This lab is not assessed. It’s there to give you some practice working with Prolog before I ask you to do anything that is assessed because I know that some people initially find it challenging, and I want to get over that initial challenge before I start assigning marks to things. For that reason I have put a lot of small exercises in. Once you feel comfortable with one group of exercises, move on. Doing them all could take some time, and once you’ve got a grip on one topic there’s not much point in doing a million more of the same kind of thing: if you don’t get it first time, then do the second, and if you don’t get it after the second do the third, and ..., but once you have understood one bit move on to the next. If I’ve explicitly said that something is tricky/challenging and you get stuck on it, move on to the next – I’ll go through the solutions in a lecture, and I don’t want you spending hours on things that are just completely defeating you.

I strongly suggest 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

```
(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))
(setq prolog-indent-width 4)
```

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). If you really want to use SWI rather than SICStus then change the line

```
(setq prolog-system 'sicstus)
```

to

```
(setq prolog-system 'swipl)
```
But my later exercises might not work under SWI

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

Unification and identity: this exercise 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 (don’t type the \( | ?- \) bit – that’s just the Prolog prompt asking you to type something.

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.

%% Simple unification examples: reversibility, delay in binding
| ?- X = Y, Z = 10, Y = X.
| ?- X = Y, Y = Z, 10 = Z.
| ?- X = Y, Y = Z, Z = a.
| ?- X = Y, Y = Z, Z = a, X = b.

%% difference between equality and unifiability
| ?- X == Y.
| ?- X == Y, X = Y.
| ?- X = Y, X == Y. %%% What’s the difference between this and the %%% previous one)

%% Lists come in two pieces, even when they look like they don’t
| ?- [1 | X] = [1,2,3].
| ?- [1, X] = [1,2,3].
| ?- [A, B, C] = [H | T], A = 1, C = 3, A = B.

%% Difference between ’is’ (arithmetic evaluation) and unification
| ?- X is 3*4.
Building terms and taking them apart  Again, just some things to try.

Recursion on lists  A lot of programs fit the following pattern: to turn one list into another, have a look at its head, see if you like it, and if so turn it into whatever you want it to be. If you put a cut after the test then it won’t backtrack to stupid solutions

(i) Write a predicate to turn a list of numbers into a list of the squares of those numbers. The test is vacuous, so you can omit it. And since the test is vacuous you don’t actually need the third case. The filter should be a predicate that unifies the square of its first argument with its second argument, e.q.
Prolog programs tend to consist of lots of quite small pieces. A predicate is in many ways like a function/procedure in another language, but you very seldom write a predicate that is more than about 10 lines long. My rule-of-thumb for writing programs in Python or Java is that a function that won’t all fit on the screen at once is probably too long and should be broken up. My rule-of-thumb for Prolog is that a rule that is more than 10 lines long is probably too long and should be broken up.

(ii) Write a predicate to turn a list that contains a mixture of lists and other things into a list that just contains the lists. The test will require a predicate called islist that succeeds if its sole argument unifies with either [] or something that has a head and a tail.

(iii) Check whether a list contains two copies of the same thing:
One way of doing this would be to make a supplementary version of multicopies that has an extra argument, which is to be a list of things you’ve seen so far. Then the first clause of \texttt{multicopies/2} will test whether the head of the list that you’re interested in is a member of the list of things you’ve seen so far, and the second will add the head of the list that you’re interested in to the list of things you’ve seen so far and then recurse down the tail.

Prolog doesn’t mind having predicates with the same name and different numbers of arguments – when it’s trying to prove something it will only use rules whose heads unify with the thing it’s trying to prove, so that will mean that only the version with the right number of arguments will be considered. You \textbf{don’t} have to specify the /2 part of the name \texttt{multicopies/2} (in fact you’re not allowed to!)

(iv) Lists can contain lists (which can contain lists (which can contain lists (...))). Write a predicate to test whether a nested list contains a specified item (the target item could itself be a list):

\begin{verbatim}
?\:- nested(a, [[[b, [a]]]]).
yes
?\:- nested(c, [[[b, [a]]]]).
no
?\:- nested([a], [[[b, [a]]]]).
yes
?\:- nested([b], [[[b, [a]]]]).
no

?\:- nested([X], [[[b, [a]]]])..
X = [[[b, [a]]]] ? ;
X = [b, [a]] ? ;
X = [a] ? ;
X = a ? ;
no
\end{verbatim}

This one needs three cases: are the target and the thing you are comparing with it the same (unifiable), is the target nested inside the head of the thing you are comparing it with, is it nested inside the tail of the thing you are comparing it with?

What happens if you call

\begin{verbatim}
?\:- nested(X, [[[b, [a]]]])
\end{verbatim}

and repeatedly use ; to force it to give you different answers?

(v) Use \texttt{nested} to define a predicate that checks whether two nested lists have some item in common.
There is a slight complication here, because the simplest version of `nested` will say that `[]` is nested somewhere inside any list, and if you allow that then you’ll find that `[x]` and `[p, [q], r]` both contain `[]`. So you’ll have to rule that out.

(vi) We can extend our pattern to handle arbitrary terms.

Challenging task: define a predicate `tmember` which will test whether an arbitrary term contains some target item. My solution to this is 12 lines of code, but it’s tricky to find
it. You’ve got all the machinery you need to do it, but working out what to do with that machinery is not straightforward. It’s a sort of combination of what you did for nested and the filter I outlined just above. You will be doing the recursion on the second argument, and you won’t be building something. If you find that you’ve written half a page of code you’re on the wrong lines – throw it away and start again.

?- tmember(3, f(g([1,2,3]))).
yes
?- tmember(4, f(g([1,2,3]))).
no

**Database**  
(i) The Fibonacci series is defined by the following set of recurrence relations:

1. \( fib(0) = 1 \)
2. \( fib(1) = 1 \)
3. \( fib(i + 2) = fib(i) + fib(i + 1) \)

This looks pretty innocent, but it has two interesting properties: the result grows exponentially as the argument grows, and the time taken grows exponentially if you implement it in the most obvious way:

(i-a) Write a predicate that will calculate \( fib \).

?- fib(2, P).
P = 2 ?
yes
?- fib(3, P).
P = 3 ?
yes
?- fib(4, P).
P = 5 ?
yes
?- fib(5, P).
P = 8 ?
yes
?- fib(6, P).
P = 13 ?
yes
?- fib(10, P).


P = 89 ?
yes
| ?- fib(20, P).
P = 10946 ?
yes
| ?- fib(25, P).
P = 121393 ?

(i-b) It gets perceptibly slower as you increase the size of the argument, and somewhere
around 29 or 30 it just collapses in a heap. Think about why it gets slower.

(i-c) Write version of fib callled fibd which uses the database to store intermediate results,
and see what happens when you call it with largeish numbers:

(slight complication: if you ask Prolog about something for which it has no rules, it’s not
clear what it should do. Should it throw an exception, on the grounds that asking it about
things for which it has no rules is a fairly silly thing to do, or should it just no because you
asked it to do something and it couldn’t do it because it doesn’t have any relevant facts and
rules?

They’re both reasonable things to do. The default behaviour is to throw an exception. If
you want it to just fail (which you will want here) then you should include the statement

:- prolog_flag(unknown, _, fail).

That tells Prolog that if you ask it to do something and it has no facts or rules then it should
just fail)

| ?- fibd(30, P).
P = 1346269 ?
yes
| ?- fibd(40, P).
P = 165580141 ?
yes
| ?- fibd(50, P).
P = 20365011074 ?
yes
| ?- fibd(100, P).
P = 573147844013817084101 ?
yes
| ?- fibd(1000, P).
P = 703303677114228158218352548771835497701812698363587327426049050871545371181969335
This one is hard: think about it, don’t get despondent if you get stuck.

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.

connected1(buxton, leek, 11).
connected1(buxton, longnor, 9).
connected1(longnor, ’earl sterndale’, 3).
connected1(longnor, leek, 10).
connected1(leek, congleton, 6).
connected1(longnor, butterton, 4).
...

connected(X, Y, N) :-
    connected1(X, Y, N).
connected(X, Y, N) :-
    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

route([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

route([leek, buxton], 11).
route([longnor, buxton], 9).

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

route([’earl sterndale’, longnor, buxton], 12).
route([butterton, longnor, buxton], 13).
route([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

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

2. 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.

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, ...

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).
```