Prolog is useful in two situations: when you want to use unification for partial matching of structures, and when you want to search through a space of possible solutions. We haven’t yet got to the point where we’ve got serious tasks that have these properties, so these exercises are just programming exercises to get you used to these ideas, and to generally working with Prolog.

I am strongly suggesting that you use Emacs as the IDE, and that you use SICStus as the Prolog interpreter. School machines provide two interpreters, SICStus and SWI. For the initial exercises it doesn’t matter which you use, but differences may arise later on, so it is best to use the one which will support the later exercises from the outset.

To use Emacs, you need to do a little preparation. You may or may not have a file called .emacs in your home directory. If you’ve got one, add the lines

\begin{verbatim}
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
(autoload 'mercury-mode "prolog" "Major mode for editing Mercury programs." t)
(setq prolog-system 'sicstus)
(setq auto-mode-alist (append '(("\.pl$" . prolog-mode)
                             "\.m$" . mercury-mode))
          auto-mode-alist))
\end{verbatim}

at the end, if you haven’t got one create one and put these lines in it. Emacs will then work as an IDE for Prolog (it also provides quite a nice IDE for Python, and a reasonable one for LaTeX. I never leave it!) (cut and paste this code from the online version of the notes or from prologemacs in /opt/info/courses/COMP24412/PROGRAMS)

Copy /opt/info/courses/COMP24412/PROGRAMS/ancestors.pl into your directory for working on this course and open it using Emacs. You can then consult (≈ compile) it either by typing &C &F or by choosing consult from the ‘Prolog’ option on the menubar.

Exercise 1 carries 0 marks, but will help you think about unification and how Prolog variables work. Exercises 2 & 3 are each worth 6 marks, and question 4 is worth 8 marks. If you get a question partly right you will get some of the marks for it, but you will lose a mark for every singleton variable message that the interpreter produces when you load your program.
1. Unification and identity: this exercise carries 0 marks, but is worth doing to help you reflect on how Prolog variables and terms behave and on what unification does. The key here is to get an understanding of the difference between \( X = Y \) (do \( X \) and \( Y \) unify?), \( X == Y \) (are \( X \) and \( Y \) the same thing?) and \( X \text{ is } Y \) (does \( X \) unify with the value of the arithmetic expression \( Y \)?)

Think about what the following queries will do, and then try them out in the Prolog interpreter. **Think first, try second.** If you just type one in and then say ‘Oh, I wonder why that happened’ you’ll learn less than if you look at it, think about what will happen, and then try it.

If something you don’t expect happens, **stop and think about it.** Whatever it is will happen when you write your own programs, so if you don’t understand it now, when it doesn’t matter, you won’t understand it later, when it will.

```
| ?- X = Y, Z = 10, Y = X. |
| ?- X = Y, Z = 10, Y = Z. |
| ?- X == Y. |
| ?- X == Y, X = Y. |
| ?- X = Y, X == Y. |
| ?- X = Y, Y = Z, Z = a. |
| ?- X = Y, Y = Z, Z = a, X = b. |
| ?- [1 | X] = [1,2,3]. |
| ?- [1, X] = [1,2,3]. |
| ?- [A, B, C] = [H | T], A = 1, C = 3, A = B. |
| ?- X is 3*4. |
| ?- X = 3*4. |
| ?- X = 3*4. |
```
?- X = 3*4, Y is X.

?- X is abs(6-8).

?- X = 9, Y is X*X.

?- 30 is 5*6.

?- 30 = 5*6.

?- 5*6 is 30.

?- 5*6 = 6*5.

?- 31 is 5*6.
2. Backtracking and search: my program for finding out who is whose mother covers some very simple family relations, but there are all sorts of relations that you could calculate—aunt, great aunt, cousin, niece, second cousin, stepfather, . . . [6 marks]

Extend the program ancestors.pl from
/opt/info/courses/COMP24412/PROGRAMS
so that it covers the following:

**Sister** you are someone’s sister if you are female and have the same mother and father.

**Brother** like sister but you have to be male.

**Aunt** you are someone’s aunt if you are female and are their parent’s brother or sister or are married to their parent’s brother or sister.

**Uncle** like aunt, but you must be male.

**Niece** you are someone’s niece if you are female and they are your aunt or uncle.

**Nephew** like niece, but you must be male.

**Cousin** you are someone’s cousin if their parent is your parent’s brother or sister.

**Second cousin** A is B’s second cousin if A and B’s parents are cousins.

You should try your program with the following goals. Some of these should succeed, some should succeed multiple times with different instantiations of the variables, some should fail: you get a mark for each query where the program does what it’s supposed to do. If you get one answer, force Prolog to give you others until it gives up.

| ?- brother(jim, sam). |
| ?- brother(jim, mike). |
| ?- aunt(X, Y). |
| ?- nephew(X, Y). |
| ?- cousin(X, Y). |
| ?- secondCousin(X, Y). |
3. Lists:

(a) Define a predicate which will tell you whether a list contains two or more instances of a single item (clue: look at the definition of `member` in `lists.pl` to remind yourself of how it works. What you want is something which walks along the list looking at the head and asking whether it is a member of the tail):

```prolog
| ?- multielements([a,b,c,d,a]). % should succeed
| ?- multielements([a,b,c,d]). % should fail
```

3 marks

(b) Define a predicate which will join a list of lists together (clue: look at the definition of `append` in `lists.pl` and think about how it works. What you want is something which returns `[]` if you give it an empty list, and which appends its first element to the result of appending all the items in its tail together if not):

```prolog
| ?- multiappend([[1,2,3], [a,b,c], [i]], L).
L = [1,2,3,a,b,c,i] ?
```

3 marks
4. Database:

Suppose I had a collection of places and information about roads that connected them, and that I want to find the shortest route between them.

\[
\text{connected1}(\text{buxton, leek, 11}).
\text{connected1}(\text{buxton, longnor, 9}).
\text{connected1}(\text{longnor, 'earl sterndale', 3}).
\text{connected1}(\text{longnor, leek, 10}).
\text{connected1}(\text{leek, congleton, 6}).
\text{connected1}(\text{longnor, butterton, 4}).
\...
\]

\[
\text{connected}(X, Y, N) :-
\quad \text{connected1}(X, Y, N).
\]

\[
\text{connected}(X, Y, N) :-
\quad \text{connected1}(Y, X, N).
\]

I could find some route by backtracking through all the possibilities, but it wouldn’t necessarily be the shortest route.

To find the shortest route from A to B, find all the places you could get to from A and stick them in the database. So if I’m trying to get from Buxton to Congleton, I start by putting a partial route which ends in Buxton and is of zero length into the database, as

\[
\text{route}([\text{buxton}], 0).
\]

I then repeatedly retract the shortest route currently in the database and extend it to all the places you could get to from its end point. So after the first round I’ve got

\[
\text{route}([\text{leek, buxton}], 11).
\text{route}([\text{longnor, buxton}], 9).
\]

The shortest of these is the one to Longnor, so I retract it and add its possible extensions:

\[
\text{route}([\text{'earl sterndale', longnor, buxton}], 12).
\text{route}([\text{butterton, longnor, buxton}], 13).
\text{route}([\text{leek, buxton}], 11).
\]
The shortest one now is the one to Leek, so I retract it and add its possible extensions:

route([congleton, leek, buxton], 17).
route(['earl sterndale', longnor, buxton], 12).
route([butterton, longnor, buxton], 13).

To do this, you need to

(a) Find the shortest route currently in the database: look for a route, hope that you can’t find another shorter one

(b) Find all the places you can get to from the head of that one: look for somewhere you can get to from X, add that as an extension of this route, tell Prolog you don’t like this answer. When it can’t find any more, move on to the next round. Remember that you can use fail to force Prolog to try alternative solutions, so the key to this bit is to define a predicate called addAnExtension which adds an extended version of the current route, and then have something like

extendCurrentRoute(R) :-
    addAnExtension(R),
    fail.
extendCurrentRoute(_R).

This will call addAnExtension (which you have to define) to put a new extended version of the route into the database. The call of fail will then force addAnExtension to backtrack if it can, in which case fail will be called again, and then ...Eventually there won’t be any more solutions to addAnExtension, at which point it will try the second clause for extendCurrentRoute, which will succeed.

[8 marks: partial marks will be given for partial answers]

One mark will be subtracted for every singleton variable message that your program generates when it is compiled.