KØBENHAVNS UNIVERSITET

Advanced Programming

AP20 Assignment3 Groups 26

Assignment 3

csb847, hcx121

Datalogisk Institut30. september 2020

1 Design and Implementation

1.1 Level 0

First we implement predicate **follows**. We have to check whether a person B is in the follow list of A in graph G. So we implement a predicate **inside**, to find out if an element E is in the list L. Then we just use the **inside** to first check if A is in graph and then if B is in A's follow list.

Then it comes to predicate **ignores**. We have to check both B follows A and A doesn't follow B. The former question can be done as the predicate **follows**, and for the latter one, we implement a predicate **notInside**. For **notInside**, we have to implement a series of predicates **different**, **getRest**, **getName**, where **getName** is for getting all the names in graph G, **getRest** is for getting all the names in graph G except name A, **different** is for check whether name A and B are in graph G and A is not equal to B. With these predicates, we can implement **notInside** to check if name A is in the complement of B's follow list.

1.2 Level 1

For predicate **popular**, we get the list that A follows and find out whether all people in the list follows A. We implement a recursive predicate **checkPop** for this purpose.

For predicate **outcast**, we implement a recursive predicate **checkOut** to see if all the people in A's follow list ignores A.

To implement the next two predicates, we implement **checkFollow** first. It takes a graph G, a name A, a name list N and a subset of N who follows A. Then for **friendly**, we implement a predicate **checkFriendly** which takes two lists L1 and L2 and check if all the elements in L1 is in L2 (i.e. if L1 is the list who follows A, L2 is the list A follows, then it returns true if A follows all who follows him). For **hostile**, we implement **checkHostile** to check if all the elements in L1 is not in L2 (i.e. if L1 is the list who follows A, L2 is the list A follows, then it returns true if A ignores all who follows him).

In **friendly**, we get a list of names who follows A in graph G, and use **checkFriendly** to check if all those people are followed by A. And In **hostile**, we get a list of names who follows A in graph G, and use **checkHostile** to check if all those people are ignored by A.

1.3 Level 2

For **aware**, we have to first check if the user is asking about someone aware of himself, and returns false if so. Then we use Breadth First Search **bfs** to look for people he is aware of. If we are asking if A is aware of B, we first put A into the queue, and check if B is in the follow list of A, if so, terminate, else we sign A as visited, put all the names A follows and have not been visited into the queue, pop A from the queue, and then repeat. We ignore those who are signed visited in the queue. In **bfs** we implement **checkAware** and **checkUnaware** to help. Also we implement **addToTail** to implement the function of append, and **notInList** to get a list of elements in L1 but not in L2.

For **ignorant**, we also have to check the user is asking about someone aware of himself, and returns false if so. Then we use Breadth First Search **awareList** to get the aware list of some person A. We implement the Breadth First Search as stated above, except that we do not terminate on finding some B, but traverse the entire graph to get a list of names aware by A.

1.4 Level 3

For **same_world**, we first check if the two graphs G1 and G2 has the same number of elements (i.e. there are the same number of people). If the number of people are not the same, it cannot be a same world. So we implement a predicate **comprLen** to count. Then we have to check that the list of keys have the same number and order as G1 by implementing **checkK**. We use **replace** and **subReplace** to replace all the names in G1 with keys. Then we have to check if the replaced list is the same as G2. Here we implement two predicates **checkSame** and **sameList**, where **sameList** is to check if all elements in list L1 are in list L2 regardless of the order, and **checkSame** is for world list. We have to use both **sameList** and **checkSame** on L1, L2 and L2, L1 to make sure L1 and L2 are the same.

2 Assessment of the code

2.1 Test in the instatest.pl

In order to testify the correctness of all the required functionality, we wrote 31 test cases. We carefully selected all the possible cases including these may cause faults.

To testify the predicate in level 0, we tested predicate follow() with parameters that can generates fail, a list, or nondet. Similarly, for ignores(), we tested the cases that would generate the same results. The testing results show that the predicate can correctly analyse the pairwise relationship.

To testify the predicate in level 1, we tested the predicates popular(G, X), outcast(G, X), friendly(G, X), hostile(G, X). We selected several most representing cases , for example, we used the only 'outcasted' element bruce to test the outcast predicate. It is proven to be applicable for all these predicates to judge the local connections for an element.

To testify the predicate in level 2, we also chose these most representing cases. Since the element bruce is the only element that can result in the success of predicate ignore as a target in the given _G, we had limited choices.

And finally ,to testify the predicate in level 3, we construct two worlds to test if they are isomorphism sets. It is proven to be valid in all cases including an empty world, worlds with different relationships while the number of relationships and names stay same, and the replaceable same world.

2.2 Apart from the tests in the instatest.pl

We also used online TA to assess our code and passed all the tests.

In summary, we think our code can get a 6/6 score.

Appendix

warmup.pl

```
% Advanced Programming Assignment 3
% Skeleton for warm−up part. Predicates to implement:

% add(N1, N2, N)
add(z,N2,N2).
add(s(N1),N2,s(Sum)):−add(N1,N2,Sum).


mult(z, N2, z).
mult(s(N1),N2,Res):− mult(N1,N2,Res1),add(Res1,N2,Res).


comp(z, z, eq).
comp(s(N1), z, gt).
comp(z, s(N2), lt).
comp(s(N1), s(N2), A):− comp(N1,N2,A).



% insert(N, TI, TO):−
insert (N, node(N1,L_Leaf,R_Leaf), node(N1,L_Leaf,R_Leaf)):− comp(N,N1,eq).
insert (N, node(N1,L_Leaf,R_Leaf), node(N1,TO,R_Leaf)):− comp(N,N1,lt),insert(N,L_Leaf,TO).
insert (N, node(N1,L_Leaf,R_Leaf), node(N1,L_Leaf,TO)):− comp(N,N1,gt),insert(N,R_Leaf,TO).
insert (N, leaf , node(N,leaf, leaf )).



% insertlist (Ns, TI, TO)
 insertlist ([], TI, TI).
 insertlist ([Head|Tail], TI,TO):− insert(Head,TI,TO1), insertlist(Tail,TO1,TO).
```

instahub.pl

```
% Advanced Programming Assignment 3
% Skeleton for main part. Predicates to implement:

%%% level 0 %%%

inside(X, [H|T]) :- inside_(T, X, H).
inside_(_, E, E).
inside_([H|T], E, _) :- inside_(T, E, H).

getName([], []) .
getName([person(Name, _)|R], [Name|N]) :- getName(R, N).

getRest(X, [Head|Tail], Rest) :- getRest_(Tail, Head, X, Rest).

getRest_(Tail, Head, Head, Tail).
getRest_([Head2|Tail], Head, X, [Head|Rest]) :- getRest_(Tail, Head2, X, Rest).

different(G, X, Y) :-
getName(G, N),
inside(X, N),
inside(Y, N),
getRest(X, N, R),
inside(Y, R).

notInside(G, X, []) .

notInside(G, X, [H|T]) :- notInside_(G, T, X, H).

notInside_(G, _, E, E) :- inside(E, []) .

notInside_(G, [], E, Y) :- different(G, E, Y).

notInside_(G, [H|T], E, Y) :-
different(G, E, Y),
notInside_(G, T, E, H).

follows(G, X, Y) :-
inside(person(X, Z), G),
inside(Y, Z).

ignores(G, X, Y) :-
inside(person(Y, Z), G),
inside(X, Z),
inside(person(X, A), G),
notInside(G, Y, A).

%%% level 1 %%%
checkPop(G, X, []).

checkPop(G, X, [Y|Z]) :-
follows(G, Y, X),
checkPop(G, X, Z).

popular(G, X) :-
inside(person(X, Z), G),
checkPop(G, X, Z).
```

```
checkOut(G, X, []).

checkOut(G, X, [Y|Z]) :-
ignores(G, Y, X),
checkOut(G, X, Z).

outcast(G, X) :-
inside(person(X, Z), G),
checkOut(G, X, Z).

%%% checkFollow: find people who follows X %%%
checkFollow(G, X, [], []) .

checkFollow(G, X, [X|T], R) :-
checkFollow(G, X, T, R).

checkFollow(G, X, [H|T], [H|R]) :-
different (G, X, H),
follows(G, H, X),
checkFollow(G, X, T, R).

checkFollow(G, X, [H|T], R) :-
different (G, X, H),
inside(person(H, Z), G),
notInside(G, X, Z),
checkFollow(G, X, T, R).

%%% checkFriendly: all people in X are in Y %%%

checkFriendly ([], _).

checkFriendly([X|T], Y) :-
inside(X, Y),
checkFriendly(T, Y).

friendly(G, X) :-
getName(G, N),
checkFollow(G, X, N, R),
inside(person(X, Z), G),
checkFriendly(R, Z).

checkHostile(G, [], _).

checkHostile(G, [X|T], Y) :-
notInside(G, X, Y),
checkHostile(G, T, Y).

hostile(G, X) :-
getName(G, N),
checkFollow(G, X, N, R),
inside(person(X, Z), G),
checkHostile(G, R, Z).

%%% level 2 %%%

addToTail([], L, L).
addToTail(L, [], L).
addToTail([H|T], L, [H|R]) :- addToTail(T, L, R).
```

```
checkAware(G, X, Y) :-
inside(person(X, Z), G),
inside(Y, Z).

checkUnaware(G, X, Y) :-
inside(person(X, Z), G),
notInside(G, Y, Z).

bfs(G, [X|T], Visited, Y) :-
inside(X, Visited),
bfs(G, T, Visited, Y).

bfs(G, [X|T], Visited, Y) :-
notInside(G, X, Visited),
checkAware(G, X, Y).

bfs(G, [X|T], Visited, Y) :-
notInside(G, X, Visited),
checkUnaware(G, X, Y),
addToTail([X], Visited, V),
inside(person(X, Z), G),
notInList(G, Z, V, R),
addToTail(R, T, TT),
bfs(G, TT, V, Y).

aware(G, X, Y) :-
different (G, X, Y),
bfs(G, [X], [], Y).

aware(_, X, X) :- inside(X, []) .

%%% notInList: elements in X but not in Y %%%
notInList(G, X, [], X).

notInList(G, [], _, []) .

notInList(G, [X|T], Y, R) :-
inside(X, Y),
notInList(G, T, Y, R).

notInList(G, [X|T], Y, [X|R]) :-
notInside(G, X, Y),
notInList(G, T, Y, R).


awareList(_, [], _, []) .

awareList(G, [X|T], Visited, L) :-
inside(X, Visited),
awareList(G, T, Visited, L).

awareList(G, [X|T], Visited, [X|L]) :-
notInside(G, X, Visited),
addToTail([X], Visited, V),
inside(person(X, Z), G),
notInList(G, Z, V, R),
addToTail(R, T, TT),
awareList(G, TT, V, L).

awareList1(G, [X|T], Visited, L) :-
notInside(G, X, Visited),
addToTail([X], Visited, V),
inside(person(X, Z), G),
addToTail(Z, T, TT),
awareList(G, TT, V, L).

ignorant(G, X, Y) :-
different(G, X, Y),
awareList1(G, [X], [], L),
notInside(G, Y, L).

%%% level 3 %%%

subReplace([], K, []).

subReplace([X|T], K, [Y|R]) :-
inside(p(X, Y), K),
subReplace(T, K, R).

replace([], K, []).

replace([person(X, L)|T], K, [person(Y, Z)|R]) :-
inside(p(X, Y), K),
subReplace(L, K, Z),
replace(T, K, R).

sameList([X1|L1], L2) :-
inside(X1, L2),
sameList(L1, L2).

sameList([], _).

checkSame([], _).

checkSame([person(X, Z)|L1], L2) :-
inside(person(X, L), L2),
sameList(Z, L),
sameList(L, Z),
checkSame(L1, L2).

comprLen([], []).
comprLen([H1|T1], [H2|T2]) :- comprLen(T1, T2).

checkK([person(X, Z)|G], [p(X, _)|K]) :-
checkK(G, K).

checkK([], []).

same_world(G, H, K) :-
comprLen(G, H),
checkK(G, K),
replace(G, K, R),
checkSame(R, H),
checkSame(H, R).
```

% optional!
% different_world(G, H)