Imperial College London – Department of Computing

276 Introduction to Prolog

Exercise 6 (Unassessed)

Practice Lexis Test

This exercise is intended to give you some practice and some idea of what to expect in the formal 'Lexis Test' that you will take in week 1 of the Spring Term. *Check the timetables.* It is *your* responsibility to ensure that you turn up at the right time and place.

The 'Lexis Test' The Lexis Test is taken under examination conditions. It counts for 75% of the total marks for the Prolog module. (The assessed exercise you did counts for the other 25%.) You will have access to the Sicstus Prolog environment, a text editor, and the on-line Sicstus Prolog manual. You will also be allowed to bring into the examination room one (double-sided) sheet of paper with your own notes. (More than that is counter-productive. You have access to the on-line manual.)

The Lexis Test will last for *TWO HOURS* plus an extra 20 minutes of reading time. You will have 20 minutes to read through the questions and plan your answers, after which you can log in to the Lexis exam system and write your programs.

This exercise This practice exercise is unassessed. Because of the timing of the lectures, it is not possible for us to mark your solutions and return detailed annotated submissions before the end of term. The practice test will not be marked and there is nothing you need to submit. However: a model answer will be made available on CATE. There will also be a set of test queries that you can use to compare your solution against the model answer, together with a set of instructions on how you can automate most of the comparison.

Obviously, this practice exercise will be most useful if you do it under the same conditions as the actual test: you should work *on your own*, and you should set yourself the same *TWO HOUR* time limit (plus 20 minutes reading time).

It is difficult to judge the length of these exercises. If you find you cannot finish in time, do not despair. Account will be taken of that during the marking.

INSTRUCTIONS

There are *TWO* questions. The marks allocated for each question and each part-question are shown.

Parts of questions are related but do not depend on one another except where indicated. You can still obtain full marks for later parts even if you do not manage to complete earlier ones.

To answer the questions add new clauses to the two supplied files

`practice_Q1_prison.pl`, `practice_Q2_graphs.pl`

Your edited versions should contain all the code you want to submit as your answer to the given questions, including any comments you regard as necessary to justify your solutions. You can include comments to outline your intended method if you cannot get your programs to work. You are not required to do anything else; the files will be taken directly from your working space once you have logged out from your session. Any other files you make as copies or backups will be ignored.

(The instructions above apply to the Lexis test itself. For this practice exercise there is nothing you need to submit.)

Important Note You can still get credit for fragmentary answers or incomplete code that does not work fully. Comments, fragments of code, and other written answers will be read and assessed. Make sure than any supplementary comments and incomplete code fragments are *commented out* so that the submitted file compiles without errors.

Ensure that your solution *COMPILES and EXECUTES* without errors on the Linux Sicstus implementation.

Do *NOT* use any of the Sicstus libraries, except where indicated.

Question 1 (*55% marks*)

The file `prisonDB_lexis.pl` contains a database of facts about a (fictional, unnamed) prison. It defines the following predicates:

```
cells/1,   crimes/1,   prisoner/6,   psychopath/2,   female_name/1
```

`cells/1` gives the number of cells in the prison. `crimes/1` gives the list of possible crimes.

`prisoner(Surname, FirstName, Cell, Crime, Sentence, ToServe)` represents data about the prisoners. A prisoner is uniquely identified by his or her first name and surname. The other arguments represent the following information for each prisoner

`Cell`	the prisoner's cell number (a positive integer)
`Crime`	the crime for which the prisoner was convicted
`Sentence`	the number of years (positive integer) for which the prisoner was originally convicted
`ToServe`	the number of years (positive integer) left to serve of the prisoner's orginal sentence

There may be more than one prisoner in a cell and there may be cells that are empty.

`psychopath(Surname,FirstName)` holds when the prisoner with that name is a psychopath. (Psychopaths are not necessarily kept in cells by themselves.)

`female_name/1` is used to identify the female prisoners. A prisoner `(Surname,FirstName)` is female if (and only if) `female_name(FirstName)` holds.

You will see that `practice_Q1_prison.pl` automatically loads `prisonDB_lexis.pl`. Do not edit `prisonDB_lexis.pl`. Answer the question by adding clauses to `practice_Q1_prison.pl`.

Part (a) (6 marks)

The file `practice_Q1_prison.pl` contains the following definition of `cell/1`:

```
cell(N) :- cells(Cells), in_range(1,Cells,N).
```

Write a program `in_range(Min,Max,N)` which, given integers `Min` and `Max`, will generate the integers `N` such that `Min` $\leq$ `N` $\leq$ `Max`. For example, the query `?- in_range(2,4,N)` should give answers `N = 2`, `N = 3`, and `N = 4` (obtained by backtracking, as usual). The query `?- in_range(5,5,N)` should give a single answer `N = 5`. `in_range(Min,Max,N)` should fail if it is not the case that `Min` $\leq$ `Max`. You can assume that integers `Min` and `Max` will be given when `in_range/3` is called, and that both are integers.

`in_range/3` corresponds to the Sicstus library predicate `between/3`. If you want to skip this part, load the Sicstus library module `library(between)` and define:

```
in_range(Min,Max,N) :- between(Min,Max,N).
```

This will allow you to answer the remaining parts of the question. Instructions are in `practice_Q1_prison.pl`.

Part (b) (3 marks)

Give a one clause definition of `empty_cell/1` such that `empty_cell(Cell)` holds when `Cell` is an empty cell in the prison.

Part (c) (6 marks)

Define `all_female_cell/1` such that `all_female_cell(Cell)` holds when `Cell` is a (non-empty) cell in the prison containing only female prisoners.

You can use Prolog's negation-as-failure primitive `\+` or the utility 'meta-predicate' `forall/2`:

```
forall(C1,C2) :- \+ (C1, \+ C2).
```

Part (d) (8 marks)

How many female prisoners are there in the prison? Define `female_prisoners/1` such that `female_prisoners(N)` holds when `N` is the number of female prisoners in the prison.

(We suggest that you check your answer by querying also how many non-female prisoners there are in the prison, and how many prisoners there are in total.)

Note When counting solutions, here and in other parts of the question, you can assume without checking that prisoners are uniquely identified by their first name and surname, i.e., that there is no more than one prisoner with any given first name and surname combination.

If you choose to use `setof/3` make sure that variables are existentially quantified correctly. Alternatively, you may prefer to deal with the required quantification by defining auxiliary predicates.

`length/2` is a built-in predicate in Sicstus and so can be used without loading any libraries.

Part (e) (8 marks)

Define `cell_occupancy/2` such that `cell_occupancy(Cell,N)` holds when `N` is the number of prisoners in cell `Cell`. Your program should generate solutions on backtracking if `Cell` is a variable when the program is called.

Part (f) (8 marks)

Which of the cells contain the greatest number of prisoners? Define `fullest_cell/1` such that `fullest_cell(Cell)` holds when there is no other cell in the prison with more prisoners than `Cell`. Note that `fullest_cell(Cell)` is not necessarily unique: your program should generate all answers on backtracking.

Part (g) (8 marks)

Which of the psychopaths is serving the longest sentence and for which crime?

Define `worst_psychopath/4` such that `worst_psychopath(S,F,Crime,T)` holds when `(S,F)` is a psychopath (`psychopath(S,F)` holds) serving a sentence of length `T` years for crime `Crime` and there is no other psychopath in the prison serving a sentence longer than `T`. Note that `worst_psychopath/4` is not necessarily unique: your program should generate all answers on backtracking.

Part (h) (8 marks)

How many murderers are there in the prison? How many plagiarists? Define `criminals/2` such that `criminals(Crime,N)` holds when there are `N` prisoners in the prison who have been convicted for the crime `Crime`. Your program should generate solutions on backtracking if `Crime` is a variable when the program is called. (This question is quite hard. You can use `setof/3` but you need to know how to quantify variables inside the call.)

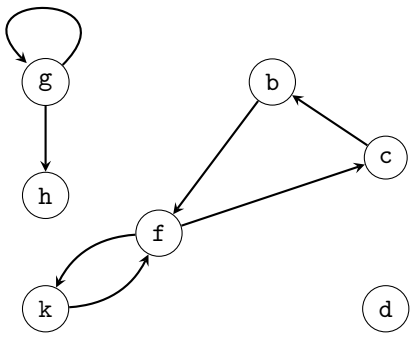
Question 2 (*45% marks*)

A (directed) graph is a set of nodes (also called 'vertices') and a set of edges (also called 'arcs'), where each edge is a pair of nodes.

One way of representing graphs in Prolog is to represent each edge separately by a clause (fact). (Obviously, isolated nodes cannot be represented.) For many tasks however it is more convenient to represent the whole graph as a single data object. One common method is to represent the graph by a term of the form

```
graph(Nodes, Edges)
```

where `Nodes` and `Edges` are both ordered, duplicate-free lists of nodes and edges, respectively. An edge is represented by a term of the form `e(X,Y)` where `X` and `Y` represent nodes. A node can be any (ground) term. (This includes arbitrary compound terms, such as `city('London',4711)`, although in this question all nodes in example graphs will be Prolog atoms.) Note that in order to simulate sets, the lists are kept sorted and without duplicated elements. The ordering is the standard Prolog order as given by `sort/2` (and `@</2`). We will call this the *graph-term* form. For example, the graph



would be represented by

```
graph([b,c,d,f,g,h,k], [e(b,f),e(c,b),e(f,c),e(f,k),e(g,g),e(g,h),e(k,f)])
```

Another representation method which is sometimes more convenient is to associate with each node the set (ordered list) of nodes that are adjacent to it. We call this the *adjacency-list* form. In this representation the graph is represented by an ordered, duplicate-free list of terms of the form

```
n(Node,AdjNodes)
```

where `Node` represents a node and `AdjNodes` is an ordered, duplicate-free list of nodes representing the nodes adjacent to `Node` in the graph. In the example:

```
[n(b,[f]), n(c,[b]), n(d,[]), n(f,[c,k]), n(g,[g,h]), n(h,[]), n(k,[f])]
```

These two representations are well suited for automated processing but their syntax is not very user-friendly. A more compact and human-readable notation represents a graph by a list of terms of the form `X > Y` to represent edges; any other (ground) terms—atoms or compound terms with a functor other than `>/2`—represent nodes. The endpoints `X` and `Y` of `X > Y` terms are automatically defined as nodes. We will call this the *human-friendly* form. In this representation the example graph could be written as:

```
[b > f, f > c, c > b, g > h, g > g, d, b, f > k, k > f, f > c]
```

Note that the list does not have to be sorted and may even contain the same edge and the same node multiple times. Notice the isolated node `d`. The term `b` does not represent an isolated node because it is the endpoint of an `X > Y` term (actually, more than one).

Part (a) (15 marks)

Write a Prolog program

```
merge_ordered( Left, Right, Merged )
```

which given two (ground) ordered, duplicate-free lists `Left` and `Right` produces a single, ordered duplicate-free list `Merged` containing the elements of `Left` and `Right`. The ordering to be used is the standard Prolog order as given by `sort/2` (and `@</2`). For example, the following query should produce a single solution as shown

```
?-  merge_ordered( [a,g,p,e(b,k)], [c,g,q,e(a,a),e(b,f)], Merged ).
Merged = [a,c,g,p,q,e(a,a),e(b,f),e(b,k)] ;
no
```

Note how Prolog's `sort/2` and `@</2` orders terms like `e(b,f)`.

You can assume without checking that `Left` and `Right` are both (ground) ordered, duplicate-free lists when the program is called.

If you wish to skip this part of the question, you can define `merge_ordered/3` as follows:

```
merge_ordered(Left,Right,Merged) :-
  append(Left,Right,Both),
  sort(Both,Merged).
```

This will allow you to use `merge_ordered/3` in other parts of the question. (It is slightly more general than needed since it does not assume that `Left` and `Right` are ordered and duplicate-free.)

Suggestion: It might be easier to begin this way, and return to finish part (a) after the other parts of the question have been completed.

Part (b) (15 marks)

Write a Prolog program

```
hf_to_graph_term(Hform, Graph)
```

to convert a graph `Hform` in *human-friendly* form to its (unique) *graph-term* representation `Graph`.

You can assume without checking that `Hform` is a valid representation of a graph.

The problem can be solved using `member/2`, `findall/3`, `sort/2` and/or `setof/3`. However, full credit will be given to recursive programs that perform the translation in one pass through the list `Hform`. (*Hint:* Use 'accumulators' and `merge_ordered/3`.)

Part (c) (15 marks)

Write a Prolog program

```
graph_term_to_adj_list(Graph, AdjList)
```

to convert a graph `Graph` in *graph-term* form to its (unique) *adjacency-list* representation `AdjList`.

You can assume without checking that `Graph` is a valid graph-term representation of a graph.

Extra credit will be given for tail-recursive solutions.

Submission

To answer the questions add new clauses to the two supplied files

`practice_Q1_prison.pl`, `practice_Q2_graphs.pl`

Your edited versions should contain all the code you want to submit as your answer to the given questions, including any comments you regard as necessary to justify your solutions. You can include comments to outline your intended method if you cannot get your programs to work. You are not required to do anything else; the files will be taken directly from your working space once you have logged out from your session. Any other files you make as copies or backups will be ignored. The file `prisonDB_lexis.pl` will also be ignored. (Your programs will be tested on a different set of data.)

(The above instructions obviously do not apply to this practice test. There is nothing to submit.)

Ensure that your submitted file *COMPILES WITHOUT ERRORS* on the Linux Sicstus implementation.

Do not use any of the Sicstus libraries, except where indicated. (You will lose marks.)

Model solution

A model solution for these exercises is available in CATE, together with some test queries and a set of instructions on how you can execute them automatically.