Intro to Logic Programming

Example 1

Question 1:Family tree

Using your set of facts of members in your family and the set of definitions you have defined above; do the following queries: spouse(X, Y)- find all couples who are s pous es in your family. uncle(X, yourself) - find all your uncles. Turn on the trace to check your definitions. Show the facts and definitions you defined, and the trace of the queries in your logbook.

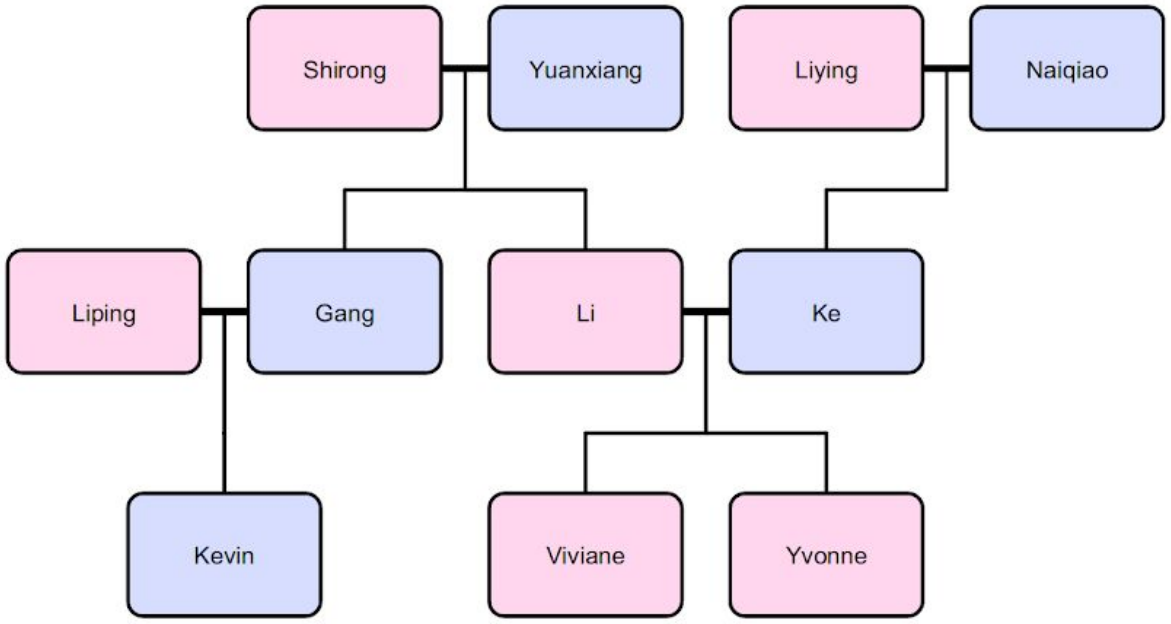

Figure 1: Viviane's Family

This is my family tree with 11 family members. The female family members are coloured in pink and the male members are colored in blue. The logic/relationship of my family will be now translated into prolog code. I tried to use as little facts as possible, since it is stated in the task that one would only need one fact per family entry. This is why I left out fact's like sister(viviane,yvonne), since it can be deduced from the fact female(viviane) and sister(yvonne,viviane).

Code	**Comments**

`female(li).`
`female(viviane).`
`female(yvonne).`
`female(liping).`
`female(shirong).`
`female(liying).`
	My mother
	Me
	My sister
	My aunt
	My grandmother
	My grandmother

`male(kevin).`	My cousin
`male(yuanxiang).`	My grandfather
`male(ke).`	My father
`male(naiqiao).`	My grandfather
`male(gang).`	My uncle
`sister(yvonne,viviane).`	yvonne is my sister.
`brother(gang,li).`	My uncle gang is the brother of my mother.
`parent(shirong,li).`	My mother's parents are shirong and yuanxiang.
`parent(yuanxiang,li).`	My parents are li and ke.
`parent(li,viviane).`	
`parent(ke,viviane).`	My father's parents are liying and naiqiao.
`parent(liying,ke).`	
`parent(naiqiao,ke).`	My cousin's parent's are liying and gang.
`parent(gang,kevin).`	
`parent(liping,kevin).`	

`parent_of(X,Y):-`
`  parent(X,Y);`
`  sibling(Y,Z),`
`  parent(X,Z).`
	I am differentiating between facts and relationship of parenthood. The relationship parent_of(X,Y) can be derived from the parents of one's siblings.

`father(X,Y):-`
`  parent_of(X,Y),`
`  male(X).`
	Your father is your male parent.

`mother(X,Y):-`
`  parent_of(X,Y),`
`  female(X).`
	Your mother is your female parent.

`son(X,Y):-`
`  parent_of(Y,X),`
`  male(X).`
	Your son is your male child.

`daugther(X,Y):-`
`  parent_of(Y,X),`
`  female(X).`
	Your daughter is your female child.

`grandfather(X,Y):-`
`  parent_of(X,Z),`
`  parent_of(Z,Y),`
`  male(X).`
	Your grandfather is the male parent of your parent.

`sibling(X,Y):-`
`  sister(X,Y);`
`  brother(X,Y);`
`  sister(Y,X);`
`  brother(Y,X).`
	Your sibling is either your brother/sister or you are the brother/sister of your sibling.

`aunt(X,Y):-`
`  sibling(X,Z),`
`  parent_of(Z,Y),`
`  female(X);`
`  spouse(X,T),`
`  sibling(T,G),`
`  parent_of(G,Y),`
`  female(X).`
	Your aunt is either a female sibling of one of your parents or the female spouse of your uncle, who is the sibling of one of your parents.

`uncle(X,Y):-`
`  sibling(X,Z),`
`  parent_of(Z,Y),`
`  male(X);`
`  spouse(X,T),`
`  sibling(T,G),`
`  parent_of(G,Y),`
`  male(X).`
	The same as aunt, just make him male.

`spouse(X,Y):-`
`  parent_of(X,Z),`
`  parent_of(Y,Z),`
`  X\==Y.`
	Your spouse is someone that has the same child as you, but is not you. Assuming all the married couples in your family have at least one child.

`cousin(X,Y):-`
`  parent_of(Z,X),`
`  uncle(Z,Y).`
	Your cousin is the son of your uncle. Assuming your aunt and uncle have not split up.

Running the query **spouse(X,Y)** results in:

Logbook	**Comments**

`?- spouse(X,Y).`
`X = shirong,`
`Y = yuanxiang ;`
`X = yuanxiang,`
`Y = shirong ;`
`X = li,`
`Y = ke ;`
`X = ke,`
`Y = li ;`
`X = liying,`
`Y = naiqiao ;`
`X = naiqiao,`
`Y = liying ;`
`X = gang,`
`Y = liping ;`
`X = liping,`
`Y = gang ;`
	The pair (shirong,yuanxing) and (ke,li) appear twice, since those couples have two kids and each child produces a different case where the query spouse(X,Y) is evaluated to true.

```
X = li,
Y = ke ;
X = ke,
Y = li ;
X = shirong,
Y = yuanxiang ;
X = yuanxiang,
Y = shirong ;
false.
```

Let's take a closer look with the trace function:

Trace output	**Comments**

`[trace]  ?- spouse(X,Y).`
`Call: (8) spouse(_3178, _3180) ? creep`
`Call: (9) parent_of(_3178, _3434) ? creep`
`Call: (10) parent(_3178, _3434) ? creep`
`Exit: (10) parent(shirong, li) ? creep`
`Exit: (9) parent_of(shirong, li) ? creep`
`Call: (9) parent_of(_3180, li) ? creep`
`Call: (10) parent(_3180, li) ? creep`
`Exit: (10) parent(shirong, li) ? creep`
`Exit: (9) parent_of(shirong, li) ? creep`
`Call: (9) shirong\==shirong ? creep`
`Fail: (9) shirong\==shirong ? creep`
`Redo: (10) parent(_3180, li) ? creep`
`Exit: (10) parent(yuanxiang, li) ? creep`
`Exit: (9) parent_of(yuanxiang, li) ? creep`
`Call: (9) shirong\==yuanxiang ? creep`
`Exit: (9) shirong\==yuanxiang ? creep`
`Exit: (8) spouse(shirong, yuanxiang) ? creep`
`X = shirong,`
`Y = yuanxiang ;`
`Redo: (9) parent_of(_3180, li) ? creep`
`Call: (10) sibling(li, _3434) ? creep`
`Call: (11) sister(li, _3434) ? creep`
`Fail: (11) sister(li, _3434) ? creep`
`Redo: (10) sibling(li, _3434) ? creep`
`Call: (11) brother(li, _3434) ? creep`
`Fail: (11) brother(li, _3434) ? creep`
`Redo: (10) sibling(li, _3434) ? creep`
`Call: (11) sister(_3432, li) ? creep`
`Fail: (11) sister(_3432, li) ? creep`
`Redo: (10) sibling(li, _3434) ? creep`
`Call: (11) brother(_3432, li) ? creep`
`Exit: (11) brother(gang, li) ? creep`
`Exit: (10) sibling(li, gang) ? creep`
`Call: (10) parent(_3180, gang) ? creep`
`Fail: (10) parent(_3180, gang) ? creep`
	Prolog will us a search algorithm and backtracking to find all the cases the query evaluates to true.
	
	For finding the spouse one must first find parents of the same child.
	
	spouse(shirong,shirong) results to false since it is the same parent/person.
	
	found another parent that is not shirong.
	shirong and yuanxiang are spouses.
	
	The second case, find parents of siblings of li.
	
	cannot find another sibling

Fail: (9) parent_of(_3180, li) ? creep	of li->quit.
Redo: (10) parent(_3178, _3434) ? creep	look for parent of gang.
Exit: (10) parent(yuanxiang, li) ? creep	
Exit: (9) parent_of(yuanxiang, li) ? creep	
Call: (9) parent_of(_3180, li) ? creep	
Call: (10) parent(_3180, li) ? creep	
Exit: (10) parent(shirong, li) ? creep	
Exit: (9) parent_of(shirong, li) ? creep	
Call: (9) yuanxiang\==shirong ? creep	
Exit: (9) yuanxiang\==shirong ? creep	
Exit: (8) spouse(yuanxiang, shirong) ? creep	
X = yuanxiang,	Found the same pair but the other way around.
Y = shirong ;	
Redo: (10) parent(_3180, li) ? creep	
Exit: (10) parent(yuanxiang, li) ? creep	
Exit: (9) parent_of(yuanxiang, li) ? creep	
Call: (9) yuanxiang\==yuanxiang ? creep	
Fail: (9) yuanxiang\==yuanxiang ? creep	
Redo: (9) parent_of(_3180, li) ? creep	
Call: (10) sibling(li, _3434) ? creep	
Call: (11) sister(li, _3434) ? creep	
Fail: (11) sister(li, _3434) ? creep	
Redo: (10) sibling(li, _3434) ? creep	
Call: (11) brother(li, _3434) ? creep	
Fail: (11) brother(li, _3434) ? creep	
Redo: (10) sibling(li, _3434) ? creep	
Call: (11) sister(_3432, li) ? creep	
Fail: (11) sister(_3432, li) ? creep	
Redo: (10) sibling(li, _3434) ? creep	
Call: (11) brother(_3432, li) ? creep	
Exit: (11) brother(gang, li) ? creep	
Exit: (10) sibling(li, gang) ? creep	
Call: (10) parent(_3180, gang) ? creep	
Fail: (10) parent(_3180, gang) ? creep	
Fail: (9) parent_of(_3180, li) ? creep	
Redo: (10) parent(_3178, _3434) ? creep	
Exit: (10) parent(li, viviane) ? creep	Search finished with li as a child.
Exit: (9) parent_of(li, viviane) ? creep	
Call: (9) parent_of(_3180, viviane) ? creep	li is the parent of viviane, find the second parent.
Call: (10) parent(_3180, viviane) ? creep	
Exit: (10) parent(li, viviane) ? creep	
Exit: (9) parent_of(li, viviane) ? creep	
Call: (9) li\==li ? creep	
Fail: (9) li\==li ? creep	
Redo: (10) parent(_3180, viviane) ? creep	
Exit: (10) parent(ke, viviane) ? creep	
Exit: (9) parent_of(ke, viviane) ? creep	
Call: (9) li\==ke ? creep	
Exit: (9) li\==ke ? creep	
Exit: (8) spouse(li, ke) ? creep	

X = li, Y = ke ; Redo: (9) parent_of(_3180, viviane) ? creep Call: (10) sibling(viviane, _3434) ? creep Call: (11) sister(viviane, _3434) ? creep Fail: (11) sister(viviane, _3434) ? creep Redo: (10) sibling(viviane, _3434) ? creep Call: (11) brother(viviane, _3434) ? creep Fail: (11) brother(viviane, _3434) ? creep Redo: (10) sibling(viviane, _3434) ? creep Call: (11) sister(_3432, viviane) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(viviane, yvonne) ? creep Call: (10) parent(_3180, yvonne) ? creep Fail: (10) parent(_3180, yvonne) ? creep Redo: (10) sibling(viviane, _3434) ? creep Call: (11) brother(_3432, viviane) ? creep Fail: (11) brother(_3432, viviane) ? creep Fail: (10) sibling(viviane, _3434) ? creep Fail: (9) parent_of(_3180, viviane) ? creep Redo: (10) parent(_3178, _3434) ? creep Exit: (10) parent(ke, viviane) ? creep Exit: (9) parent_of(ke, viviane) ? creep Call: (9) parent_of(_3180, viviane) ? creep Call: (10) parent(_3180, viviane) ? creep Exit: (10) parent(li, viviane) ? creep Exit: (9) parent_of(li, viviane) ? creep Call: (9) ke\==li ? creep Exit: (9) ke\==li ? creep Exit: (8) spouse(ke, li) ? creep	Second parent is ke, so ke is li's spouse. Find the parent of viviane's siblings now.
X = ke, Y = li ; Redo: (10) parent(_3180, viviane) ? creep Exit: (10) parent(ke, viviane) ? creep Exit: (9) parent_of(ke, viviane) ? creep Call: (9) ke\==ke ? creep Fail: (9) ke\==ke ? creep Redo: (9) parent_of(_3180, viviane) ? creep Call: (10) sibling(viviane, _3434) ? creep Call: (11) sister(viviane, _3434) ? creep Fail: (11) sister(viviane, _3434) ? creep Redo: (10) sibling(viviane, _3434) ? creep Call: (11) brother(viviane, _3434) ? creep Fail: (11) brother(viviane, _3434) ? creep Redo: (10) sibling(viviane, _3434) ? creep Call: (11) sister(_3432, viviane) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(viviane, yvonne) ? creep Call: (10) parent(_3180, yvonne) ? creep Fail: (10) parent(_3180, yvonne) ? creep	Found another pair!

Redo: (10) sibling(viviane, _3434) ? creep Call: (11) brother(_3432, viviane) ? creep Fail: (11) brother(_3432, viviane) ? creep Fail: (10) sibling(viviane, _3434) ? creep Fail: (9) parent_of(_3180, viviane) ? creep Redo: (10) parent(_3178, _3434) ? creep Exit: (10) parent(liying, ke) ? creep Exit: (9) parent_of(liying, ke) ? creep Call: (9) parent_of(_3180, ke) ? creep Call: (10) parent(_3180, ke) ? creep Exit: (10) parent(liying, ke) ? creep Exit: (9) parent_of(liying, ke) ? creep Call: (9) liying\==liying ? creep Fail: (9) liying\==liying ? creep Redo: (10) parent(_3180, ke) ? creep Exit: (10) parent(naiqiao, ke) ? creep Exit: (9) parent_of(naiqiao, ke) ? creep Call: (9) liying\==naiqiao ? creep Exit: (9) liying\==naiqiao ? creep Exit: (8) spouse(liying, naiqiao) ? creep X = liying, Y = naiqiao ; Redo: (9) parent_of(_3180, ke) ? creep Call: (10) sibling(ke, _3434) ? creep Call: (11) sister(ke, _3434) ? creep Fail: (11) sister(ke, _3434) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) brother(ke, _3434) ? creep Fail: (11) brother(ke, _3434) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) sister(_3432, ke) ? creep Fail: (11) sister(_3432, ke) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) brother(_3432, ke) ? creep Fail: (11) brother(_3432, ke) ? creep Fail: (10) sibling(ke, _3434) ? creep Fail: (9) parent_of(_3180, ke) ? creep Redo: (10) parent(_3178, _3434) ? creep Exit: (10) parent(naiqiao, ke) ? creep Exit: (9) parent_of(naiqiao, ke) ? creep Call: (9) parent_of(_3180, ke) ? creep Call: (10) parent(_3180, ke) ? creep Exit: (10) parent(liying, ke) ? creep Exit: (9) parent_of(liying, ke) ? creep Call: (9) naiqiao\==liying ? creep Exit: (9) naiqiao\==liying ? creep Exit: (8) spouse(naiqiao, liying) ? creep X = naiqiao, Y = liying Redo: (10) parent(_3180, ke) ? creep	Jump to finding parent of ke. naiqiao and liying are parents of ke. Since ke has no siblings, the second case fails. But taking the fact, naiqiao is the parent of ke into account, we find the

Exit: (10) parent(naiqiao, ke) ? creep Exit: (9) parent_of(naiqiao, ke) ? creep Call: (9) naiqiao\==naiqiao ? creep Fail: (9) naiqiao\==naiqiao ? creep Redo: (9) parent_of(_3180, ke) ? creep Call: (10) sibling(ke, _3434) ? creep Call: (11) sister(ke, _3434) ? creep Fail: (11) sister(ke, _3434) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) brother(ke, _3434) ? creep Fail: (11) brother(ke, _3434) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) sister(_3432, ke) ? creep Fail: (11) sister(_3432, ke) ? creep Redo: (10) sibling(ke, _3434) ? creep Call: (11) brother(_3432, ke) ? creep Fail: (11) brother(_3432, ke) ? creep Fail: (10) sibling(ke, _3434) ? creep Fail: (9) parent_of(_3180, ke) ? creep Redo: (10) parent(_3178, _3434) ? creep Exit: (10) parent(gang, kevin) ? creep Exit: (9) parent_of(gang, kevin) ? creep Call: (9) parent_of(_3180, kevin) ? creep Call: (10) parent(_3180, kevin) ? creep Exit: (10) parent(gang, kevin) ? creep Exit: (9) parent_of(gang, kevin) ? creep Call: (9) gang\==gang ? creep Fail: (9) gang\==gang ? creep Redo: (10) parent(_3180, kevin) ? creep Exit: (10) parent(liping, kevin) ? creep Exit: (9) parent_of(liping, kevin) ? creep Call: (9) gang\==liping ? creep Exit: (9) gang\==liping ? creep Exit: (8) spouse(gang, liping) ? creep X = gang, Y = liping ; Redo: (9) parent_of(_3180, kevin) ? creep Call: (10) sibling(kevin, _3434) ? creep Call: (11) sister(kevin, _3434) ? creep Fail: (11) sister(kevin, _3434) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) brother(kevin, _3434) ? creep Fail: (11) brother(kevin, _3434) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) sister(_3432, kevin) ? creep Fail: (11) sister(_3432, kevin) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) brother(_3432, kevin) ? creep Fail: (11) brother(_3432, kevin) ? creep Fail: (10) sibling(kevin, _3434) ? creep	commutative second pair (naiqiao, liying) Again failing to find siblings of ke. Move to kevin, gang is his parent. so is liping. Found the pair (gang,liping) Fails to find siblings since kevin is a only child.

Fail: (9) parent_of(_3180, kevin) ? creep Redo: (10) parent(_3178, _3434) ? creep Exit: (10) parent(liping, kevin) ? creep Exit: (9) parent_of(liping, kevin) ? creep Call: (9) parent_of(_3180, kevin) ? creep Call: (10) parent(_3180, kevin) ? creep Exit: (10) parent(gang, kevin) ? creep Exit: (9) parent_of(gang, kevin) ? creep Call: (9) liping\==gang ? creep Exit: (9) liping\==gang ? creep Exit: (8) spouse(liping, gang) ? creep X = liping, Y = gang ;	
Redo: (10) parent(_3180, kevin) ? creep Exit: (10) parent(liping, kevin) ? creep Exit: (9) parent_of(liping, kevin) ? creep Call: (9) liping\==liping ? creep Fail: (9) liping\==liping ? creep	liping is parent of kevin.
Redo: (9) parent_of(_3180, kevin) ? creep Call: (10) sibling(kevin, _3434) ? creep Call: (11) sister(kevin, _3434) ? creep Fail: (11) sister(kevin, _3434) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) brother(kevin, _3434) ? creep Fail: (11) brother(kevin, _3434) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) sister(_3432, kevin) ? creep Fail: (11) sister(_3432, kevin) ? creep Redo: (10) sibling(kevin, _3434) ? creep Call: (11) brother(_3432, kevin) ? creep Fail: (11) brother(_3432, kevin) ? creep Fail: (10) sibling(kevin, _3434) ? creep Fail: (9) parent_of(_3180, kevin) ? creep Redo: (9) parent_of(_3178, _3434) ? creep Call: (10) sibling(_3432, _3434) ? creep Call: (11) sister(_3432, _3434) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(yvonne, viviane) ? creep Call: (10) parent(_3178, viviane) ? creep Exit: (10) parent(li, viviane) ? creep Exit: (9) parent_of(li, yvonne) ? creep Call: (9) parent_of(_3180, yvonne) ? creep Call: (10) parent(_3180, yvonne) ? creep Fail: (10) parent(_3180, yvonne) ? creep Redo: (9) parent_of(_3180, yvonne) ? creep Call: (10) sibling(yvonne, _3434) ? creep Call: (11) sister(yvonne, _3434) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(yvonne, viviane) ? creep Call: (10) parent(_3180, viviane) ? creep	Done this case already. Fails to find sibling again. Look at yvonne, found sibling viviane.

Exit: (10) parent(li, viviane) ? creep Exit: (9) parent_of(li, yvonne) ? creep Call: (9) li\==li ? creep Fail: (9) li\==li ? creep Redo: (10) parent(_3180, viviane) ? creep Exit: (10) parent(ke, viviane) ? creep Exit: (9) parent_of(ke, yvonne) ? creep Call: (9) li\==ke ? creep Exit: (9) li\==ke ? creep Exit: (8) spouse(li, ke) ? creep X = li, Y = ke ; Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) brother(yvonne, _3434) ? creep Fail: (11) brother(yvonne, _3434) ? creep Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) sister(_3432, yvonne) ? creep Fail: (11) sister(_3432, yvonne) ? creep Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) brother(_3432, yvonne) ? creep Fail: (11) brother(_3432, yvonne) ? creep Fail: (10) sibling(yvonne, _3434) ? creep Fail: (9) parent_of(_3180, yvonne) ? creep Redo: (10) parent(_3178, viviane) ? creep Exit: (10) parent(ke, viviane) ? creep Exit: (9) parent_of(ke, yvonne) ? creep Call: (9) parent_of(_3180, yvonne) ? creep Call: (10) parent(_3180, yvonne) ? creep Fail: (10) parent(_3180, yvonne) ? creep Redo: (9) parent_of(_3180, yvonne) ? creep Call: (10) sibling(yvonne, _3434) ? creep Call: (11) sister(yvonne, _3434) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(yvonne, viviane) ? creep Call: (10) parent(_3180, viviane) ? creep Exit: (10) parent(li, viviane) ? creep Exit: (9) parent_of(li, yvonne) ? creep Call: (9) ke\==li ? creep Exit: (9) ke\==li ? creep Exit: (8) spouse(ke, li) ? creep X = ke, Y = li ; Redo: (10) parent(_3180, viviane) ? creep Exit: (10) parent(ke, viviane) ? creep Exit: (9) parent_of(ke, yvonne) ? creep Call: (9) ke\==ke ? creep Fail: (9) ke\==ke ? creep Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) brother(yvonne, _3434) ? creep Fail: (11) brother(yvonne, _3434) ? creep	yvonne's parent's are li and ke who are spouses. Same applies to the other way around. Have found all siblings of yvonne.

Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) sister(_3432, yvonne) ? creep Fail: (11) sister(_3432, yvonne) ? creep Redo: (10) sibling(yvonne, _3434) ? creep Call: (11) brother(_3432, yvonne) ? creep Fail: (11) brother(_3432, yvonne) ? creep Fail: (10) sibling(yvonne, _3434) ? creep Fail: (9) parent_of(_3180, yvonne) ? creep Redo: (10) sibling(_3432, _3434) ? creep Call: (11) brother(_3432, _3434) ? creep Exit: (11) brother(gang, li) ? creep Exit: (10) sibling(gang, li) ? creep Call: (10) parent(_3178, li) ? creep Exit: (10) parent(shirong, li) ? creep Exit: (9) parent_of(shirong, gang) ? creep Call: (9) parent_of(_3180, gang) ? creep Call: (10) parent(_3180, gang) ? creep Fail: (10) parent(_3180, gang) ? creep Redo: (9) parent_of(_3180, gang) ? creep Call: (10) sibling(gang, _3434) ? creep Call: (11) sister(gang, _3434) ? creep Fail: (11) sister(gang, _3434) ? creep Redo: (10) sibling(gang, _3434) ? creep Call: (11) brother(gang, _3434) ? creep Exit: (11) brother(gang, li) ? creep Exit: (10) sibling(gang, li) ? creep Call: (10) parent(_3180, li) ? creep Exit: (10) parent(shirong, li) ? creep Exit: (9) parent_of(shirong, gang) ? creep Call: (9) shirong\==shirong ? creep Fail: (9) shirong\==shirong ? creep Redo: (10) parent(_3180, li) ? creep Exit: (10) parent(yuanxiang, li) ? creep Exit: (9) parent_of(yuanxiang, gang) ? creep Call: (9) shirong\==yuanxiang ? creep Exit: (9) shirong\==yuanxiang ? creep Exit: (8) spouse(shirong, yuanxiang) ? creep X = shirong, Y = yuanxiang ; Redo: (10) sibling(gang, _3434) ? creep Call: (11) sister(_3432, gang) ? creep Fail: (11) sister(_3432, gang) ? creep Redo: (10) sibling(gang, _3434) ? creep Call: (11) brother(_3432, gang) ? creep Fail: (11) brother(_3432, gang) ? creep Fail: (10) sibling(gang, _3434) ? creep Fail: (9) parent_of(_3180, gang) ? creep Redo: (10) parent(_3178, li) ? creep Exit: (10) parent(yuanxiang, li) ? creep Exit: (9) parent_of(yuanxiang, gang) ? creep	Next is gang, he has a sibling li. His parent's are shirong and yuanxiang, therefore this pair are spouses.

```
   Call: (9) parent_of(_3180, gang) ? creep
   Call: (10) parent(_3180, gang) ? creep
   Fail: (10) parent(_3180, gang) ? creep
   Redo: (9) parent_of(_3180, gang) ? creep
   Call: (10) sibling(gang, _3434) ? creep
   Call: (11) sister(gang, _3434) ? creep
   Fail: (11) sister(gang, _3434) ? creep
   Redo: (10) sibling(gang, _3434) ? creep
   Call: (11) brother(gang, _3434) ? creep
   Exit: (11) brother(gang, li) ? creep
   Exit: (10) sibling(gang, li) ? creep
   Call: (10) parent(_3180, li) ? creep
   Exit: (10) parent(shirong, li) ? creep
   Exit: (9) parent_of(shirong, gang) ? creep
   Call: (9) yuanxiang\==shirong ? creep
   Exit: (9) yuanxiang\==shirong ? creep
   Exit: (8) spouse(yuanxiang, shirong) ? creep
X = yuanxiang,
Y = shirong ;
   Redo: (10) parent(_3180, li) ? creep
   Exit: (10) parent(yuanxiang, li) ? creep
   Exit: (9) parent_of(yuanxiang, gang) ? creep
   Call: (9) yuanxiang\==yuanxiang ? creep
   Fail: (9) yuanxiang\==yuanxiang ? creep
   Redo: (10) sibling(gang, _3434) ? creep
   Call: (11) sister(_3432, gang) ? creep
   Fail: (11) sister(_3432, gang) ? creep
   Redo: (10) sibling(gang, _3434) ? creep
   Call: (11) brother(_3432, gang) ? creep
   Fail: (11) brother(_3432, gang) ? creep
   Fail: (10) sibling(gang, _3434) ? creep
   Fail: (9) parent_of(_3180, gang) ? creep
   Redo: (10) sibling(_3432, _3434) ? creep
   Call: (11) sister(_3432, _3434) ? creep
   Exit: (11) sister(yvonne, viviane) ? creep
   Exit: (10) sibling(viviane, yvonne) ? creep
   Call: (10) parent(_3178, yvonne) ? creep
   Fail: (10) parent(_3178, yvonne) ? creep
   Redo: (10) sibling(_3432, _3434) ? creep
   Call: (11) brother(_3432, _3434) ? creep
   Exit: (11) brother(gang, li) ? creep
   Exit: (10) sibling(li, gang) ? creep
   Call: (10) parent(_3178, gang) ? creep
   Fail: (10) parent(_3178, gang) ? creep
   Fail: (9) parent_of(_3178, _3434) ? creep
   Fail: (8) spouse(_3178, _3180) ? creep
false.
```

The other way around, spouse(yuanxiang,shirong) also returns true.

We have found every pair of spouses. So the search algorithm will not be able to find any more pairs. Everything will fail.

Next, we look at the query **`uncle(X,me)`** where me is viviane:

Query Output	Comments
?- uncle(X,viviane). X = gang ; false.	I have only one uncle - gang. No one else fulfills the uncle conditions.

Again, we look a closer look at the search and backtracking algorithm of prolog in trace mode.

Trace	Comments
[trace] ?- uncle(X,viviane). Call: (8) uncle(_4248, viviane) ? creep Call: (9) sibling(_4248, _4468) ? creep Call: (10) sister(_4248, _4468) ? creep Exit: (10) sister(yvonne, viviane) ? creep Exit: (9) sibling(yvonne, viviane) ? creep Call: (9) parent_of(viviane, viviane) ? creep Call: (10) parent(viviane, viviane) ? creep Fail: (10) parent(viviane, viviane) ? creep Redo: (9) parent_of(viviane, viviane) ? creep Call: (10) sibling(viviane, _4468) ? creep Call: (11) sister(viviane, _4468) ? creep Fail: (11) sister(viviane, _4468) ? creep Redo: (10) sibling(viviane, _4468) ? creep Call: (11) brother(viviane, _4468) ? creep Fail: (11) brother(viviane, _4468) ? creep Redo: (10) sibling(viviane, _4468) ? creep Call: (11) sister(_4466, viviane) ? creep Exit: (11) sister(yvonne, viviane) ? creep Exit: (10) sibling(viviane, yvonne) ? creep Call: (10) parent(viviane, yvonne) ? creep Fail: (10) parent(viviane, yvonne) ? creep Redo: (10) sibling(viviane, _4468) ? creep Call: (11) brother(_4466, viviane) ? creep Fail: (11) brother(_4466, viviane) ? creep Fail: (10) sibling(viviane, _4468) ? creep Fail: (9) parent_of(viviane, viviane) ? creep Redo: (9) sibling(_4248, _4468) ? creep Call: (10) brother(_4248, _4468) ? creep Exit: (10) brother(gang, li) ? creep Exit: (9) sibling(gang, li) ? creep Call: (9) parent_of(li, viviane) ? creep Call: (10) parent(li, viviane) ? creep Exit: (10) parent(li, viviane) ? creep Exit: (9) parent_of(li, viviane) ? creep Call: (9) male(gang) ? creep Exit: (9) male(gang) ? creep Exit: (8) uncle(gang, viviane) ? creep X = gang ; Redo: (10) parent(li, viviane) ? creep	yvonne is sister of viviane, so she can't be the uncle. Her sister is viviane herself, but viviane is not her own parent. viviane has no brother. Found brother of li so gang is also the sibling of li. li is the parent of viviane, so the first case evaluates to true, the uncle is the male sibling of one of viviane's parents.

```
Fail: (10) parent(li, viviane) ? creep
Redo: (9) parent_of(li, viviane) ? creep
Call: (10) sibling(viviane, _4468) ? creep
Call: (11) sister(viviane, _4468) ? creep
Fail: (11) sister(viviane, _4468) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) brother(viviane, _4468) ? creep
Fail: (11) brother(viviane, _4468) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) sister(_4466, viviane) ? creep
Exit: (11) sister(yvonne, viviane) ? creep
Exit: (10) sibling(viviane, yvonne) ? creep
Call: (10) parent(li, yvonne) ? creep
Fail: (10) parent(li, yvonne) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) brother(_4466, viviane) ? creep
Fail: (11) brother(_4466, viviane) ? creep
Fail: (10) sibling(viviane, _4468) ? creep
Fail: (9) parent_of(li, viviane) ? creep
Redo: (9) sibling(_4248, _4468) ? creep
Call: (10) sister(_4466, _4248) ? creep
Exit: (10) sister(yvonne, viviane) ? creep
Exit: (9) sibling(viviane, yvonne) ? creep
Call: (9) parent_of(yvonne, viviane) ? creep
Call: (10) parent(yvonne, viviane) ? creep
Fail: (10) parent(yvonne, viviane) ? creep
Redo: (9) parent_of(yvonne, viviane) ? creep
Call: (10) sibling(viviane, _4468) ? creep
Call: (11) sister(viviane, _4468) ? creep
Fail: (11) sister(viviane, _4468) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) brother(viviane, _4468) ? creep
Fail: (11) brother(viviane, _4468) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) sister(_4466, viviane) ? creep
Exit: (11) sister(yvonne, viviane) ? creep
Exit: (10) sibling(viviane, yvonne) ? creep
Call: (10) parent(yvonne, yvonne) ? creep
Fail: (10) parent(yvonne, yvonne) ? creep
Redo: (10) sibling(viviane, _4468) ? creep
Call: (11) brother(_4466, viviane) ? creep
Fail: (11) brother(_4466, viviane) ? creep
Fail: (10) sibling(viviane, _4468) ? creep
Fail: (9) parent_of(yvonne, viviane) ? creep
Redo: (9) sibling(_4248, _4468) ? creep
Call: (10) brother(_4466, _4248) ? creep
Exit: (10) brother(gang, li) ? creep
Exit: (9) sibling(li, gang) ? creep
Call: (9) parent_of(gang, viviane) ? creep
Call: (10) parent(gang, viviane) ? creep
```

Searches through family tree but there is no one who fulfills the conditions to be viviane's uncle other than gang. → Fail

```
   Fail: (10) parent(gang, viviane) ? creep
   Redo: (9) parent_of(gang, viviane) ? creep
   Call: (10) sibling(viviane, _4468) ? creep
   Call: (11) sister(viviane, _4468) ? creep
   Fail: (11) sister(viviane, _4468) ? creep
   Redo: (10) sibling(viviane, _4468) ? creep
   Call: (11) brother(viviane, _4468) ? creep
   Fail: (11) brother(viviane, _4468) ? creep
   Redo: (10) sibling(viviane, _4468) ? creep
   Call: (11) sister(_4466, viviane) ? creep
   Exit: (11) sister(yvonne, viviane) ? creep
   Exit: (10) sibling(viviane, yvonne) ? creep
   Call: (10) parent(gang, yvonne) ? creep
   Fail: (10) parent(gang, yvonne) ? creep
   Redo: (10) sibling(viviane, _4468) ? creep
   Call: (11) brother(_4466, viviane) ? creep
   Fail: (11) brother(_4466, viviane) ? creep
   Fail: (10) sibling(viviane, _4468) ? creep
   Fail: (9) parent_of(gang, viviane) ? creep
   Fail: (8) uncle(_4248, viviane) ? creep
false.
```

Question 2: Family.pl

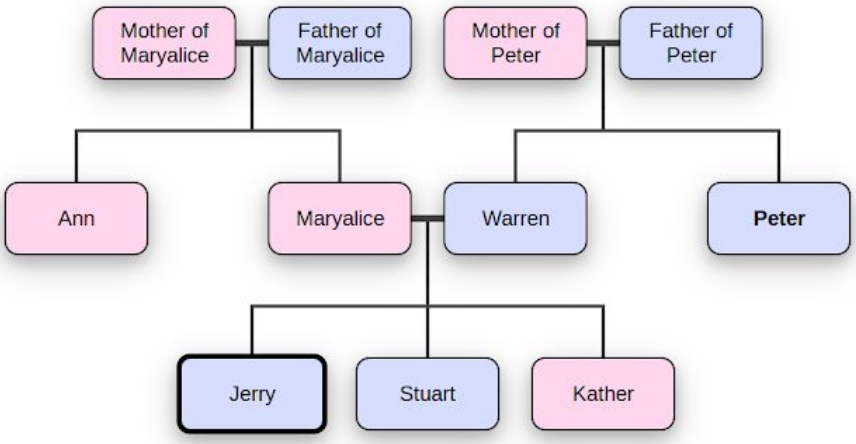

Figure 2: Family tree of the family in family.pl

Using the definitions you have created and the family facts below (given in family.pl), make a query ?- **`parent_of(X, Y).`**

Code	**Query parent_of(X,Y)**

`male(jerry).`
`male(stuart).`
`male(warren).`
`male(peter).`
`female(kather).`
`female(maryalice).`
`female(ann).`
`brother(jerry,stuart).`
`brother(jerry,kather).`
`brother(peter, warren).`
`sister(ann, maryalice).`
`sister(kather,jerry).`
`parent(warren,jerry).`
`parent(maryalice,jerry).`
 	
`parent_of(X,Y):-`
`parent(X,Y);`
`sibling(Y,Z),`
`parent(X,Z).`
 	
`sibling(X,Y):-`
`sister(X,Y);`
`brother(X,Y);`
`sister(Y,X);`
	`?- parent_of(X,Y).`
	`X = warren,`
	`Y = jerry ;`
	`X = maryalice,`
	`Y = jerry ;`
	`X = warren,`
	`Y = kather ;`
	`X = maryalice,`
	`Y = kather ;`
	`X = warren,`
	`Y = stuart ;`
	`X = maryalice,`
	`Y = stuart ;`
	`X = warren,`
	`Y = kather ;`
	`X = maryalice,`
	`Y = kather ;`
	`false.`

`brother(Y,X).`	

Code	Comments

`?- parent_of(X,Y).`
`X = warren,`
`Y = jerry ;`
`X = maryalice,`
`Y = jerry ;`
`X = warren,`
`Y = kather ;`
`X = maryalice,`
`Y = kather ;`
`X = warren,`
`Y = stuart ;`
`X = maryalice,`
`Y = stuart ;`
`X = warren,`
`Y = kather ;`
`X = maryalice,`
`Y = kather ;`
`false.`
	Using the same parent_of(X,Y) and sibling(X,Y) definition as in Question 1 and changing the parent_of(warren,jerry) and parent_of(maryalice,jerry) to facts: parent(warren,jerry) and parent(maryalice,jerry), we get this output for the query parent_of(x,y).

a) Do a trace of the matching process in the AND-OR tree to show the search process.

I have specified the first argument to be warren, so that the tree does not grow too large. For the general query parent_of(X,Y), one would need to check all cases for X. I think checking parent_of(warren,X) is sufficient for getting a general understanding about how to draw a decision tree and how the search algorithm works.

```
parent_of(warren,X)
      parent(warren,X)
            X=jerry
                  parent(warren,jerry)
                        true, X = jerry.
      sibling(X,Z),parent(warren,Z)
            sister(X,Z),parent(warren,Z)
                  X=ann,Z=maryalice
                        parent(warren,maryalice)
                                    false.
                  X=kather,Z=jerry
                        parent(warren,jerry)
                                    true, X = kather.
            brother(X,Z),parent(warren,Z)
                  X=jerry,Z=stuart
                        parent(warren,stuart)
```

```
                        false.
      X=jerry,Z=kather
            parent(warren,kather)
                        false.
      X=peter,Z=warren
            parent(warren,warren)
                        false.
sister(Z,X),parent(warren,Z)
      X=maryalice,Z=ann
            parent(warren,ann)
                        false.
      X=jerry,Z=kather
            parent(warren,kather)
                        false.
brother(Z,X),parent(warren,Z)
      X=stuart,Z=jerry
            parent(warren,jerry)
                        true, X = stuart.
      X=kather,Z=jerry
            parent(warren,jerry)
                        true, X = kather.
      X=warren,Z=peter
            parent(warren,peter)
                        false.
```

b) Reorder the facts in the following way: 10, 9, 8, 2, 4, 3, 1, 5, 7, 6, 12 , 11, 14 , 13 . Repeat the query and do a trace on the new query. Are query results the same? Are the traces identical? If not, explain why.

All of the result pair (X,Y) that appear in the version without reordering also appear in the version with reordering. However, there is no fail at the end. Since we have modified the order of the knowledge base, the order search tree or elements that are traversed during the search has changed.

Without reordering Case 1	**Number of solution**	**With reordering Case 2**	**Number of solution**
?- parent_of(X,Y). X = warren, Y = jerry ; X = maryalice, Y = jerry ; X = warren, Y = kather ; X = maryalice, Y = kather ; X = warren, Y = stuart ; X = maryalice,	 (1) (2) (3)/(7) (4)/(8) (5) (6)	?- parent_of(X,Y). X = maryalice, Y = jerry ; X = warren, Y = jerry ; X = maryalice, Y = kather ; X = warren, Y = kather ; X = maryalice, Y = kather ; X = warren,	 (2) (1) (4)/(8) (3)/(7) (4)/(8) (3)/(7)

Y = stuart ; X = warren, Y = kather ; X = maryalice, Y = kather ; false.	(3)/(7) (4)/(8)	Y = kather ; X = maryalice, Y = stuart ; X = warren, Y = stuart.	(6) (5)

Without reordering **Case 1**	**With reordering** **Case 2**
[trace] ?- parent_of(X,Y). Call: (8) parent_of(_3060, _3062) ? creep Call: (9) parent(_3060, _3062) ? creep Exit: (9) parent(warren, jerry) ? creep Exit: (8) parent_of(warren, jerry) ? creep X = warren, Y = jerry ; Redo: (9) parent(_3060, _3062) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, jerry) ? creep X = maryalice, Y = jerry ; Redo: (8) parent_of(_3060, _3062) ? creep Call: (9) sibling(_3062, _3316) ? creep Call: (10) sister(_3062, _3316) ? creep Exit: (10) sister(ann, maryalice) ? creep Exit: (9) sibling(ann, maryalice) ? creep Call: (9) parent(_3060, maryalice) ? creep Fail: (9) parent(_3060, maryalice) ? creep Redo: (10) sister(_3062, _3316) ? creep Exit: (10) sister(kather, jerry) ? creep Exit: (9) sibling(kather, jerry) ? creep Call: (9) parent(_3060, jerry) ? creep Exit: (9) parent(warren, jerry) ? creep Exit: (8) parent_of(warren, kather) ? creep X = warren, Y = kather ; Redo: (9) parent(_3060, jerry) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, kather) ? creep X = maryalice, Y = kather ; Redo: (9) sibling(_3062, _3316) ? creep Call: (10) brother(_3062, _3316) ? creep Exit: (10) brother(jerry, stuart) ? creep Exit: (9) sibling(jerry, stuart) ? creep Call: (9) parent(_3060, stuart) ? creep Fail: (9) parent(_3060, stuart) ? creep	[trace] ?- parent_of(X,Y). Call: (8) parent_of(_3060, _3062) ? creep Call: (9) parent(_3060, _3062) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, jerry) ? creep X = maryalice, Y = jerry Unknown action: M (h for help) Action? . [trace] ?- parent_of(X,Y). Call: (8) parent_of(_3060, _3062) ? creep Call: (9) parent(_3060, _3062) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, jerry) ? creep X = maryalice, Y = jerry ; Redo: (9) parent(_3060, _3062) ? creep Exit: (9) parent(warren, jerry) ? creep Exit: (8) parent_of(warren, jerry) ? creep X = warren, Y = jerry ; Redo: (8) parent_of(_3060, _3062) ? creep Call: (9) sibling(_3062, _3312) ? creep Call: (10) sister(_3062, _3312) ? creep Exit: (10) sister(kather, jerry) ? creep Exit: (9) sibling(kather, jerry) ? creep Call: (9) parent(_3060, jerry) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, kather) ? creep X = maryalice, Y = kather ; Redo: (9) parent(_3060, jerry) ? creep Exit: (9) parent(warren, jerry) ? creep Exit: (8) parent_of(warren, kather) ? creep X = warren, Y = kather ; Redo: (10) sister(_3062, _3312) ? creep

```
   Redo: (10) brother(_3062, _3316) ? creep
   Exit: (10) brother(jerry, kather) ? creep
   Exit: (9) sibling(jerry, kather) ? creep
   Call: (9) parent(_3060, kather) ? creep
   Fail: (9) parent(_3060, kather) ? creep
   Redo: (10) brother(_3062, _3316) ? creep
   Exit: (10) brother(peter, warren) ? creep
   Exit: (9) sibling(peter, warren) ? creep
   Call: (9) parent(_3060, warren) ? creep
   Fail: (9) parent(_3060, warren) ? creep
   Redo: (9) sibling(_3062, _3316) ? creep
   Call: (10) sister(_3314, _3062) ? creep
   Exit: (10) sister(ann, maryalice) ? creep
   Exit: (9) sibling(maryalice, ann) ? creep
   Call: (9) parent(_3060, ann) ? creep
   Fail: (9) parent(_3060, ann) ? creep
   Redo: (10) sister(_3314, _3062) ? creep
   Exit: (10) sister(kather, jerry) ? creep
   Exit: (9) sibling(jerry, kather) ? creep
   Call: (9) parent(_3060, kather) ? creep
   Fail: (9) parent(_3060, kather) ? creep
   Redo: (9) sibling(_3062, _3316) ? creep
   Call: (10) brother(_3314, _3062) ? creep
   Exit: (10) brother(jerry, stuart) ? creep
   Exit: (9) sibling(stuart, jerry) ? creep
   Call: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(warren, jerry) ? creep
   Exit: (8) parent_of(warren, stuart) ? creep
X = warren,
Y = stuart ;
   Redo: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(maryalice, jerry) ? creep
   Exit: (8) parent_of(maryalice, stuart) ? creep
X = maryalice,
Y = stuart ;
   Redo: (10) brother(_3314, _3062) ? creep
   Exit: (10) brother(jerry, kather) ? creep
   Exit: (9) sibling(kather, jerry) ? creep
   Call: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(warren, jerry) ? creep
   Exit: (8) parent_of(warren, kather) ? creep
X = warren,
Y = kather ;
   Redo: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(maryalice, jerry) ? creep
   Exit: (8) parent_of(maryalice, kather) ? creep
X = maryalice,
Y = kather ;
   Redo: (10) brother(_3314, _3062) ? creep
   Exit: (10) brother(peter, warren) ? creep
```

```
   Exit: (10) sister(ann, maryalice) ? creep
   Exit: (9) sibling(ann, maryalice) ? creep
   Call: (9) parent(_3060, maryalice) ? creep
   Fail: (9) parent(_3060, maryalice) ? creep
   Redo: (9) sibling(_3062, _3312) ? creep
   Call: (10) brother(_3062, _3312) ? creep
   Exit: (10) brother(peter, warren) ? creep
   Exit: (9) sibling(peter, warren) ? creep
   Call: (9) parent(_3060, warren) ? creep
   Fail: (9) parent(_3060, warren) ? creep
   Redo: (10) brother(_3062, _3312) ? creep
   Exit: (10) brother(jerry, kather) ? creep
   Exit: (9) sibling(jerry, kather) ? creep
   Call: (9) parent(_3060, kather) ? creep
   Fail: (9) parent(_3060, kather) ? creep
   Redo: (10) brother(_3062, _3312) ? creep
   Exit: (10) brother(jerry, stuart) ? creep
   Exit: (9) sibling(jerry, stuart) ? creep
   Call: (9) parent(_3060, stuart) ? creep
   Fail: (9) parent(_3060, stuart) ? creep
   Redo: (9) sibling(_3062, _3312) ? creep
   Call: (10) sister(_3310, _3062) ? creep
   Exit: (10) sister(kather, jerry) ? creep
   Exit: (9) sibling(jerry, kather) ? creep
   Call: (9) parent(_3060, kather) ? creep
   Fail: (9) parent(_3060, kather) ? creep
   Redo: (10) sister(_3310, _3062) ? creep
   Exit: (10) sister(ann, maryalice) ? creep
   Exit: (9) sibling(maryalice, ann) ? creep
   Call: (9) parent(_3060, ann) ? creep
   Fail: (9) parent(_3060, ann) ? creep
   Redo: (9) sibling(_3062, _3312) ? creep
   Call: (10) brother(_3310, _3062) ? creep
   Exit: (10) brother(peter, warren) ? creep
   Exit: (9) sibling(warren, peter) ? creep
   Call: (9) parent(_3060, peter) ? creep
   Fail: (9) parent(_3060, peter) ? creep
   Redo: (10) brother(_3310, _3062) ? creep
   Exit: (10) brother(jerry, kather) ? creep
   Exit: (9) sibling(kather, jerry) ? creep
   Call: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(maryalice, jerry) ? creep
   Exit: (8) parent_of(maryalice, kather) ? creep
X = maryalice,
Y = kather ;
   Redo: (9) parent(_3060, jerry) ? creep
   Exit: (9) parent(warren, jerry) ? creep
   Exit: (8) parent_of(warren, kather) ? creep
X = warren,
Y = kather ;
```

Exit: (9) sibling(warren, peter) ? creep Call: (9) parent(_3060, peter) ? creep Fail: (9) parent(_3060, peter) ? creep Fail: (8) parent_of(_3060, _3062) ? creep false.	Redo: (10) brother(_3310, _3062) ? creep Exit: (10) brother(jerry, stuart) ? creep Exit: (9) sibling(stuart, jerry) ? creep Call: (9) parent(_3060, jerry) ? creep Exit: (9) parent(maryalice, jerry) ? creep Exit: (8) parent_of(maryalice, stuart) ? creep X = maryalice, Y = stuart ; Redo: (9) parent(_3060, jerry) ? creep Exit: (9) parent(warren, jerry) ? creep Exit: (8) parent_of(warren, stuart) ? creep X = warren, Y = stuart.

By comparing the length of the trace, we can observe that the trace output of the version without reordering is shorter. The reordering is the reason that the solution are found later than in the case without reordering. The first version could be pruned, whereas in the second version the algorithm has to go through every possibility, which is why it takes longer. With pruning I mean that if a Fail appears sooner, you would not have to check the conditions that are followed by the one we are checking, due to the And-Introduction.

Also, by reordering the conditions that need to be checked can have an impact on the computation time. For example, if we would rewrite the definition of parent_of(X,Y) to:

```
parent_of(X,Y):-
      parent(X,Y);
      parent(X,Z),
      sibling(Y,Z).
```

the computation time would improve, since it is easier to check parent(X,Z) than sibling(Y,Z). It is easier, since parent(X,Y) is a fact that is directly in the KB, whereas for sibling(X,Y) one would have to check all four cases sister(X,Y), brother(X,Y), sister(Y,X), brother(Y,X). We can therefore prune earlier in the search and check less conditions. The trace output has 69 lines of trace output compared to 93 lines in Case 1 and 101 lines in Case 2.

Prolog Declarative Knowledge Assignment

Exercise 1

Question 1.1

Translate the natural language statements above describing the dealing within the Smartphone industry in to First Order Logic, (FOL).

Natural Language

SumSum, a competitor of Appy, developed some nice smart phone technology called Galactica-S3, all of which was stolen by Stevey, who is a Boss. It is unethical for a Boss to steal business from rival companies. A competitor of Appy is a rival. Smartphone technology is a business.

Terms

smartphonetech(x) = {x is a smartphone technology.}
boss(b) = {b is a boss.}
rival(r) = {r is a rival.}
business(b) = {b is a business}
competitor(x,y) = {x is a competitor of y.}
steals(x,y,z) = {x steals y from z.}
developed(t,d) = {Technology t was developed by developer d.}
unethical(x) = {x is unethical.}

Knowledge Base

boss(Stevey)
smartphonetech(Galactica-S3)
steals(Stevey,Galactica-S3,SumSum)
competitor(SumSum, Appy)
developed(Galactica-S3,SumSum)

Rules

$\forall x, \forall y, \forall z \;\; boss(x) \wedge steals(x,y,z) \wedge business(y) \wedge rival(z) \Rightarrow unethical(x)$
$\forall x \; competitor(x, Appy) \Rightarrow rival(x)$
$\forall x \; smartphonetech(x) \Rightarrow business(x)$

Question 1.2

Write these FOL statements as Prolog clauses.

Prolog

Code	Comments
boss(s). smartphonetech(s3). competitor(ss,a). developed(s3,ss). steals(s,s3,ss).	KB with facts written in a similar way as in Section Knowledge Base. s refers to Stevey, s3 = Galactica-S3, ss = SumSum, a = Appy
unethical(X):- boss(X), business(Y), rival(Z), steals(X,Y,Z).	Someone is unethical if he is a boss stealing a business from a rival.
rival(X):- competitor(X,a).	A rival is a competitor of Appy.
business(X):- smartphonetech(X).	All smartphone technologies are businesses.

Question 1.3

Using the prolog search engine, prove that Stevey is unethical. Show a trace of your proof.

Trace	Comment
?- unethical(s). true.	Query: Is Stevey unethical? → Yes he is!
?- trace. true.	Turn on trace mode.
[trace] ?- unethical(s). Call: (8) unethical(s) ? creep Call: (9) boss(s) ? creep Exit: (9) boss(s) ? creep Call: (9) business(_3236) ? creep Call: (10) smartphonetech(_3236) ? creep Exit: (10) smartphonetech(s3) ? creep Exit: (9) business(s3) ? creep Call: (9) rival(_3236) ? creep Call: (10) competitor(_3236, a) ? creep Exit: (10) competitor(ss, a) ? creep Exit: (9) rival(ss) ? creep Call: (9) steals(s, s3, ss) ? creep Exit: (9) steals(s, s3, ss) ?	Need to check whether s is a boss first. Yes he is, it's a fact in the KB. Second condition, is there a business? Since no business in KB, find smartphonetech with implies business. s3 is smartphonetech therefore also a business. ss is a competitor of a, therefore it is also a rival. Does s steal s3 from ss? Yes, found it in KB.

creep Exit: (8) unethical(s) ? creep true.	All conditions are true→ Stevey is unethical. Return true.

Exercise 2

The old Royal succession rule states that the throne is passed down along the male line according to the order of birth before the consideration along the female line – similarly according to the order of birth. Queen Elizabeth, the monarch of United Kingdom, ha s four offsprings; namely:- Prince Charles, Princess Ann, Prince Andrew and Prince Edward – listed in the order of birth.

Question 2.1

Define their relations and rules in a prolog rule base. Hence, define the old Royal succession rule. Using this old succession rule determine the line of succession based on the information given. Do a trace to show your results.

Code	**Comments**
child(charles, elizabeth). child(ann, elizabeth). child(andrew, elizabeth). child(edward, elizabeth).	Charles, Ann, Andrew and Edward are children of Queen Elizabeth.
male(charles). male(andrew). male(edward).	Sons of the Queen.
female(ann). female(elizabeth).	Daughters of the Queen.
older(charles,ann). older(ann, andrew). older(andrew, edward).	Age relations.

higherrank(X,Y):-
    male(X),
    male(Y),
    is_older(X,Y).
	higherrank means X will be considered as the successor before Y. This happens if both are male and X is older than Y.

higherrank(X,Y):-
    female(X),
    female(Y),
    X\==elizabeth,
    Y\==elizabeth.
    is_older(X,Y).
	Leave out the Queen.
	
	In case that both X and Y are female, X will be higher in rank if X is older.

higherrank(X,Y):- female(Y),	Last case is if Y is female and X is male, then X will be higher in rank, given that Y is not the

`male(X),`
`Y\==elizabeth.`
 	
`is_older(X,Y):-`
`older(X,Y);`
`older(X,Z),`
`is_older(Z,Y).`
 	
`successionList(X, List):-`
`findall(Y,child(Y,X),`
`Children),`
`succession_sort(Children,`
`List).`
 	
`succession_sort([],[]).`
`succession_sort([A|B], Sorted):-`
`succession_sort(B, SortedTail),`
`insert(A,SortedTail, Sorted).`
 	
`insert(A, [B|C], [B|D]) :-`
`not(higherrank(A,B)),!,`
`insert(A, C, D).`
 	
`insert(A, C, [A|C]).`
	Queen.
	
	This relationship is implemented to differentiate between facts and relationships. Someone is older if there is the fact that he/she is older or if there is someone in between who is older than Y and younger than X.
	
	Given a person X, initialize the order of successors in List by finding all of X's children and sorting them after the rule.
	
	If both lists are empty, do nothing.
	Sorted is empty during initialization, [A|B] the list of all children. Sort the List by inserting element wise.
	
	Insertion following the higherrank/succession rule.

Trace	**Comments**

`[trace]  ?- successionList(elizabeth,List).`
`Call: (8) successionList(elizabeth, _3498) ?`
`creep`
`^  Call: (9) findall(_3704, child(_3704,`
`elizabeth), _3728) ? creep`
`Call: (14) child(_3704, elizabeth) ? creep`
`Exit: (14) child(charles, elizabeth) ? creep`
`Redo: (14) child(_3704, elizabeth) ? creep`
`Exit: (14) child(ann, elizabeth) ? creep`
`Redo: (14) child(_3704, elizabeth) ? creep`
`Exit: (14) child(andrew, elizabeth) ? creep`
`Redo: (14) child(_3704, elizabeth) ? creep`
`Exit: (14) child(edward, elizabeth) ? creep`
`^  Call: (14)`
`call('$bags':'$destroy_findall_bag') ? creep`
`^  Exit: (14)`
`call('$bags':'$destroy_findall_bag') ? creep`
`^  Exit: (9) findall(_3704, user:child(_3704,`
`elizabeth), [charles, ann, andrew, edward]) ?`
`creep`
`Call: (9) succession_sort([charles, ann,`
	Find all children of elizabeth.
	
	They are now stored in the Children list in the order of them appearing in the

andrew, edward], _3498) ? creep
  Call: (10) succession_sort([ann, andrew, edward], _3784) ? creep
  Call: (11) succession_sort([andrew, edward], _3784) ? creep
  Call: (12) succession_sort([edward], _3784) ? creep
  Call: (13) succession_sort([], _3784) ? creep
  Exit: (13) succession_sort([], []) ? creep
  Call: (13) insert(edward, [], _3786) ? creep
  Exit: (13) insert(edward, [], [edward]) ? creep
  Exit: (12) succession_sort([edward], [edward]) ? creep
  Call: (12) insert(andrew, [edward], _3792) ? creep
^ Call: (13) not(higherrank(andrew, edward)) ? creep
  Call: (14) higherrank(andrew, edward) ? creep
  Call: (15) male(andrew) ? creep
  Exit: (15) male(andrew) ? creep
  Call: (15) male(edward) ? creep
  Exit: (15) male(edward) ? creep
  Call: (15) is_older(andrew, edward) ? creep
  Call: (16) older(andrew, edward) ? creep
  Exit: (16) older(andrew, edward) ? creep
  Exit: (15) is_older(andrew, edward) ? creep
  Exit: (14) higherrank(andrew, edward) ? creep
^ Fail: (13) not(user:higherrank(andrew, edward)) ? creep
  Redo: (12) insert(andrew, [edward], _3792) ? creep
  Exit: (12) insert(andrew, [edward], [andrew, edward]) ? creep
  Exit: (11) succession_sort([andrew, edward], [andrew, edward]) ? creep
  Call: (11) insert(ann, [andrew, edward], _3798) ? creep
^ Call: (12) not(higherrank(ann, andrew)) ? creep
  Call: (13) higherrank(ann, andrew) ? creep
  Call: (14) male(ann) ? creep
  Fail: (14) male(ann) ? creep
  Redo: (13) higherrank(ann, andrew) ? creep
  Call: (14) female(ann) ? creep
  Exit: (14) female(ann) ? creep
  Call: (14) female(andrew) ? creep
	knowledge base (by age).
	Recursion starts.
	
	Recursion went to deepest layer (empty list).
	Insert edward into empty list.
	
	Sorted.
	
	Insert Andrew.
	
	Look whether Andrew is higher in rank than Edward.
	
	Yes he is, so insert him in the beginning.
	
	List is sorted.
	
	Insert Ann.

`   Fail: (14) female(andrew) ? creep`
`   Redo: (13) higherrank(ann, andrew) ? creep`
`   Call: (14) female(andrew) ? creep`
`   Fail: (14) female(andrew) ? creep`
`   Fail: (13) higherrank(ann, andrew) ? creep`
`^  Exit: (12) not(user:higherrank(ann, andrew))`
`? creep`
`   Call: (12) insert(ann, [edward], _3782) ?`
`creep`
`^  Call: (13) not(higherrank(ann, edward)) ?`
`creep`
`   Call: (14) higherrank(ann, edward) ? creep`
`   Call: (15) male(ann) ? creep`
`   Fail: (15) male(ann) ? creep`
`   Redo: (14) higherrank(ann, edward) ? creep`
`   Call: (15) female(ann) ? creep`
`   Exit: (15) female(ann) ? creep`
`   Call: (15) female(edward) ? creep`
`   Fail: (15) female(edward) ? creep`
`   Redo: (14) higherrank(ann, edward) ? creep`
`   Call: (15) female(edward) ? creep`
`   Fail: (15) female(edward) ? creep`
`   Fail: (14) higherrank(ann, edward) ? creep`
`^  Exit: (13) not(user:higherrank(ann, edward))`
`? creep`
	Ann is lower in rank than Andrew. Need to check if she is also lower in rank than Edward.

`   Call: (13) insert(ann, [], _3800) ? creep`
`   Exit: (13) insert(ann, [], [ann]) ? creep`
`   Exit: (12) insert(ann, [edward], [edward,`
`ann]) ? creep`
`   Exit: (11) insert(ann, [andrew, edward],`
`[andrew, edward, ann]) ? creep`
`   Exit: (10) succession_sort([ann, andrew,`
`edward], [andrew, edward, ann]) ? creep`
`   Call: (10) insert(charles, [andrew, edward,`
`ann], _3498) ? creep`
`^  Call: (11) not(higherrank(charles, andrew))`
`? creep`
`   Call: (12) higherrank(charles, andrew) ?`
`creep`
`   Call: (13) male(charles) ? creep`
`   Exit: (13) male(charles) ? creep`
`   Call: (13) male(andrew) ? creep`
`   Exit: (13) male(andrew) ? creep`
`   Call: (13) is_older(charles, andrew) ? creep`
`   Call: (14) older(charles, andrew) ? creep`
`   Fail: (14) older(charles, andrew) ? creep`
`   Redo: (13) is_older(charles, andrew) ? creep`
`   Call: (14) older(charles, _3856) ? creep`
`   Exit: (14) older(charles, ann) ? creep`
`   Call: (14) is_older(ann, andrew) ? creep`
	Sublist [andrew, edward] already sorted, insert ann at the end.
	
	Insert Charles.

`Call: (15) older(ann, andrew) ? creep`
`Exit: (15) older(ann, andrew) ? creep`
`Exit: (14) is_older(ann, andrew) ? creep`
`Exit: (13) is_older(charles, andrew) ? creep`
`Exit: (12) higherrank(charles, andrew) ? creep`
`^ Fail: (11) not(user:higherrank(charles, andrew)) ? creep`
`Redo: (10) insert(charles, [andrew, edward, ann], _3498) ? creep`
`Exit: (10) insert(charles, [andrew, edward, ann], [charles, andrew, edward, ann]) ? creep`
`Exit: (9) succession_sort([charles, ann, andrew, edward], [charles, andrew, edward, ann]) ? creep`
`Exit: (8) successionList(elizabeth, [charles, andrew, edward, ann]) ? creep`
`List = [charles, andrew, edward, ann].`
	Charles is higher in rank than Andrew, therefore insert him in front of Andrew.
	
	Now the list of successors is sorted.

The succession order following the succession rule is [charles, andrew, edward, ann]. Ann is last in the queue despite being the second oldest due to her gender.

Question 2.2

Recently, the Royal succession rule has been modified. The throne is now passed down according to the order of birth irrespective of gender. Modify your rules and prolog knowledge base to handle the new succession rule. Explain the necessary changes to the knowledge needed to represent the new information. Use this new succession rule to determine the new line of succession based on the same knowledge given. Show your results using a trace.

Just modify the higherrank relationship to:

```
higherrank(X,Y):-
      is_older(X,Y).
```

If the only purpose of this knowledge base is to find out the successor order, one can remove all the facts related to gender since they are not relevant anymore.

```
[trace]  ?- successionList(elizabeth,List).
   Call: (8) successionList(elizabeth, _3498) ? creep
^  Call: (9) findall(_3704, child(_3704, elizabeth), _3728) ? creep
   Call: (14) child(_3704, elizabeth) ? creep
   Exit: (14) child(charles, elizabeth) ? creep
   Redo: (14) child(_3704, elizabeth) ? creep
   Exit: (14) child(ann, elizabeth) ? creep
   Redo: (14) child(_3704, elizabeth) ? creep
   Exit: (14) child(andrew, elizabeth) ? creep
```

```
   Redo: (14) child(_3704, elizabeth) ? creep
   Exit: (14) child(edward, elizabeth) ? creep
^  Call: (14) call('$bags':'$destroy_findall_bag') ? creep
^  Exit: (14) call('$bags':'$destroy_findall_bag') ? creep
^  Exit: (9) findall(_3704, user:child(_3704, elizabeth), [charles, ann,
andrew, edward]) ? creep
   Call: (9) succession_sort([charles, ann, andrew, edward], _3498) ?
creep
   Call: (10) succession_sort([ann, andrew, edward], _3784) ? creep
   Call: (11) succession_sort([andrew, edward], _3784) ? creep
   Call: (12) succession_sort([edward], _3784) ? creep
   Call: (13) succession_sort([], _3784) ? creep
   Exit: (13) succession_sort([], []) ? creep
   Call: (13) insert(edward, [], _3786) ? creep
   Exit: (13) insert(edward, [], [edward]) ? creep
   Exit: (12) succession_sort([edward], [edward]) ? creep
   Call: (12) insert(andrew, [edward], _3792) ? creep
^  Call: (13) not(higherrank(andrew, edward)) ? creep
   Call: (14) higherrank(andrew, edward) ? creep
   Call: (15) is_older(andrew, edward) ? creep
   Call: (16) older(andrew, edward) ? creep
   Exit: (16) older(andrew, edward) ? creep
   Exit: (15) is_older(andrew, edward) ? creep
   Exit: (14) higherrank(andrew, edward) ? creep
^  Fail: (13) not(user:higherrank(andrew, edward)) ? creep
   Redo: (12) insert(andrew, [edward], _3792) ? creep
   Exit: (12) insert(andrew, [edward], [andrew, edward]) ? creep
   Exit: (11) succession_sort([andrew, edward], [andrew, edward]) ?
creep
   Call: (11) insert(ann, [andrew, edward], _3798) ? creep
^  Call: (12) not(higherrank(ann, andrew)) ? creep
   Call: (13) higherrank(ann, andrew) ? creep
   Call: (14) is_older(ann, andrew) ? creep
   Call: (15) older(ann, andrew) ? creep
   Exit: (15) older(ann, andrew) ? creep
   Exit: (14) is_older(ann, andrew) ? creep
   Exit: (13) higherrank(ann, andrew) ? creep
^  Fail: (12) not(user:higherrank(ann, andrew)) ? creep
   Redo: (11) insert(ann, [andrew, edward], _3798) ? creep
   Exit: (11) insert(ann, [andrew, edward], [ann, andrew, edward]) ?
creep
   Exit: (10) succession_sort([ann, andrew, edward], [ann, andrew,
edward]) ? creep
   Call: (10) insert(charles, [ann, andrew, edward], _3498) ? creep
^  Call: (11) not(higherrank(charles, ann)) ? creep
   Call: (12) higherrank(charles, ann) ? creep
   Call: (13) is_older(charles, ann) ? creep
   Call: (14) older(charles, ann) ? creep
   Exit: (14) older(charles, ann) ? creep
   Exit: (13) is_older(charles, ann) ? creep
```

```
   Exit: (12) higherrank(charles, ann) ? creep
^  Fail: (11) not(user:higherrank(charles, ann)) ? creep
   Redo: (10) insert(charles, [ann, andrew, edward], _3498) ? creep
   Exit: (10) insert(charles, [ann, andrew, edward], [charles, ann,
andrew, edward]) ? creep
   Exit: (9) succession_sort([charles, ann, andrew, edward], [charles,
ann, andrew, edward]) ? creep
   Exit: (8) successionList(elizabeth, [charles, ann, andrew, edward]) ?
creep
List = [charles, ann, andrew, edward].
```

The first part with finding the Children list remains the same, only the sorting process changes since the higherrank(X,Y) relation has been modified and simplified. Since we do not need to check the gender anymore, the computation time in the second case is a bit shorter, as well as the trace output. We would only need to check the is_older(X,Y) relationship. In this case, we can also remove the higherrank(X,Y) relation and use is_older(X,Y) as the relation used for sorting to simplify the code.