Projet prolog

LIU Zhuying p1306849

Partie1:

Toutes les fonctions tournent mais l'affichage est different que celui donné dans le sujet.
Pour que les règles soient plus logiques, j'ai défini les deux fonctions « si » et « alors » qui me servent ensuite pour insérer les conclusions directement dans la base des faits. Par exemple,quand j'appelle la fonction regle(r1), ça vérifie si les deux premisses « a » et « b »sont dans la base, et si oui, « c » est ajouté directement dans la base.
Pour éviter qu'un fait soit ajouté plusieurs fois, dans la fonction « fait » avant d'ajouter un fait j'enlève d'abord ce fait de la base.
Pour pouvoir dérouler plusieurs fois « saturer », « marque » et « changement » doivent pouvoir être ré-initialisés, donc dans la fonction « raz » démarque toutes les règles et initialise « changement » à vrai.

```
?- [tp5].
true.

?- raz.
true.

?- fait([a,c,d]).
true.

?- saturer.
f est prouvé par r2
g est prouvé par r4
b est prouvé par r5
e est prouvé par r3
true.

?- listing(vrai(_)).
:- dynamic vrai/1.

vrai(a).
vrai(c).
vrai(d).
vrai(f).
vrai(g).
vrai(b).
```

vrai(e).

true.

Partie2:

L’affichage est légèrement différent, mais le principe est respecté. C’est une fonction récursive: pour une conclusion, je vérifie soit elle dans la base, soit ses deux prémisses peuvent être satisfaites, si oui, cette conclusion est aussi ajoutée dans la base.
Pour faciliter la récupération des deux premisses d’une conclusion, j’ai fait une fonction comme suit : regle(R,P1,P2,C):-clause(regle(R),(si([P1,P2]),alors([C]))). Donc P1,P2 sont récupérés directement depuis une règle.

?- raz.
true.

?- fait([a,c,d]).
true.

?- satisfait(e).
parcourir règle r3 :
parcourir règle r2 :
c est dans la base
d est dans la base
f est ajouté dans la base
fin de parcours r2
parcourir règle r5 :
parcourir règle r4 :
f est dans la base
a est dans la base
g est ajouté dans la base
fin de parcours r4
f est dans la base
b est ajouté dans la base
fin de parcours r5
e est ajouté dans la base
fin de parcours r3
true.

Partie3:

Les deux fonctions «observable» et «terminal» sont faites facilement avec une negation. La fonction «go» marche avec l'exemple « fait([a,c,d])». Je n'ai pas pu testé tous les cas, mais le test « fait([c,d])» marche. Et quand l'utilisateur répond non, le test marche.
Comme dans prolog il n'y a pas de variable de retour, « uninconnu » est une fonction qui trouve la prémisse inconnue parmi les deux, et qui appelle directement le chainage arrière de la prémisse inconnue.
Comme dans prolog il n'y a pas de switch condition, «traiter(Y,X)» est une fonction qui vérifie la valeur de Y saisie par l'utilisateur, et traiter X(la variable que l'on voulait traiter avant la saisie).

```
/*test exemple*/
?- raz.
true.

?- fait([a,c,d]).
true.

?- go.
f est prouvé par r2
g est prouvé par r4
b est prouvé par r5
e est prouvé par r3
h à prouver !
Est ce que h ? (o/n/i)
|: o.
h car observé
l est prouvé par r6
m est prouvé par r7
false.


/*test fait([c,d])*/
?- raz.
true.

?- fait([c,d]).
true.

?- go.
f est prouvé par r2
a à prouver !
Est ce que a ? (o/n/i)
|: o.
```

a car observé
g est prouvé par r4
b est prouvé par r5
e est prouvé par r3
h à prouver !
Est ce que h ? (o/n/i)
|: o.
h car observé
l est prouvé par r6
m est prouvé par r7
false.

/*test quand utilisateur répond non*/
?- raz.
true.

?- fait([c,d]).
true.

?- go.
f est prouvé par r2
a à prouver !
Est ce que a ? (o/n/i)
|: n.
a faux
false.