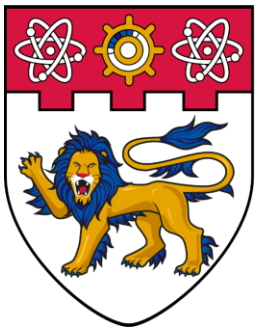

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ3005 ARTIFICIAL INTELLIGENCE

LEARNING TO USE PROLOG AS A LOGIC PROGRAMMING TOOL

ASSIGNMENT 3 REPORT

Shearman Chua Wei Jie (U1820058D)

LAB GROUP: TSP1

Contents

Exercise 1: The Smart Phone Rivalry 3
Question 1 3
Question 2 4
Question 3 4
Conclusions on Exercise 1: The Smart Phone Rivalry 5
Question 1 7
Trace 10
Question 2 12
Trace 14

Exercise 1: The Smart Phone Rivalry

Question 1

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. Smart phone technology is business.

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

Based purely from the natural language statements in the paragraph above, without any facts derived from external human interpretation and inference, these are the FOL constants and statements derived:

Constants (For FOL, constants are written with their first letters capitalized) :

- Sumsum
- Appy
- Galactica-s3
- Stevey

FOL Statements Derived

Statement	First-Order Logic	Reason/Type
"sumsum, a competitor of appy"	Competitor(Sumsum, Appy)	Atomic sentence, refer to relation between Sumsum and Appy
"developed some nice smart phone technology called galactica-s3"	SmartPhoneTechnology(Galactica-s3), Develop(Sumsum, Galactica-s3)	Term, Atomic sentence, refer to relation between Sumsum and Galactica-s3
"all of which was stolen by stevey, who is a boss"	Steal(Stevey, Galactica-s3, Sumsum), Boss(Stevey)	Atomic sentence in the form "x steals y from z", Term: Boss(Stevey)
"It is unethical for a boss to steal business from rival companies"	$\text{Boss}(x) \wedge \text{Steal}(x, y, z) \wedge \text{Business}(y) \wedge \text{Rival}(z) \Rightarrow \text{Unethical}(x)$	If x is boss, x steals y from z, where y is business, and z is a rival company, then x is unethical
"A competitor of appy is a rival"	$\text{Competitor}(x, \text{Appy}) \Rightarrow \text{Rival}(x)$	Logical statement
"Smart phone technology is business"	$\text{SmartPhoneTechnology}(x) \Rightarrow \text{Business}(x)$	Logical statement

Question 2

Write these FOL statements as Prolog clauses.

For Prolog, logical statements are written a little different from FOL notation. In FOL notation, we write the premise (antecedent) on the left-hand side of the statement, and the conclusion (consequent) on the right-hand side of the statement, in the form of X ⇒ Y.

Whereas for Prolog, for logical statements, we write the conclusion (consequent), or "goal" as it is known in Prolog, on the left-hand side, and the premise (antecedent) on the right-hand side of the statement, in the form of "conclusion" :- "premise 1", "premise 2".

Also, for Prolog, Variables are written as Capitalized letters or words, while Constants are written in all lower-case letters. In FOL notation, "and" is represented by " ∧ " while in Prolog, it is just represented by a comma ",".

```
competitor(sumsum,appy).
develop(sumsum,galactica-s3).
smart_phone_technology(galactica-s3).
steal(stevey,galactica-s3,sumsum).
boss(stevey).
business(X):- smart_phone_technology(X).
rival(X) :- competitor(X,appy).
unethical(X):- boss(X),steal(X,Y,Z),business(Y),rival(Z).
```

Prolog clauses

Question 3

Using Prolog, prove that Stevey is unethical. Show a trace of your proof.

```
[trace]  ?- unethical(stevey).
   Call: (10) unethical(stevey) ? creep
   Call: (11) boss(stevey) ? creep
   Exit: (11) boss(stevey) ? creep
   Call: (11) steal(stevey, _19136, _19138) ? creep
   Exit: (11) steal(stevey, galactica-s3, sumsum) ? creep
   Call: (11) business(galactica-s3) ? creep
   Call: (12) smart_phone_technology(galactica-s3) ? creep
   Exit: (12) smart_phone_technology(galactica-s3) ? creep
   Exit: (11) business(galactica-s3) ? creep
   Call: (11) rival(sumsum) ? creep
   Call: (12) competitor(sumsum, appy) ? creep
   Exit: (12) competitor(sumsum, appy) ? creep
   Exit: (11) rival(sumsum) ? creep
   Exit: (10) unethical(stevey) ? creep
true.
```

Prolog Trace

Above is the Prolog trace when we consult the Prolog clauses derived for the Smart Phone industry. From the trace, we can see that when we consult unethical(stevey), we pass stevey into the unethical() function.

The Prolog program then first check the first premise of the unethical() function, which is whether stevey is a boss by calling boss(stevey), the Prolog clause boss(stevey) is present in the Prolog knowledge base, therefore it returns true. Next, the second premise of the unethical() function is

checked, where we check if there exist a Prolog clause where steal(stevey, X, Y) for some X, Y is true. The Prolog clause steal(stevey, galactica-s3, sumsum) is present in the Prolog knowledge base, therefore it returns true. Then, the third premise of the unethical() function is checked, where the business(galactica-s3) function is called. Prolog in turn checks if galactica-s3 is a smart phone technology by calling smart_phone_technology(galactica-s3). Then, by Modus Ponens, business(galactica-s3) returns true based on the Prolog clause business(X) :- smart_phone_technology(X). Next, we check the final premise of the unethical() function, to check if sumsum is a rival company by calling rival(sumsum). Prolog in turn checks if sumsum is a competitor of appy by calling competitor(sumsum, appy) which returns true. Then, by Modus Ponens, rival(sumsum) returns true based on the Prolog clause rival(X) :- competitor(X, appy). As all of the premises of unethical(stevey) returns true, by Modus Ponens, unethical(stevey) returns true.

Conclusions on Exercise 1: The Smart Phone Rivalry

From the questions in Exercise 1, we derived the First Order Logic (FOL) statements directly from the natural language statements in the paragraph given in the exercise. We then transform the FOL statements into Prolog clauses which are then used to derive if Stevey is unethical, based on querying the Prolog Knowledge base that we have defined using the Prolog clauses.

However, we can observe that the First Order Logic (FOL) statements and Prolog clauses derived directly from the natural language statements in the paragraph given in the exercise does not allow us to build a strong and logically sound knowledge base for describing the dealing within the Smart Phone industry, if we were to extend the knowledge base to include more companies in the Smart Phone industry.

For example, given the new knowledge: “mokia, a competitor of sumsum, developed a new smart phone technology called mokia-1, all of which was stolen by stevey.” Then we would have the FOL statements:

- Competitor(Mokia, Sumsum)
- SmartPhoneTechnology(Mokia-1)
- Develop(Mokia, Mokia-1)
- Steal(Stevey, Mokia-1, Mokia)

Which when converted to Prolog will give the clauses:

- competitor(mokia, sumsum)
- smart_phone_technology(mokia-1)
- develop(mokia, mokia-1)
- steal(stevey, mokia-1, mokia)

Then, let us say that from the FOL statements derived in Question 1, we remove Steal(Stevey, Galactica-s3, Sumsum), and keep the other remaining FOL statements, and from the Prolog clauses derived in Question 2, we remove steal(stevey, galactica-s3, sumsum), and keep the other remaining Prolog clauses, and we want to find out if Stevey is unethical, when we consult Prolog unethical(stevey), the result returns false.

```
?- trace,unethical(stevey).
   Call: (11) unethical(stevey) ? creep
   Call: (12) boss(stevey) ? creep
   Exit: (12) boss(stevey) ? creep
   Call: (12) steal(stevey, _18396, _18398) ? creep
   Exit: (12) steal(stevey, mokia-1, mokia) ? creep
   Call: (12) business(mokia-1) ? creep
   Call: (13) smart_phone_technology(mokia-1) ? creep
   Exit: (13) smart_phone_technology(mokia-1) ? creep
   Exit: (12) business(mokia-1) ? creep
   Call: (12) rival(mokia) ? creep
   Call: (13) competitor(mokia, appy) ? creep
   Fail: (13) competitor(mokia, appy) ? creep
   Fail: (12) rival(mokia) ? creep
   Fail: (11) unethical(stevey) ? creep
false.
```

This is due to the fact that from the new information, there is no new information regarding if mokia is a competitor of appy. Therefore, when Prolog tries to derive whether stevey is unethical, Prolog returns false as the last premise of the goal unethical(X) returns false due to the fact that competitor(mokia, appy) returns false which resulted in Rival(mokia) returning false.

However, from a human inference and logic point of view, we know that stevey is unethical as he stole smart phone technology (mokia-1) from mokia and hence, business from mokia, but the rules and clauses defined in the knowledge base was not able to derive this. Therefore, there is a need for better defined rules in the knowledge base if we want to extend the knowledge base for describing the dealing within the Smart Phone industry for more companies.

Therefore, we can see that by just building the knowledge base of a logical agent solely from a small set of natural language statements, can result in logical fallacies when the logical agent derive the truth for some given statement from the limited knowledge in the knowledge base. For example, the FOL logical statement in Question 1, **Boss(x) ∧ Steal(x, y, z) ∧ Business(y) ∧ Rival(z) ⇒ Unethical(x)**, can lead to logical fallacies as the statement will return true as long as for some boss X, who stole Y from Z where Y is a business, and that there exist some Rival(Z). Rival(Z) can only derive true only if Competitor(Z, Appy) is true based on the knowledge base we have, meaning, as long as a boss X stole from a company Z where Z is not a Competitor of appy, he will not be unethical which is a logical fallacy. The knowledge base also have missing information like for example, from boss(stevey), we know that stevey is a boss, but we have no idea he is the boss of which company. Therefore, in the statement **Boss(x) ∧ Steal(x, y, z) ∧ Business(y) ∧ Rival(z) ⇒ Unethical(x)**, there is no consideration whether stevey is the boss of appy, as long as the company he stole from is a rival of appy, the logical agent will deem him as unethical, which means that even if stevey is the boss of another company, as long as the item he stole belongs to a rival company of appy, he is deemed unethical which seems illogical in a real-world context.

A better knowledge base representing the dealing within the Smart Phone industry will look something like this:

FOL notation:

- Competitor(x, y)
- SmartPhoneTechnology(x)
- Develop(x, y)
- Steal(x, y, z)
- Boss(x, y)
- Boss(x, w) ∧ Steal(x, y, z) ∧ Business(y) ∧ Competitor(z, w) ⇒ Unethical(x)
- SmartPhoneTechnology(x) ⇒ Business(x)

From the new FOL statement, **Boss(x, w) ∧ Steal(x, y, z) ∧ Business(y) ∧ Competitor(z, w) ⇒ Unethical(x)**, we have, if a Boss x of company w, steals y from company z, where y is business, and z is a competitor of w, then Boss x is unethical. This way, we can extend the knowledge base to include more companies without having to change the rule. We have also removed Competitor(x, appy) ⇒ Rival(x) and also not consider having Competitor(x, y) ⇒ Rival(x,y) as the Competitor(x,y) predicate is enough to show that two companies are rivals.

Therefore, it is important to have a strong knowledge base and correct logical statements to ensure that there are no logical fallacies when the logical agent derive statements from the knowledge in the knowledge base.

Exercise 2: The Royal Family

Question 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.

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, has four offsprings; namely:- prince charles, princess ann, prince andrew and prince edward – listed in the order of birth.

```
queen(elizabeth).
monarch(elizabeth).
prince(charles).
prince(andrew).
prince(edward).
princess(ann).
offspring(elizabeth,charles).
offspring(elizabeth,ann).
offspring(elizabeth,andrew).
offspring(elizabeth,edward).

older(charles,ann).
older(ann,andrew).
older(andrew,edward).

male_heir(X,Y):-prince(Y),offspring(X,Y).
female_heir(X,Y):-princess(Y),offspring(X,Y).

is_older(X,Y):-older(X,Y).
is_older(X,Y):-older(X,Z),is_older(Z,Y).

birth_order([A|B],Sorted):-birth_order(B,SortedTail),insert(A,SortedTail,Sorted).
birth_order([],[]).

insert(A,[B|C],[B|D]):-not(is_older(A,B)),insert(A,C,D).
insert(A,C,[A|C]).

succession_list(X,SuccessionList):- findall(Y,male_heir(X,Y),MaleLine),
                                    findall(Y,female_heir(X,Y),FemaleLine),
                                    birth_order(MaleLine,MaleLineSorted),
                                    birth_order(FemaleLine,FemaleLineSorted),
                                    append(MaleLineSorted,FemaleLineSorted,SuccessionList).
```

Prolog Clauses and Rules

The figure above shows the Prolog rule base implemented based on the natural language statements for the old Royal succession rule in the paragraph above. First, we define the terms of the royal succession rule base:

- queen(elizabeth).
- monarch(elizabeth).
- prince(charles).
- prince(andrew).
- prince(edward).
- princess(ann).

To state that elizabeth is a queen, as well as the monarch of the royal family. The terms following then defines that charles, andrew, and edward are princes and ann is a princess in the royal family.

Next, we define the facts given by the natural language statements:

- offspring(elizabeth, charles).
- offspring(elizabeth, ann).
- offspring(elizabeth, andrew).
- offspring(elizabeth, edward).

- older(charles, ann).
- older(ann, andrew).
- older(andrew, edward).

Where from the Prolog clauses **offspring(X, Y)** above we can interpret that, elizabeth, has 4 offsprings, namely, charles, ann, andrew and edward and from the clauses **older(X, Y)**, we build the knowledge base of who is older than whom in the royal family.

Old Royal Succession Rule

```
male_heir(X,Y):-prince(Y),offspring(X,Y).
female_heir(X,Y):-princess(Y),offspring(X,Y).

is_older(X,Y):-older(X,Y).
is_older(X,Y):-older(X,Z),is_older(Z,Y).

birth_order([A|B],Sorted):-birth_order(B,SortedTail),insert(A,SortedTail,Sorted).
birth_order([],[]).

insert(A,[B|C],[B|D]):-not(is_older(A,B)),insert(A,C,D).
insert(A,C,[A|C]).

succession_list(X,SuccessionList):- findall(Y,male_heir(X,Y),MaleLine),
                                    findall(Y,female_heir(X,Y),FemaleLine),
                                    birth_order(MaleLine,MaleLineSorted),
                                    birth_order(FemaleLine,FemaleLineSorted),
                                    append(MaleLineSorted,FemaleLineSorted,SuccessionList).
```

The **succession_list**(X, SuccessionList) rule is the old Royal succession rule used to determine the line of succession based on the rule base defined by the Prolog clauses. It takes in an argument X and returns a succession list in the order of succession precedence.

For the goal **succession_list**(X, SuccessionList), we first call the function **findall**(Y, male_heir(X, Y), MaleLine), where **findall** is a built-in list function in Prolog, that returns all Y that satisfy the function **male_heir**(X, Y). The **male_heir**(X, Y) function determines if firstly, Y is a prince, and then checks if Y is the offspring of X, if both premises are true, the male heir of X is Y. Using **findall**(Y, male_heir(X, Y), MaleLine), we will find a list of constants that belongs to the male line of the royal family, return in the list MaleLine.

We then call the function **findall**(Y, female_heir(X, Y), FemaleLine), where **findall** is a built-in list function in Prolog, that returns all Y that satisfy the function **female_heir**(X, Y). The **female_heir**(X,

Y) function determines if firstly, Y is a princess, and then checks if Y is the offspring of X, if both premises are true, the female heir of X is Y. Using **findall**(Y, female_heir(X, Y), FemaleLine), we will find a list of constants that belongs to the female line of the royal family, return in the list FemaleLine.

Next, for the third premise of the **succession_list**(X, SuccessionList) rule, to satisfy the condition given in the natural language statement "the throne is passed down along the male line according to the order of birth", we call the function **birth_order**(MaleLine, MaleLineSorted), to sort the MaleLine list in birth order, where the oldest in the list will be placed at the head of the list and the youngest in the list placed as the last element of the list in the returned list MaleLineSorted. When we first call the **birth_order**() function, we trigger the first **birth_order**() rule, given by **birth_order**([A|B], Sorted) :- **birth_order**(B, SortedTail), **insert**(A, SortedTail, Sorted) where the first premise of the rule will recursively call the **birth_order**([A|B], Sorted) rule until the list that is being passed in as the first argument to the function becomes empty, in which case the **birth_order**([], []) rule is triggered. When that happens, we return back to the **birth_order**() rule that triggered the **birth_order**([], []) rule, and we enter the second premise of the rule, which is, **insert**(A, SortedTail, Sorted), which calls the **insert**() function. Similarly, the **insert**() function has two rules, where for the rule **insert**(A,[B|C],[B|D]):-not(is_older(A,B)),insert(A,C,D) has two premises and the first premise checks if the first argument of the **insert** function, A, is older than the head of the list of the second argument, B. If the first premise is true, we then recursively call **insert**(A,C,D) to recursively sort the list. The is_older(A,B) function in the **insert**(A,[B|C],[B|D]):-not(is_older(A,B)),insert(A,C,D) rule triggers the **is_older**() function which has two rules. The first **is_older**(X, Y) rule checks if X is older than Y, if this rule returns false, the second **is_older**(X, Y) rule is triggered to check that if X is older than some Z, and Z is older than Y, by Modus Ponens, we can infer that X is older than Y. The reason we need this rule is that in the rule base, we did not explicitly make a predicate for all the other offsprings a constant is older than. For the **older**(X, Y) clauses, we only state the predicates where for some X, Y is the next sibling to be born. After the list is sorted in birth order, the second premise of **birth_order**([A|B], Sorted) is fulfilled and the sorted list is return to **birth_order**(MaleLine, MaleLineSorted) as the list MaleLineSorted.

For the fourth premise of the **succession_list**(X, SuccessionList) rule, similar to the third premise, we call the function **birth_order**(FemaleLine, FemaleLineSorted) to sort the female line of the royal family in birth order where the oldest in the list will be placed at the head of the list and the youngest in the list placed as the last element of the list in the returned list FemaleLineSorted.

Finally, in the last fourth premise of the **succession_list**(X, SuccessionList) rule, we call the function **append**(MaleLineSorted, FemaleLineSorted, SuccessionList) where **append** is a built-in list function of Prolog where the first 2 arguments are lists and the second list is appended to the back of the first list and the result is returned as the third argument of the function. In order to fulfil the statement "passed down along the male line according to the order of birth before the consideration along the female line", we pass in the MaleLineSorted with the male line of the royal family in birth order as the first argument and the FemaleLineSorted with the female line of the royal family in birth order and we append the FemaleLineSorted list to the back of the MaleLineSorted list and store the final results in SuccessionList. Therefore, for the end result, we have a list called SuccessionList whereby the front of the list are the male descendants of the royal line in birth order and the back of the list are the female descendants of the royal line in birth order, to give us a list of the line of succession based on the old royal succession rule.

Trace

```
?- trace,succession_list(elizabeth,SuccessionList).
   Call: (11) succession_list(elizabeth, _6804) ? creep
^  Call: (12) findall(_7286, male_heir(elizabeth, _7286), _7346) ? creep
   Call: (17) male_heir(elizabeth, _7286) ? creep
   Call: (18) prince(_7286) ? creep
   Exit: (18) prince(charles) ? creep
   Call: (18) offspring(elizabeth, charles) ? creep
   Exit: (18) offspring(elizabeth, charles) ? creep
   Exit: (17) male_heir(elizabeth, charles) ? creep
   Redo: (18) prince(_7286) ? creep
   Exit: (18) prince(andrew) ? creep
   Call: (18) offspring(elizabeth, andrew) ? creep
   Exit: (18) offspring(elizabeth, andrew) ? creep
   Exit: (17) male_heir(elizabeth, andrew) ? creep
   Redo: (18) prince(_7286) ? creep
   Exit: (18) prince(edward) ? creep
   Call: (18) offspring(elizabeth, edward) ? creep
   Exit: (18) offspring(elizabeth, edward) ? creep
   Exit: (17) male_heir(elizabeth, edward) ? creep
^  Exit: (12) findall(_7286, user:male_heir(elizabeth, _7286), [charles, andrew, edward]) ? creep
^  Call: (12) findall(_7286, female_heir(elizabeth, _7286), _8196) ? creep
   Call: (17) female_heir(elizabeth, _7286) ? creep
   Call: (18) princess(_7286) ? creep
   Exit: (18) princess(ann) ? creep
   Call: (18) offspring(elizabeth, ann) ? creep
   Exit: (18) offspring(elizabeth, ann) ? creep
   Exit: (17) female_heir(elizabeth, ann) ? creep
^  Exit: (12) findall(_7286, user:female_heir(elizabeth, _7286), [ann]) ? creep
   Call: (12) birth_order([charles, andrew, edward], _8586) ? creep
   Call: (13) birth_order([andrew, edward], _8630) ? creep
   Call: (14) birth_order([edward], _8674) ? creep
   Call: (15) birth_order([], _8718) ? creep
   Exit: (15) birth_order([], []) ? creep
   Call: (15) insert(edward, [], _8808) ? creep
   Exit: (15) insert(edward, [], [edward]) ? creep
   Exit: (14) birth_order([edward], [edward]) ? creep
   Call: (14) insert(andrew, [edward], _8946) ? creep
^  Call: (15) not(is_older(andrew, edward)) ? creep
   Call: (16) is_older(andrew, edward) ? creep
   Call: (17) older(andrew, edward) ? creep
   Exit: (17) older(andrew, edward) ? creep
   Exit: (16) is_older(andrew, edward) ? creep
^  Fail: (15) not(user:is_older(andrew, edward)) ? creep

   Redo: (14) insert(andrew, [edward], _9272) ? creep
   Exit: (14) insert(andrew, [edward], [andrew, edward]) ? creep
   Exit: (13) birth_order([andrew, edward], [andrew, edward]) ? creep
   Call: (13) insert(charles, [andrew, edward], _9410) ? creep
^  Call: (14) not(is_older(charles, andrew)) ? creep
   Call: (15) is_older(charles, andrew) ? creep
   Call: (16) older(charles, andrew) ? creep
   Fail: (16) older(charles, andrew) ? creep
   Redo: (15) is_older(charles, andrew) ? creep
   Call: (16) older(charles, _9690) ? creep
   Exit: (16) older(charles, ann) ? creep
   Call: (16) is_older(ann, andrew) ? creep
   Call: (17) older(ann, andrew) ? creep
   Exit: (17) older(ann, andrew) ? creep
   Exit: (16) is_older(ann, andrew) ? creep
   Exit: (15) is_older(charles, andrew) ? creep
^  Fail: (14) not(user:is_older(charles, andrew)) ? creep
   Redo: (13) insert(charles, [andrew, edward], _10044) ? creep
   Exit: (13) insert(charles, [andrew, edward], [charles, andrew, edward]) ? creep
   Exit: (12) birth_order([charles, andrew, edward], [charles, andrew, edward]) ? creep
   Call: (12) birth_order([ann], _10180) ? creep
   Call: (13) birth_order([], _10224) ? creep
   Exit: (13) birth_order([], []) ? creep
   Call: (13) insert(ann, [], _10314) ? creep
   Exit: (13) insert(ann, [], [ann]) ? creep
   Exit: (12) birth_order([ann], [ann]) ? creep
   Call: (12) lists:append([charles, andrew, edward], [ann], _6804) ? creep
   Exit: (12) lists:append([charles, andrew, edward], [ann], [charles, andrew, edward, ann]) ? creep
   Exit: (11) succession_list(elizabeth, [charles, andrew, edward, ann]) ? creep
SuccessionList = [charles, andrew, edward, ann].
```

Prolog Trace for Old Royal Succession Rule

The figure above shows the trace when we consult the Prolog rule base with **trace**, **succession_list**(elizabeth, SuccessionList) where we trace all the function and rule calls of the function **succession_list**(elizabeth, SuccessionList), where we find the line of royal succession for the current monarch, queen elizabeth. In **trace**, the “Exit:” trace shows that a premise or predicate returns as true and the “Fail:” trace shows that a premise or predicate returns as false.

From the trace, we can see that the function calls are done in accordance to the Old Royal Succession Rule we have formulated in the Prolog rule base, where the first function called is the **succession_list**() function, where elizabeth replaces the variable X, and the first premise of the

succession_list() rule is activated and the **findall**() function is called to find all constants in the Prolog rule base that satisfy **male_heir**(elizabeth, Y), where the **male_heir**() function first find a constant that satisfy the premise **prince**() and thereafter, check if the constant is an offspring of elizabeth, then by Modus Ponens, we can determine if the constant is a male heir of queen elizabeth. After all the possible constants that satisfy male_heir(elizabeth, Y) are found, they are returned in a list as MaleLine.

Next, we go to the second premise of the **succession_list**() rule, where we call the **findall**() function to find all constants in the Prolog rule base that satisfy female_heir(elizabeth, Y), where the female_heir() function first find a constant that satisfy the premise princess() and thereafter, check if the constant is an offspring of elizabeth, then by Modus Ponens, we can determine if the constant is a female heir of queen elizabeth. After all the possible constants that satisfy female_heir(elizabeth, Y) are found, they are returned in a list as FemaleLine.

Then, the **trace** shows that the **succession_list**() rule proceeds to call the third premise, where the **birth_order**() function is called where the MaleLine list is passed in as first argument to sort the MaleLine list in birth order. As can be seen from the trace, the **birth_order**() function is recursively called until both the arguments in **birth_order**() are empty lists. Then, the second premise of the **birth_order**() is called, where we recursively call the **insert**() function to sort the MaleLine list in birth order. For the trace above, as the **findall**() returned a MaleLine list already in birth order, we can see that the predicate **not**(**is_older**(X, Y)) always returns false. The **birth_order**() function the returns the MaleLine list sorted in birth order as MaleLineSorted. One interesting thing we can observe in the trace is that when **is_older**(charles, andrew) is called, as there is no **older**(charles, andrew) predicate in the Prolog rule base, the **is_older**() function checks **older**(charles, ann) and **older**(ann, andrew) to return **is_older**(charles, andrew) as true by Modus Ponens.

Then, the **trace** shows that the **succession_list**() rule proceeds to call the fourth premise, where the **birth_order**() function is called where the FemaleLine list is passed in as first argument to sort the FemaleLine list in birth order. The **birth_order**() function the returns the FemaleLine list sorted in birth order as FemaleLineSorted.

Lastly, the **trace** shows that the **succession_list**() rule proceeds to call the fifth premise, where the **append**() function is called to append the list, [ann], to the back of the list [charles, andrew, edward], to give the SuccessionList, [charles, andrew, edward, ann].

With all the premises of the **succession_list**() rule satisfied, the **succession_list**() rule returns the succession list given by SuccessionList = [charles, andrew, edward, ann], to give us the line of succession by the old royal succession rule.

Question 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.

```
queen(elizabeth).
monarch(elizabeth).
prince(charles).
prince(andrew).
prince(edward).
princess(ann).
offspring(elizabeth,charles).
offspring(elizabeth,ann).
offspring(elizabeth,andrew).
offspring(elizabeth,edward).

older(charles,ann).
older(ann,andrew).
older(andrew,edward).

male_heir(X,Y):-prince(Y),offspring(X,Y).
female_heir(X,Y):-princess(Y),offspring(X,Y).

is_older(X,Y):-older(X,Y).
is_older(X,Y):-older(X,Z),is_older(Z,Y).

birth_order([A|B],Sorted):-birth_order(B,SortedTail),insert(A,SortedTail,Sorted).
birth_order([],[]).

insert(A,[B|C],[B|D]):-not(is_older(A,B)),insert(A,C,D).
insert(A,C,[A|C]).

succession_list_new(X,SuccessionList):- findall(Y,offspring(X,Y),RoyalLine),
                                        birth_order(RoyalLine,SuccessionList).
```

New Succession Rule Prolog Knowledge Base

The figure above shows the Prolog Knowledge Base for the new royal succession rule. As can be seen from the Prolog Knowledge Base above, most of the predicates in the knowledge base remains the same as elizabeth is still a queen, and the monarch of the United Kingdom, charles, andrew and edward are still princes, and ann is still a princess. The offsprings of elizabeth are not changed as well as elizabeth still has charles, ann, andrew and edward as her offsprings. Consequently, the birth order in the royal family does not change too, as the offsprings are still born in the same order, so for all **older**(X, Y) predicates in the Prolog Knowledge Base, we do not have to change them.

For the logical statements **male_heir**(X, Y) and **female_heir**(X, Y), we do not have to remove them or change them as the it is still a fact that for a given Y where Y is a prince and Y is the offspring of X, the male heir of X is Y and given Y where Y is a princess and Y is the offspring of X, the female heir of X is Y. The logical statements **male_heir**(X, Y) and **female_heir**(X, Y) can be kept in case a user want to consult the Prolog Knowledge Base, for some given X, Y, does **male_heir**(X, Y) or **female_heir**(X, Y) returns true.

Similarly, the logical statements for **is_older**(X, Y) are not required to be changed as the logic surrounding **is_older**(X, Y) still remains the same. For a given **is_older**(X, Y), if we can find a predicate **older**(X, Y), then **is_older**(X, Y) returns true. If not using the second **is_older**(X, Y) logical statement, we can recursively check that for a given X, where **older**(X, Z) is true and **is_older**(Z, Y) returns true, by Modus Ponens, **is_older**(X, Y) returns true.

As the modified royal succession rule still checks birth order from the statement “throne is now passed down according to the order of birth irrespective of gender” the logic behind the **birth_order**() function does not change as we still sort the royal succession line by birth order, meaning that the two **birth_order**() rules in the Prolog Knowledge Base does not have to change. Given that, consequently, since we have to check for the royal succession list who is older than whom, the logic behind the two **insert**() rules remains the same and **insert**() rules in the Prolog Knowledge Base does not have to change.

The only change that we have to make to the Prolog Knowledge Base is the **succession_list**() rule. To distinguish the new rule, we renamed the rule as **succession_list_new**(). From the natural language statement “The throne is now passed down according to the order of birth irrespective of gender”, the succession line is now based purely on birth order, therefore, we do not have to split the royal line into a male line and a female line, therefore, we can combine the **findall**(Y, male_heir(X, Y), MaleLine) and **findall**(Y, female_heir(X, Y), FemaleLine) premises of the old **succession_list**() rule into one premise, **findall**(Y, offspring(X, Y), RoyalLine), for the new **succession_list_new**() rule. Then since we do not have 2 separate lists to sort in birth order, we just need one **birth_order**(RoyalLine, SuccessionList) to sort the royal succession list in birth order. The **append**() function can also be removed as there is only one royal succession list instead of one royal succession list for males and one royal succession list for females. With that, we have the new royal succession rule in the Prolog Knowledge Base to represent the new information as:

succession_list_new(X, SuccessionList):- **findall**(Y, offspring(X,Y),RoyalLine),

birth_order(RoyalLine, SuccessionList).

Trace

```
?- trace,succession_list_new(elizabeth,SuccessionList).
   Call: (11) succession_list_new(elizabeth, _6804) ? creep
^  Call: (12) findall(_7286, offspring(elizabeth, _7286), _7346) ? creep
   Call: (17) offspring(elizabeth, _7286) ? creep
   Exit: (17) offspring(elizabeth, charles) ? creep
   Redo: (17) offspring(elizabeth, _7286) ? creep
   Exit: (17) offspring(elizabeth, ann) ? creep
   Redo: (17) offspring(elizabeth, _7286) ? creep
   Exit: (17) offspring(elizabeth, andrew) ? creep
   Redo: (17) offspring(elizabeth, _7286) ? creep
   Exit: (17) offspring(elizabeth, edward) ? creep
^  Exit: (12) findall(_7286, user:offspring(elizabeth, _7286), [charles, ann, andrew, edward]) ? creep
   Call: (12) birth_order([charles, ann, andrew, edward], _6804) ? creep
   Call: (13) birth_order([ann, andrew, edward], _7886) ? creep
   Call: (14) birth_order([andrew, edward], _7930) ? creep
   Call: (15) birth_order([edward], _7974) ? creep
   Call: (16) birth_order([], _8018) ? creep
   Exit: (16) birth_order([], []) ? creep
   Call: (16) insert(edward, [], _8108) ? creep
   Exit: (16) insert(edward, [], [edward]) ? creep
   Exit: (15) birth_order([edward], [edward]) ? creep
   Call: (15) insert(andrew, [edward], _8246) ? creep
^  Call: (16) not(is_older(andrew, edward)) ? creep
   Call: (17) is_older(andrew, edward) ? creep
   Call: (18) older(andrew, edward) ? creep
   Exit: (18) older(andrew, edward) ? creep
   Exit: (17) is_older(andrew, edward) ? creep
^  Fail: (16) not(user:is_older(andrew, edward)) ? creep
   Redo: (15) insert(andrew, [edward], _8572) ? creep
   Exit: (15) insert(andrew, [edward], [andrew, edward]) ? creep
   Exit: (14) birth_order([andrew, edward], [andrew, edward]) ? creep
   Call: (14) insert(ann, [andrew, edward], _8710) ? creep
^  Call: (15) not(is_older(ann, andrew)) ? creep
   Call: (16) is_older(ann, andrew) ? creep
   Call: (17) older(ann, andrew) ? creep
   Exit: (17) older(ann, andrew) ? creep
   Exit: (16) is_older(ann, andrew) ? creep
^  Fail: (15) not(user:is_older(ann, andrew)) ? creep
   Redo: (14) insert(ann, [andrew, edward], _9036) ? creep
   Exit: (14) insert(ann, [andrew, edward], [ann, andrew, edward]) ? creep
   Exit: (13) birth_order([ann, andrew, edward], [ann, andrew, edward]) ? creep
   Call: (13) insert(charles, [ann, andrew, edward], _6804) ? creep
^  Call: (14) not(is_older(charles, ann)) ? creep
   Call: (15) is_older(charles, ann) ? creep
   Call: (16) older(charles, ann) ? creep
   Exit: (16) older(charles, ann) ? creep
   Exit: (15) is_older(charles, ann) ? creep
^  Fail: (14) not(user:is_older(charles, ann)) ? creep
   Redo: (13) insert(charles, [ann, andrew, edward], _6804) ? creep
   Exit: (13) insert(charles, [ann, andrew, edward], [charles, ann, andrew, edward]) ? creep
   Exit: (12) birth_order([charles, ann, andrew, edward], [charles, ann, andrew, edward]) ? creep
   Exit: (11) succession_list_new(elizabeth, [charles, ann, andrew, edward]) ? creep
SuccessionList = [charles, ann, andrew, edward].
```

Prolog Trace for New Royal Succession Rule

When we consult the Prolog rule base with **trace**, **succession_list_new**(elizabeth, SuccessionList) where we trace all the function and rule calls of the function **succession_list_new**(elizabeth, SuccessionList), where we find the line of royal succession for the current monarch, queen elizabeth, using the new royal succession rule. In **trace**, the "Exit:" trace shows that a premise or predicate returns as true and the "Fail:" trace shows that a premise or predicate returns as false.

From the trace, we can see that the function calls are done in accordance to the New Royal Succession Rule we have formulated in the Prolog rule base, where the first function called is the **succession_list_new**() function, where elizabeth replaces the variable X, and the first premise of the **succession_list_new**() rule is activated and the **findall**() function is called to find all constants in the Prolog rule base that satisfy offspring(elizabeth, Y). The Prolog trace for the New Royal Succession Rule is significantly shorter than the trace for the Old Royal Succession Rule as Prolog now does not need to distinguish between male and female heirs and all the constants that satisfy the function offspring(elizabeth, Y) are returned in one single list using the **findall**() function. The Prolog program just have to search the offspring(X, Y) predicates in the Knowledge Base for the **findall**() function.

Once we have the list returned by the **findall**() function, as can be seen from the trace, the program moves to the second premise of the **succession_list_new**() function, where the function **birth_order**(RoyalLine, SuccessionList) is called where the list obtained by the **findall**() function is passed to the second premise as the first argument of **birth_order**(RoyalLine, SuccessionList) to sort

the royal succession list in birth order. As the **findall**() function returned a list in birth order for this instance of the Prolog consult, all the **not**(**is_older**(A,B)) function calls return false for the **insert**() function, and the **birth_order**() function returns the same list in birth order as the list SuccessionList.

With all the premises of the **succession_list_new**() function satisfied, the **succession_list_new**() rule returns the succession list given by SuccessionList = [charles, ann, andrew, edward], to give us the line of succession by the new royal succession rule.