Imperial College London – Department of Computing

# 276: Introduction to Prolog
## Exercise 4 — Assessed

**Issued:** 19 November 2013 **Due:** 29 November 2013

### Introduction

This is a variation of the classic farmer-wolf-goat-cabbage puzzle. We have added another item (a bag of fertilizer) and a number of other features.

A river has a north bank and a south bank. On the north bank there stands a farmer with a wolf, a goat, a cabbage, and a bag of fertilizer.

The farmer's goal is to transport all of these items and himself to the south bank. He can cross the river in a boat, either alone or accompanied by just *one* other item. However, it is not *safe* to leave behind the wolf with the goat, as the wolf might eat the goat if the farmer is not present. Likewise, it is not *safe* to leave behind the goat with the cabbage (as the goat might eat the cabbage). The farmer will therefore have to make several crossings back and forth, in such a way that all items get transported and nothing gets eaten. The wolf will not eat the cabbage, and neither the wolf nor the goat will eat the bag of fertilizer.

Suppose, for this exercise, that constants `f`, `w`, `g`, `c` and `b` denote the farmer, wolf, goat, cabbage, and bag of fertilizer, respectively, and that a *state*, describing the situation immediately before or after a river crossing, is represented by a term of the form `North-South`, where `North` is a list of the items currently on the north bank of the river and `South` is a list of the items currently on the south bank. The items in the lists `North` and `South` can occur in any order, except that `f`, if present, must be the first item in the list. The initial state could thus be represented by the term `[f,w,g,c,b]-[]`, for example. (This is not the only possible representation of a state, nor even the best one. It is chosen for the purposes of the exercise.) Note that the function symbol `-` is an 'infix' operator in Prolog: the term `X-Y` is shorthand for `-(X,Y)`.

For each journey across the river the farmer has to pay a fee, depending on how many items (including himself) are transported in that crossing. He wants to find all the possible ways of achieving the goal and the total cost of each.

**Note** You may use any Sicstus built-in predicate, such as `member/2`, `append/3`, `length/2`, `sort/2`, ... etc., but *DO NOT* use any of the Sicstus libraries.

The file `crossings.pl` includes a definition of `forall/2` and also a simple utility program `write_list/1` which you might find useful for printing out (long) lists when testing. Please do *not* include calls to `write_list/1` in your submitted solution.

Complete the exercise by adding your code to `crossings.pl`.

## What to do

**Step 1** Write a Prolog program for the predicate `safe(Bank)` which holds when `Bank` is a given list of items, possibly including the farmer, that is *safe* as defined above. For example, `safe([w,c])` and `safe([f,g,c])` should succeed; `safe([g,c,b])` should fail.

Remember that `f`, if present, must be the first element of the list but that other items (if any) may occur in any order — allow for this. You can assume that `Bank` is always ground when `safe(Bank)` is called.

**Step 2** Write a Prolog program for the predicate `safe_state(State)` which holds when `State` represents a state in which both of the banks are *safe* as defined above. You can assume that `State` is always ground when `safe_state(State)` is called and that it represents a valid state.

**Step 3** Write a Prolog program for the predicate `equiv(State1, State2)` which holds when the two (given, ground) states `State1` and `State2` are equivalent, that is, when they have the same items on their north banks and the same items on their south banks. For example, `[b,c]-[f,w,g]` and `[c,b]-[f,g,w]` are equivalent in this sense, but `[b]-[f,w,c,g]` and `[c,b]-[f,g,w]` are not.

**Step 4** Write a Prolog program for the predicate `goal(State)` which holds when the (given, ground) term `State` represents the goal state. For example, `goal([]-[f,c,w,g,b])` should succeed. Remember that items can occur in lists in any order, except for the farmer. You can assume that `State` is always ground when `goal(State)` is called.

**Step 5** A *history* is a list of states. Write a Prolog program for the predicate `visited(State, History)` which holds when a given state `State` is equivalent (as defined above) to some member of a given list of states `History`. You can assume that both `State` and `History` are ground.

**Step 6** Write a Prolog program for the predicate `remove(X, List, Remainder)` which holds when `X` is an element of the given (ground) list `List` and `Remainder` is the list obtained when `X` is removed from `List`. There may be multiple occurrences of `X` in `List`. For example:

```
?- remove(X, [r,i,v,e,r], Rem).
X = r, Rem = [i,v,e,r] ;
X = i, Rem = [r,v,e,r] ;
X = v, Rem = [r,i,e,r] ;
X = e, Rem = [r,i,v,r] ;
X = r, Rem = [r,i,v,e] ;
no
```

`remove/3` is a generalisation of the built-in predicate `member/2`. You should give a recursive definition of `remove/3`. The file `crossings.pl` contains an (inefficient) definition of `remove/3` in terms of `append/3`, called `app_remove/3`. If you wish to skip this part of the exercise, or leave it until the end, you can define

```
remove(X, List, Remainder) :-
   app_remove(X, List, Remainder).
```

This will allow you to make use of `remove/3` in later parts of the question.

**Step 7** Write a Prolog program for the predicate `crossing(State1, Move, State2)` which holds when `Move` represents one of the possible (not necessarily safe) river crossings that the farmer can make in (given) state `State1` and `State2` is the state that then results when `Move` is made.

Suppose that the possible river crossings ('moves') are represented by the constant `f`, representing a journey in which the farmer crosses the river alone, or a term of the form `f(X)` representing a journey in which the farmer transports item `X` where `X` is one of `w`, `g`, `c`, or `b`.

For example:

```
?- crossing([f,w,b]-[g,c], Move, Next).
Move = f,    Next = [w,b]-[f,g,c] ;
Move = f(w), Next = [b]-[f,w,g,c] ;
Move = f(b), Next = [w]-[f,b,g,c] ;
no
```

Note that in general there can be many other possible values for `Next` for any given value of `Move` because items in the resulting state can appear in any order. Your program should generate all the possible values of `Move` in any given state but just one representation of the resulting state in each case. It does not matter which one you pick.

You can assume that the first argument of `crossing/3` will be a ground list representing a valid (but not necessarily safe) possible state.

**Step 8** A *sequence* is a list of possible 'moves' (terms representing possible river crossings as defined above). Write a Prolog program for the predicate `succeeds(Sequence)` which returns as output a `Sequence` of 'moves' that leads, in order, from the initial state to the goal state, and is such that each intermediate state is *safe* (nothing gets eaten).

In order to ensure that `Sequence` contains no 'loops', `succeeds(Sequence)` should maintain a history of states visited so far and call the predicate `visited/2` defined earlier. Accordingly, define `succeeds/1` in terms of an auxiliary predicate `journey/3` as follows:

```
succeeds(Sequence) :-
   journey([f,w,g,c,b]-[], [], Sequence).
```

The second argument in `journey/3`, initially empty, represents the list of states visited so far.

The recursive clause for `journey/3` should look for a possible crossing ('move') to some next state, check that this state is safe and has not already been visited, add it to the current list of visited states, and continue until the goal state (base case) is reached.

In order to aid testing, the file `crossings.pl` contains a set of facts for the predicate `solution/1` giving the (four) successful solutions to this problem. If you wish to skip this part of the exercise, or cannot get your solution to work correctly, you can define

```
succeeds(Sequence) :-
   solution(Sequence).
```

This will allow you to attempt the remaining part of the exercise.

**Step 9** The file `crossings.pl` defines a predicate `fees(F1, F2)` which specifies the fees for each river crossing: `F1` is the fee if the farmer crosses alone, and `F2` is the fee if he transports one of the items.

Write a program for the predicate `journey_cost(Sequence, Cost)` such that `succeeds(Sequence)` holds and `Cost` is the sum of the fees for all the individual crossings ('moves') in `Sequence`. Your program should generate all solutions on backtracking.

(Credit will be given for tail recursive solutions.)

## Submission

Your Prolog program should be named `crossings.pl`. This file is to be submitted electronically via the CATE system. Add your code to the supplied file `crossings.pl`.

**You may submit this coursework in pairs if you wish.** One of the pair is nominally the 'team leader' (it does not matter which); CATE will ask the 'team leader' to identify the partner. If you are submitting an exercise on your own, you are the 'team leader' and there is no partner.

*DO NOT* include any print or write statements in your submitted code. Use them by all means for tracing/debugging but do not include them in your submitted program. (Otherwise the autotester produces reams of output.)

Do not use the Sicstus `library(lists)`.

## Marking

Your solutions will be marked mainly on the correctness of your solutions. As such you should test your programs on suitable examples before submission. (Your program will be tested on a different version of the prison database than is supplied in the support file.)

Ensure that your submission *COMPILES AND EXECUTES* without errors on the Linux Sicstus system installed on the lab machines. 20% of marks will be deducted for programs that do not compile or that generate run-time errors.

A percentage of marks is awarded to the style of your solutions. This includes layout and appropriate use of comments. Marks will be deducted for overcomplicated solutions.

This exercise is worth 25% of the marks allocated to this module. (75% is for the Lexis test in January.)

## Return of Work and Feedback

Feedback on your solution will be given on your returned submission. General feedback will be given during lectures after the deadline.