Solutions to some exercises in "The Art of Prolog"

These are my solutions to some of the exercises given in "The Art of Prolog" by Leon Sterling and Ehud Shapiro. The exercise numbers correspond to the Third printing, March 1987. I understand that they have changed in later editions. (Programs, but not exercises, from a later edition can be found here.) Unless otherwise stated, the syntax I have used corresponds to that of Strawberry Prolog, which I believe agrees with the ISO standard. My apologies to those who think solutions should not be given, as they could help students do their homework. But not only students read books! Any errors or improvements will be gratefully received.

Exercise 2.1 (i)

```
sister(Sister, Sib):-
  parent(Parent, Sister),
  parent(Parent, Sib),
  female(Sister),
  Sister =\= Sib.

niece(Niece, Person):-
  parent(Parent, Niece),
  sibling(Parent, Person),
  female(Niece).

sibling(Sib1, Sib2):-
  father(Father, Sib1),
  father(Father, Sib2),
  mother(Mother, Sib1),
  mother(Mother, Sib2),
  Sib1 =\= Sib2.
```

Exercise Index Home Page

Exercise 2.1 (ii)

```
mother_in_law(MotherInLaw, Husband):-
  married_couple(Wife, Husband),
  mother(MotherInLaw, Wife).
mother_in_law(MotherInLaw, Wife):-
  married_couple(Wife, Husband),
  mother(MotherInLaw, Husband).

brother_in_law(BrotherInLaw, Husband):- % Wife's brother
  married_couple(Wife, Husband),
  brother(BrotherInLaw, Wife).
brother_in_law(BrotherInLaw, Wife):-    % Husband's brother
  married_couple(Wife, Husband),
  brother(BrotherInLaw, Husband).
brother_in_law(BrotherInLaw, Person):-  % Sister's husband
  sister(Sister, Person),
  married_couple(Sister, BrotherInLaw).

son_in_law(SonInLaw, Parent):-
  married_couple(Daughter, SonInLaw),
  parent(Parent, Daughter).
```

Exercise Index Home Page

Exercise 2.1 (iii)

```
or_gate(Input1, Input2, Output):-
  transistor(Input1, Output, ground),
  resistor(power, Output).
or_gate(Input1, Input2, Output):-
  transistor(Input2, Output, ground),
  resistor(power, Output).

nor_gate(Input1, Input2, Output):-
  or_gate(Input1, Input2, X),
  inverter(X, Output).
```

Exercise Index Home Page

Exercise 3.1 (i) - lt/2, gt/2 and ge/2

```
/* lt(X,Y) is true if X and Y are natural numbers such that X is less than */
/*   Y.                                                                    */
lt(0,s(X)):-natural_number(X).
lt(s(X),s(Y)):-lt(X,Y).

/* gt(X,Y) is true if X and Y are natural numbers such that X is greater   */
/*   than Y.                                                               */
gt(s(X),0):-natural_number(X).
gt(s(X),s(Y)):-gt(X,Y).

/* ge(X,Y) is true if X and Y are natural numbers such that X is greater   */
/*   than or equal to Y.                                                   */
ge(X,0):-natural_number(X).
ge(s(X),s(Y)):-ge(X,Y).

/* natural_number(X) is true if X is a natural number.                     */
natural_number(0).
natural_number(s(X)):-natural_number(X).
```

Exercise Index Home Page

Exercise 3.1 (iv) - even/1 and odd/1

```
/* even(X) is true if X is an even natural number.                         */
even(0).
even(s(s(X))):-even(X).

/* odd(X) is true if X is an odd natural number.                           */
odd(s(0)).
odd(s(s(X))):-odd(X).
```

Exercise Index Home Page

Exercise 3.1 (v) - Fibonacci Number

```
/* fib(N,F) is true if F is the Nth Fibonacci number.                      */
fib(0,0).
fib(s(0),s(0)).
fib(s(s(X)),F):-
  fib(X,D),
  fib(s(X),E),
  plus(D,E,F).

/* plus(X,Y,Z) is true if X, Y and Z are natural numbers such that Z is    */
/*   the  sum of X and Y.                                                  */
plus(0,X,X):-natural_number(X).
plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

Exercise Index Home Page

Exercise 3.1 (vi) - Integer Quotients

```
/* int_div(X,Y,Z) is true if Z is the quotient of the integer division of  */
/*   X by Y.                                                               */
int_div(X,Y,0):-gt(Y,X).
int_div(X,Y,s(Z)):-plus(Y,X1,X),int_div(X1,Y,Z).
```

Exercise Index Home Page

Exercise 3.1 (vii) - Greatest Common Divisor

```
/* gcd(X,Y,Z) is true if Z is the greatest common divisor of X and Y.      */
gcd(X,0,X):-gt(X,0).
gcd(0,X,X):-gt(X,0).
gcd(X,Y,G):-gt(Y,0),ge(X,Y),plus(Y,X1,X),gcd(X1,Y,G).
gcd(X,Y,G):-gt(X,0),gt(Y,X),plus(X,Y1,Y),gcd(X,Y1,G).
```

Exercise Index Home Page

Exercise 3.2 (ii) - adjacent/3 and last/2

```
/* adjacent(X,Y,Zs) is true if the elements X and Y are adjacent in the    */
/*   list Zs.                                                              */
adjacent(X,Y,[X,Y|Zs]).
adjacent(X,Y,[Z|Zs]):-adjacent(X,Y,Zs).

/* last(X,Xs) is true if X is the last element in the list Xs.             */
last(X,[X]).
last(X,[Y|Xs]):-last(X,Xs).
```

Exercise Index Home Page

Exercise 3.2 (iii) - double/2

```
/* double(Xs,Ys) is true if every element in the list Xs appears twice in  */
/*   the list Ys.                                                          */
double([],[]).
double([X|Xs],[X,X|Ys]):-double(Xs,Ys).
```

Exercise Index Home Page

Exercise 3.2 (v) - sum/2

(a)

```
/* sum(Is,S) is true if S is the sum of the list of integers Is.           */
sum([],0).
sum([I|Is],S):-
  sum(Is,S0),
  plus(I,S0,S).
```

(b)

```
/* sum(Is,S) is true if S is the sum of the list of integers Is.           */
sum([],0).
sum([0|Is],S):-sum(Is,S).
sum([s(I)|Is],s(Z)):-sum([I|Is],Z).
```

Exercise Index Home Page

Exercise 3.3 (i) - substitute/4

```
/* substitute(X,Y,Xs,Ys) is true if the list Ys is the result of          */
/*   substituting Y for all occurrences of X in the list Xs.               */
substitute(X,Y,[],[]).
```

```
substitute(X,Y,[X|Xs],[Y|Ys]):-substitute(X,Y,Xs,Ys).
substitute(X,Y,[Z|Xs],[Z|Ys]):-X\=Z, substitute(X,Y,Xs,Ys).
```

Exercise Index Home Page

Exercise 3.3 (ii) - select/3

```
/* select(X,Xs,Ys) is true if Ys is the result of removing the first       */
/*   occurrence of X from Xs.                                              */
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-X\=Y,select(X,Ys,Zs).
```

Exercise Index Home Page

Exercise 3.3 (iii) - no_doubles/2

```
/* no_doubles(Xs, Ys) is true if Ys is the list of the elements appearing  */
/*   in Xs without duplication.  The elements in Ys are in the same order  */
/*   as in Xs with the last duplicate values being kept.                   */
no_doubles([], []).
no_doubles([X|Xs], Ys):-
  member(X, Xs),
  no_doubles(Xs, Ys).
no_doubles([X|Xs], [X|Ys]):-
  nonmember(X, Xs),
  no_doubles(Xs, Ys).

/* member(X,Xs) is true if X is a member of the list Xs.                   */
member(X,[X|Xs]).
member(X,[Y|Ys]):-member(X,Ys).

/* nonmember(X,Xs) is true if X is not a member of the list Xs.            */
nonmember(_, []).
nonmember(X, [Y|Ys]):-X =\= Y, nonmember(X, Ys).
```

Exercise Index Home Page

Exercise 3.3 (iv) - Even and Odd Permutations

```
/* even_permutation(Xs, Ys) is true if Ys is an even permutation of Xs.    */
even_permutation(Xs, Ys):-
  permute(Xs, Ys),
  sign_of_product_of_differences(Xs, 1, D),
  sign_of_product_of_differences(Ys, 1, E),
  D = E.

/* odd_permutation(Xs, Ys) is true if Ys is an odd permutation of Xs.      */
odd_permutation(Xs, Ys):-
  permute(Xs, Ys),
  sign_of_product_of_differences(Xs, 1, D),
  sign_of_product_of_differences(Ys, 1, E),
  D =\= E.

/* permute(Xs, Ys) is true if Ys is a permutation of the list Xs.          */
permute([], []).
permute([X|Xs], Ys1):-
  permute(Xs, Ys),
  pick(X, Ys1, Ys).
```

```
/* pick(X, Ys, Zs) is true if Zs is the result of removing one occurrence  */
/*   of the element X from the list Ys.                                    */
pick(X, [X|Xs], Xs).
pick(X, [Y|Ys], [Y|Zs]):-
  pick(X, Ys, Zs).

sign_of_product_of_differences([], D, D).
sign_of_product_of_differences([Y|Xs], D0, D):-
  sign_of_product_of_differences_1(Xs, Y, D0, D1),
  sign_of_product_of_differences(Xs, D1, D).

sign_of_product_of_differences_1([], _, D, D).
sign_of_product_of_differences_1([X|Xs], Y, D0, D):-
  Y =\= X,
  D1 is D0 * (Y - X) // abs(Y - X),
  sign_of_product_of_differences_1(Xs, Y, D1, D).
```

Exercise Index Home Page

Exercise 3.3 (v) - Merge Sort

```
/* merge_sort(Xs, Ys) is true if the list Ys is a sorted permutation of    */
/*   the list Xs.                                                          */
merge_sort([], []).
merge_sort([X], [X]).
merge_sort([Odd,Even|Xs], Ys):-
  split([Odd,Even|Xs], Odds, Evens),
  merge_sort(Odds, Os),
  merge_sort(Evens, Es),
  ordered_merge(Os, Es, Ys).

/* split(Xs, Os, Es) is true if Os is a list containing the odd positioned */
/*   elements of the list Xs, and Es is a list containing the even         */
/*   positioned elements of Xs.                                            */
split([], [], []).
split([X|Xs], [X|Os], Es):-split(Xs, Es, Os).

/* ordered_merge(Xs, Ys, Zs) is true if Zs is an ordered list obtained     */
/*   from merging the ordered lists Xs and Ys.                             */
ordered_merge([], Ys, Ys).
ordered_merge([X|Xs], [], [X|Xs]).
ordered_merge([X|Xs], [Y|Ys], [X|Zs]):-X < Y,  ordered_merge(Xs, [Y|Ys], Zs).
ordered_merge([X|Xs], [Y|Ys], [Y|Zs]):-X >= Y, ordered_merge([X|Xs], Ys, Zs).
```

Exercise Index Home Page

Exercise 3.4 (i) - subtree/2

```
/* subtree(Subtree,Tree) is true if Subtree is a subtree of the binary     */
/*   tree Tree.                                                            */
subtree(T,T).
subtree(S,tree(X,L,R)):-subtree(S,L).
subtree(S,tree(X,L,R)):-subtree(S,R).
```

Exercise Index Home Page

Exercise 3.4 (ii) - sum_tree/2

```
/* sum_tree(Tree,S) is true if S is the sum of the elements of the tree    */
/*   Tree.                                                                 */
sum_tree(void,0).
sum_tree(tree(X,L,R),S):-
  sum_tree(L,S1),
  sum_tree(R,S2),
  S is X+S1+S2.
```

Exercise Index Home Page

Exercise 3.4 (iii) - ordered_tree/1

```
/* ordered_tree(Tree) is true if Tree is an ordered tree.                  */
ordered_tree(void).
ordered_tree(tree(X,L,R)):-
  ordered_left(X,L),
  ordered_right(X,R).

/* ordered_left(X,Tree) is true if Tree is an ordered tree, and X is       */
/*   greater than all the elements of Tree.                                */
ordered_left(X,void).
ordered_left(X,tree(Y,L,R)):-
  X>Y,
  ordered_tree(tree(Y,L,R)),
  ordered_tree(tree(X,R,void)).

/* ordered_right(X,Tree) is true if Tree is an ordered tree, and X is less */
/*   than all the elements of Tree.                                        */
ordered_right(X,void).
ordered_right(X,tree(Y,L,R)):-
  X < Y,
  ordered_tree(tree(Y,L,R)),
  ordered_tree(tree(X,void,L)).
```

Exercise Index Home Page

Exercise 3.4 (iv) - tree_insert/3

```
/* tree_insert(X,Tree,Tree1) is true if Tree1 is the result of inserting   */
/*   the element X into the ordered tree Tree.                             */
tree_insert(X,void,tree(X,void,void)).
tree_insert(X,tree(X,L,R),tree(X,L,R)).
tree_insert(X,tree(Y,L,R),tree(Y,L1,R)):-
  X < Y,
  tree_insert(X,L,L1).
tree_insert(X,tree(Y,L,R),tree(Y,L,R1)):-
  X > Y,
  tree_insert(X,R,R1).
```

Exercise Index Home Page

Exercise 7.5 (i) - no_doubles/2

```
/* no_doubles(Xs, Ys) is true if Ys is the list of the elements appearing  */
/*   in Xs without duplication.  The elements in Ys are in the reverse     */
/*   order of Xs with the first duplicate values being kept.               */
no_doubles(Xs, Ys):-no_doubles_1(Xs, [], Ys).

no_doubles_1([], Ys, Ys).
```

```
no_doubles_1([X|Xs], As, Ys):-
  member(X, As),
  no_doubles_1(Xs, As, Ys).
no_doubles_1([X|Xs], As, Ys):-
  nonmember(X, As),
  no_doubles_1(Xs, [X|As], Ys).

/* member(X,Xs) is true if X is a member of the list Xs.                    */
member(X,[X|Xs]).
member(X,[Y|Ys]):-member(X,Ys).

/* nonmember(X,Xs) is true if X is not a member of the list Xs.            */
nonmember(X,[Y|Ys]):-X=\=Y, nonmember(X,Ys).
nonmember(X,[]).
```

Exercise Index Home Page

Exercise 8.2 (i) - triangle/2

```
/* triangle(N,T) is true if T is the Nth triangular number.                */
triangle(0,0).
triangle(N,T):-N>0, N1 is N - 1, triangle(N1,T1), T is T1+N.
```

Exercise Index Home Page

Exercise 8.2 (ii) - power/3

```
/* power(X,N,V) is true if V is X to the Nth power.                        */
power(0,N,0):-N>0.
power(X,0,1):-X>0.
power(X,N,V):-X>0, N>0, N1 is N - 1, power(X,N1,V1), V is V1*X.
```

Exercise Index Home Page

Exercise 8.3 (i) - triangle/2

```
/* triangle(N,T) is true if T is the Nth triangular number.                */
triangle(N,T):-triangle(N,0,T).

triangle(0,A,A).
triangle(N,A,T):-N>0, N1 is N - 1, B is A+N, triangle(N1,B,T).
```

Exercise Index Home Page

Exercise 8.3 (ii) - power/3

```
/* power(X,N,V) is true if V is X to the Nth power.                        */
power(X,N,V):-power(X,N,1,V).

power(0,N,A,0):-N>0.
power(X,N,A,V):-X>0, N>0, N1 is N - 1, B is A*X, power(X,N1,B,V).
power(X,0,A,A):-X>0.
```

Exercise Index Home Page

Exercise 8.3 (iii) - between/3

```
/* between(I,J,K) is true if K is an integer between I and J inclusive.   */
between(I,J,J):-J>=I.
```

```
between(I,J,K):-J>I, J1 is J - 1, between(I,J1,K).
```

Exercise Index Home Page

Exercise 8.3 (iv) - timeslist/2

```
/* timeslist(Is,Prod) is true if Prod is the product of the list Is.       */
timeslist([I|Is],Prod):-timeslist(Is,I,Prod).

timeslist([],Prod,Prod).
timeslist([I|Is],Temp,Prod):-Temp1 is Temp*I, timeslist(Is,Temp1,Prod).
```

Exercise Index Home Page

Exercise 8.3 (v) - Area of Polygon

```
/* area(Points,Area) is true if Area is the area of the polygon enclosed   */
/*   by the list of points Points, where the coordinates of each point are */
/*   represented by a pair p(X,Y) of integers.                             */
area(Ps,Area):- area(Ps,0,Area).

area([Pair],Area,Area).
area([p(X1,Y1),p(X2,Y2)|XYs],Temp,Area):-
  Temp1 is Temp + (X1*Y2 - Y1*X2)/2,
  area([p(X2,Y2)|XYs],Temp1,Area).
```

Exercise Index Home Page

Exercise 8.3 (vi) - min/2

```
/* min(Is,M) is true if M is the smallest element in the list Is.          */
min([I|Is],M):-min(Is,I,M).

min([],Min,Min).
min([I|Is],Temp,Min):-min_1(I,Temp,Temp1), min(Is,Temp1,Min).

min_1(I,J,I):-J >= I, !.
min_1(I,J,J):-J < I.
```

Exercise Index Home Page

Exercise 8.3 (vii) - length/2

```
/* length(Xs,L) is true if L is the length of the list Xs.                 */
length(Xs,L):-length(Xs,0,L).

length([],L,L).
length([X|Xs],L0,L):-L1 is L0+1, length(Xs,L1,L).
```

Exercise Index Home Page

Exercise 8.3 (viii) - range/3

```
/* range(I,J,Ks) is true if Ks is the list of integers between I and J     */
/*   inclusive.                                                            */
range(I,J,Ks):-range(I,J,[],Ks).

range(I,I,Ks,[I|Ks]):-!.
range(I,J,As,Ks):-I < J, J1 is J - 1, range(I,J1,[J|As],Ks).
```

Exercise Index Home Page

Exercise 9.1 (i) - flatten/2

This executes correctly using LPA Win-Prolog.

```
/* flatten(Xs,Ys) is true if Ys is a list of the elements in Xs.          */
/* e.g. flatten([[[3,c],5,[4,[]]],[1,b],a],[3,c,5,4,1,b,a]).              */
flatten(Xs,Ys):-flatten(Xs,[],Ys).

flatten([X|Xs],As,Ys):-flatten(Xs,As,As1), flatten(X,As1,Ys).
flatten(X,As,[X|As]):-integer(X).
flatten(X,As,[X|As]):-atom(X), X\=[].
flatten([],Ys,Ys).
```

Exercise Index Home Page

Exercise 9.2 (iv) - functor/3 and arg/3

This item is coded using LPA Win-Prolog syntax.

```
/* functor_(Term,F,Arity) is true if Term is a term whose principal       */
/*   functor has name F and arity Arity.                                  */
/* e.g. functor_(a(1,b),a,2).                                             */
functor_(Term,F,N):-Term=..[F|Args], length(Args,N).

/* arg_(N,Term,Arg) is true if Arg is the Nth argument of Term.           */
/* e.g. arg_(2,a(b,c,d),c).                                               */
arg_(N,Term,Arg):-Term=..[F|Args], position(N,Args,Arg).

/* position(N,Xs,X) is true if X is the Nth element in the list Xs.       */
position(1,[X|Xs],X).
position(N,[X|Xs],Y):-N>1, N1 is N-1, position(N1,Xs,Y).
```

Exercise Index Home Page

Exercise 12.5 (i) - abolish/2

This item is coded using LPA Win-Prolog syntax.

```
/* abolish_(F, N) retracts all clauses for the procedure F of arity N.    */
abolish_(F, N):-functor(T, F, N), repeat, abolish_(T), !.

abolish_(T):-retract(T), fail.
abolish_(T):-retract((T:-U)), fail.
abolish_(T).
```

Exercise Index Home Page

Exercise 14.1 (i) - Integer Square Root

```
/* squareroot(N, I) is true if I is the integer square root of the natural */
/*   number N.                                                             */
squareroot(N, I):-
  N >= 0,
  between(0, N, I),
  I * I =< N,
  (I + 1) * (I + 1) > N,
  !.
```

```
/* between(I, J, K) is true if K is an integer between I and J inclusive.  */
between(I, J, I):-I =< J.
between(I, J, K):-I < J, I1 is I + 1, between(I1, J, K).
```

Exercise Index Home Page

Exercise 14.1 (ii) - The Stable Marriage Problem

```
/* Goal: generate_and_test([a,b,c,d,e], [p,q,r,s,t], Xs).                  */
/*   or: backtrack([a,b,c,d,e], [p,q,r,s,t], Xs).                          */

/* preferences(Person, List) is true if the Person prefers people of the   */
/*   other sex in the order given in the List.                             */
preferences(a, [q,t,p,r,s]).  preferences(p, [e,a,d,b,c]).
preferences(b, [p,q,r,s,t]).  preferences(q, [d,e,b,a,c]).
preferences(c, [q,r,t,s,p]).  preferences(r, [a,d,b,c,e]).
preferences(d, [p,r,q,s,t]).  preferences(s, [c,b,d,a,e]).
preferences(e, [t,r,q,p,s]).  preferences(t, [d,b,c,e,a]).

/* stable(Men, Women, Marriages) is true if Marriages is a set of stable   */
/*   marriages between the Men and the Women.                              */
stable([], _, _).
stable([Man|Men], Women, Marriages):-
  stable_1(Women, Man, Marriages),
  stable(Men, Women, Marriages).

stable_1([], _, _).
stable_1([Woman|Women], Man, Marriages):-
  not(unstable(Man, Woman, Marriages)),
  stable_1(Women, Man, Marriages).

/* unstable(Man, Woman, Marriages) is true if the Man and the Woman both   */
/*   prefer each other to their spouses as defined by the set of Marriages.*/
unstable(Man, Woman, Marriages):-
  married(Man, Wife, Marriages),
  married(Husband, Woman, Marriages),
  prefers(Man, Woman, Wife),
  prefers(Woman, Man, Husband).

/* married(Man, Woman, Marriages) is true if the Man and the Woman are     */
/*   married as defined by the set of Marriages.                           */
married(Man, Woman, Marriages):-
  rest(m(Man, Woman), Marriages, _).

/* prefers(Person, OtherPerson, Spouse) is true if the Person prefers the  */
/*   OtherPerson to his Spouse.                                            */
prefers(Person, OtherPerson, Spouse):-
  preferences(Person, Preferences),
  rest(OtherPerson, Preferences, Rest),
  rest(Spouse, Rest, _).

/* rest(X, Ys, Zs) is true if X is a member of the list Ys, and the list   */
/*   Zs is the rest of the list following X.                               */
rest(X, [X|Ys], Ys):-!.
rest(X, [_|Ys], Zs):-rest(X, Ys, Zs).

/* select(X, Ys, Zs) is true if Zs is the result of removing one           */
/*   occurrence of the element X from the list Ys.                         */
select(X, [X|Ys], Ys).
```

```
select(X, [Y|Ys], [Y|Zs]):-select(X, Ys, Zs).

/*
 * Generate and Test
 */
/* generate_and_test(Men, Women, Marriages) is true if Marriages is a set  */
/*   of stable marriages between the Men and the Women.                    */
/* e.g.  generate_and_test([a,b,c,d,e], [p,q,r,s,t], Xs).                  */
generate_and_test(Men, Women, Marriages):-
  generate(Men, Women, Marriages),
  stable(Men, Women, Marriages).

/* generate(Men, Women, Marriages) is true if Marriages is a set of        */
/*   possible marriages between the Men and the Women.                     */
generate([], [], []).
generate([Man|Men], Women, [m(Man,Woman)|Marriages]):-
  select(Woman, Women, Women1),
  generate(Men, Women1, Marriages).

/*
 * Backtrack
 */
/* backtrack(Men, Women, Marriages) is true if Marriages is a set of       */
/*   stable marriages between the Men and the Women.                       */
/* e.g.  backtrack([a,b,c,d,e], [p,q,r,s,t], Xs).                          */
backtrack(Men, Women, Marriages):-
  backtrack_1(Men, Women, Men, Women, [], Marriages).

backtrack_1([], _, _, _, Marriages, Marriages).
backtrack_1([Man|Men], Women, Men0, Women0, Marriages0, Marriages):-
  select(Woman, Women, Women1),
  Marriages1 = [m(Man,Woman)|Marriages0],
  stable(Men0, Women0, Marriages1),
  backtrack_1(Men, Women1, Men0, Women0, Marriages1, Marriages).
```

Exercise Index Home Page

Exercise 14.1 (iv) - Houses Puzzle

```
solve:-
  clues(Houses),
  queries(Houses).

clues(Houses):-
  house(A, Houses), colour(A, red), nationality(A, english),          /*  1 */
  house(B, Houses), nationality(B, spaniard), pet(B, dog),            /*  2 */
  house(C, Houses), colour(C, green), drink(C, coffee),               /*  3 */
  house(D, Houses), nationality(D, ukranian), drink(D, tea),          /*  4 */
  immed_to_right(Houses, E, F), colour(E, green), colour(F, ivory),   /*  5 */
  house(G, Houses), smoke(G, winston), pet(G, snails),                /*  6 */
  house(H, Houses), smoke(H, kools), colour(H, yellow),               /*  7 */
  middle(Houses, I), drink(I, milk),                                  /*  8 */
  first(Houses, J), nationality(J, norwegian),                        /*  9 */
  next_to(Houses, K, L), smoke(K, chesterfields), pet(L, fox),        /* 10 */
  next_to(Houses, M, N), smoke(M, kools), pet(N, horse),              /* 11 */
  house(O, Houses), smoke(O, luckystrike), drink(O, orangejuice),     /* 12 */
  house(P, Houses), nationality(P, japanese), smoke(P, parliaments),  /* 13 */
  next_to(Houses, Q, R), nationality(Q, norwegian), colour(R, blue). /* 14 */

queries(Houses):-
```

```
  house(X, Houses), pet(X, zebra), nationality(X, Nationality1),
  write("The "), write(Nationality1), write(" owns the zebra"), nl,
  house(Y, Houses), drink(Y, water), nationality(Y, Nationality2),
  write("The "), write(Nationality2), write(" drinks water"), nl.

colour(house(C,_,_,_,_), C).
nationality(house(_,N,_,_,_), N).
pet(house(_,_,P,_,_), P).
drink(house(_,_,_,D,_), D).
smoke(house(_,_,_,_,S), S).

first(houses(X,_,_,_,_), X).

middle(houses(_,_,X,_,_), X).

immed_to_right(houses(L,R,_,_,_), R, L).
immed_to_right(houses(_,L,R,_,_), R, L).
immed_to_right(houses(_,_,L,R,_), R, L).
immed_to_right(houses(_,_,_,L,R), R, L).

next_to(Xs, X, Y):-
  immed_to_right(Xs, X, Y).
next_to(Xs, X, Y):-
  immed_to_right(Xs, Y, X).

house(X, houses(X,_,_,_,_)).
house(X, houses(_,X,_,_,_)).
house(X, houses(_,_,X,_,_)).
house(X, houses(_,_,_,X,_)).
house(X, houses(_,_,_,_,X)).
```

Exercise Index Home Page

Exercise 14.3 (i) - an NPDA

```
/* p(Xs) is true if Xs is a list consisting of n a's followed by n b's.    */
p(Xs):-p_1(q0,Xs,[]).

p_1(q1,[],[]).
p_1(q0,[a|Xs],S):-p_1(q0,Xs,[a|S]).
p_1(q0,[b|Xs],[a|S]):-p_1(q1,Xs,S).
p_1(q1,[b|Xs],[a|S]):-p_1(q1,Xs,S).
```

Exercise Index Home Page

Exercise 15.1 (i) - flatten/2

This executes correctly using LPA Win-Prolog.

```
/* flatten(Xs,Ys) is true if Ys is a list of the elements of Xs (in        */
/*   reversed order).                                                      */
/* e.g. flatten([[[3,c],5,[4,[]]],[1,b],a],[a,b,1,4,5,c,3]).              */
flatten(Xs,Ys):-flatten(Xs,[],Ys).

flatten([X|Xs],As,Ys):-flatten(X,As,As1), flatten(Xs,As1,Ys).
flatten(X,As,[X|As]):-integer(X).
flatten(X,As,[X|As]):-atom(X), X\=[].
flatten([],Ys,Ys).
```

Exercise Index Home Page

Exercise 15.1 (ii) - Binary Trees

```
/* pre_order(Tree,L) is true if L is a pre-order traversal of the binary   */
/*   tree Tree.                                                            */
pre_order(T,L):-pre_order(T,L,[]).

pre_order(void,Xs,Xs).
pre_order(tree(X,L,R),[X|Xs],Zs):-
  pre_order(L,Xs,Ys),
  pre_order(R,Ys,Zs).

/* in_order(Tree,L) is true if L is an in-order traversal of the binary    */
/*   tree Tree.                                                            */
in_order(T,L):-in_order(T,L,[]).

in_order(void,Xs,Xs).
in_order(tree(X,L,R),Xs,Zs):-
  in_order(L,Xs,[X|Ys]),
  in_order(R,Ys,Zs).

/* post_order(Tree,L) is true if L is a post-order traversal of the binary */
/*   tree Tree.                                                            */
post_order(T,L):-post_order(T,L,[]).

post_order(void,Xs,Xs).
post_order(tree(X,L,R),Xs,Zs):-
  post_order(L,Xs,Ys),
  post_order(R,Ys,[X|Zs]).
```

Exercise Index Home Page

Exercise 15.1 (iii) - Towers of Hanoi

```
/* hanoi(N,A,B,C,Moves,[]) is true if Moves is the sequence of moves       */
/*   required to move N discs from peg A to peg B using peg C as an        */
/*   intermediary according to the rules of the Tower of Hanoi puzzle.     */
hanoi(1,A,B,C,[to(A,B)|Zs],Zs).
hanoi(N,A,B,C,Xs,Zs):-
  N>1, N1 is N - 1,
  hanoi(N1,A,C,B,Xs,[to(A,B)|Ys]),
  hanoi(N1,C,B,A,Ys,Zs).
```

Exercise Index Home Page

Exercise 17.1 (i) - intersect/3

```
intersect(Xs,Ys,Zs):-
  findall(Z, (member(Z,Xs),member(Z,Ys)), Zs).
```

Exercise Index Home Page

Exercise 18.1 (ii) - Missionaries and Cannibals

```
/* p(WhereTheBoatIs, MissionariesOnLeft, CannibalsOnLeft)                  */
/* m(MissionariesInBoat, CannibalsInBoat)                                  */

initial_state(p(left, 3, 3)).

final_state(p(right, 0, 0)).
```

```
move(p(left,  M, _), m(1, 0)):-M >= 1.
move(p(left,  _, C), m(0, 1)):-C >= 1.
move(p(left,  M, C), m(1, 1)):-M >= 1, C >= 1.
move(p(left,  M, _), m(2, 0)):-M >= 2.
move(p(left,  _, C), m(0, 2)):-C >= 2.
move(p(right, M, _), m(1, 0)):-(3 - M) >= 1.
move(p(right, _, C), m(0, 1)):-(3 - C) >= 1.
move(p(right, M, C), m(1, 1)):-(3 - M) >= 1, (3 - C) >= 1.
move(p(right, M, _), m(2, 0)):-(3 - M) >= 2.
move(p(right, _, C), m(0, 2)):-(3 - C) >= 2.

update(p(left, M0, C0), m(MB, CB), p(right, M, C)):-
  M is M0 - MB, C is C0 - CB.
update(p(right, M0, C0), m(MB, CB), p(left, M, C)):-
  M is M0 + MB, C is C0 + CB.

/* This uses an (under)estimation of the number of remaining voyages as    */
/*   the evaluation function.                                              */
value(p(_,     M, C), 1):-M + C =:= 1, !.
value(p(left,  M, C), L):-L is (M + C - 2) * 2 + 1.
value(p(right, M, C), L):-L is (M + C) * 2.

/* Ensures that, on each bank, the cannibals are not outnumbered */
legal(p(_, _, 3)):-!.
legal(p(_, _, 0)):-!.
legal(p(_, M, M)).
```

Exercise Index Home Page

Exercise 18.1 (iii) - Five Jealous Husbands

```
/* p(WhereTheBoatIs, PeopleOnIsle, PeopleOnBank) */

initial_state(p(isle, [h1,h2,h3,h4,h5,w1,w2,w3,w4,w5], [])).

final_state(p(bank, [], [h1,h2,h3,h4,h5,w1,w2,w3,w4,w5])).

move(p(isle, I, _), [P1]):-       /* Move one person from isle to bank     */
  rest(I, P1, _).
move(p(isle, I, _), [P1,P2]):-    /* Move two people from isle to bank     */
  rest(I, P1, I1), rest(I1, P2, _),
  legal_1([P1,P2]).
move(p(isle, I, _), [P1,P2,P3]):- /* Move three people from isle to bank   */
  rest(I, P1, I1), rest(I1, P2, I2), rest(I2, P3, _),
  legal_1([P1,P2,P3]).
move(p(bank, _, B), [P1]):-       /* Move one person from bank to isle     */
  rest(B, P1, _).
move(p(bank, _, B), [P1,P2]):-    /* Move two people from bank to isle     */
  rest(B, P1, B1), rest(B1, P2, _),
  legal_1([P1,P2]).
move(p(bank, _, B), [P1,P2,P3]):- /* Move three people from bank to isle   */
  rest(B, P1, B1), rest(B1, P2, B2), rest(B2, P3, _),
  legal_1([P1,P2,P3]).

update(p(isle, I1, B1), Boat, p(bank, I2, B2)):-
  ordered_delete(Boat, I1, I2), ordered_insert(Boat, B1, B2).
update(p(bank, I1, B1), Boat, p(isle, I2, B2)):-
  ordered_delete(Boat, B1, B2), ordered_insert(Boat, I1, I2).
```

```
/* This uses an (under)estimation of the number of remaining voyages as    */
/*   the evaluation function.                                               */
value(p(isle, Xs, _), L):-length(Xs, 0, M), L is (M // 2) * 2 - 1.
value(p(bank, Xs, _), L):-length(Xs, 0, M), L is (M + 1) // 2 * 2.

legal(p(_, Xs, Ys)):-legal_1(Xs), legal_1(Ys).

legal_1(Xs):-only_wives(Xs), !.
legal_1(Xs):-wives_with_husbands(Xs, Xs).

only_wives([]).
only_wives([W|Xs]):-couple(_, W), only_wives(Xs).

wives_with_husbands([], _).
wives_with_husbands([H|Xs], Ys):-
  couple(H, _), !, wives_with_husbands(Xs, Ys).
wives_with_husbands([W|Xs], Ys):-
  couple(H, W), rest(Ys, H, _), !, wives_with_husbands(Xs, Ys).

couple(h1,w1). couple(h2,w2). couple(h3,w3). couple(h4,w4). couple(h5,w5).

/* ordered_delete(Xs, Ys, Zs) is true if Zs is the ordered list obtained   */
/*   by deleting the ordered list Xs from the ordered list Ys.             */
ordered_delete([], Ys, Ys).
ordered_delete([X|Xs], [X|Ys], Zs):-!,
  ordered_delete(Xs, Ys, Zs).
ordered_delete([X|Xs], [Y|Ys], Zs):-
  X > Y, !, Zs = [Y|Ws], ordered_delete([X|Xs], Ys, Ws).
ordered_delete([_|Xs], [Y|Ys], [Y|Zs]):-
  ordered_delete(Xs, Ys, Zs).

/* ordered_insert(Xs, Ys, Zs) is true if Zs is the ordered list obtained   */
/*   by inserting the ordered list Xs in the ordered list Ys.              */
ordered_insert([], Ys, Ys).
ordered_insert([X|Xs], [Y|Ys], Zs):-
  Y =< X, !, Zs = [Y|Ws], ordered_insert([X|Xs], Ys, Ws).
ordered_insert([X|Xs], Ys, [X|Zs]):-
  ordered_insert(Xs, Ys, Zs).

/* rest(Xs, Y, Zs) is true if Zs is the list of elements following the     */
/*   element Y in the list Xs.                                             */
rest([X|Xs], X, Xs).
rest([_|Xs], Y, Zs):-rest(Xs, Y, Zs).

/* length(Xs, L0, L) is true if L is equal to L0 plus the number of        */
/*   elements in the list Xs.                                              */
length([], L, L).
length([_|Xs], L0, L):-L1 is L0 + 1, length(Xs, L1, L).
```

Exercise Index Home Page

Exercise 18.1 (iv) - Breadth First Framework

This item is coded using LPA Win-Prolog syntax.

```
breadth(Moves):-
  initial_state(Position),
  empty_queue(Queue0),
  enqueue(b(Position, []), Queue0, Queue),
  breadth_first(Queue, [], Moves).
```

```
breadth_first(Queue, _, FinalMoves):-
  dequeue(b(Position, Path), Queue, _),
  final_state(Position),
  reverse(Path, [], FinalMoves).
breadth_first(Queue0, History, FinalMoves):-
  dequeue(b(Position, Path), Queue0, Queue1),
  findall(Move, move(Position, Move), Moves),
  filter(Moves, Position, Path, History, Queue1, Queue),
  breadth_first(Queue, [Position|History], FinalMoves).

filter([], _, _, _, Queue, Queue).
filter([Move|Moves], Position, Path, History, Queue0, Queue):-
  update(Position, Move, Position1),
  legal(Position1),
  not(member(Position1, History)),
  !,
  enqueue(b(Position1, [Move|Path]), Queue0, Queue1),
  filter(Moves, Position, Path, History, Queue1, Queue).
filter([_|Moves], Position, Path, History, Queue0, Queue):-
  filter(Moves, Position, Path, History, Queue0, Queue).

empty_queue(q(zero, Ys, Ys)).

enqueue(X, q(N, Ys, [X|Zs]), q(s(N), Ys, Zs)).

dequeue(X, q(s(N), [X|Ys], Zs), q(N, Ys, Zs)).
```

Exercise Index Home Page

Exercise 18.1 (v) - Eight Queens Puzzle

```
initial_state(posn(8,[])).

final_state(posn(N,Qs)):-length(Qs, N).

move(posn(Q,Qs), Q):-
  not(member(Q, Qs)),
  not(attack(Qs, 1, Q)).
move(posn(N,Qs), Q):-
  N > 1,
  N1 is N - 1,
  move(posn(N1,Qs), Q).

attack([Q0|_], I, Q):-
  Q is Q0 + I, !.
attack([Q0|_], I, Q):-
  Q is Q0 - I, !.
attack([_|Qs], I, Q):-
  I1 is I + 1,
  attack(Qs, I1, Q).

update(posn(N,Qs), Q, posn(N,[Q|Qs])).

legal(_).

length(Xs, L):-length_1(Xs, 0, L).

length_1([_|Xs], L0, L):-L1 is L0 + 1, length_1(Xs, L1, L).
length_1([], L, L).
```

```
member(X, [X|_]).
member(X, [_|Xs]):-member(X, Xs).
```

Exercise Index Home Page