Question 1:

multiple_lines(S):- stop(X,_,S), stop(Z,_,S),\+Z=X.

For multiple lines, I used two stops making sure the line colour is different (\+Z=X). The value of the stops doesn't matter but the letter S is the same for the letter of the stations being the same. The reason I did this is for a stop to have multiple lines it must have a different coloured line with the same station.

Question 2

notmaximum(C,N,L):- stop(C, N,L), stop(C,F,E), N<F.

notmaximum defines a stop where there is at least 1 stop with a larger number, hence N<F.

notminimum(C,N,L):- stop(C, N,L), stop(C,F,E), N>F.

notminimum defines a stop where there is another stop at least 1 smaller number, hence N>F.

termini(C,S1,S2):- stop(C,R,S1), \+notminimum(C,R,S1).

This termini defines the S1, this uses negation of notminimum meaning it is the minimum value.

termini(C,S1,S2):- stop(C,R,S2), \+notmaximum(C,R,S2).

This termini defines the S2, this uses negation of notmaximum meaning it is the maximum value.

Question 3

highestvalue(C,VALUE):-termini(C,1,S2),stop(C,VALUE,S2).

highestvalue returns the VALUE of the line that is the highest. S2 in the termini, defined earlier to return the maximum value.

orderednumlist(L,List):-highestvalue(L,Value), numlist(1,Value,List).

Orderednumlist returns a list of [1..n] n. n is Value, which is the highest value of line L.

list_stops(C,List):- orderednumlist(C,Olist), maplist(stop(C),Olist,List).

list_stops maps the Olist of [1..n] to stops and returns them as a List of the stations.

Question 4

edge(C, X, Y):- stop(C,M,X), stop(C,N,Y), N>M.

I used the example of pathBuilder in the workshop but I had to make many changes, the first being edge, this had to be redefined as a stop on the same line (C) with different values (M, N). N is larger than N because you can't move back on the line. X and Y are the start station and destination respectively.

path(X,Y,Path):-pathBuilderHelper(X,Y,[],Path).

This calls the pathBuilder helper with a blank list as the VISITED.

pathBuilderHelper(X,Y,VISITED,Path):-edge(C,X,Y),Path = [segment(C,X,Y)],segment_adds_cycle(segment(C,X,Y),VISITED).

edge(C, X, Y) makes sure the X and Y are on the same line and the value of Y is larger than X. The path list adds the segment(C,X,Y), segment_adds_cycle takes in the current segment, and the list of visited segments and compares their stations traversed, if there are any clashes it returns false this prevents the builder visiting a station more than once in a single path.

pathBuilderHelper(X,Y,VISITED,Path):- edge(C,X,Z),segment_adds_cycle(segment(C,X,Z),VISITED), pathBuilderHelper(Z,Y,[segment(C,X,Z)|VISITED],ZYPath),Path=[segment(C,X,Z)|ZYPath],\+ member(segment(C,_,_),ZYPath).

edge(C, X, Z) is used if there isn't an edge(C, X, Y). segment(C,X,Z) is then checked with VISITED in segment_adds_cycle this will fail if the visited path already contains any stations between X and Z. pathBuilderHelper is called again replacing X with Z, segment(C,X,Z) is added to VISITED and ZYPath. The final check is that the line isn't visited twice in one path, this is done by checking if there is a member of ZYPath containing segment(C,_,_).

segment_edge(C,X,Y):- stop(C,M,X), stop(C,N,Y), N is M+1.

This is defined for the segment_to_path_helper. This finds 2 stations that the two stations X and Y are next to each other. The value of one is exactly 1 more than the other

segment_to_path(segment(C,X,Y),Path):- segment_to_path_helper(C,X,Y,Path).

segment_to_path calls the segment_to_path_helper with the input of a segment(…).

segment_to_path_helper(C,X,Y,Path):- segment_edge(C,X,Y),Path = [X].

segment_to_path_helper here checks of X and Y are a segment_edge and if they are then Path with X added is returned to the Segment. This helper doesn't include the last station of the segment as they are added in the path at the beginning of the segment after it, the reason I chose to do this is to prevent duplicate stations.

segment_to_path_helper(C,X,Y,Path):- segment_edge(C,X,Z), segment_to_path_helper(C,Z,Y,ZYPath), Path=[X|ZYPath].

This finds a segment_edge that is 1 station closer to the Y from X and appends it to the list called Path. It then calls the helper until the segment_edge is Y.

stations_traversed(Path,Set):- maplist(segment_to_path(),Path,Setlists), append(Setlists,Set).

This maps the list Path (Path being a list of segments) to the segment_to_path previously mentioned. The append puts all the individual segment lists into one big list. Set is returned which is the list of stations visited in the path.

stations_traversed(segment(C,S1,S2),Set):- segment_to_path(segment(C,S1,S2),Set).

This does the same as the function above but with a segment input instead of a path. There is no need to append at the end as there is only one segment not a list of them.

compare_list(List1, List2):-intersection(List1,List2,X), \+dif(X,[]).

Compare_list creates a list of the intersecting elements called X and then the list X is compared to an empty list to see if they are different. If they are the same the negation means the compare_list will return true but if they are different then it will be false.

segment_adds_cycle(segment(C,S1,S2),Path):- stations_traversed(segment(C,S1,S2),Segmentset), stations_traversed(Path,Listset), compare_list(Segmentset,Listset).

This takes in a segment and a path and determines whether or not the segment can be added to the path. The segment and the path both have their stations_traversed and then the two lists are input into compare_list this will come out false if there are clashing stations in the two lists and true if they both have unique stations.

Question 5

minimum_line_changes(S1,S2,Smallest):-findall(List,path(S1,S2,List),ListofLists), smallest_path(ListofLists,Smallest).

Minimum_line_changes returns the number equal to the fewest line changes needed to go from S1 to S2. This is done using a findall for the paths from S1 to S2 then returned in a ListofLists. This is then put into smallest_path() to find the smallest value of the number of changes.

smallest_path(List,Size):- maplist(length(),List,Sizes),min_list(Sizes,Size).

A list is input, and a value is returned. The list is mapped to finding the lengths of each list and then put into a list called Sizes, once this list is found the smallest value in the list Sizes is found using min_list() and returned as Size.

minimum_path(S1,S2,Path):- minimum_line_changes(S1,S2,Linechanges), path(S1,S2,Path), length(Path,X), X==Linechanges.

Minimum_path calls the minimum_line_changes to get the value of the fewest line changes, then calls path for all the paths from S1 to S2. The length of each path is determined as X and the final check is that if X is equal to Linechanges (the value of fewest line changes).