**STATEMENT OF NON-PLAGIARISM**

I hereby declare that all information in this assignment has been obtained and presented in accordance with academic rules and ethical conduct and the work I am submitting in this document, except where I have indicated, is my own work.

**Name:** Joanna Elaijah Talvo

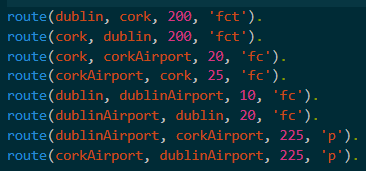
**Student Number:** 18342523

**Course:** CASE 2

**Date:** 9/12/2019

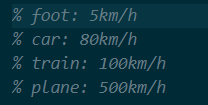
**CA208 Logic Assignment**

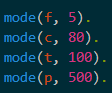
**GIVEN FACTS**



*Figure 1*

*route(Source, Destination, Distance, Mode)*





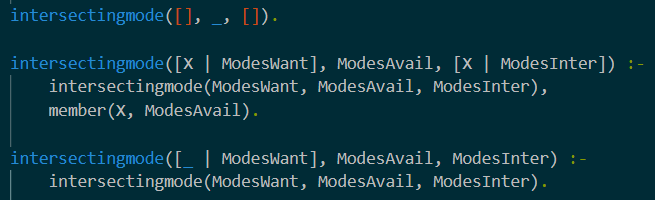
*Figure 2*

*f = foot – 5 km/h*

*c = car – 80 km/h*

*t = train – 100 km/h*

*p = plane – 500 km/h*

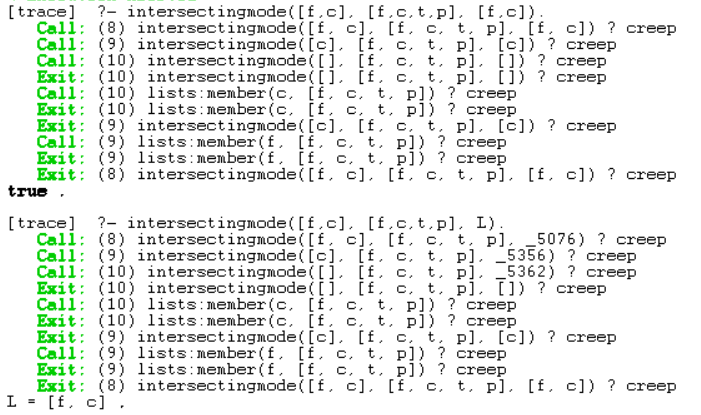


*Finding the intersecting modes.*

The **intersectingmode()** predicate gets the list of modes (ModesInter) that are both in the list of the modes I want to use (ModesWant) and the given modes available (ModesAvail) in that route (Figure 1). This predicate takes in 3 arguments; first being the list of modes we want to use, second being the list of modes available in the given route (Figure 1), and the third is the list of modes that are both in the first and second list.

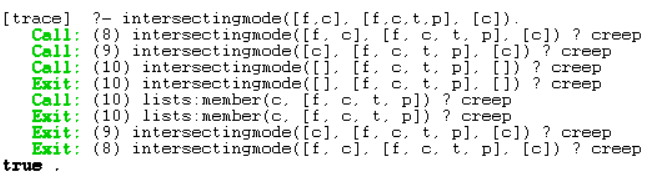
The first case is the base case. This shows that if the first list is empty, no matter what the second list has, the third list is always going to be empty since there is no element in common with the second list.

The second case is one of the recursion cases in this predicate. This checks if the heads of the ModesWant and ModesInter lists are the same and check if head is a member of the ModesAvail list. I used the built-in function *member,* which checks if the element is a member of the list. This goes on until there is no more element to check (base case is reached). Using trace, I am able to check my code in an elaborate way. So whenever I call the predicate intersectingmode recursively, it scans which case among the cases satisfies the current condition.

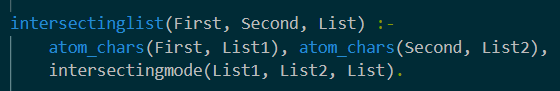


*Tracing the intersectingmode predicate recursion case 1*

The third case is also a recursion case but with different approach. The anonymous variable “\_” is used in case the head of the ModesWant is not the same as the head of the ModesIntersect. Using recursion, same as the second case, it scans which case satisfies the current condition. From the example below, when c is seen, it goes to the second case since the head of the ModesWant is the same as the ModesInter which is the second case is for. The loop stops when it reached the base case.

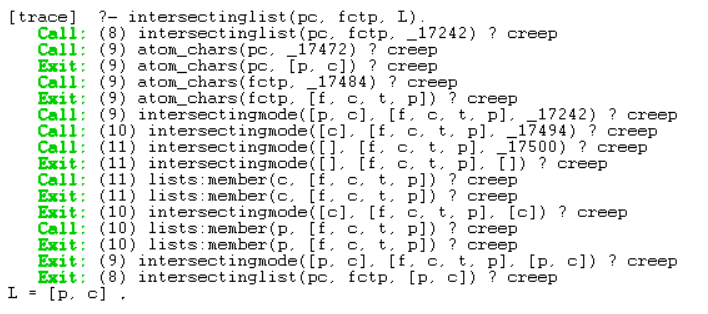


*Tracing the intersectingmode predicate case 2*

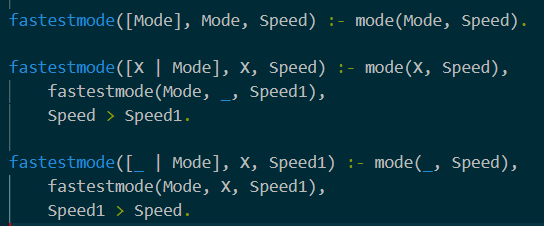


*Converting strings into list*

The **intersectinglist**() predicate reads strings and add them into a list. This predicate takes in 3 arguments; first and second both contain a string and list is just a list. The built-in function *atom\_chars* converts strings into a list. In the predicate intersectingmode, we have to input a list which needs a bit of effort to type in and could also be a way of getting an error. With intersectinglist, strings can be used as in input and it will do the work to change it into a list. This predicate also calls the intersectingmode in which checks if the modes we want are in the modes available (See above). The other reason why I decided to create this predicate is because later on, in our journey predicate, the input for modes is going to be a string and not a list.



*Tracing intersectinglist predicate*

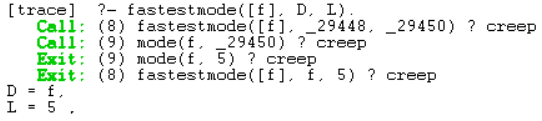


*Getting the fastest mode*

The **fastestmode()** predicate gets the fastest mode among the input of modes. I called the **mode()** predicate (Figure 2) in order to get the average speed each mode and from this, we can deduce which has the fastest speed. If we were to look at it, plane would be the fastest mode since it’s average speed is 500 compared to the other modes.

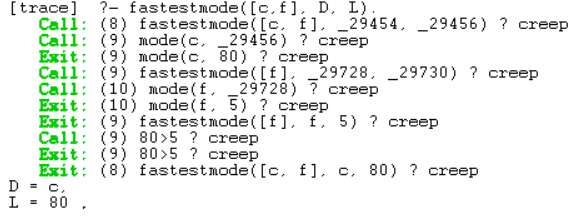
This predicate takes in 3 arguments; first is the list of modes we input, second gives the fastest mode and the last one gives the speed of the fastest mode. There are also 3 cases for this predicate.

The first case is the base case which basically says that if there is only one element in the input list, then it is the fastest mode. As said above, mode predicate gives the speed of the mode.



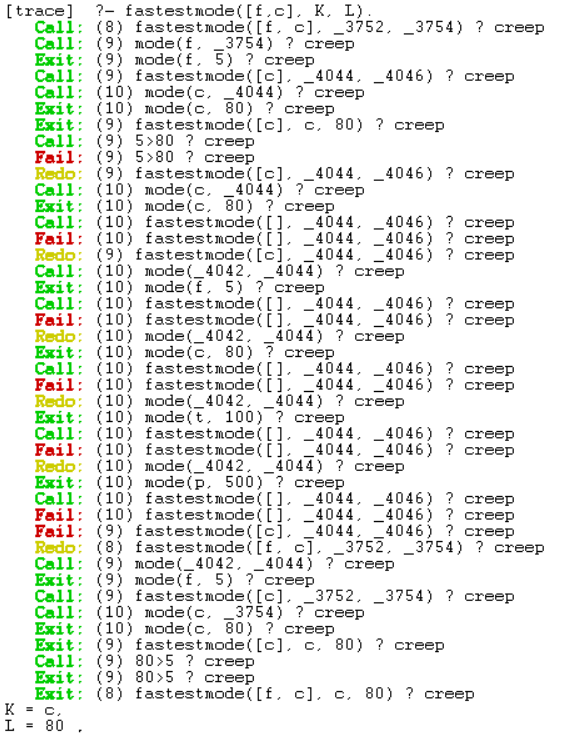
*Tracing fastestmode case 1 (base case)*

The second case is recursive case. This condition is true if the head of the list has a greater speed than the next element. From the tracing below, it gives us the idea how this code works. Speed is the speed of the first element/ head of the list and Speed1 is the speed of the next element. Once all elements have been checked (base case is reached), we can check which has the greater speed using “Speed > Speed1”. I called Speed in the head of the predicate since I want to get the result of the higher speed.



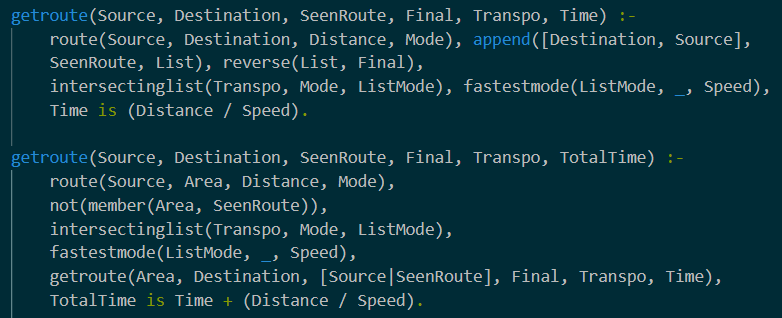
*Tracing fastestmode case 2 (recursive case)*

The third case is also recursive. However, unlike the second case, this checks if the mode that has a fastest speed is along in the list. Anonymous variable “\_” is used again since we are unsure which element has the fastest speed. This case is needed in case the second case fails i.e. the head of the list isn’t the fastest mode. So to go through the code, we check the speed of the first element using mode predicate and its speed being “Speed”. The predicate is called again and so the next element is checked. This time the next element’s speed is in Speed1. Once the base case is reached, only then we can compare their speeds. Since I called Speed1 in the head of my predicate, the mode with a faster speed will be shown as a result.



*Tracing fastestmode case 3 (recursive case)*

\*the reason why the trace is quite long for this case is because Prolog went over to check if the condition satisfies the second case since it goes before the third case. It went through all modes until it gets to the third case. The result still gives the right answer.



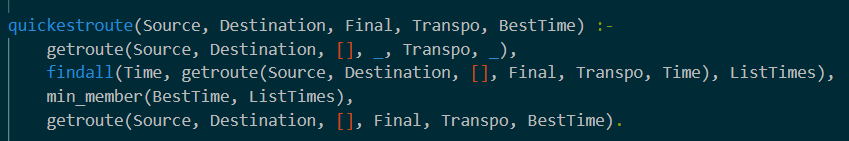
*Getting Routes*

The **getroute()** predicate gets all the possible routes that can be taken from one place to another. I consider this as my main predicate since most of the other predicates I already mentioned are called in this predicate. This predicate takes in 6 arguments. Source, Destination and Transpo arguments are all have to be inputted. SeenRoute is an empty list. Final gives the list of all possible routes. Time gives the time it takes to travel from one place to another (including all the interconnecting routes).

The first case is the base case. This works when there is a direct route from the source to destination. The route predicate is called in first. This predicate has 4 arguments; source, destination, distance and mode (See Figure 1). We use this predicate for us to verify if the inputs we made exist. I used the built-in function append to add the source and destination into List. The problem here is that the routes are appended the other way around so I had to call a built-in function called reverse to get the output of the list the way I wanted. SeenRoute is just an empty list. Intersectinglist as mentioned earlier, converts string into list and checks for the intersecting modes between the first and second list and the result will be put in the third list. Since now we have the intersecting modes, we can use this to get the fastest mode. Fastestmode is now called to get which mode among the list of intersecting modes (from intersectinglist predicate) has the quickest speed. Time is equivalent to distance, from the route predicate, divided by the speed, from fastestmode predicate.

The second case is the recursive case. Now this is for when there is no direct route from one place to another. The approach is clearly different from the base case. In the route predicate, I used a variable called Area which serves as a destination and a source. For example, Dublin to Cork, Cork to Cork Airport. Cork serves as a destination for the first route and a source for the other one. Then, I check if Area is not a member of SeenRoute. The reason is because later on when we call the getroute() predicate, we want to add the source into the SeenRoute list as the head of the list as to keep track which places have we already been to. Intersectinglist and fastestmode are called. And then we call the getroute predicate again. As I have mentioned, the source from the previous route is added as the head of the SeenRoute list just to keep track of places already taken. The rest just works as the same. TotalTime is the time taken for all routes travelled. The Time + (distance / speed) for each route is summed up to get Total Time.

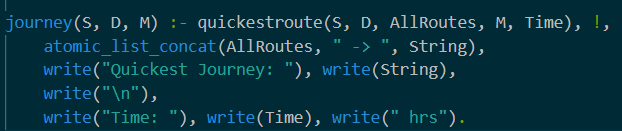
This predicate does not get the quickest route yet despite of me calling the fastest mode predicate. This gets me all the possible routes and the travelled time taken. Now, I have to get the quickest time from all those possible routes in order to get the quickest journey.



*Getting the quickest journey*

The **quickestroute()** predicate is what I consider my final predicate. This gets the quickest journey and the time is takes to get from the source to destination. This predicate takes in 5 arguments; Source, Destination, Transpo are all needed to be inputted. Final is the result of the quickest journey while BestTime is the time taken of the quickest journey.

First, I called in getroute predicate to get me Source, Destination and Transpo. The rest of the arguments are anonymous “\_” since I’m not using them for the mean time. Then I used the built-in function findall which takes in 3 arguments; a generating template which is what you want to search for, a matching template which where you want to search through and a list of generated predicates i.e. the result you searched for. I want to get all the times so I put Time as my generating template and since getroute() gets me all the times of possible routes, I used it as my matching template and ListTimes is where all the times will be put into. Next, I used another built-in function min\_member() to check which member in the list has the smallest value. BestTime is the smallest value in the ListTimes list. Now that I have the quickest time, I have to match that time to its taken route. I called the getroute predicate again and now I have to mention Final which has the list of all the routes in order to match the time of that route to the BestTime.



*Journey predicate*

The **journey()** predicate takes in 3 arguments; source, destination and mode which are all have to be inputted. Since quickestroute gives me the quickest time and quickest journey, I have to call it into journey. In quickestroute predicate, I used the same variable as the ones in journey (S, D, M) to test those inputs into my code. I used the built-in function atomic\_list\_concat which converts list into strings. It takes in 3 arguments; a list, a separator and the generated string. AllRoutes is list of routes and I want to display the routes into strings and not a list to look nicer. I used the “->” as a separator. I used another built-in function write which just prints the arguments inside it. I want to display the string of the quickest route and quickest time. I used a cut to avoid backtracking. The result is shown in the photo below.

