui n'est pas dans la base de connaissances est donc considéré comme faux.

D'où le programme :

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

Des questions :

```
|?- membre(1, [1,2,3]).
      true ?a
      yes
|?- membre(X, [1,2,3]).
      X = 1 ? a
      X = 2
      X = 3
```

Cette question est intéressante car elle permet d'énumérer les différents éléments d'une liste. C'est une fonctionnalité qui nous sera utile par la suite (*cf* programmation par essais successifs).

```
|?- membre(X, L).
      L = [_,X|_] ? ;
      L = [_,_,X|_] ? ;
      L = [_,_,_,X|_] ? ;
      …
```

Évidemment cette dernière question entraine une infinité de réponses (et boucle), ce qui en soit est correct.

Synthèse sur la réversibilité

☞ *Un prédicat peut admettre plusieurs possibilités d'appels (de combinaisons de paramètres libres et liés), ces possibilités découlent de la sémantique opérationnelle de Prolog, la résolution de l'appel (la question) doit se terminer.*

☞ *Il peut être intéressant d'indiquer explicitement les possibilités d'appel en commentaires, les conventions suivantes sont utilisées pour chaque argument* a_i *d'un prédicat P d'arité n (*$i \in 1..n$*) :*

+a_i *si l'argument doit être lié*
-a_i *si l'argument doit être libre*
?a_i *si l'instanciation est indifférente*

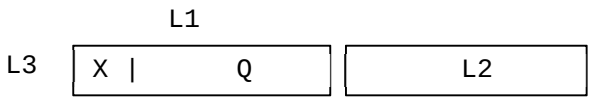
exemple : cf doc GProlog), `/* member(?E,+L) */`

➢ *Mini-TP 1 : Concaténer deux listes*

Le prédicat `conc(L1, L2, L3)` est vrai si `L3` est la concaténation des listes `L1` et `L2`. Par exemple, `conc([1,2], [3,4], [1,2,3,4])` est vrai.

Raisonnons en considérant le premier argument :

- Si `L1` est vide : `L3 = L2`
- Si `L1` est non vide, `L1` est de la forme `[X|Q]` :



⇨ `L3` est de la forme `X` suivi de la concaténation de `Q` et de `L2`

Ci-dessus aucune hypothèse n'a été faite sur `L2` (qui peut être vide ou non).

➢ *Mini-TP 2 : Construction d'une sous-liste*

Définir le prédicat `impair(L1, L2)` tel que `L1` soit une liste contenant un nombre pair d'éléments (exemple : [0, 1, 2, 3]) et L2 soit la liste des éléments de rang impair de `L1` ([0, 2]).

- Si `L1` est vide : `L2` est vide
- Si `L1` est non vide, `L1` est de la forme `[X,Y|Q]`, `L1` contient au moins 2 éléments
  - `X` est le premier élément de `L2` qui est donc de la forme `[X|Q2]`
  - Par ailleurs, les éléments de rang impair de `Q` correspondent à `Q2`