Dixième partie X

Introduction à la programmation logique avec Prolog

Plan

En bref ...

Syntaxe et terminologie

Sémantique d'un programme Prolog

Les listes

La coupure

Quelques prédicats prédéfinis de SWI-Prolog

Syntaxe et terminologie

Syntaxe et terminologie

- Un programme Prolog est constitué d'un ensemble de clauses
- Une clause est une affirmation portant sur des atomes logiques
- Un atome logique exprime une relation entre des termes
- Les termes sont les objets de l'univers.

Les termes

Les objets manipulés par un programme Prolog (les "données" du programme) sont appelés des termes. On distingue trois sortes de termes :

1. Les variables représentent des objets inconnus de l'univers
2. Les termes élémentaires (ou termes atomiques) représentent les objets simples connus de l'univers
3. Les termes composés représentent les objets composés (structurés) de l'univers

Les variables

- Les variables représentent des objets inconnus de l'univers. Syntaxiquement, une variable est une chaîne alpha-numérique commençant par :
  - une majuscule par exemple :
    - `Var`, `X`, `Var_longue_2`
  - par un sous-ligné par exemple ;
    - `_objet`, `_21`
- La variable anonyme est notée "_" et représente un objet dont on ne souhaite pas connaître la valeur

Attention

Une variable Prolog s'apparente plus à une variable mathématique qu'à une variable informatique. Elle représente toujours le même objet (inconnu) tout au long de sa durée de vie et ne peut pas changer de valeur.

Les termes élémentaires

- Les termes élémentaires (ou termes atomiques) représentent les objets simples connus de l'univers. On distingue trois sortes de termes élémentaires :
  1. les nombres : entiers ou flottants
  2. les identificateurs (parfois appelés atomes)
     - un identificateur est une chaîne alpha-numérique commençant par une minuscule (e.g., `toto`, `aX12`, `jean_Paul_2`)
  3. les chaînes de caractères entre guillemets (e.g., "le prolog c'est bien", "123").

Les termes composés

- Les termes composés représentent les objets composés (structurés) de l'univers. Syntaxiquement, un terme composé est de la forme :

$$foncteur(t_1, \ldots, t_n)$$

où
  - *foncteur* est une chaîne alpha-numérique commençant par une minuscule
  - $t_1, \ldots, t_n$ sont des termes, i.e., variables, termes élémentaires ou termes composés
- Le nombre d'arguments n est appelé arité du terme

Exemple de termes composées

Soit le terme composé suivant :

```
adresse(18, "rue des lilas", Ville)
```

- le foncteur est `adresse` et a une arité de 3
- les deux premiers arguments sont les termes élémentaires `18` et `"rue des lilas"`
- le troisième argument est la variable `Ville`

De même, soit le terme composé suivant :

```
cons(a, cons(X, nil))
```

- le foncteur est `cons` et a une arité de 2
- le premier argument est le terme élémentaire `a`
- le deuxième argument le terme composé `cons(X,nil)`

Les relations, ou atomes logiques

- Un atome logique exprime une relation entre des termes
- Cette relation peut être vraie ou fausse.
- Syntaxiquement, un atome logique est de la forme :

$$predicat(t_1, \ldots, t_n)$$

où
  - *predicat* est une chaîne alpha-numérique commençant par une minuscule
  - $t_1, \ldots, t_n$ sont des termes
- Le nombre d'arguments n est appelé arité de l'atome logique.

Exemple de relations

Soit la relation suivante :

```
pere(toto, paul)
```

- la relation `pere` est d'arité 2 entre les termes élémentaires `toto` et `paul`
- la relation peut être interprétée par *" toto est le père de paul "*

De même, soit la relation suivante :

```
habite(X, adresse(12, "rue r", nice))
```

- la relation `habite` est d'arité 2 entre la variable `X` et le terme composé `adress(12, "rue r", nice)`
- la relation peut être interprétée par *" une personne inconnue X habite à l'adresse (12, "rue r", nice) "*

Les clauses

- Une clause est une affirmation qui peut être :
  1. inconditionnelle on parle alors de fait
  2. conditionnelle on parle alors de règle.

Les faits

- Un fait est de la forme :

  `A.`

  où `A` est un atome logique, et signifie que la relation définie par `A` est vraie (sans condition). Par exemple, le fait

  `pere(toto, paul).`

  indique que la relation " *toto est le père de paul* " est vraie.

Remarque

Une variable dans un fait est quantifiée universellement, e.g., le fait

`egal(X, X).`

indique que la relation "`X` *est égal à* `X`" est vraie pour toute valeur (tout terme) que `X` peut prendre.

Les règles (1/2)

- Une règle est de la forme :

  `A0 :- A1 , ... , An.`

  ou `A0`, `A1`, ..., `An` sont des atomes logiques.
- Une telle règle signifie que la relation `A0` est vraie si les relations `A1` et ... et `An` sont vraies
- `A0` est appelé tête de clause et `A1`, ..., `An` est appelé corps de clause

Les règles (2/2)

- Une variable apparaissant dans la tête d'une règle (et éventuellement dans son corps) est quantifiée universellement
- Une variable apparaissant dans le corps d'une clause mais pas dans sa tête est quantifiée existentiellement, e.g., la clause

  `meme_pere(X, Y) :- pere(P, X), pere(P, Y).`

  se lit

  " *pour tout* `X` *et pour tout* `Y`, `meme_pere(X,Y)` *est vrai s'il existe un* `P` *tel que* `pere(P,X)` *et* `pere(P,Y)` *soient vrais* "

Les programmes Prolog

- Un programme Prolog est constitué d'une suite de clauses regroupées en paquets.
- L'ordre dans lequel les paquets sont définis n'est pas significatif.
- Chaque paquet définit un prédicat et est constitué d'un ensemble de clauses dont l'atome de tête a le même symbole de prédicat et la même arité.
- L'ordre dans lequel les clauses sont définies est significatif.
- Intuitivement, deux clauses d'un même paquet sont liées par un ou logique, e.g., le prédicat `personne` défini par les deux clauses suivantes :

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

  se lit

  " pour tout `X`*,* `personne(X)` *est vrai si* `homme(X)` *est vrai ou* `femme(X)` *est vrai "*

Exécution d'un programme Prolog (2/2)

- La réponse de Prolog est alors l'ensemble des valeurs des variables pour lesquelles la question est une conséquence logique du programme, e.g.,, la question

  ```
  ?- pere(toto, X), pere(X, Y).
  ```

  se lit

  " est-ce qu'il existe un X et un `Y` *tels que* `pere(toto,X)` *et* `pere(X,Y)` *soient vrais ".*

- La réponse de Prolog est l'ensemble des valeurs de `X` et `Y` qui vérifient cette relation
- Autrement dit, la réponse de Prolog à cette question devrait être l'ensemble des enfants et petits-enfants de `toto` si `toto` est effectivement grand-père

Exécution d'un programme Prolog (1/2)

- *Exécuter* un programme Prolog consiste à poser une question à l'interprète PROLOG
- Une question (ou but ou activant) est une suite d'atomes logiques séparés par des virgules
- La réponse de Prolog est " `yes` " si la question est une conséquence logique du programme, ou " `no` " si la question n'est pas une conséquence logique du programme
- Une question peut comporter des variables, quantifiées existentiellement

La sémantique

Le concepte de substitution

- Une substitution, s, est une fonction de l'ensemble des variables dans l'ensemble des termes, e.g.,

$$s = X \leftarrow Y, Z \leftarrow f(a, Y)$$

  est la substitution qui " *remplace* " X par Y, Z par $f(a, Y)$, et laisse inchangée toute autre variable que X et Z
- Par extension, une substitution peut être appliquée à un atome logique, e.g.,

$$s(p(X, f(Y, Z))) = p(s(X), f(s(Y), s(Z))) = p(Y, f(Y, f(a, Y)))$$

Le concepte d'instance

- Une instance d'un atome logique A est le résultat $s(A)$ de l'application d'une substitution s sur A, e.g.,

```
pere(toto, paul)
```

  est une instance de

```
pere(toto, X).
```

Le concepte d'unificateur

- Un unificateur de deux atomes logiques A_1 et A_2 est une substitution s telle que $s(A_1) = s(A_2)$, e.g., soit

$$A_1 = p(X, f(X, Y))$$

  et

$$A_2 = p(a, Z)$$

  et

$$s = X \leftarrow a, Z \leftarrow f(a, Y)$$

  est un unificateur de A_1 et A_2 car $s(A1) = s(A2) = p(a, f(a, Y))$

Le concepte d'unificateur le plus général

- Un unificateur s de deux atomes logiques A_1 et A_2 est le plus général (upg) si pour tout autre unificateur s' de A_1 et A_2, il existe une autre substitution s'' telle que $s' = s''(s)$
- Par exemple,

$$s = X \leftarrow Y$$

  est un upg de $p(X, b)$ et $p(Y, b)$, tandis que

$$s' = X \leftarrow a, Y \leftarrow a$$

  n'est pas un upg de $p(X, b)$ et $p(Y, b)$
- L'algorithme de Robinson calcule un upg de deux termes, ou rend " *echec* " si les deux termes ne sont pas unifiables

Rappel : L'algorithme d'unification de Robinson (1/2)

Algorithme

```
funtion UNIFY(x, y, sigma) return a subtitution to make x and y identical or failure
    inputs : x, a terme, i.e, a variable, a constant, a function, or a list
             y, a terme, i.e, a variable, a constant, a function or a list
             σ, a subtutution initially empty {}
    if σ = failure then return failure
    if x = y then return σ
    if VARIABLE ?(x) then return UNIFY-VAR(x,y,σ)
    if VARIABLE ?(y) then return UNIFY-VAR(y,x,σ)
    if FUNCTION ?(x) and FUNCTION ?(x) then
        if FUNCTOR(x) ≠ FUNCTOR(y) then return failure
        return UNIFY(AGRS[x],AGRS[y],σ)
    if LIST ?(x) and LIST ?(x) then
        if LENGTH(x) ≠ LENGTH(y) then return failure
        return UNIFY(REST[x], REST[y], FIRST[x], FIRST[y], sigma))
    return failure
```

Rappel : L'algorithme d'unification de Robinson (2/2)

Algorithme

```
funtion UNIFY-VAR(var, x, sigma) return a subtitution
    inputs : var, a variable
             x, a terme, i.e, a variable, a constant, a function or a list
             σ, a subtutution
    if {var/val} ∈ sigma then return UNIFY(val,x,σ)
    if {x/val} ∈ sigma then return UNIFY(var,val,σ)
    if OCCUR-CHECK ?(var,x) then return failure
    else return add {var/x} to σ
```

La dénotation d'un programme Prolog

- La dénotation d'un programme Prolog P est l'ensemble des atomes logiques qui sont des conséquences logiques de P
- Ainsi, la réponse de Prolog à une question est l'ensemble des instances de cette question qui font partie de la dénotation
- Cet ensemble peut être " calculé " par une approche ascendante, dite en chaînage avant
  - on part des faits (autrement dit des relations qui sont vraies sans condition), et on applique itérativement toutes les règles conditionnelles pour déduire de nouvelles relations jusqu'à ce qu'on ait tout déduit

Exemple de chaînage avant (1/3)

- Considérons par exemple le programme Prolog suivant :

```
parent(paul,jean).
parent(jean,anne).
parent(anne,marie).
homme(paul).
homme(jean).

pere(X,Y) :- parent(X,Y), homme(X).

grand_pere(X,Y) :- pere(X,Z), parent(Z,Y).
```

Exemple de chaînage avant (2/3)

- L'ensemble des relations vraies sans condition dans P est E_0 :

  E_0 = `parent(paul,jean), parent(jean,anne), parent(anne,marie), homme(paul), homme(jean)`

  à partir de E_0 et P, on déduit l'ensemble des nouvelles relations vraies E_1 :

  E_1 = `pere(paul,jean), pere(jean,anne)`

  à partir de E_0, E_1 et P, on déduit l'ensemble des nouvelles relations vraies E_2 :

  E_2 = `grand_pere(paul,anne), grand_pere(jean,marie)`

  à partir de E_0, E_1, E_2 et P, on ne peut plus rien déduire de nouveau
- L'union de E_0, E_1 et E_2 constitue la dénotation (l'ensemble des conséquences logiques) de P.

Exemple de chaînage avant (3/3)

- Malheureusement, la dénotation d'un programme est souvent un ensemble infini et n'est donc pas calculable de façon finie. Considérons par exemple le programme P suivant :

```
plus(0,X,X).
plus(succ(X),Y,succ(Z)) :- plus(X,Y,Z).
```

- L'ensemble des atomes logiques vrais sans condition dans P est

  E_0 = `plus(0,X,X)`

  à partir de E_0 et P, on déduit

  E_1 = `plus(succ(0),X,succ(X))`

  à partir de E_0, E_1 et P, on déduit

  E_2 = `plus(succ(succ(0)),X,succ(succ(X)))`

  à partir de E_0, E_1, E_2 et P, on déduit

  E_3 = `plus(succ(succ(succ(0))),X,succ(succ(succ(X))))`

  etc.

Signification opérationnelle

- D'une façon générale, on ne peut pas calculer l'ensemble des conséquences logiques d'un programme par l'approche ascendante
  - ce calcul serait trop couteux, voire infini
- En revanche, on peut démontrer qu'un but (composé d'une suite d'atomes logiques) est une conséquence logique du programme, en utilisant une approche descendante, dite en chaînage arrière

Démontrer un but par chaînage arrière (1/4)

- Pour prouver un but composé d'une suite d'atomes logiques, e.g.,

  $$But = [A_1, A_2, \ldots A_n])$$

  l'interprête Prolog commence par prouver le premier de ces atomes logiques (A_1).
- Pour cela, il cherche une clause dans le programme dont l'atome de tête s'unifie avec le premier atome logique à prouver, e.g., la clause

  $$A'_0 : -A'_1, A'_2, \ldots, A'_r$$

  telle que $upg(A_1, A'_0) = s$
- Puis l'interprête Prolog remplace le premier atome logique à prouver (A_1) dans le but par les atomes logiques du corps de la clause, en leur appliquant la substitution (s).

Démontrer un but par chaînage arrière (2/4)

- Le nouveau but à prouver devient

$$But = [s(A'_1), s(A'_2), \ldots, s(A'_r), s(A_2), \ldots, s(A_n)]$$

- L'interprête Prolog recommence alors ce processus, jusqu'à ce que le but à prouver soit vide, i.e., jusqu'à ce qu'il n'y ait plus rien à prouver.
- A ce moment, l'interprête Prolog a prouvé le but initial :
  - si le but initial comportait des variables, il affiche la valeur de ces variables obtenue en leur appliquant les substitutions successivement utilisées pour la preuve

Remarque

Il existe généralement plusieurs clauses dans le programme Prolog dont l'atome de tête s'unifie avec le premier atome logique à prouver. Ainsi, l'interprête Prolog va successivement répéter ce processus de preuve pour chacune des clauses candidates. Par conséquent, l'interprête Prolog peut trouver plusieurs réponses à un but.

Démontrer un but par chaînage arrière (3/4)

- Ce processus de preuve en chaînage arrière est résumé par la fonction suivante :

Algorithme de l'interpréteur

```
procedure prouver(But: liste d'atomes logiques )
  si But = [] alors
  /* le but initial est prouve */
  /* afficher les valeurs des variables du but initial */
  sinon soit But = [A_1, A_2, .., A_n]
    pour toute clause (A'_0 :- A'_1, A'_2, ..,A'_r) du programme:
        (ou les variables ont ete renommees)
      s <- upg(A_1 ,A'_0)
      si s != echec alors
        prouver([s(A'_1), s(A'_2), .. s(A'_r), s(A_2), .. s(A_n)], s(But-init}))
      finsi
    finpour
  finsi
fin prouver
```

Démontrer un but par chaînage arrière (4/4)

- Quand on pose une question à l'interprète Prolog, celui-ci exécute dynamiquement l'algorithme précédent. L'arbre constitué de l'ensemble des appels récursifs à la procédure est appelé arbre de recherche
- La stratégie de recherche n'est pas complète, dans la mesure ou l'on peut avoir une suite infinie d'appels récursifs
- La stratégie de recherche dépend d'une part de l'ordre de définition des clauses dans un paquet (si plusieurs clauses peuvent être utilisées pour prouver un atome logique, on considère les clauses selon leur ordre d'apparition dans le paquet), et d'autre part de l'ordre des atomes logiques dans le corps d'une clause (on prouve les atomes logiques selon leur ordre d'apparition dans la clause)

Les listes

Les listes

- La liste est un terme composé particulier de symbole de fonction " `.` " et d'arité 2
  - Le premier argument est l'élément de tête de la liste
  - le deuxième argument est la queue de la liste
- La liste vide est notée " `[]` "

Exemple de notation

- Exemples :
  - la liste `.(X,L)` est également notée `[X|L]`
  - la liste `.(X1, .(X2, L))` est également notée `[X1, X2|L]`
  - la liste `.(X1, .(X2,..., .(Xn, L)... ))` est également notée `[X1, X2,..., Xn|L]`
  - la liste `[X1, X2, X3,..., Xn|[]]` est également notée `[X1, X2, X3,..., Xn]`
- Autres exemples :
  - la liste `[a,b,c]` correspond à la liste `.(a,.(b,.(c,[])))` et contient les 3 éléments `a`, `b` et `c`
  - La liste `[a,b|L]` correspond à la liste `.(a,.(b,L))` et contient les 2 éléments `a` et `b`, suivis de la liste (inconnue) `L`

Notion et représentation

- Une liste est une structure récursive :
  - la liste `Liste = [X|L]` est composée d'un élément de tête `X` et d'une queue de liste `L` qui est elle-même une liste
  - la définition récursive implique que les relations Prolog qui " manipulent " les listes seront généralement définies par :
    1. une ou plusieurs clauses récursives, définissant la relation sur la liste `[X|L]` en fonction de la relation sur la queue de liste `L`
    2. une ou plusieurs clauses non récursives assurant la terminaison de la manipulation, et définissant la relation pour une liste particulière, e.g., la liste vide, ou la liste dont l'élément de tête vérifie une certaine condition etc.

La coupure

Signification opérationnelle

- La coupure, aussi appelée " *cut* ", est notée !
- La coupure est un prédicat sans signification logique (la coupure n'est ni vraie ni fausse), utilisée pour " couper " des branches de l'arbre de recherche.
- La coupure est toujours " prouvée " avec succès dans la procédure `prouver`
- La " preuve " de la coupure a pour effet de bord de modifier l'arbre de recherche
  - elle coupe l'ensemble des branches en attente créées depuis l'appel de la clause qui a introduit la coupure

Exemple d'utilisation d'une coupure (1/4)

- Considérons par exemple le programme Prolog suivant :

```
p(X,Y) :- q(X), r(X,Y).
p(c,c1).
q(a).
q(b).

r(a,a1).
r(a,a2).

r(b,b1).
r(b,b2).
r(b,b3).
```

Exemple d'utilisation d'une coupure (2/4)

- L'arbre de recherche construit par Prolog pour le but `p(Z,T)` est :

Arbre de recherche

```
prouver([p(Z,T)])
  b1 : s = {Z <- X, T <- Y}
  ---- prouver([q(X),r(X,Y)])
    b11 : s = {X <- a}
    ----- prouver([r(a,Y)])
       b111 : s = {Y <- a1}
       ------ prouver([],[p(a,a1)] --> solution1 = {Z=a, T=a1}
         b112 : s = {Y <- a2}
         ------ prouver([]) --> solution2 = {Z=a, T=a2}
    b12 : s = {X <- b}
    ----- prouver([r(b,Y)])
      b121 : s = {Y <- b1}
      ------ prouver([]) --> solution3 = {Z=b, T=b1}
      b122 : s = {Y <- b2}
      ------ prouver([]) --> solution4 = {Z=b, T=b2}
      b123 : s = {Y <- b3}
      ------ prouver([]) --> solution5 = {Z=b, T=b3}
  b2 : s = {Z <- c, T <- c1}
  ---- prouver([]) --> solution6 = {Z=c, T=c1}
```

Exemple d'utilisation d'une coupure (3/4)

- En fonction de l'endroit ou l'on place une coupure dans la définition de p, cet arbre de recherche est plus ou moins élagué et des solutions sont supprimées :
  - si on définit p par

    ```
    p(X,Y) :- q(X), r(X,Y),!.
    p(c,c1).
    ```

    ↪ Prolog donne la solution 1 puis coupe toutes les branches en attente (b112, b12 et b2)
  - si on définit p par

    ```
    p(X,Y) :- q(X),!, r(X,Y).
    p(c,c1).
    ```

    ↪ Prolog donne les solutions 1 et 2 puis coupe les branches en attente (b12 et b2)
  - si on définit p par

    ```
    p(X,Y) :-!, q(X), r(X,Y).
    p(c,c1).
    ```

    ↪ Prolog donne les solutions 1, 2, 3, 4 et 5 puis coupe la branche en attente (b2).

Exercice

Que répond Prolog aux questions suivantes ?

1. `?- element(L,[[a,b],[c,d,e]]), element(X,L).`
2. `?- element(L,[[a,b],[c,d,e]]), element(X,L),!.`
3. `?- element(L,[[a,b],[c,d,e]]),!, element(X,L).`
4. `?-!, element(L,[[a,b],[c,d,e]]), element(X,L).`

Quelques prédicats prédéfinis

Les applications de la coupure

1. Recherche de la première solution
2. Exprimer le déterminisme
3. La négation
4. Le " si-alors-sinon "

Édition et chargement de programmes

```
edit(toto)      % appelle l'editeur vi avec le fichier toto
[fichier]       % charge le fichier toto dans la base
listing         % liste l'ensemble des predicats de la base
```

Comparaison de termes

```
T1 == T2        % reussit si T1 est identique a T2
T1 \== T2       % reussit si T1 n'est pas identique a T2
T1 = T2         % reussit si T1 est unifiable avec T2
T1 \= T2        % reussit si T1 n'est pas unifiable avec T2
T1 @< T2        % reussit si T1 est inferieur a T2
T1 @=< T2       % reussit si T1 est inferieur ou egal a T2
T1 @> T2        % reussit si T1 est superieur a T2
T1 @>= T2       % reussit si T1 est superieur ou egal a T2
```

Arithmétique

`Expr1 =:= Expr2`

réussit si le résultat de l'évaluation de l'expression arithmétique `Expr1` est égal au résultat de l'évaluation de l'expression arithmétique `Expr2`

`Var is Expr`

unifie le résultat de l'évaluation de l'expression arithmétique `Expr` avec la variable `Var`

Entrées/Sorties (1/2)

```
nl                % saut de ligne
tab(N)            % affiche N espaces
get(C)            % lit un caratere et l'unifie avec C
get_single_char(C) % idem get(C), mais n'attends pas de retour charriot
read(T)           % lit un terme et l'unifie avec T
write(Format, L)  % Format est une chaine de carateres a afficher
                  % L est une liste de termes a afficher
```

`Format` peut contenir des séquences particulières permettant d'afficher des caractères spéciaux :

```
\n                % <NL> est affiche
\t                % <TAB> est affiche
\\                % '\' est affiche
\%                % '%' est affiche
```

Entrées/Sorties (2/2)

`Format` peut contenir des séquences particulières pour inclure des élément de la list `L`. La séquence dépend de la nature de l'élément à afficher :

```
%t                % si l'element est un term
%n                % si l'element est un code ASCII
%s                % si l'element est une chaine de caracteres
```

Par exemple, l'exécution de

```
writef("ceci %t l'%s %t  n et j'affiche X=%t",
       [est,"exemple numero",45,X])
```

affichera la séquence :

```
ceci est l'exemple numero 45
et j'affiche X=toto
```

si la variable `X` a pour valeur `toto`