Principles of Programming Languages, Spring 2015
Assignment 5
Logic Programming

Submission instructions:

1. You are not allowed to use Prolog's arithmetic or negation in this assignment. Also, you are not allowed to use functors or lists in questions marked as Relational LP. All these constraints will be checked.
2. Submit an archive file named ***id1_id2.zip*** where *id1* and *id2* are the IDs of the students responsible for the submission (or ***id1.zip*** for one student in the group).
3. The contract must be included for every implemented procedure.
4. Use exact procedure and file names, as your code is tested automatically.
5. Answer theoretical questions in 'ex5.pdf'. You can scan hand-drawn trees and add them to the submitted work.
6. Answer coding questions in the appropriate files:
   a. Question 3 in 'q3.pl'
   b. Question 4a, 4b, 4c in 'q4.pl'
   c. Question 5b2 in 'substitute.pl'
   d. Question 5b3 in 'unify.pl'
   e. Question 7a in 'q7.pl'

   The files include contracts and examples for all the procedures you are required to implement. Some files use auxiliary data loaded automatically you can find in 'ex5-aux.pl'.
7. You may use Relational LP (only) primitive procedures and add auxiliary procedures, if needed. Any order of answers of a query is acceptable, as long as all the correct answers are returned, only correct answers are returned and infinite computations are avoided. Also avoid duplicate answers unless they are allowed explicitly.
8. Before you start implementing, read all the contracts provided in the source files.

Question 1: Comprehension Questions

Answer in 'ex5.pdf' file.

a. Specify the differences and similarities between predicate symbols, individual constant symbols and functor symbols in LP.
b. Is every proof tree for programs and queries in RLP finite? Explain.
c. Consider the proof tree algorithm in RLP. Suggest a modification, so that it always terminates. That is, infinite branches are "cropped" at some point in which it is obvious that the branch is infinite. You can assume that the value N(P, Q), of the number of all atomic formulas over the given program P and a query Q up to variable renaming is given. Describe your suggested solution.
d. Can we apply the modification of (c) to proof tree of LP? Apply or explain the problem.
e. Define and give examples for a program with (1) success finite proof tree, (2) success infinite proof tree and (3) failure proof tree.
f. Given a procedure `not_member/2` which tests if an element does not appear in a list:

```
not_member(X,Xs) :-
     member(X,Xs), !,       %1
     fail.                  %2
not_member(_,_).
```

Michael noticed that there are multiple uses of row 1 in his code and decided to factor out the call to a helper procedure called `member_check/2`, and rewrote `not_member/2`:

```
member_check(X,Xs) :-
     member(X,Xs), !.
not_member(X,Xs) :-
     member_check(X,Xs),
     fail.
not_member(_,_).
```

Recall the meta-circular interpreter. Will this modification work? Explain why?

<u>Question 2. Concrete and abstract syntax.</u>

Answer in 'ex5.pdf' file.
The `if` statement in Prolog has the following form:

(*condition* -> *do_if_true* ; *do_if_false*)

For example,

```
red(a).
blue(b).
blue(c).
green(d).

color_kind(P,Color) :- (red(P) -> Color = warm ; Color = cool).

?- color_kind (b, X).
X = cool.
?- color_kind (a, X).
X = warm.
```

1. The first step in the addition of this construct to the RLP concrete syntax could be as follows: replace the BNF rules

   ```
   <body> -> (<atomic-formula>',')* <atomic-formula>
   <query> -> '?-' (<atomic-formula>',')* <atomic-formula> '.'
   ```

   by the rules

   ```
   <body> -> (<statement>',')* <statement>
   <query> -> '?-' (<statement>',')* <statement> '.'
   <statement> -> <atomic-formula> | <if-statement>
   ```

   Complete the process.

2. Add the `if` construct to the RLP abstract syntax.
3. Draw an and/or tree (ASD) for the `<if-statement>` category (recalling Section 2.1.2 from the lecture notes).

4. Rachel wishes to modify this construct to the following form:

   `(if condition then do_if_true else do_if_false)`

   Accordingly, what are the changes that should be performed to your previous answers?

Question 3: Relational Logic Programming and Structured Query Operations

Answer in 'q3.pl' file.
Relational databases are useful for managing structured information. Each table in the database represents a relation. Elementary structured query operations, such as select, project, join and Cartesian product enable data access. You are allowed to use only Relational LP in this question, unless it is mentioned explicitly. You may use the procedure `not_member`, appearing in the provided file '`ex5-aux.pl`'.
A relational database for Wikipedia management is given in the file 'ex5-aux.pl'. The database consists of four tables, represented as fact-based procedures:

1) The table `page(Page_id, Page_namespace, Page_title, Page_len)/4` contains information about pages in Wikipedia. Each page has an id (that identifies it uniquely), a namespace number (used for article, user page, book, category, help, and so on), a title (where namespace and title pair is a unique key again), and length in bytes. For example, `page(12345, 1, 'DOS', 345).`
2) The table `namespaces(Ns_number, Ns_name)/2` is the namespaces list.
3) The table `category(Cat_id, Cat_title, Cat_hidden)/3` describes the categories. Each category has id (same as in page table), name (also same as in page), and a boolean value that determines if the category name the page belongs to is displayed on the page bottom.
4) The table `categorylinks(Cl_from, Cl_to)`/2 describes the category links. Each inclusion of page to category is characterized by Cl_from (same as Page_id in page table) and Cl_to (which is the category title in page).

a. Write a procedure (i.e. set of axioms) `page_in_category(Name, Id/2)` that defines the relation between a page name and its category id, such that the category is non-hidden only.
b. Write a procedure `splitter_category(Id)`/1 which defines a category with at least two pages. Multiple right answers are allowed if the program does not enter to infinite loop.
c. Write a procedure `namespace_list(Name, List)/2` which is a relationship between a namespace name and list of all the pages IDs in it.

Question 4: Logic programming: Lists and functors

Answer in 'q4.pl' file (except d., which should be answered in 'ex5.pdf' file)

a. Write a procedure `allDiff(List,Dups)/2` that succeeds if and only if `Dups` contains all the elements that appear more than once in `List`. **Notice:** `Dups` may not contain duplicate members! For example,

```
?- allDiff([2, 3, 4, 5],Dups).
Dups = [];
false.
?- allDiff([2, 3, 4, 2, 8, 9, 2],Dups).
Dups = [2];
false.
```

b. Write a procedure `noDups(List, WithoutDups)/2` that succeeds if and only if `WithoutDups` contains the same elements of `List` with duplications removed. Meaning, for every element X in `List` you should keep the first appearance of X and remove the rest. For example:

```
?- noDups([1,2,3,4,5],X).
X = [1,2,3,4,5].
?- noDups([1,2,3,4,5,3,4,5],X).
X = [1,2,3,4,5].
?- noDups([1,2,3,4,5,3,4,5],[1,2,3,4,5]).
true.
?- noDups([1,2,3,4,5,3,4,5],[1,2,4,5,3]).
false.
```

c. The following context-free grammar describes a partial yet infinite subset of English. S is the start nonterminal (variable). Others represent syntax components such as nouns, verbs, prepositions, adjectives and determinants.

```
S --> Np Vp
Np --> Det Adjs N
Vp --> V Np Pp
Adjs -> ε | Adj Adjs
Pp --> ε | P Np
Adj --> beautiful | funny | tall | big
Det --> a | the
N --> cat | dog | mouse | house | table
V --> saw | ate
P --> in | from | on | with
```

In the file 'ex5.pl' you are given a program that implements the above grammar. Read the contract and run the examples.
Write a procedure `nondupCFG(Text)/1` that defines the relation such that $s \in nondupCFG$ iff $s \in L(G)$ and also s does not include duplications. Assume that `text` is fully bound.

d. The language described by the given (original) grammar is infinite. Notice that the given implementation of the context free grammar can be used as a sentence generator for the language it describes. That is, the query

```
?- s(L).
```

returns infinite instantiations of variable `L`, such that each one is a list that represents a sentence in the grammar. Can such a procedure guarantee that every legal sentence will appear after a finite number of answers? If yes, explain why. Otherwise, provide a counterexample.

Question 5: Unification

Answer in 'ex5.pdf' file (except b2 and b3., which should be answered in 'substitute.pl' and 'unify.pl' files respectively).

a. What is the result of these operations? Provide all the algorithm steps. Explain in case of failure.

```
1. unify[m(p(A, p(d(0), word3), X)), m(p(d(B), p(B, word3), C))]
2. unify[m(p(A, p(d(0), p), X)), m(p(d(B), p(B, word3), C))]
3. unify[m(p(A, p(d(A), word3), X)), m(p(d(B), p(B, word3), C))]
4. unify[m(p(A, p(d(0), word3), X)), m(p(d(B), p(B, word3), word3))]
```

b. Recall the implementation of the `unify` algorithm for Prolog from the lecture notes.

1. In the procedure `substitute/3` there are two cases to handle a variable, one of which contains a red cut. Is this cut necessary? Why? If it is – give an example which will lead to an error if it is removed.
2. Let's define two Prolog procedures as *equivalent*, if:
   i. For any sequence of input arguments the set of answer substitution will be the same, when each answer is returned the same number of times, in any order.
   ii. For any sequence of input arguments we'll get the same sequence of output arguments.
   iii. For any sequence of input arguments the result will be success or fail in both procedures.
   iv. For any sequence of input arguments either both or none of them will enter an infinite loop.

   Remove exactly one rule from the substitute/3 procedure such that the modified procedure would be equivalent to the original.
3. Consider a call `unify[expr_a, expr_b]`. Assume that `expr_b` does not contain variables. Modify the implementation in order to make it more efficient, accordingly.

Question 6: Answer-query algorithm,

Answer in 'ex5.pdf' file.

Unary numbers provide symbolic representation for the natural numbers. They are defined inductively as follows: zero is the atom `[]`, and for a Unary number `c`, `[1|c]` represents the number `c+1`. The numbers 0, 1, 2, 3 etc. are represented by the terms `[], [1], [1, 1], [1, 1, 1]`, and so forth. The following program is given.

```
% Signature: unary_number(L)/1
% Purpose: L is a unary number
unary_number([]).           %1
unary_number ([1|A]) :-     %2
      unary_number (A).     %3

% Signature:  unary_plus(X,Y,Z)/3
% Purpose: X append Y = Z
unary_plus(X, [], X) :-          %1
      unary_number(X).           %2
unary_plus (X, [A|Y], [A|Z]):-   %3
      unary_plus(X, Y, Z).       %4
```

a. Draw the proof tree for the query below and the given program. For success leaves, calculate the substitution composition and report the answer at each success leaf.
   `?- unary_plus ([1|Y], [1,1|X], [1,1,1,1]).`
b. What are the answers of the answer-query algorithm for this query?
c. Is this a success or a failure proof tree? Explain.
d. Is this tree finite or infinite? Explain.

Question 7: Meta-circular LP interpreter

Answer in 'ex5.pdf' file (except a, which should be answered in 'q7.pl' file).

a. Recall the proof meta-circular interpreter given in class (code below). Change the interpreter such that every answer is returned four times instead of once (obviously, the order is irrelevant).

The order by which answers are returned does not matter – as long as **all** answers are retuned.

```
% Signature: solve(Exp, Proof)/2
solve(true, true).                                    %1
solve([], []).                                        %2
solve([A|B], [ProofA|ProofB]) :- solve(B, ProofB),    %3
                                 solve(A, ProofA).
solve(A, node(A, Proof)) :- rule(A, B),               %4
                            solve(B, Proof).
```

For example:

```
?- solve(true,_).
true;
true;
true;
true;
false.

?- solve(member(X,[1,2]),_).
X=1;
X=1;
X=1;
X=1;
X=2;
X=2;
X=2;
X=2;
false.
```

b. Recall the waiting list meta-circular interpreter given in class.

```
% Signature: solve(Exp)/1
solve(Goal) :- solve(Goal, []).                       %1

% Signature: solve(Exp, Goals)/2
solve(true, Goals) :- solve([], Goals).               %1
solve([], []).                                        %2
solve([], [G|Goals]) :- solve(G, Goals).              %3
solve([A|B], Goals) :-
      extend_waiting_list([A|B], Goals, NewGoals),    %4
      solve([], NewGoals).
solve(A, Goals) :- rule(A, B),                        %5
                   solve(B, Goals).
```

Assume that the `extend_waiting_list` rule was implemented as append, as usual, but after that the result is randomly permuted (shuffled). For example,

```
extend_waiting_list([a(X, X, Y, 3, 4, 5)],
       [m(R, R, R), b(Z, T), c(A, B, X)],
       [m(R, R, R), a(X, X, Y, 3, 4, 5), b(Z, T), c(A, B, X)]).
```

Consider the proof tree that will be created for some query in both versions. Can any variable substitution be achieved in one version only? Explain. Can any true leaf move to the other side of the most left infinite branch? Explain. In both questions give an example if the answer is positive.

c. (bonus 4 points) Solve the question in `(b)` – for the solver with failure proof.