Computer Science 324 16 March, 2007
St. George Campus University of Toronto

Homework Assignment #5
Prolog Warm-up Assignment
Due: Thursday, 29 March, 2007, 1:00 PM

Important Note: Late assignments will **not** be accepted after Saturday March 31 at 1:00 PM.

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.*

Total Marks: There are 100 marks available in this assignment. This assignment represents 5% of the course grade.

Handing in this Assignment
What to hand in on paper: No paper submission is required for this assignment.

What to hand in electronically: In addition to your paper submission, you must submit your code electronically. The predicates (including helper predicates) for each question are to be submitted separately, with the filename as stated in the question. To submit these files electronically, use the CDF secure website:
`https://www.cdf.utoronto.ca/students`

Warning: marks will be deducted for incorrect submission.

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact predicate names and argument(s) (including the order of arguments) specified,
- use the exact file names specified in the questions,
- not load any file in any of your submitted files,
- not display anything but the predicate output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Marking Questions will be both automarked and inspected manually. Marks will be awarded for program correctness as well as for style and documentation. Note that we may use a *different* flight network than the one given here to automark your code. Your solutions are expected to work for *any* legal flight network configuration.

Clarification Page and Newsgroup Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 5 Clarification page, linked from the CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

The Flight Scheduling System

This assignment is a warm-up to get you used to programming in Prolog and to thinking as a logic programmer. The assignment requires you to write a series of Prolog predicates for a *flight scheduling system*. You may use helper predicates as needed in defining these predicates. You may **not** use any of the following:

; (“or”)
→ (“if-then”)
!(cut)

or any other special operators in Prolog (which generally subvert the pure logic programming paradigm). If you are in doubt about what’s allowed, check the Assignment Clarification page or email the newsgroup; if you haven’t seen a certain notation in class or tutorial, it’s probably not a good idea to use it without checking with me or the TAs.

A flight network consists of a set of airport-cities and a set of flights connecting these airports. We shall assume that each airport has an airport tax and a minimum security delay in minutes associated with it. Similarly, each flight is associated with a particular airline and has a price and a duration measured in minutes. We shall make a few, sometimes counter-intuitive, assumptions about our system:

1. We will not model departure/arrival times for the flights and simply assume that it is always possible to take a flight at any point in time.
2. We will assume that flights are always *two-way* or *reversible* so that if an airline has a flight from city *A* to city *B*, then the same airline also has a flight from city *B* to city *A*.
3. We will assume that airport taxes are paid at the origin’s airport and at every airport where there is a change of airline.
4. We shall assume that a flight network gives *complete* information about all the flights available. This means that if a flight is not described in the network, then it does not exist.[1]

Prolog Notational Conventions: Predicate and Mode Spec

We shall use the following notation when *referring* to Prolog predicates: Predicates in Prolog are distinguished by their name and their arity. The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `append/3` specifies the predicate named “`append`” that takes 3 arguments.

Sometimes, we may be interested in specifying how specific predicates are meant to be used. To that end, we shall present a predicate’s usage with a *mode spec* which has the form: `name(arg1, ..., argn)` where each `argi` denotes how that argument should be instantiated when a goal to `name/n` is called. `argi` has one of the following forms:

+ArgName This argument should be instantiated to a non-variable term.

-ArgName This argument should be uninstantiated.

?ArgName This argument may or may not be instantiated.

For example `delete(+List,?Elem,?NewList)` states that, when using `delete/3`, the first argument should be instantiated whereas the second and third arguments may or may not be instantiated.

Note that these Prolog notational conventions provide a convenient way to *specify* Prolog predicates and their usage. They do not represent in any way the form of your actual code. E.g., when defining predicate `wellFormedPath/1` in Question 3, do not use `‘‘+Path’’` as the argument in your code.

[1]This assumption is usually referred to as the “*closed-world assumption*”.

Do's and Don'ts

1. You should not use any special Prolog/SWI-Prolog features like `assert, retract, arg, ...` Put differently, the only predicates you may use apart from the ones you implement yourself are `not/1, append/3, member/2, length/2 'is'`, arithmetic symbols (`'=', '+', '-', '*', '/'`), the symbols `'\=', ';'`, and of course `','`. Of course you may also use output predicates like `writeln/1` for testing and debugging, but these have to be removed prior to submission. If there is another built-in predicate you wish to use, please consult the newsgroup before using it.

2. Avoid *singleton* variables! A singleton variable is a variable that only occurs once in the arguments or the body of a predicate and is thus useless. For example in the following two definitions:

   ```
   doit( X, Y, Z) :- Y is X*2.
   doitagain( X, Y ) :- Z = doesitmatter, Y is X*2.
   ```

   `Z` is a singleton variable and should be prefixed by an underscore (`_Z`) or removed entirely if possible. Note that it usually won't be possible to simply remove a singleton argument as the arity of the predicate is often fixed:

   ```
   times( [], [], Z) :- !.
   times( [H1|L1], [H2|L2], Z ) :- H2 is H1*Z, times( L1, L2, Z).
   ```

   Here in the first definition `Z` is singleton but cannot be removed. Therefore we should replace it with '`_Z`' or just '`_`'.

   Although not harmful, it is useful not to have any singleton variables to keep the code easy. SWI-Prolog will point out any singleton variables you have. This is very useful information because it often helps you finding typos, a common source of bugs in Prolog. For instance if we want to increase a number we could write:

   ```
   inc( FirstNr, Result ) :- Result is Firstnr+1.
   ```

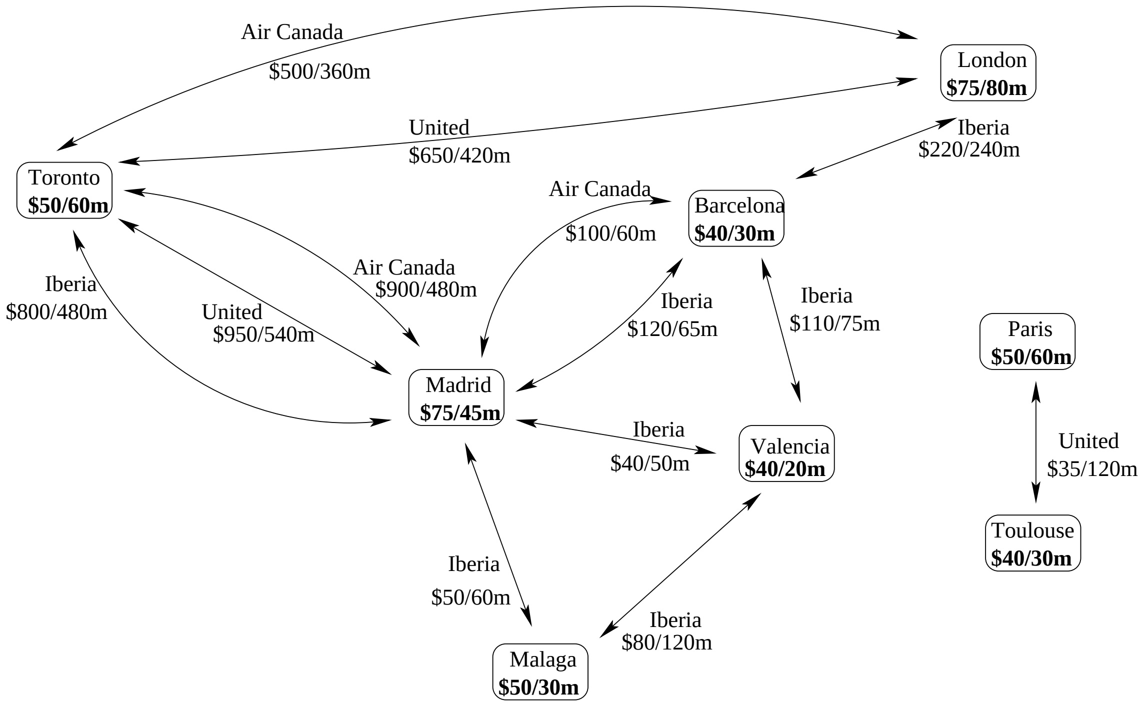
   Here both `FirstNr` and `Firstnr` are singletons and SWI-Prolog will tell you so, which in turn will make you realize that you have a typo (`Firstnr` should be spelled with a capital 'N').

   Marks will be deducted for each singleton you have in your code!

Question 1. (15 marks) The Database.

Submit your database in a file called **a5-flightdb.pl**

A database can be naturally represented in Prolog as a set of facts. In this exercise, you are to create a Prolog database representing a particular flight network. The network in question is depicted by the following graph:



Each node denotes an airport-city with its corresponding tax and minimum security delay. Each link denotes a flight and is labelled with its corresponding airline name, price, and duration. You should change any names to lower-case letters (e.g., `madrid`) and remove spaces in names (e.g., `aircanada`).

We will assume that a flight database is stored in Prolog using a predicate `flight/5`, and possibly a helper predicate to avoid writing redundant information. The tax and security delay at each airport can be stored in Prolog using a predicate `airport/3`.

Required interface for the flight database:

a) the query

```
?- flight(From, Airline, To, Price, Duration).
```

must produce all the possible flights *in both directions* with the price in dollars and the duration in minutes. Note that your database should only list each flight link once.

b) the query

```
?- airport(City, AirportTax, MinSecurityDelay).
```

must produce all the cities with the airport tax in dollars and the minimum security delay in minutes.

Question 2. (20 marks) Queries and Rules.

Submit your answers to this question in a plain text file **a5-q2.txt**.

Assuming flight/5 and airport/3 given, write Prolog clauses (facts, rules or queries) to express the following sentences:

(a) Is there a flight from Toronto to Madrid?

(b) A flight from city *A* to city *B* with airline *C* is cheap if its price is less than $400.

(c) Is it possible to go from Toronto to Paris in two flights?

(d) A flight from city *A* to city *B* with airline *C* is preferred if it's cheap (see (b)) or it's with Air Canada.

(e) If there is a flight from city *A* to city *B* with United, then there is a flight from city *A* to city *B* with Air Canada.

Question 3. (20 marks) List Manipulation.

Submit your code in a file called **a5-listman.pl**

In this question you are to write a predicate that tests whether a fight path is well-formed and another predicate to reverse a flight path. We first define what a flight leg and a flight path are.

Flight Leg

A *flight leg* is described by a triple list of the form `[City1,Airline,City2]` where `City1` is the city where the leg commences, the *origin*; `Airline` is the airline used for the leg; and `City2` is the city where the leg ends, the *destination* of the leg.

Flight Path

A *flight path* is a possibly empty list of (consecutive) legs. For example, the following list represent a path that starts in Toronto and finishes in Valencia:

```
[[toronto,aircanada,madrid], [madrid,iberia,barcelona], [barcelona,iberia,valencia]] (1)
```

Well-formed Flight Path

A flight path is *well-formed* or *legal* if the destination city of each leg in the path matches the origin city of the subsequent leg in the path. The above path is well-formed but the following one is **not**:

```
[[toronto,aircanada,madrid], [valencia,iberia,barcelona], [barcelona,iberia,valencia]]
```

Note that the empty list `[]` is considered to be a well-formed path.

a. (7 marks) Write a Prolog predicate `wellFormedPath(+Path)` that holds if `Path` is a *well-formed* flight path, where `Path` is always instantiated to a flight path.

For example,

```
?- wellFormedPath([[toronto,aircanada,madrid]]).
YES

?- wellFormedPath([[toronto,aircanada,madrid],[valencia,iberia,barcelona]]).
NO

?- wellFormedPath([[toronto,aircanada,madrid],[madrid,iberia,barcelona]]).
YES
```

b. (13 marks) Write a Prolog predicate `reversePath(+Path1,?Path2)` that holds if `Path2` is the *reverse* flight path of `Path1`. That is, `Path2`'s origin is `Path1`'s destination, `Path2`'s destination is `Path1`'s origin, and `Path2` uses the same airlines and intermediate cities in each leg as `Path1` does. For example,

```
?- reversePath([[toronto,aircanada,madrid],[madrid,iberia,barcelona]],X).
X = [[barcelona,iberia,madrid],[madrid,aircanada,toronto]];
NO

?- reversePath([[toronto,aircanada,windsor]],[[windsor,aircanada,toronto]]).
YES

?- reversePath([[toronto,aircanada,windsor]],[[windsor,united,toronto]]).
NO
```

Question 4. (45 marks) Tools for the flight system.

Submit your code in a file called **a5-ftools.pl**

In this question again assume that a database of flight/5 and airport/3 is given. This might, however, not be the same database as the one you have defined in Question 1. Your predicates should work with any well-defined flight database. Plese do not copy the database from Question 1 into any of the other files that you submit, nor load the database into them. If you do so, marks will be deducted for incorrect submission.

Recall our well-formed flight path from the previous page:

```
[[toronto,aircanada,madrid],[madrid,iberia,barcelona],[barcelona,iberia,valencia]] (1)
```

We can associate a *total price* and a *total duration* to every possible flight path. We may also be interested in the *number of different airlines* that are used by a flight path.

- The total *price* of a flight path, measured in dollars, is the sum of each leg's price, the tax at the initial airport, and the taxes at each airport where there is a change of airline. For example, the total price of path (1) is calculated by adding the price of each of the three legs and the airport taxes at Toronto and Madrid (notice there is no tax charge at Barcelona since there is no change of airline).
- The total *duration* of a path, measured in minutes, is the sum of the duration of each leg plus the security delay at each airport visited (including the initial origin airport and excluding the final destination airport). For example, for the path (1) above, the duration is calculated by adding the duration of the three corresponding legs plus the security delays at Toronto, Madrid, and Barcelona.
- The *number of (different) airlines* is the number of distinct airlines used by the path in question. For example, the above path (1) uses 2 different airlines, namely, Air Canada and Iberia. Observe that the following path also uses only 2 different airlines:

```
[[toronto,iberia,madrid],[madrid,aircanada,barcelona],
 [barcelona,iberia,london],[london,aircanada,toronto]]
```

Note that in order to calculate the total price and duration of a path, one needs to query the flight database using both predicates `flight/5` and `airport/3`. On the other hand, the total number of different airlines can be simply obtained by inspecting the flight path in question without using the underlying database.

We now define a complex structure referred to as a *flight route*.

Flight Route

A (detailed) *flight route* is a list of the following form:

`[Cost, Duration, NoAirlines, Path]`

where `Path` is a *non-empty* and *well-formed* flight path, and `Cost`, `Duration` and `NoAirlines` stand for the total price, total duration, and number of different airlines of the flight path in question, respectively. For example, the following list term represents a flight route that has a total price of $1255, a total duration of 755 minutes, uses 2 different airlines, and whose actual path is the above path (1):

```
[1255,755,2,[[toronto,aircanada,madrid],[madrid,iberia,barcelona], [barcelona,iberia,valencia]]]
```

a. (20 marks) Write a Prolog predicate `noAirlines(+Path,?N)` that holds if `N` is the number of different airlines used in well-formed flight path `Path`. For example,

```
?-noAirlines([[toronto,aircanada,madrid],[madrid,iberia,barcelona],[barcelona,iberia,valencia]],N).
N = 2;
NO

?-noAirlines([[toronto,iberia,madrid],[madrid,iberia,barcelona], [barcelona,iberia,valencia]],1).
YES

?-noAirlines([[toronto,united,madrid],[madrid,iberia,barcelona], [barcelona,iberia,valencia]],3).
NO
```

b. (25 marks) Write a Prolog predicate `addLeg(+Route,+To,+Airline,-NewRoute)` that adds a new leg to the end of the path of an existing (non-empty) flight route. `addLeg/4` takes as input a flight route `Route`, a destination city `To` and an airline `Airline`. It produces a forth argument `NewRoute`, which is the route obtained by inserting a new leg to the end of the path in `Route`. The new leg's origin is the final destination of the existing path in `Route`, the new leg's airline is `Airline` and the destination of the new leg is `To`. You may assume that the new leg that you are constructing is one that exists.

As an example of how `addLeg/4` should work, assume we start with the following route from Toronto to Madrid:

`[950,540,1,[[toronto,aircanada,madrid]]]`

Suppose we want to add to this route a flight to Barcelona using Air Canada. From the above flight network, we know that the the leg `[madrid,aircanada,barcelona]` has a cost of $100 and a duration of 60*min*. Also, we know that the security delay at Madrid is 45*min*. Then, we can have the following Prolog query:

```
?-addLeg([950,540,1,[[toronto,aircanada,madrid]]],barcelona,aircanada, X).
X=[1050,645,1,[[toronto,aircanada,madrid],[madrid,aircanada,barcelona]]] ;
NO
```

Suppose further that we want to add to the resulting new route a flight to Valencia using Iberia. Again, from the above flight network, we know that the the leg `[barcelona,iberia,valencia]` has a cost of $110 and a duration of 75*min*.. We also know that the airport tax at Barcelona is $40 and the security delay is 30*min*.:

```
?-addLeg([1050,645,1,[[toronto,aircanada,madrid],[madrid,aircanada,barcelona]]],valencia,iberia,X).
X=[1255,755,2,[[toronto,aircanada,madrid],[madrid,aircanada,barcelona],[barcelona,iberia,valencia]]];
NO
```