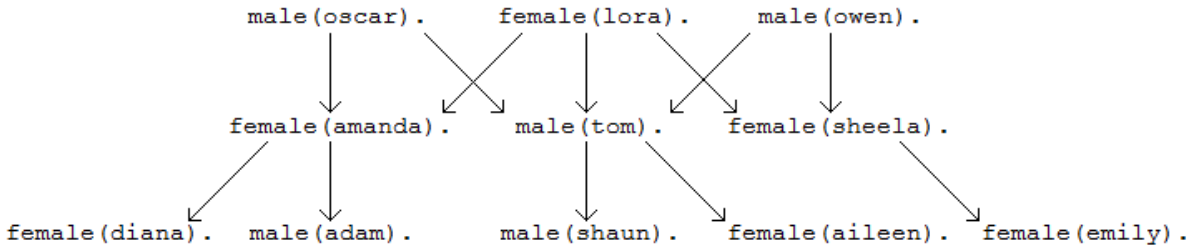
1 – Family Tree

Q1. *Draw the family tree representing the following facts in Prolog.* **[20 marks]**



Above you can see the individual members of the family displayed with their "gender" predicate and the lines are used to define the "`parent(X, Y)`" predicate in that the arrow's direction shows their order – for example, the predicate "`parent(oscar, amanda)`" is shown by the arrow pointing from "`male(oscar)`" to "`female(amanda)`".

All of the following statements are displayed here, as defined by the Prolog in the assignment brief:

- Oscar is the father of Amanda and Tom
- Owen is the father of Tom and Sheela
- Lora is the mother of Amanda, Tom and Sheela
- Amanda is the mother of Diana and Adam
- Tom is the father of Shaun and Aileen
- Sheela is the mother of Emily

For Office Use Only

2 – Predicate Design

Q2. *Fill in the bodies of the following eleven predicates so that they correctly encode the relationships between the people in the fact section. For the purpsoes of this assignment, sister, brother and sibling should be defined to include persons related to each through either one or both parents (i.e. "sister" inclues half-sister, "brother" includes half-brother, and "sibling" includes both sister and brother).* **[20 marks]**

```
mother(X, Y) :- female(X), parent(X, Y).
father(X, Y) :- male(X), parent(X, Y).
```

The 'mother(X, Y)' and 'father(X, Y)' predicates are very easy to define, as they are simply the 'parent(X, Y)' predicate with gender verification on X – as X is the parent of Y, then to define whether X is a mother or a father requires knowing whether they're female or male, respectively.

```
child(X, Y) :- parent(Y, X).
daughter(X, Y) :- female(X), parent(Y, X).
son(X, Y) :- male(X), parent(Y, X).
```

These three predicates for checking whether one object is a child of another object is exactly the same as the previous three, but with the parameters of the 'parent' predicate check being reversed – as if X is a child of Y, then it stands to reason that Y must be a parent of X. Additional gender verification done on the 'daughter' and 'son' predicates, as before.

```
sister(X, Y) :- dif(X, Y), female(X), parent(B, X), parent(B, Y).
brother(X, Y) :- dif(X, Y), male(X), parent(B, X), parent(B, Y).
sibling(X, Y) :- dif(X, Y), parent(B, X), parent(B, Y).
```

As the assignment brief specified these three in particular should include half-brothers and half-sisters, these predicates are easy to define too – they simply need to check whether X and Y have a parent in common, and then add additional gender verification on X for daughter and son.

There's also the additional check for 'dif(X, Y)', in order to make sure Prolog doesn't pick up the same object as it's own sibling – diana should not be defined as a sister of diana, for instance.

```
uncle(X, Y) :- male(X), parent(B, Y), sibling(B, X).
aunt(X, Y) :- female(X), parent(B, Y), sibling(B, X).
cousin(X, Y) :- parent(B, X), ( uncle(B, Y) ; aunt(B, Y) ).
```

The 'uncle' and 'aunt' predicates simply check whether a parent of Y is also the sibling of X (in addition to the actual gender verification on X), and doesn't require anything else. The 'cousin' predicate expands on this, and states that Y can only be the cousin of X if one of X's parents is also the uncle/aunt of Y.

For Office Use Only

3 – Predicate Evaluation

Q3. *Show that using your rules; you can infer that shaun, aileen and emily are diana's cousins, and that diana and adam are brother and sister. Create some queries that test every predicate, and include these in your report.* **[20 marks]**

Input	Expected Output	Output	Pass
`cousin(diana, shaun).`	`true.`	`true.`	YES
`cousin(diana, aileen).`	`true.`	`true.`	YES
`cousin(diana, emily).`	`true.`	`true.`	YES
`sibling(diana, adam).`	`true.`	`true.`	YES
`brother(adam, diana).`	`true.`	`true.`	YES
`sister(diana, adam).`	`true.`	`true.`	YES

Evaluation

Input	Expected Output	Output	Pass

`mother(lora, X).`	`X = amanda`
	`X = tom`
	`X = sheela`
		`X = amanda`
		`X = tom`
		`X = sheela`
			YES

`mother(amanda, X).`	`X = diana` `X = adam`	`X = diana` `X = adam`	YES
`mother(tom, X).`	`false.`	`false.`	YES
`father(oscar, X).`	`X = amanda` `X = tom`	`X = amanda` `X = tom`	YES
`father(owen, X).`	`X = sheela`	`X = sheela`	YES
`father(amanda, X).`	`false.`	`false.`	YES
`child(amanda, oscar).`	`X = lora` `X = oscar`	`X = lora` `X = oscar`	YES

child(adam, X).	X = amanda	X = amanda	YES
son(adam, X).	X = amanda	X = amanda	YES
son(diana, X).	false.	false.	YES
daughter(emily, X).	X = sheela	X = sheela	YES
daughter(shaun, X).	false.	false.	YES
sister(amanda, X).	X = tom X = sheela	X = tom X = sheela	YES
sister(tom, X).	false.	false.	YES
brother(tom, X).	X = amanda X = sheela	X = amanda X = sheela	YES
brother(amanda, X).	false.	false.	YES
sibling(diana, X).	X = adam	X = adam	YES
sibling(oscar, X).	false.	false.	YES
uncle(tom, adam).	true.	true.	YES
uncle(tom, shaun).	false.	false.	YES
aunt(amanda, shaun).	true.	true.	YES
aunt(amanda, diana).	false.	false.	YES
cousin(diana, shaun).	true.	true.	YES
cousin(diana, adam).	false.	false.	YES

For Office Use Only

4 – The 'relative' Predicate

Q4. *Write a predicate relative(X, Y) that is able to determine correctly whether two people are related, and do it in such a way that you can use the predicate to query the database for all of a person's relatives (the query must terminate, rather than return an infinite sequence of answers). Your predicate should be defined to include only blood relations (people who are related to one another by birth but not by marriage). For example, adam and sheela are relatives (because lora is the mother of both amanda and sheela, and amanda is the mother of adam) but adaman and owen are not relatives.* **[20 marks]**

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(B, Y), ancestor(X, B).
relative(X, Y) :- dif(X, Y), ancestor(X, Y) ; ancestor(Y, X).
relative(X, Y) :- dif(X, Y), ancestor(B, X), ancestor(B, Y).
```

Though this assignment only specified the creation of a 'relation' predicate for testing whether someone is directly related through blood, I was coming across a lot of issues with backtracking and stack overflow when implementing it as a recursive predicate.

To get over this issue, I implemented a second predicate called 'ancestor' that determines whether someone is a parent of Y, and parent of a parent of Y, etcetera. This works recursively, so that it isn't limited to just parents and grandparents.

The 'relative' predicate then takes this and returns true if X or Y is a direct ancestor of the other (i.e. if X is the parent of Y, or Y is the parent of X, or X is the grandparent of Y, or Y is the grandparent of X, etcetera). It also returns true if people have ancestors in common, for example if two people have the same great grandfather then they are related by blood.

Evaluation

Input	Expected Output	Output	Pass
relative(shaun, X).	X can be oscar, lora, amanda, tom, sheela, diana, adam, aileen or emily	X can be oscar, lora, amanda, tom, sheela, diana, adam, aileen or emily	YES
relative(shaun, owen)	false.	false.	YES

The current problem with this is because of the backtracking that Prolog does, resulting in repeated correct possibilities for the value of X when queried. This could be fixed in future iterations through appropriate use of the cut operator '!'. It does still terminate however, so it meets the question's standards.

For Office Use Only

5 – The 'modolo' predicate

Q5. *Write a recursive predicate modolo(X, D, R) that takes as input a positive integer X, a divisor D<=X and returns the remainder R of the interger quotient X/D. For instance, if X = 6 and D = 4, then R = 2. You have to assume that only positive integers are allowed in the inputs and outputs of your function.* **[20 marks]**

```
modolo(X, 0, X).
modolo(X, D, R) :- X < D, R is X.
modolo(X, D, R) :- X >= D, modolo(X - D, D, R), !.
```

The 'modolo' predicate takes an integer X, and gives R the value of the remainder of X divided by D. To do this, two boundary cases need to be defined – what happens if you divide by 0 (in which case the remainder is equal to X) and what happens if X is smaller than the number you're dividing by, in which case the remainder is again equal to X.

The general case for this predicate is simple – if X is larger than or equal to D, it recursively negates the value of D from X until that is no longer the issue, in which case the previous boundary case is found to be true. The use of '!' then cuts it off from trying to find other values, which is appropriate as R can really only ever be one value, as with any mathematical equation.

The use of X >= D as opposed to just X > D (as the question dictates that D <= X) has a reason behind it too – instead of having to define another case, where X would be equal to D or equal to 0 and giving the remainder as 0, this simply negates the value again and goes to the boundary case with X = 0, giving a remainder of 0 anyway without the need for another case to be written.

Evaluation

Input	Expected Output	Output	Pass
X = 7 D = 5	R = 2	R = 2	YES
X = 10 D = 3	R = 1	R = 1	YES
X = 72 D = 3	R = 0	R = 0	YES