Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)
MRes in Advanced Computing

531: Prolog
Laboratory Test (ASSESSED)

THREE HOURS

INSTRUCTIONS

There are *THREE* 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 supplied file

`lexis.pl`

Your edited version 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.

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 (*40% marks*)

The file `utilities.pl` contains a database of facts about a (fictional, unnamed) prison. It is similar to the database used in the Prison lab exercise except that in this exercise you will not be simulating warders locking and unlocking cells. There are a few other small differences.

`utilities.pl` defines the following predicates:

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

`cells/1` gives the number of cells in the prison.

`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 `lexis.pl` automatically loads `utilities.pl`. Do not edit `utilities.pl`. Answer the question by adding clauses to `lexis.pl`.

Part (a) (4 marks)

The file `lexis.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.

Part (b) (2 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) (4 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) (6 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.

In some cases you may need to allow for duplicate solutions. 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) (6 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) (6 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) (6 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) (6 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.

Question 2 (*30% marks*)

In a substitution cipher, the encrypted message (the 'ciphertext') is obtained by replacing every character of the original message (the 'plaintext') by its corresponding character in a given 'cipher alphabet'. For example:

plain alphabet : `abcdefghijklmnopqrstuvwxyz`
cipher alphabet: `JLPAWIQBCTRZYDSKEGFXHUONVM`

plaintext : `more paper please`
ciphertext: `YSGW KJKWG KZWJFW`

By convention, plaintext is written in lower case and ciphertext in upper case.

In this question you will write Prolog programs to encrypt and decrypt text represented as Prolog *strings*. Recall that a 'string' in Prolog is merely notation for a list of integers corresponding to ASCII character codes. Thus the string `"SICStus"` denotes exactly the same list as `[83,73,67,83,116,117,115]`.

The file `utilities.pl` contains some examples you can use for testing your program, and some simple utilities for printing out strings in readable form.

Part (a) (4 marks)

Define `upper_case_string/1` such that `upper_case_string(String)` holds when `String` is a non-empty string of upper case characters or spaces, i.e., when `String` is a non-empty list of integers each of which is the standard ASCII code of an upper case character or of the space character.

For reference, `65 ... 90` are the codes for the upper case characters `A ... Z` and `97 ... 122` are the codes for the lower case characters `a ... z`, respectively. The code for the space character is `32`.

(You can easily find the code for any character by a simple Prolog query. For instance, the query `?- "b" = X` gives answer `X = [98]`.)

Part (b) (6 marks)

Write a Prolog program

```
subst_string( Input, Subst, Output )
```

which copies the Prolog string `Input` to a string `Output` performing character-by-character substitutions as specified by `Subst`. `Subst` is represented by a pair of equal-length strings `In-Out`. Any character in `Input` that appears in the string `In` is replaced by the character at the corresponding position of `Out`; any other characters in `Input` are copied to `Output` without change. For example:

```
?-  subst_string( "Lexis Test", "xeiT"-"!XY?", Output ).
Output = "LX!Ys ?Xst" ;
no
```

You can assume that `Input` is a Prolog string and that `Subst` is a pair `In-Out` of non-empty, equal-length, duplicate-free strings.

Part (c) (6 marks)

Write a Prolog program

```
encrypt_string( PlainText, Key, CipherText )
```

that encrypts the string `PlainText` to the string `CipherText` using the cipher alphabet `Key`, also represented as a string. For example, the query corresponding to the example above

```
?- encrypt_string( "more paper please",
                   "JLPAWIQBCTRZYDSKEGFXHUONVM",
                   CipherText ).
```

should produce a single solution `CipherText = "YSGW KJKWG KZWJFW"`.

You can assume that `PlainText` is a Prolog string and that `Key` does represent a cipher alphabet, i.e., that it is a list of 26 distinct integers representing upper case characters.

Remember that by convention, plaintext is written in lower case and ciphertext in upper case. Any upper case characters in `PlainText` should be converted to lower case during encryption. Any other characters in `PlainText` outside the range `a...z` should be encrypted without change. For example, with the cipher alphabet shown above you should obtain:

plaintext : `More paper. Please!!`
ciphertext: `YSGW KJKWG. KZWJFW!!`

The problem can be solved by performing two separate passes through the `PlainText` string, first to convert any upper case letters to lower case, and then separately to perform the actual encryption using `Key`. However, full credit will be given to solutions that perform the upper case to lower case conversion during the encryption, in one single pass through `PlainText`.

Although `encrypt_string/3` will be defined in terms of `subst_string/3` or something similar you will need to make some modifications to the program of part (b) to deal with the possibility of mixed upper and lower case plaintext in the input. (You may prefer to answer the part (d) of the question first as decryption is slightly simpler than encryption.)

Part (d) (4 marks)

Write a Prolog program

```
decrypt_string( CipherText, Key, PlainText )
```

that decrypts the string `CipherText` into the string `PlainText` using the cipher alphabet `Key`, also represented by a Prolog string as in `encrypt_string/3`. For `decrypt_string/3` you can assume in addition that the input string `CipherText` will not contain lower case characters (since it will have been produced by an execution of `encrypt_string/3`).

`decrypt_string/3` is simpler than `encrypt_string/3` since encryption has to deal with the possibility of mixed upper and lower case strings whereas decryption does not. You will either need two different versions of `subst_string/3` of part (b) or a single version with an extra parameter to allow for this difference.

Part (e) of Question 2 continues overleaf.

Part (e) (10 marks)

A common way of producing a cipher alphabet is by means of a keyphrase. For example, to use `BORING LECTURE` as a keyphrase, ignore all spaces and repeated characters (`BORINGLECTU`), use this as the beginning of the cipher alphabet, and then continue with the remaining unused characters of the plain alphabet, in their correct order, starting where the keyphrase ends. For the example keyphrase `BORING LECTURE`, the generated cipher alphabet would be as follows:

plain alphabet : `abcdefghijklmnopqrstuvwxyz`
cipher alphabet: `BORINGLECTUVWXYZADFHJKMPQS`

Write a Prolog program

```
keyphrase_cipher( KeyPhrase, Key )
```

that generates a cipher alphabet `Key` (a string representing 26 distinct upper case characters) from the string `KeyPhrase`. You may assume that `KeyPhrase` is a non-empty string containing only upper case characters in the range `A` ... `Z` or spaces (i.e., that `upper_case_string(KeyPhrase)` from part(a) holds).

The file `lexis_Q2_ciphers.pl` contains a utility program `encrypt/2` which you can use to test your programs. You will see that it calls `upper_case_string/1` of part(a). If you did not manage to define `upper_case_string/1` in part(a), comment out the first line of the `encrypt/2` program.

You will see that `encrypt` is declared to be an infix operator. This is so that you can type test queries in this form:

```
?-  "BORING LECTURE" encrypt "More paper. Please!!".
```

without having to type brackets.

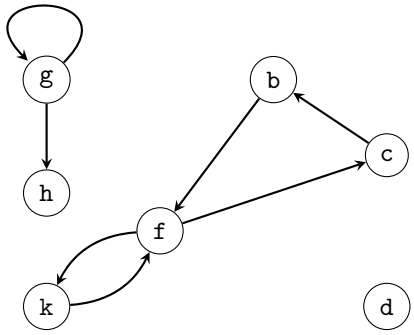
Question 3 (*30% marks*)

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) (10 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,ebk,g,p], [c,ebf,g,q], Merged ).
Merged = [a,c,ebf,ebk,g,p,q] ;
no
```

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.)

Part (b) (10 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) (10 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 supplied file

`lexis.pl`

Your edited version 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 `utilities.pl` will also be ignored. (Your programs will be tested on a different set of data.)

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

Do not use any of the Sicstus libraries.