MSc Computing Science
Imperial College London

Prolog Assessed Exercise
November 2016

You are required to:
Submit an extension of the program in the Prolog file **ug.pl** according to the specifications (a)-(e), below. Do not include in your submission any definition for the predicate 'route/2'[1] (but feel free to use 'route/2' in the bodies of your clauses and to add your own definitions in your local file for testing purposes).

It is very important that you use predicates exactly as specified (spelling, lower and upper case letters and the number and position of the arguments). Your answers will be auto-tested.

Of course, you can and should define your own auxiliary (helper) predicates in order to reduce the size of predicate definitions. Write comments to explain your program.

You will lose substantial marks if your programs cannot be auto-tested, because you have submitted wrong files or you have used wrong predicate names or you left definitions of 'route/2'. Please copy and paste the given predicate names from the specifications below or from the supplied ug.pl file into your program to avoid misspelling.

Marks will also be deducted for lack of clarity of code and bad formatting.

[1] This means that your program will not work by itself: the auto-tester will add a set of rules to define 'route/2'. So make sure your program works for any 'route/2' configuration and not just for the one given in the skeleton file.

Questions

The given file for this question is ug.pl. Extend it by adding your programs to it and submit it.

The given file contains facts of the form
route(Underground_Line, List_of_Stations).
These facts represent the information that each adjacent pair of stations on **List_of_Stations** is connected by the underground line **Underground_Line**.

For example the fact
route(green, [a,b,c,d,e,f])
indicates that the *green* line connects station *a* to station *b*, *b* to *c*, *c* to *d*, etc.

Remember to **remove all theses facts before submitting** your answer (along with any other facts or rules of the form 'route(UL, LS).' or 'route(UL, LS) :- …' that you might add for testing purposes).

Add definitions for the following relations described in (a)-(e) below to the file. You can define auxiliary relations if you need them. You can use ***findall***, ***setof***, and Prolog conditional and negation operators, if needed, together with any comparison primitives you may need (e.g. <, >), and list predicates, ***member, length, append***, but no other built-in predicates. Do not use the cut, ***!***.

Your programs should work correctly whatever underground stations and connections are given in the form above, and not just for the particular instances given in the file ug.pl.

The questions (a)-(e) below can be done independently of each other. But you are also allowed to use the programs in earlier questions in your answers to later questions.

a. **lines(Station, Lines)**
to mean **Lines** is the alphabetically ordered (low to high) list of all underground lines passing through station **Station**.
If no lines pass through **Station** then **Lines** should be the empty list. You can assume in all calls to **lines Station** is always given (i.e. is ground).

Example:
```
| ?- lines(i, Ls).
Ls = [blue,red,silver] ? ;
no
```

b. **disjointed_lines(Lines)**
to mean **Lines** is the list of all pairs of underground lines that have no station in common. The list **Lines** should have no repetitions. In particular if it contains (line1, line2), for whichever lines line1 and line2, it should not contain (line2, line1). **disjointed_lines** should fail if no pairs of lines exist that share no stations. You can assume that in all calls **Lines** is a variable.

Example:
```
| ?- disjointed_lines(Ls).
Ls = [(yellow,blue),(yellow,green),(yellow,red),(yellow,silver)] ? ;
no
```

The solution Ls = [(blue, yellow),(green, yellow),(red, yellow),(silver, yellow)]
is also acceptable.
The order of the elements in the list Ls is not important.

c. **direct(S1, S2, Line, Route)**
to mean there is a direct route between stations **S1** and **S2** on underground line **Line**, and the route is **Route**. You may make use of the definition of **rev** given in the file ug.pl that reverses lists, i.e. rev(GList, RList), for a given list GList, computes its reverse RList. You do not need to cater for the case where S1 and S2 are the same.

Examples:

```
| ?- direct(a,e, L, R).
L = green,
R = [a,b,c,d,e] ? ;
no
```

```
| ?- direct(e,a, L, R).
L = green,
R = [e,d,c,b,a] ? ;
no
```

```
| ?- direct(a,m, L, R).
no
```

d. **changes(C, PR)**
where **PR** is a 'planned route' and **C** is the list of underground line changes on the planned route **PR**, as follows.
PR should be of the form of a list of tuples each of which is of the form
(Line, List_of_stations). For example PR might be
[(green, [a,b,c,d,e]), (red, [e,i]), (silver, [i,k,m]), (red, [m,n])].
This is a planned route from station *a* to station *n*, by going from *a* to *e* on the green line, then at *e* changing to the red line, and going from *e* to *i*, then changing at *i* to the silver line and going from *i* to *m*, and finally changing to the red line at *m* and going from *m* to *n*.

You can assume that in all calls to **changes** **PR** is ground and is a *correct* planned route, in that a journey is indeed possible following this route in the order of the stations and changes given, left to right. Given such a **PR** then **C** is the list of all the changes in **PR** in the form of a list of tuples of the form (L1, L2, S) indicating that a change is to be done from Line L1 to line L2 at station S. The tuples in **C** should be in the order the changes are to take place.

Note that a correct planned route may be given as, for example
[(green, [a,b]), (green, [b,c,d,e]), (red, [e,i])].
In this case there is no change required at station *b*, i.e. two adjacent tuples involving the same line do not constitute a change.

Examples:

```
| ?- changes(C, [(green, [a,b,c,d,e]),(red, [e,i])]).
C = [(green,red,e)]

| ?- changes(C, [(green, [a,b,c,d,e]),(red, [e,i]), (silver, [i,k,m]), (red, [m,n])]).
C = [(green,red,e),(red,silver,i),(silver,red,m)]
```

```
| ?- changes(C, [(green, [a,b]), (green, [b,c,d,e]),(red, [e,i])]).
C = [(green,red,e)]
```

e. **noReps(PR)**
to mean that **PR**, which you can assume is ground and a correct planned route in any call to **noReps**, does not mention any underground line in two different places.

Examples:

```
| ?- noReps([(green, [a,b]), (green, [b,c,d,e]),(red, [e,i])]).
no
| ?- noReps( [(green, [a,b,c,d,e]),(red, [e,i]), (silver, [i,k,m]), (red, [m,n])]).
no
| ?- noReps( [(green, [a,b]),(blue, [b,c,h,i]), (silver, [i,k,m]), (red, [m,n])]).
yes
```

Questions a- e have 10% , 25%, 25%, 25%, 15% of the total marks, respectively.

You can also try question f, below, for fun.

f. **jp(S1, S2, PR)**
Here stations **S1** and **S2** are assumed given (ground in the call to **jp**, journey planner), and your program should generate a correct planned route **PR** (in the sense of question d, above) that will use each underground line at most once. For this part you can assume, for simplicity, that journey is possible only in the order of the stations given in the **route** facts. For example it is possible to go from *a* to *b*, but not from *b* to *a*.

Examples:

```
| ?- jp(a, f, X).
X = [(green,[a,b,c,d,e,f])] ? ;
no

| ?- jp(a, m, X).
X = [(green,[a,b]),(blue,[b,c,h,i]),(silver,[i,k,m])] ? ;
X = [(green,[a,b]),(blue,[b,c,h,i]),(red,[i,m])] ? ;
X = [(green,[a,b,c]),(blue,[c,h,i]),(silver,[i,k,m])] ? ;
X = [(green,[a,b,c]),(blue,[c,h,i]),(red,[i,m])] ? ;
X = [(green,[a,b,c,d,e]),(red,[e,i,m])] ? ;
X = [(green,[a,b,c,d,e]),(red,[e,i]),(silver,[i,k,m])] ? ;
X = [(green,[a,b,c,d,e,f]),(silver,[f,i,k,m])] ? ;
X = [(green,[a,b,c,d,e,f]),(silver,[f,i]),(red,[i,m])] ? ;
no
```

The order in which you get the different answers is not important.