Aller un peu plus loin avec Prolog

Dans ce TP, nous allons étudier plus en détail le fonctionnement de Prolog et la manière d'en tirer parti pour résoudre des problèmes.

1 Unification

Dans le TP précédent, nous avons utilisé des variables sans trop nous poser de question, mais le mécanisme de gestion des variables en Prolog est assez différent de ce dont on a l'habitude.

Ouvrez l'interpréteur Prolog et tapez

```
?- X=2.
```

Prolog vous répondra `X=2`, ce qui ne devrait pas trop vous étonner. Pourtant, ce que vous venez de faire n'est pas à proprement parler une affectation de variable. Vous n'avez pas décrété "Je décide que X vaut 2", mais vous avez demandé à Prolog "En fonction de ce que tu sais déjà, se peut-il que X=2 ?".

Pour mieux comprendre ce phénomène essayons d'autres tentatives[1] :

```
?- X=Y.
```

La réponse de Prolog est que la valeur `X` et `Y` doit être la même, mais qu'elle n'est pas déterminée.

Plus difficile :

```
?- X=Y, Y=2.
```

Dans un langage traditionnel, on aurait ici une erreur : La valeur de Y est utilisée avant d'être définie. Prolog, lui, se souvient que les deux variables doivent avoir la même valeur et lorqu'une des deux reçoit une valeur, l'autre aussi !

```
?- X=Y, Y=2, X=3.
```

Cet ensemble d'affectations est incohérent. La réponse de Prolog est donc `Fail`.

Ceci met en évidence une caractéristique de Prolog : les variables sont *immuables*. Une fois une valeur affectée, il n'est plus possible d'en changer pour toute la durée de vie de la variable (i.e. la requête). Ceci peut paraître surprenant, mais à la réflexion c'est cohérent : si les "affectations" sont au fait des questions du genre "est-il possible que X prenne telle valeur ?", il est normal que la valeur ne puisse pas changer.

Le mécanisme qui permet de gérer les valeurs des variables s'appelle *unification*. Au fait, `X=Y` se lit "`X` s'*unifie* à `Y`".

Mais l'unification peut faire plus que tester si 2=3. Elle inclut un puissant mécanisme de *Pattern Matching*. Essayez

```
?- X+Y=1+2.
?- X+Y=f(x)*2+3*4.
?- X*3=2*Y.
```

[1] Notez que les "liaisons" de variables ne portent que sur la requête en cours ; chacune de ces requêtes est donc indépendante.

Par contre, ce *pattern matching* peut causer des surprises :

```
?- X = 2+3.
```

Pour forcer l'évaluation arithmétique de la partie droite, on doit utiliser `is` :

```
?- X is 2+3.
```

Comme vous le voyez, cela force l'évaluation de la partie droite, puis tente l'unification avec la partie gauche.

Exercice Que répond Prolog lors des tentatives d'unifications suivantes ? essayez de deviner, puis vérifiez dans l'interpréteur.

1. bread = bread
2. 'Bread' = bread
3. 'bread' = bread
4. Bread = bread
5. bread = sausage
6. food(bread) = bread
7. food(bread) = X
8. food(X) = food(bread)
9. food(bread,X) = food(Y,sausage)
10. food(bread,X,beer) = food(Y,sausage,X)
11. food(bread,X,beer) = food(Y,kahuna_burger)
12. food(X) = X
13. meal(food(bread),drink(beer)) = meal(X,Y)
14. meal(food(bread),X) = meal(X,drink(beer))

Prédicats relatifs à l'unification

x=y réussit si x et y sont unifiables. Dans ce cas, l'unification est réalisée.

x\=y réussit si x et y *ne* sont *pas* unifiables

var(x) réussit si x est une variable non instantiée

nonvar(x) réussit si x *n*'est *pas* une variable non instantiée

x==y réussit si x et y sont égaux (sans forcer l'unification)

x\==y réussit si x et y *ne* sont *pas* égaux (sans forcer l'unification)

Prédicats relatifs à l'arithmétique Pour tous les prédicats suivants, les parties évaluées doivent être entièrement instanciées. Par exemple

```
?- X=2, 2 =:= X.
```

réussit alors que

```
?- 2 =:= X, X=2.
```

provoque une erreur.

x is y est décrit ci-dessus

x=:=y est utilisé pour tester l'égalité de deux nombres ou de deux expressions arithmétiques. Force l'évaluation arithmétique des deux expressions avant la comparaison.

x<y, x=<y, x>y, x>=y forcent l'évaluation artihmétique des deux côtés et réussit si la comparaison est vraie.

Exemple : la dérivation Pour illustrer le mécanisme d'unification, nous allons implémenter une esquisse de dérivation de fonctions. Nous voudrons implémenter les règles de dérivation suivantes :

1. $\frac{d}{dx}x = 1$
2. $\frac{d}{dx}C = 0$
3. $\frac{d}{dx}x^n = nx^{n-1}$
4. $\frac{d}{dx}(u(x)+v(x)) = \frac{d}{dx}u(x) + \frac{d}{dx}v(x)$
5. $\frac{d}{dx}(u(x)v(x)) = \frac{du(x)}{dx}v(x) + u(x)\frac{dv(x)}{dx}$

Comment implémenter ceci ? la première question qui se présente est la suivante : les prédicats Prolog peuvent *réussir* ou *échouer*, mais ne retournent pas de valeur...

Nous implémenterons donc la dérivation comme une relation entre la fonction, la variable et la dérivée : `d(f(x),x,g(x))` signifiera : $\frac{df(x)}{dx} = g(x)$

La première règle pourrait donc s'implémenter ainsi :

```
d(X,Y,Z) :- X=Y, Z=1.
```

Mettez cette règle dans un fichier et lancez-le ; interrogez votre base en entrant

```
?- d(x,x,R).
```

Si tout va bien, Prolog devrait vous répondre `R=1`, qui est bien la réponse qu'on attend !

Mais comme l'unification n'est pas une affectation de variables, on peut aussi utiliser ce prédicat

– pour vérifier un calcul : `?- d(x,x,1).` devrait donner `true`.

– pour *intégrer*, c'est à dire inverser le processus de dérivation : `?- d(F,x,1).` nous donne `F=x`.

Pour les mêmes raisons, on peut écrire notre règle de manière plus économique. Ouvrez votre fichier et remplacez son unique règle par le fait

```
d(X,X,1).
```

Ceci devrait fonctionner de la même manière qu'avant !

La deuxième règle fait intervenir le prédicat `atomic`, qui réussit si son argument est... atomique (i.e. un nombre ou une constante). Les suivantes ne posent pas de problème particulier et font plein usage du *pattern matching* :

```
d(C,X,0) :- atomic(C), C\=X.
d(X^N,X,D):-N1 is N-1, D=N*X^N1.
d(U+V,X,DU+DV) :- d(U,X,DU), d(V,X,DV).
d(U*V,X,U*DV+V*DU) :- d(U,X,DU), d(V,X,DV).
```

Vous pouvez maintenant tester une dérivation :

```
?- d(3*x^2+5,x,R).
```

Le résultat n'est pas très simplifié, mais il est correct.

Remarque Malheureusement, le code ci-dessus ne peut pas être utilisé "à l'envers" pour intégrer ; en effet, l'opération `is` de la règle 3 ne marche que dans un sens et la règle numéro 2 ne peut pas être "retournée" (infinité de solutions !).

2 Le moteur en chaînage arrière

La page `http://cs.union.edu/~striegnk/learn-prolog-now/html/node20.html` décrit très bien le fonctionnement de Prolog lorsqu'on lance une requête.

Lisez ce document maintenant.

2.1 Exercice (sur papier)[2]

Considérons la base prolog suivante :

```
1 direct_flight(hongkong, beijing).
2 direct_flight(beijing, tokyo).
3 direct_flight(tokyo, sanfrancisco).
4 direct_flight(sanfrancisco, hongkong).
5 flight(X,Y) :- direct_flight(X,Y).
6 flight(X,Y) :- direct_flight(X,Z), flight(Z,Y).
```

Dessinez un arbre de recherche pour la requête

```
flight(hongkong,X).
```

Développez votre arbre de recherche jusqu'à la cinquième réponse de Prolog (incluse !).
Votre arbre de recherche devra indiquer

1. Les faits et règles utilisés (notez le n° de la ligne)
2. Les unifications effectuées. Les variables intermédiaires seront nommées _1, _2, _3, etc.

Pour économiser de la place, vous pourrez utiliser les abréviations suivantes :
- flight $\rightsquigarrow$ f
- direct_flight $\rightsquigarrow$ df
- hongkong $\rightsquigarrow$ hk
- tokyo $\rightsquigarrow$ tk
- beijing $\rightsquigarrow$ bj
- sanfrancisco $\rightsquigarrow$ sf

2.2 Exercice (sur ordinateur)

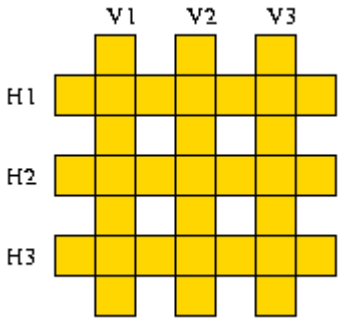
Vous aurez donc compris que Prolog cherche (en profondeur) une combinaison de l'attribution des variables qui satisfasse toutes les contraintes exprimées. Ceci permet de résoudre certains problèmes de type combinatoire très facilement.

La figure 1 présente un mot croisé simple à 6 mots. Il s'agit de trouver un placement cohérent des 6 mots dans la grille.

Indication Commencez par définir la liste des mots considérés et donner les lettres qui les composent...

```
word(abalone,a,b,a,l,o,n,e).
word(abandon,a,b,a,n,d,o,n).
word(enhance,e,n,h,a,n,c,e).
word(anagram,a,n,a,g,r,a,m).
word(connect,c,o,n,n,e,c,t).
word(elegant,e,l,e,g,a,n,t).
```

[2] inspiré de `http://www.cs.ust.hk/~oscarau/comp251_2006/notes/prolog_ex1.htm`



Mots à placer :
- abalone
- abandon
- enhance
- anagram
- connect
- elegant

FIG. 1: Un mot croisé...

... puis écrivez un prédicat `crossword(V1,V2,V3,H1,H2,H3)` qui permet de trouver le placement des mots dans la grille.

3 Structures de données et récursivité

3.1 Listes

Nous l'avons vu, Prolog connaît les *nombres entiers* (`2`, `-3`) et *réels* (`1.2`, `-1.3e5`) et les "atomes" (`a`, `toto`, ...). Il y a aussi évidemment les *chaînes de caractères* (`'ceci est une chaîne'`) et nous avons rencontré quelques structures comme `f(a(b,c))`.

En dehors de cela, la structure de données fondamentale de Prolog est la *liste* : `[a,1,toto,2.3,'hello']`.

La manière standard d'accéder à une liste en Prolog est la suivante : si L est une liste, [H|R] s'unifie à L en unifiant H au premier élément de L et R à tout le reste de L.

Essayez

```
?- [1,2,3] = [H|R].
?- [H|R] = [1].
?- [H|R] = 1.
```

3.2 Récursivité

Prolog ne propose pas de structures de contrôle telles que les boucles for, while, etc. Comme dans la plupart des langages fonctionnels et logiques, la manière "officielle" de gérer le contrôle est la *récursivité*.

Par exemple, pour écrire les nombres de 0 à N :

```
print_numbers(0) :- write(0), nl.
print_numbers(N) :-
    N1 is N-1,
    print_numbers(N1),
    write(N), nl.
```

(où `write` a un sens évident et `nl` (newline) passe à la ligne).

3.3 Accès récursif aux listes

La manière standard de manipuler des listes en Prolog est donc d'utiliser la récursivité et la notation `[H|R]`. Par exemple, un prédicat pour accéder au *n*ième élément d'une liste :

```
nth(0,[H|_], H).
nth(N,[_|R], X) :- N1 is N-1, nth(N1,R,X).
```

Ce prédicat peut de nouveau être utilisé dans plusieurs sens : `nth(2,[a,b,c], X)` permet de récupérer l'élément 2 de la liste, `nth(2,[a,b,c], c)` permet de vérifier que l'élément 2 est bien `c`, et `nth(2,L,c)` permet de construire une liste partiellement instanciée : tout ce qu'on sait de L est que son élément 2 est `c` ![3]

3.4 Exercices

3.4.1 Listes

En utilisant la notation `[H|R]` et la récursivité, écrire

1. Un prédicat[4] `double/2` qui dédouble tout les éléments d'une liste :

   ```
   ?- double([a,b,c],D).
   D = [a, a, b, b, c, c]
   ```

   Peut-on utiliser ce prédicat pour effectuer l'opération inverse (ie retrouver `[a,b,c]` à partir de `[a, a, b, b, c, c]`) ? Si oui, comment faire ?

2. Un prédicat `zip/3` qui combine deux listes de la manière suivante :

   ```
   ?- zip([a,b,c],[1,2,3],Z).
   Z = [[a, 1], [b, 2], [c, 3]]
   ```

3. Un prédicat `myMember/2` qui se comporte comme la primitive `member/2`, c'est-à-dire qui permet de déterminer si un élément aparaît dans une liste :

   ```
   ?- myMember(a,[b,a,c]).
   true
   ?- myMember(a,[b,d,c]).
   fail
   ```

4. Un prédicat `enumerate/1` qui énumère sur la sortie standard les éléments d'une liste :

   ```
   ?- enumerate([1,2,3]).
   1
   2
   3
   true.
   ```

[3] Il y aurait encore un sens d'utilisation (lequel ?) mais il ne fonctionne pas (pourquoi ?).
[4] La notation `name/n` signifie "un predicat nommé `name` et prenant `n` arguments".

3.4.2 Somme des n premiers nombres

Écrire un prédicat qui fait la somme des n premiers nombres entiers :

```
?- somprem(5,S).
S = 15
```

Pourriez-vous utiliser ce prédicat pour faire l'opération inverse (retrouver 5 à partir de 15) ? Si oui, comment ? si non, pourquoi ?

3.4.3 Rendre la monnaie

Écrire un programme Prolog qui, étant donné un montant payé et un prix, détermine comment rendre la monnaie.

Exemple :

```
?- rend(10,7.45).
A rendre:
1 piece de 0.05
1 piece de 0.5
1 piece de 2
true
```

Indications :

1. Commencer par déclarer les types de pièces disponibles...

```
...
piece(1).
piece(0.5).
...
```

2. ... puis écrire le prédicat `rend` de manière récursive.