Jules NOEL
Gabriel NOURRIT

UFR IM²AG
UNIVERSITÉ Grenoble Alpes

TP SUITE LOGIQUE

Programmation Déclarative

SOMMAIRE

I / Introduction

Pour ce tp nous devions réaliser des prédicats en prolog qui nous permettent de résoudre des suites logiques.
Pour réaliser ceci nous avons utilisé swish qui nous a permis de pouvoir tester nos prédicats.

II / Réalisation du projet

Ce tp a été fait en plusieurs étapes :

- réponse au trois types de suites données en exemple
- recherche d'autres suites logiques
- réponse à toutes ces suites logiques

Afin de répondre aux trois types de suite données en exemple nous avons commencé par réaliser un prédicat qui permet la dérivation d'une liste. Ensuite en fonction de la dérivée on test pour voir quel est le type de notre liste. Suivant le type on a une réalisation différentes, et le programme nous renvoi l'élément qu'il pense être le suivant dans la liste. Pour commencer notre programme renvoyait seulement un élément il ne regardait pas dans une liste de proposition, nous l'avons introduit par la suite.

Après avoir vérifié que tout cela marchait bien nous avons décidé de nous attaquer à d'autre types de suites que l'on pourrait résoudre grâce à notre programme. Nous avons trouvé et réalisé 6 types de suites différentes que nous expliquerons plus tard.

D'autre part, nous avons écrit un prédicat relativement complet afin de déterminer et de résoudre un ensemble assez large de cycle. Pour celui-ci nous avons cherché un algorithme, puis nous l'avons implanté sur prolog. L'algorithme que nous avons mis en place respecte la démarche suivante :

- on prend 1 liste, on considère que si le 1er élément ne réapparaît pas dans la liste on dit qu'il n'y a pas de cycle ex : [1,4,5,4,5,4,5]

Ensuite on va prendre une suite comme exemple [1,2,1,3,1,2,1,3,1,2,1,3] :

- E = 1 ; L = [2,1,3,1,2,1,3,1,2,1,3] on regarde si 1 apparaît dans la liste
- on trouve le 1 à la place numéro 1
- puis on regarde si tous les éléments suivant du 1 trouvé sont égaux aux éléments en partant du début de la liste (ici on a tout de suite 2!=3)
- si il y a une différence on blacklist ce 1 et on cherche le suivant jusqu'à ce que tout soit égal
- ici on se retrouve donc au 1 à la place 3
- quand tout est égal on supprime le premier élément (celui qu'on a sélectionner dans E) on renvoie la liste qui suit le premier élément comparé au vide. (çà marche comme on compare M de longueur inférieure à L) ici par exemple on compare :
- 2(0)=2(4) 1(1)=1(5) avec (l'indice)
- 3=3, 1=1, 2=2, 1=1, 2=2, 1=1, 3=3,
- (1=[] ?) -> [1,2,1,3] et on retourne donc ceci

III / Spécification du programme

Fonctionnalités disponibles :

- Donner une liste logique et récupérer toutes les solutions que notre programme trouve. Pour cela exécuter une requête ?- trouverX([1,2,3,4],X).
- Donner une liste logique et une liste de proposition et voir quel solution notre programme garde. Pour cela exécuter une requête ?- resolution([1,2,3,4],[100,5],X).
- Au delà de notre programme général, l'ensemble des prédicats que l'on a utilisé sont utilisable, nous avons précisé dans le code les dépendances de chaque prédicat, ce qui permet une intégration rapide et modulaire dans un autre programme. Nous avons aussi spécifié chacun d'eux afin de bien comprendre leur rôle et comment les utiliser.

Suites supportées :

Suites données en exemple :

- Additives constantes ex : [1,2,3,4,5,6]
- Multiplicatives constantes ex : [2,4,8,16,32]
- Additives cycliques ex : [1,2,4,5,7]

Autres types de suites :

- Multiplicatives cycliques ex : [10,10,20,20,40,40]
- Fibonacci et suites du type L[n+2]==L[n+1]+L[n] ex : [0,1,1,2,3,5,8,13]
- Dérivée additive jusqu'au degré n (nombre d'éléments) ex de degré 2 : [1,2,4,7,11]
- 1 dérivée additive + 1 dérivée mult coef const ex : [1,3,7,15,31]
- 1 dérivée mult + 1 dérivée additive coef const ex : [2,4,12,48,240]
- Suite de puissance successive ex : [1,4,9,16]

Piste d'amélioration : sur les cycles, notre algorithme est capable de détecter des rafales de propositions sans calculs supplémentaires, on pourrait exploiter ceci pour proposer plusieurs solution au lieu d'une seule.

VI / Conclusion

Pour conclure on remarque que Prolog et la programmation déclarative sont pratique pour manipuler des listes et les analyser. On aurait sans doute écrit beaucoup plus de lignes si on avait voulu le faire dans un langage impératif. Ce fût un projet intéressant tout en étant un peu différent de ce qu'on à l'habitude de faire. Il nous a permis de mieux apprécier la valeur de prolog et sa sans doute toute puissance dans le domaine de l'intelligence artificielle (du côté de la théorie des jeux). Enfin, nous nous sommes aussi familiariser avec swish et son excellent debugger.

V / Liens du projet

Lien swish :

- https://swish.swi-prolog.org/p/projet_prolog_noel_nourrit.pl

Lien google drive pour télécharger le fichier .pl :

- https://goo.gl/MmLwHH

```
/*
* @dependances : trouverX,isin.
* @param L une liste logique
* @param P une liste de solution possibles
* @param X la solution trouve dans la liste proposer apres passage dans notre programme
* resout quelques problemes de suite logique
* resolution(L,L1,X) vrai ssi X est dans L1 et est l'element suivant de L
*/
resolution(L,ListeSolution,X):-
    trouverX(L,X),
    isin(X,ListeSolution).

/*
* @dependances : dif.
* @param E un Element
* @param L une liste
* verifie si E n est pas dans L
* notisin(X,L) vrai ssi X n'est pas dans L
*/
notisin(_,[]):-!.
notisin(E,[F|R]):-
    dif(E,F),
    notisin(E,R).

/*
* @dependances : dif.
* @param E un Element
* @param L une liste
* verifie si E est dans L
* isin(X,L) vrai ssi X est dans L
*/
isin(E,[E|_]).
isin(E,[F|R]):- dif(E,F), isin(E,R).


/*
* @dependances : +, notisin, isin, dif.
* @param E un Element
* @param L une liste
* @param P une liste de proposition (indexes renverses)
* @param T la taille de L
* @param Blacklist une liste d index d element de la liste que l on veut ignorer
* prend un element regarde si il est dans une liste a un index qui n est pas dans la blacklist
* si l element reunis les conditions alors il est ajoute a la liste des propositions
* exist(E,L,P,T,B) vrai ssi E est dans L et P et si son indice n'est pas dans B
*/
%isin(Element,Liste,Proposition,Taille,Blacklist)
exist(_,[],[],0,_). %arret
exist(E,[E|L],[I|Reste],I,BL):- %element trouvé non BL
    exist(E,L,Reste,J,BL),
    I is J+1,
    notisin(I,BL).
exist(E,[E|L],Reste,I,BL):- %element trouvé BL
    exist(E,L,Reste,J,BL),
    I is J+1,
    isin(I,BL).
exist(E,[F|R],Reste,I,BL):- %element qui ne match pas
    dif(E,F),
    exist(E,R,Reste,J,BL),
    I is J+1.

/*
* @dependances : -.
* @param L1 une liste d indexes
* @param L2 la liste des indexes de L1 renverses
* @param T la taille de la liste L1
* @invariant res() = (x== t-y+1)
* renverse les indexes d une liste exemple ci dessous avec x un ancien index et y le nouvel index
* t:= 5 5 5 5 5
* x:= 1 2 3 4 5
* y:= 5 4 3 2 1
* reverse(L1,L2,X) vrai ssi L2 est la liste reverse de L1 de taille X
*/
reverse([],[],_).
reverse([E|L],[R|Tab],Taille):-
    reverse(L,Tab,Taille),
     R is Taille-E. %on commence a  zero

/*
* @dependances : +.
* @param L1 une liste
* @param t la taille de la liste
* calcul la taille d une liste
* mlength(L,X) vrai ssi X est la taille de la liste L
*/
mLength([],0).
mLength([_|R],N):-
    mLength(R,M),
    N is M+1.


/*
* @dependances : >=, <.
* @param L1 une liste
* @param m le max (nombre) de la liste
* calcul le maximum (nombre) d une liste
* maximum(L,X) vrai ssi X est le max de L
*/
maximum([S],S).
maximum([E|L],E):-
    maximum(L,Mp),
    E>=Mp.
maximum([E|L],Mp):-
    maximum(L,Mp),
```

```
    E<Mp.


/*
* @dependances : mlength, maximum, reverse, exist.
* @param E un element
* @param L une liste
* @param I une liste d index ou se trouve E dans L
* @param BL une blacklist
*
* calcul seul la taille de L, utilise exist, et remet en forme correctement la liste d indexe pour qu elle soit a l endroit
*/
existFinal(E,L,I,BL):-
    %calcul de la taille de tableau
    mlength(L,Llength), % ce sera soit la taille de L+1
    maximum(BL,MaxBl),  % soit le Max de la blacklist
    maximum([Llength,MaxBl],N),
    reverse(BL,NL,N), %on remet BL à l'envers pour le traitement de exist
    exist(E,L,Proposition,Taille,NL),
    reverse(Proposition,I,Taille).%on met les propositions à l'endroit pour la suite

%-----------------------------------------


/*
* @dependances : -, >=, <.
* @param L une liste
* @param M une autre liste
*
* M a -1 a tout les element de L et enleve les valeurs <0 (hors bornes)
* ex : [0,1,2,3] -> [0,1,2]
*/
majBL([E],[F]):- F is E-1.
majBL([E|L],[F|NL]):-
     F is E-1,
     F>=0,
    majBL(L,NL).
majBL([E|L],NL):- %ajout du retrait des valeurs hors bornes
    F is E-1,
     F<0,
    majBL(L,NL).


/*
* @dependances : +.
* @param L une liste
* @param I un index
* @param E un element
* donne l'element d'une liste situe a la position I  (si il existe)
* element(L,I,E) vrai ssi E est à l'indice I dans L
*/
element([E|_],0,E).
element([_|R],I,E):-
    element(R,J,E),
    I is J+1.

/*
* @dependances : mlength, algo.
* @param L une liste de derive
* @param Res une sous liste de L
* prend une liste derive et va generer une suite pour le cycle trouve (si il existe) dans Res
* final(L,R) vrai ssi R est l'élément suivant de R
*/
final(Liste,Res):-
    mlength(Liste,Length),
    algo(Liste,Res,Length,[-1]).


/*
* @dependances : existFinal, element, trIu, tr, majBL, not, eq, +, >=, <, -.
* @param L une liste de derive
* @param Res une sous liste de L
* @param Taille la longueur de L
* @param BL une liste d index d elements de L a ignorer
* prend une liste derive et va generer une suite pour le cycle trouve (si il existe) dans Res
*/
%(5)exist+1 hors bornes
algo([E|L],L,Taille,BL):-
    existFinal(E,L,TabI,BL),           %boucle
    element(TabI,0,I),
    J is I+1,
    J >= (Taille-1).

%(1)boucle if dans bornes
algo([E|L],Proposition,Taille,BL):-
    existFinal(E,L,TabI,BL),           %boucle
    element(TabI,0,I),
    J is I+1,
    J < Taille-1,
    trIu(L,J,Elem),
    tr(L,Elem),
    majBL(BL,NB),
    algo(L,Proposition,Taille-1,NB).

%(2)boucle else dans bornes
algo([E|L],Proposition,Taille,BL):-
    existFinal(E,L,TabI,BL),           %boucle
    element(TabI,0,I),
    J is I+1,
    J < Taille-1,
    trIu(L,J,Elem),
    not(tr(L,Elem)),
    eq(BBL,[I|BL]),%inversion d'indice ?
    algo([E|L],Proposition,Taille,BBL).
```

```
%---------------------
/*
* @param L une liste
* @param M une sous liste de L
* regarde si tout les element de M en partant de l index 0 sont egaux a ceux de L exemple :
* [1,2,3,4,5,6] & [1,2,3] -> ok
* tr(L,M) vrai ssi tout les element de M en partant de l index 0 sont egaux a ceux de L
*/
tr([_|_],[]).
tr([E|R],[E|L]):-
    tr(R,L).


/*
* @dependances : +.
* @param L une liste
* @param I un indice
* @param E une sous liste de L
* genere E la sous-liste de L commencant a l index I dans la liste L
* ex : L=[1,2,3,4,5,6] & I=2 -> E=[3,4,5,6]
* trIu(L,I,E) vrai ssi E est la sous-liste de L commencant a l index I dans la liste L
*/
trIu([_|L],I,E):-
    trIu(L,J,E),
    I is J+1.
trIu(L,0,L):-!.

%-----------------------------------------------

/*
* @dependances : >, /, integer.
* @param L une liste
* @param M une liste derivee multiplicative
* construit M a partir de L en divisant L[n+1] avec L[n]
* derivation(L,M) vrai ssi M est la derivee multiplicative de L
*/
derivation([_],[]).
derivation([],[]).
derivation([E,F|R],[G|Res]):-
    E>0,
    G is F/E,
    integer(G), % on ne recupere que les coef entiers pour regler les eventuelles problemes d'arrondies pouvant subvenir lors de comparaison entre ent
    derivation([F|R],Res).


/*
* @dependances : dif.
* @param L une liste
* regarde si L est une suite constante
* pascte(L) vrai ssi L n'est pas une suite constante
*/
pascte([]).
pascte([E,F|_]):-dif(E,F).
pascte([E,E|R]):- pascte([E|R]).

/*
* @dependances : additive, write.
* utile pour ne pas tout reexucuter quand une solution additive a été trouvé
*/
u(L,X):-
    additive(L,_,X),
    !,
    write("additive: Une solution possible est ":X),
    write("\n").

/*
* * @dependances : u.
* Cas liste additive
* ex:[1,2,4,7,11]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    u(L,X).


%-------------------------------------------------------------------------------


/*
* @dependances : derivation,suiteConst, recupererDernier, fois, write.
* Cas liste mutiplicative a coefficient fixé
* ex:[1,2,7,32,157]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    derivation(L,ListeDerivee),
    suiteConst([Coeff|ListeDerivee]),
    recupererDernier(L,DernierListe),
    fois(DernierListe,Coeff,X),
    write("multiplicative").


/*
* @dependances : derivee, pascte, final, recupererDernier, plus, write.
* Cas liste cyclique additif
* ex:[1,2,4,5,7]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    derivee(L,ListeDerivee), /*Calcul de la dérivée->[1,2,1,2]*/
    pascte(ListeDerivee),
    final(ListeDerivee,[Coeff|_Propositions]), /*On recupere ce que notre algo de cycle propose*/
    recupererDernier(L,DernierListe), /*on récupère le dernier de la liste*/
```

```
    plus(DernierListe,Coeff,X),
    write("cycle add"). /*X=7+1=8*/

/*
* @dependances : derivation, pascte, final, recupererDernier, fois, write.
* Cas cycle multiplicatif
* ex:[1,2,4,8]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    derivation(L,ListeDerivee), /*Calcul de la dérivée->[1,2,1,2]*/
    pascte(ListeDerivee),
    final(ListeDerivee,[Coeff|_Propositions]), /*On recupere ce que notre algo de cycle propose*/
    recupererDernier(L,DernierListe), /*on récupère le dernier de la liste*/
    fois(DernierListe,Coeff,X),
    write("cycle mult").

/*
* @dependances : integer, derivee, derivation, suiteConst, recupererDernier, fois, plus, write.
* Cas suite derive add / derivee mult
* ex: [1,3,7,15,31]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    derivee(L,ListeDerivee),
    derivation(ListeDerivee,Liste2),
    suiteConst([Raison|Liste2]),
    recupererDernier(L,DernierListe), /*on récupère le dernier de la liste*/
    recupererDernier(ListeDerivee,DernierDerivee),
    fois(DernierDerivee,Raison,X1),
    plus(X1,DernierListe,X),
    integer(X),
    write("suite derivee add - mult de coef: "),
    write(Raison).

/*
* @dependances : derivation, derivee, suiteConst, recupererDernier, plus, fois, integer, write.
* Cas suite derive mult / derivee add
* ex: [2,4,12,48,240]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    derivation(L,ListeDerivee),
    derivee(ListeDerivee,Liste2),
    suiteConst([Raison|Liste2]),
    recupererDernier(L,DernierListe), /*on récupère le dernier de la liste*/
    recupererDernier(ListeDerivee,DernierDerivee),
    plus(DernierDerivee,Raison,X1),
    fois(X1,DernierListe,X),
    integer(X),
    write("suite derivee mult - add de coef: "),
    write(Raison).


/*
* @dependances : fiboAndOth, write.
* Suite de fibonacci & autre du meme style
* ex :[0,1,1,2,3,5,8,13] ; [1,2,3,5,8]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    fiboAndOth(L,X),
    write("suite de fibonacci et autres suites additives style 1 2 3 5 8 13 ...").


/*
* @dependances : liste, write.
* Suite de puissance successives
* ex :[pow(1,1), pow(2,2), pow(3,3), pow(4,4)]
* trouverX(L,X) vrai ssi X est l'élément suivant de L
*/
trouverX(L,X):-
    liste(L,X),
    write("suite des puissances successives").


/*
* @dependances : +, is.
* @param liste L
* @param proposition P
* verifie que L est de type L[n+2] = L[n]+L[n+1] et fais une proposition P pour le prochain element de la liste
* fiboAndOth(L,Res) vrai ssi Res est l'élément suivant de la liste L
*/
%pair
fiboAndOth([X,F],Res):- Res is X+F,!. %lui ok
%impair
fiboAndOth([A,B,C], Res):- C is A+B,Res is B+C, !.
%parcours
fiboAndOth([E,F,G|R],Res) :-
    G is E+F,
    fiboAndOth([F,G|R],Res).


/*
* @package : sqrt,sqrs
* @dependances : <,-,=,*, is.
* @param entier N
* @param racine P
* calcul la racine carre entiere de N (si elle existe) et met dans P
* sqrt(N,P) vrai ssi P = racine carré de N
*/
sqrt(N,P) :-
    sqrs(N,1,P).
```

```
sqrs(N,M,P) :-
    P2 is M*M,
    N < P2,
    P is M-1,
    Z is P*P,!,
    N = Z.

sqrs(N,M,Q) :-
    M2 is M+1,
    sqrs(N, M2, Q).

/*
* @dependances : sqrt, is, +, *.
* calcul des liste de puissance ex pow(1,1) pow(2,2) pow(3,3) : 1,4,9 et propose la suite dans X
* liste(L,X) vrai ssi X est l'élément suivant de L
*/
liste([E],X):-
    sqrt(E,Xc),
    Xcp is Xc+1,
    X is Xcp*Xcp,!.
liste([A,B|L],X):-
    sqrt(A,E),
    sqrt(B,F),
    F is E+1,
    liste([B|L],X).

%-------------------------------------------------------------------------------------------
/*Savoir si liste additive ou non
additive(L,L2,Add) vrai ssi la liste L2 (L2 étant la dérivée de la liste L) est de type additive*/
additive(L,L2,Add):- %pas constante
  derivee(L,L2),
  pascte(L2),
  additive(L2,_,E),
  recupererDernier(L,G),
  Add is E+G.
additive(L,_,Add):- %constante
  derivee(L,L2),
  suiteConst([F|L2]),
  recupererDernier(L,E),
  Add is E+F.

/*Calcul de la liste dérivée
derivee(L,L1) vrai ssi la liste L1 est la liste dérivée de L*/
derivee([_E],[]). /*Cas de base, si on a un élément dans la liste la dérivée est une liste vide*/
derivee([A,B|R],[Res|R1]):-
    moins(B,A,Res), /*On fait A-B et on le stocke dans Res (que l'on met au début de la nouvelle liste)*/
    derivee([B|R],R1). /*On relance en parcourant les éléments sans le premier*/

/*Verifie si une liste est constante
suiteConst(L) vrai ssi la liste L est constante*/
suiteConst([_]). %impair
suiteConst([]):- !. % pair
suiteConst([E,E|R]):-
    suiteConst([E|R]).

/*Permet de recuperer le dernier d'une liste
recupererDernier(L,X) vrai ssi X est le dernier élément de la liste L*/
recupererDernier([E],E). /*Cas de base, un seul élément donc c'est le dernier*/
recupererDernier([_E|R],X):-
    recupererDernier(R,X). /*On cherche le dernier en parcourant la liste*/

/*Operations de base*/
/*moins(A,B,R) vrai ssi R=A-B*/
moins(A,B,R):- R is A-B.
/*plus(A,B,R) vrai ssi R=A+B*/
plus(A,B,R):- R is A+B.
/*fois(A,B,R) vrai ssi R=A*B*/
fois(A,B,R):- R is A*B.
/*eq(A,B) vrai ssi a=B*/
eq(A,B):- A=B.
```