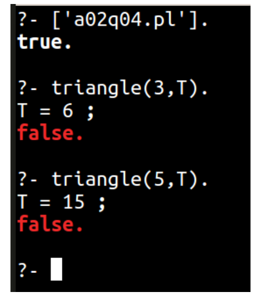
All exercise numbers refer to the Sterling and Shapiro text book.

1. Exercises 8.3.1 (i), (iii), (vi), (vii) {SAME FROM PREVIOUS ASSIGNMENT}*

(i) Write an iterative version for *triangle(N,T)*, posed as Exercise 8.2(i).

Ans.

```
% triangle(N,T): T is sum of first N natural numbers.
triangle(N,T) :- triangle(0,N,0,T).

triangle(I,N,A,T) :-
      I < N, I1 is I+1,
      A1 is A+I1, triangle(I1,N,A1,T).

triangle(N,N,T,T).
```



(iii) Rewrite Program 8.5 so that the successive integers are generated in descending order.

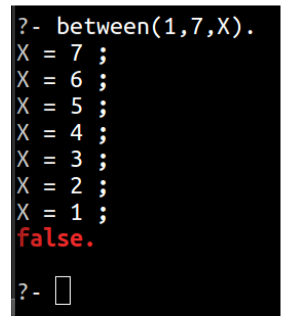
Ans.

```
% between(I,J,K): K is an integer between the integers I and J inclusive.

between(I,J,I) :- I <= J.
between(I,J,K) :- I < J, I1 is I+1, between(I1,J,K).
```

Program 8.5 Generating a range of integers

Ans.

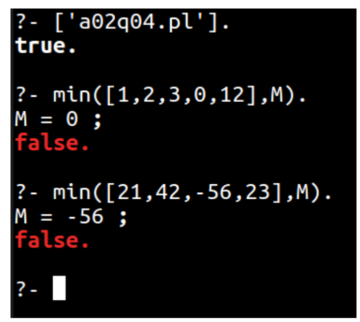
```
% between(I,J,K): K is an integer between the
%     integers I and J inclusive.

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



(vi) Write a program to find the minimum of a list of integers.

Ans.

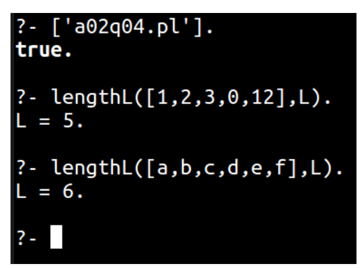
```
% min(List,least): least integer from List
%min([M|_],M) :- min([M|_],M).
min([M],M).

min([H|T],H) :-
      min(T,M), H =< M.
min([H|T],M) :-
      min(T,M), M < H.
```



(vii) Rewrite Program 8.11 for finding the length of a list so that it is iterative. (Hint: Use a counter, as in Program 8.3)

Ans.

```
% length_itr(List,A,L): L is length of List,
%     A being accumulator

lengthL([],0).
lengthL([H|T],L) :- length_itr([H|T],0,L).
length_itr([],A,A).

length_itr([H|T],A,L) :-
      A1 is A+1, length_itr(T,A1,L).
```



2. Exercises 9.2.1 (i), (ii), (iv), (v).

(i) Define a predicate *occurrences(Sub,Term,N)*, true if *N* is the number of occurrences of subterm *Sub* in *Term*. Assume that *Term* is ground.

Ans.

```
occurrences(Sub,Term,N) :-
      occurrences(Sub,Term,0,N).

occurrences(Term,Term,N,R) :-
      N1 is N+1,
      compound(Term), !,
      Term=..[_|Args],
occurrencesList(Term,Args,N1,R).

occurrences(Sub,Term,N,R) :-
      compound(Term), !,
      Term=..[_|Args],
      occurrencesList(Sub,Args,N,R).

occurrences(Term,Term,N,N1) :- N1 is N+1.
occurrences(_,_,N,N).

occurrencesList(_,[],N,N).
occurrencesList(Sub,[Arg|Args], N, N2) :-
      occurrencesList(Sub,Args,N,N1),
      occurrences(Sub,Arg,N1,N2).
```

```
?- ['hw03q02.pl'].
true.

?- subterm(a,f(X,Y)).
X = a ;
Y = a ;
false.

?- functor(length([q,w,e,r],8), F, N).
F = length,
N = 2.

?- arg(2,length([a,c,e,r],7),Y).
Y = 7.

?- occurrences(a,f(X,Y),Occs).
X = Y, Y = a,
Occs = 2 ;
Y = a,
Occs = 1 ;
X = a,
Occs = 1 ;
Occs = 0 ;
false.
```

(ii) Define a predicate *position(Subterm,Term,Position)*, where *Position* is a list of argument positions identifying *Subterm* within *Term*.

For example, the position of X in 2*sin(X) is [2,1].

Ans.

```
position(Sub,Sub,[]).
position(Sub,Term,[N|T]) :-
      compound(Term), arg(N,Term,Arg),
      subterm(Sub,Arg), position(Sub,Arg,T).
```

```
?- ['hw03q02.pl'].
true.

?- functor(2*sin(x)+length([b,c,d],a),F,N).
F =  (+),
N = 2.

?- position(a,2*sin(x)+length([b,c,d],a),L).
L = [2, 2] ;
false.

?- position(a,2*sin(a),L).
L = [2, 1] ;
false.

?-
```

(iv) Define *functor* and *arg* in terms of *univ*. How can the programs be used?

Ans.

```
%iv. functor and arg using UNIV
functr(Term, F, N) :-
      Term=..[F|T], length(T,N).

argUV(N,Term,Arg) :-
      Term=..[_|T], nth1(N,T,Arg).
```

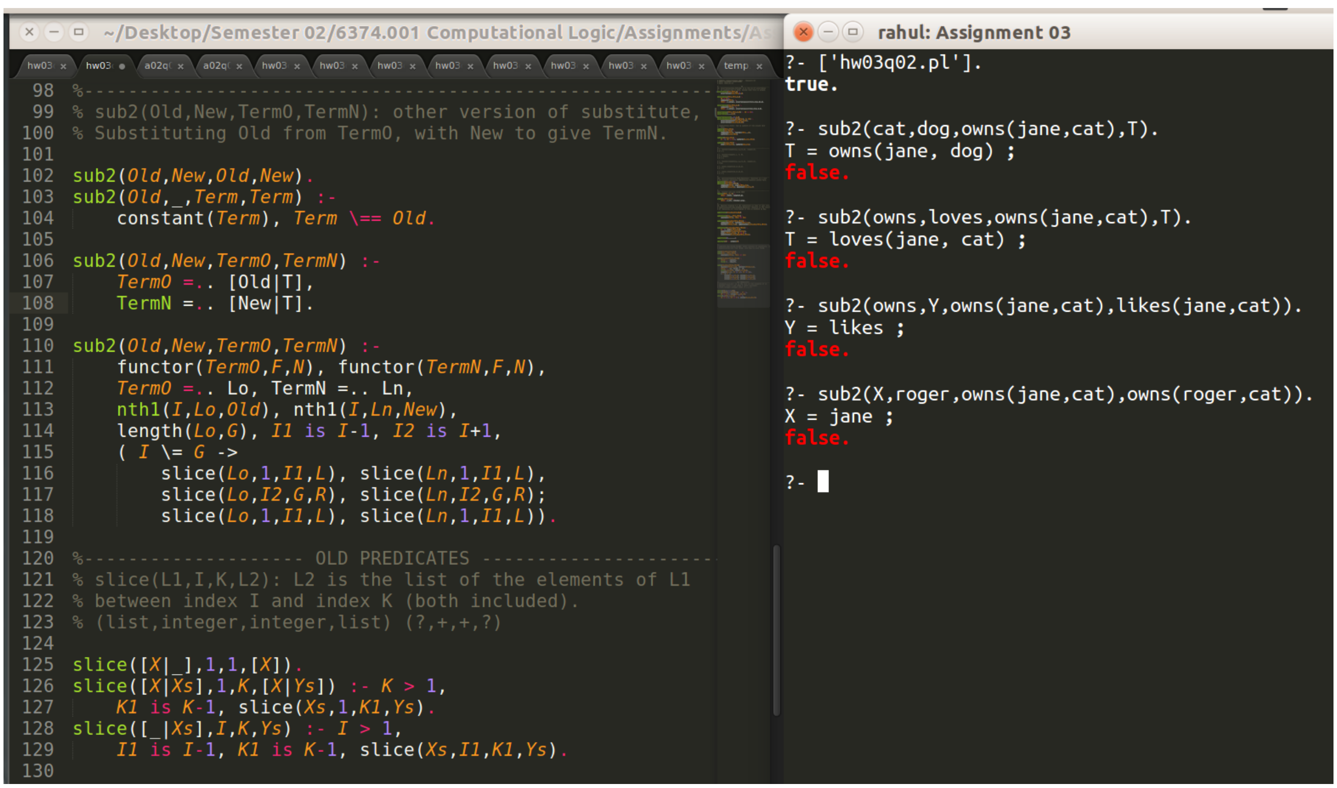
```
?- ['hw03q02.pl'].
true.

?- functor(length([a,c,e,r],7), F, N).
F = length,
N = 2.

?- functr(length([a,c,e,r],7), F, N).
F = length,
N = 2.

?- arg(2,length(X,Y,Z),What).
Y = What.

?- argUV(2,length(X,Y,Z),What).
Y = What ;
false.

?-
```

(v) Rewrite Program 9.3 for *substitute* so that it uses *univ*.

```
%substitute(Old,New,OldTerm,NewTerm):
%     NewTerm is the result of replacing all occurrences of Old in OldTerm by New.

substitute(Old,New,Old,New).

substitute(Old,_,Term,Term) :-
      constant(Term), Term \= Old.

substitute(Old,New,Term,Term1) :-
      compound(Term), functor(Term,F,N),
      functor(Term1,F,N),
      substitute(N,Old,New,Term,Term1).

substitute(N,Old,New,Term,Term1) :-
      N > 0, arg(N,Term,Arg),
      substitute(Old,New,Arg,Arg1),
      arg(N,Term1,Arg1), N1 is N-1,
      substitute(N1,Old,New,Term,Term1).
substitute(0,Old,New,Term,Term1).

constant(X) :- atomic(X).
```

Program 9.3 A program for substituting in a term.

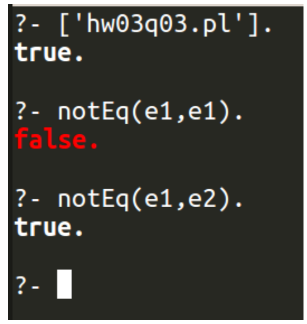
Ans.



3. Exercises 11.3 (i) and (ii)

(i) Define the system predicate \== using == and the cut-fail combination.
Ans.

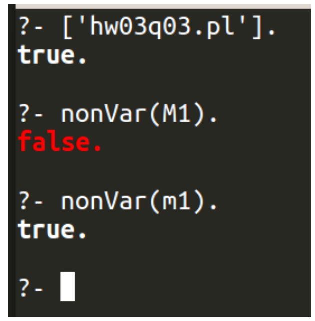
```
notEq(X,Y) :-
      neg(X==Y).

neg(P) :- P, !, fail.
neg(_).
```



```
?- ['hw03q03.pl'].
true.

?- notEq(e1,e1).
false.

?- notEq(e1,e2).
true.

?-
```

(ii) Define *nonvar* using *var* and the cut-fail combination.
Ans.

```
nonVar(Term):- var(Term), !, fail.
nonVar(Term):-
      neg(var(Term)).
```



```
?- ['hw03q03.pl'].
true.

?- nonVar(M1).
false.

?- nonVar(m1).
true.

?-
```

4. Belgian Snake Problem

```prolog
% CS6374: Computational Logic - Homework 03
% Rahul Nalawade [RSN170330]
% Date: 2018-02-14

%------------------------- Q4 --------------------------
% snake(Pattern,Row,Column): represents belgian snake
% in Pattern forming a rectangle.
snake(_,_,[]).

snake(P,R,[C]) :-
    makeFull(P,R,C,_), writeList(C).

snake(P,R,[C1,C2|Cs]) :-
    makeFull(P,R,C1,P1),
    makeFull(P1,R,Temp,P2),
    reverse(Temp,C2),
    writeList(C1), writeList(C2),
    snake(P2,R,Cs).

%--------------------------------------------------------
% rotate([a,b,c],[b,c,a]) is true.
rotate([],[]).
rotate([X|T],R) :-
    append(T,[X],R).

% makeFull(Pattern,Row,R,Pn): Fills Row with cyclic
% patterns to give R, with Pn as the last rotation.
makeFull(Plast,[],[],Plast).
makeFull([P|Ps],[_|Rs],[P|Ls],Plast) :-
    rotate([P|Ps],Pnext),
    makeFull(Pnext,Rs,Ls,Plast).

% writes the List on a new-line
writeList([]) :- nl.
writeList([H|T]) :- write(H), write(' '), writeList(T).

%------------------- OLD PREDICATES ---------------------
% append(X,Y,Z): appends Y to X to give Z.
append([],[],[]).
```

```
rahul:Assignment 03$ swipl -q hw03q04.pl
?- snake( [a,b,c,d,e], [_,_,_,_,_], [_,_,_,_]).
a b c d e
e d c b a
a b c d e
e d c b a
true ;
false.

?- snake( [a,b,c,d,e], [_,_,_,_,_], [_,_,_]).
a b c d e
e d c b a
a b c d e
true ;
false.

?- snake( [a,b,c,d,e], [_,_,_,_], [_,_,_]).
a b c d
c b a e
d e a b
true ;
false.

?- snake( [a,b,c,d], [_,_,_,_,_], [_,_,_]).
a b c d a
b a d c b
c d a b c
true ;
false.

?- snake( [a,b,c], [_,_,_,_,_], [_,_,_]).
a b c a b
a c b a c
b c a b c
true ;
false.

?-
```

5. Program the N-Queen problem from the book.

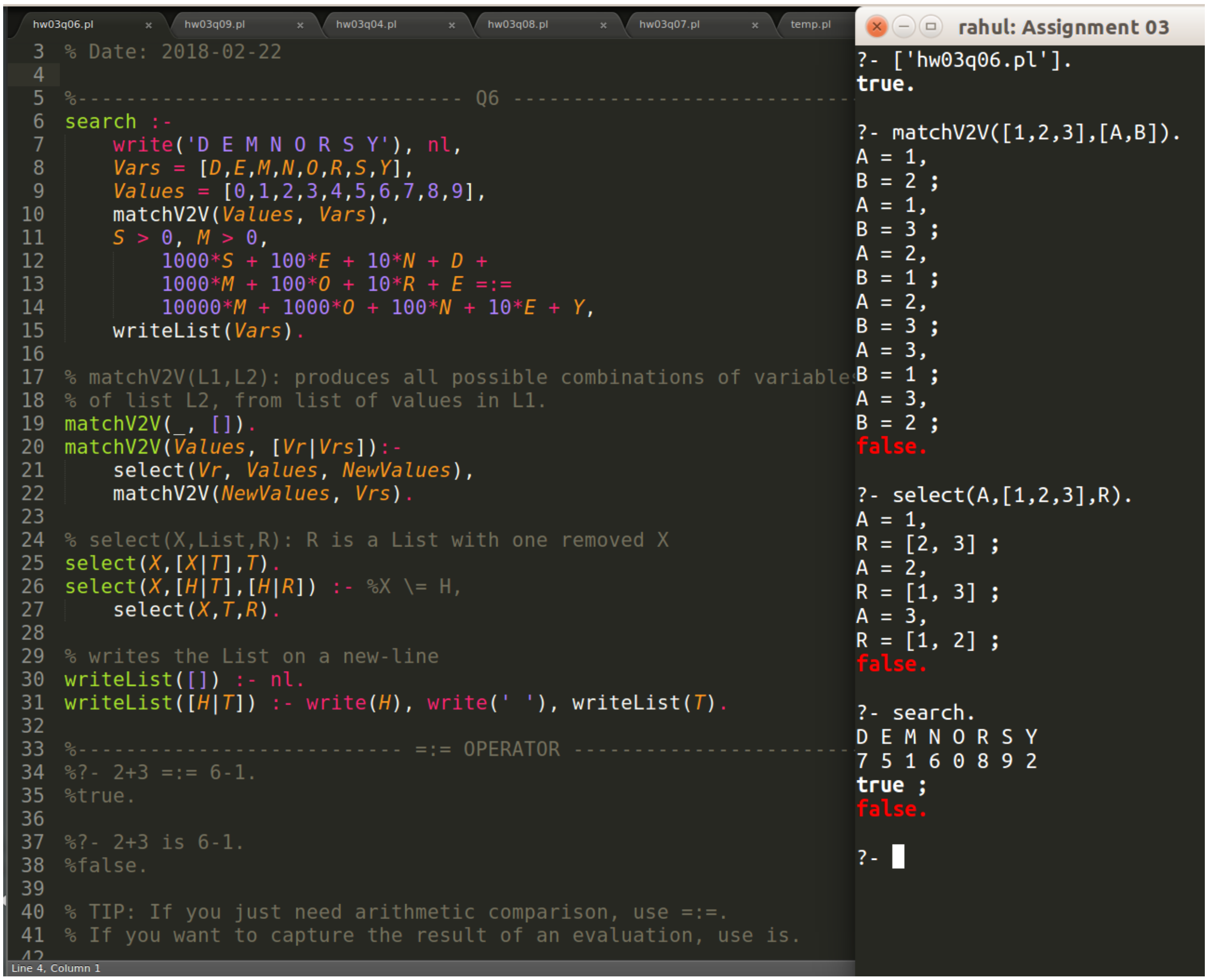
```prolog
% CS6374: Computational Logic - Homework 03
% Rahul Nalawade [RSN170330]
% Date: 2018-02-13

%------------------------- Q5 --------------------------
% queens(N,Queens): Queens is a placement that solves
% the N queens problem, represented as a permutation of
% the list of numbers [1,2,3,..,N].

queens(N,Qs) :-
    range(1,N,Ns), permutation(Ns,Qs), safe(Qs).

% The placement Qs is safe/ valid.
safe([]).
safe([Q|Qs]) :-
    safe(Qs), \+ attack(Q,Qs).

attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|_]) :-
    X is Y+N; X is Y-N.
attack(X,N,[_|Ys]) :-
    N1 is N+1, attack(X,N1,Ys).

% permutation(L1,P): P is a permutation of L1
permutation([],[]).
permutation(Xs,[Z|Zs]) :-
    select(Z,Xs,Ys), permutation(Ys,Zs).

% range(M,N,List): List is list of integers [M,N], M<N.
range(N,N,[N]).
range(M,N,[M|Ns]) :-
    M < N, M1 is M+1, range(M1,N,Ns).

% select(X,List,R): R is a List with one removed X
select(X,[X|T],T).
select(X,[H|T],[H|R]) :- %X \= H,
    select(X,T,R).
%--------------------------------------------------------
```

```
?- ['hw03q05.pl'].
true.

?- queens(5,S).
S = [1, 3, 5, 2, 4] ;
S = [1, 4, 2, 5, 3] ;
S = [2, 4, 1, 3, 5] ;
S = [2, 5, 3, 1, 4] ;
S = [3, 1, 4, 2, 5] ;
S = [3, 5, 2, 4, 1] ;
S = [4, 1, 3, 5, 2] ;
S = [4, 2, 5, 3, 1] ;
S = [5, 2, 4, 1, 3] ;
S = [5, 3, 1, 4, 2] ;
false.

?- queens(4,S).
S = [2, 4, 1, 3] ;
S = [3, 1, 4, 2] ;
false.

?- queens(4,[3,1,4,2]).
true ;
false.

?- queens(4,[3,1,2,4]).
false.

?- queens(3,S).
false.

?- queens(2,S).
false.

?-
```

6. Write a Prolog program to solve cryptarithmetic addition problems such as

```
   S E N D
 + M O R E
 ---------
 M O N E Y
```

The solution is D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2. Each letter should stand for a unique digit. If there is a solution, Prolog should return the list of letters and corresponding digits. If there is no solution, Prolog should report ‘no’.




```prolog
% Date: 2018-02-22

%-------------------------------- Q6 ----------------------------
search :-
    write('D E M N O R S Y'), nl,
    Vars = [D,E,M,N,O,R,S,Y],
    Values = [0,1,2,3,4,5,6,7,8,9],
    matchV2V(Values, Vars),
    S > 0, M > 0,
        1000*S + 100*E + 10*N + D +
        1000*M + 100*O + 10*R + E =:=
        10000*M + 1000*O + 100*N + 10*E + Y,
    writeList(Vars).

% matchV2V(L1,L2): produces all possible combinations of variables
% of list L2, from list of values in L1.
matchV2V(_, []).
matchV2V(Values, [Vr|Vrs]):-
    select(Vr, Values, NewValues),
    matchV2V(NewValues, Vrs).

% select(X,List,R): R is a List with one removed X
select(X,[X|T],T).
select(X,[H|T],[H|R]) :- %X \= H,
    select(X,T,R).

% writes the List on a new-line
writeList([]) :- nl.
writeList([H|T]) :- write(H), write(' '), writeList(T).

%------------------------- =:= OPERATOR ----------------------
%?- 2+3 =:= 6-1.
%true.

%?- 2+3 is 6-1.
%false.

% TIP: If you just need arithmetic comparison, use =:=.
% If you want to capture the result of an evaluation, use is.
```

```
?- ['hw03q06.pl'].
true.

?- matchV2V([1,2,3],[A,B]).
A = 1,
B = 2 ;
A = 1,
B = 3 ;
A = 2,
B = 1 ;
A = 2,
B = 3 ;
A = 3,
B = 1 ;
A = 3,
B = 2 ;
false.

?- select(A,[1,2,3],R).
A = 1,
R = [2, 3] ;
A = 2,
R = [1, 3] ;
A = 3,
R = [1, 2] ;
false.

?- search.
D E M N O R S Y
7 5 1 6 0 8 9 2
true ;
false.

?-
```

NOTE: Here, S and M are > 0. Hence, S,M can take values only from 1 to 9.

7. Solve the stable marriage problem in Exercise 14.1 (ii) pg. 261

Write a program to solve the stable marriage problem (Sedgewick, 1983), stated as follows:

Suppose there are *N* men and *N* women who want to get married. Each man has a list of all the women in his preferred order, and each woman has a list of all the men in her preferred order. The problem is to find a set of marriages that is stable.

A pair of marriages is *unstable,* if there are a man and woman who prefer each other to their spouses. For example, consider the pair of marriages where David is married to Paula, and Jeremy is married to Judy. If David prefers Judy to Paula. and Judy prefers David to Jeremy, the pair of marriages is unstable. This pair would also be unstable if Jeremy preferred Paula to Judy, and Paula preferred Jeremy to David.

A set of marriages is stable if there is no pair of *unstable* marriages.

Your program should have as input lists of preferences. and produce as output a stable set of marriages. It is a theorem from graph theory that this is always possible. Test the program on the following five men and five women with their associated preferences:

avraham: chana tamar zvia ruth sarah
binyamin: zvia chana ruth sarah tamar
chaim: chana ruth tamar sarah zvia
david: zvia ruth chana sarah tamar
elazar: tamar ruth chana zvia sarah

zvia: elazar avraham david binyamin chaim
chana: david elazar binyamin avraham chaim
ruth: avraham david binyamin chaim elazar
sarah: chaim binyamin david avraham elazar
tamar: david binyamin chaim elazar avraham

hw03q06.pl hw03q08.pl hw03q07.pl

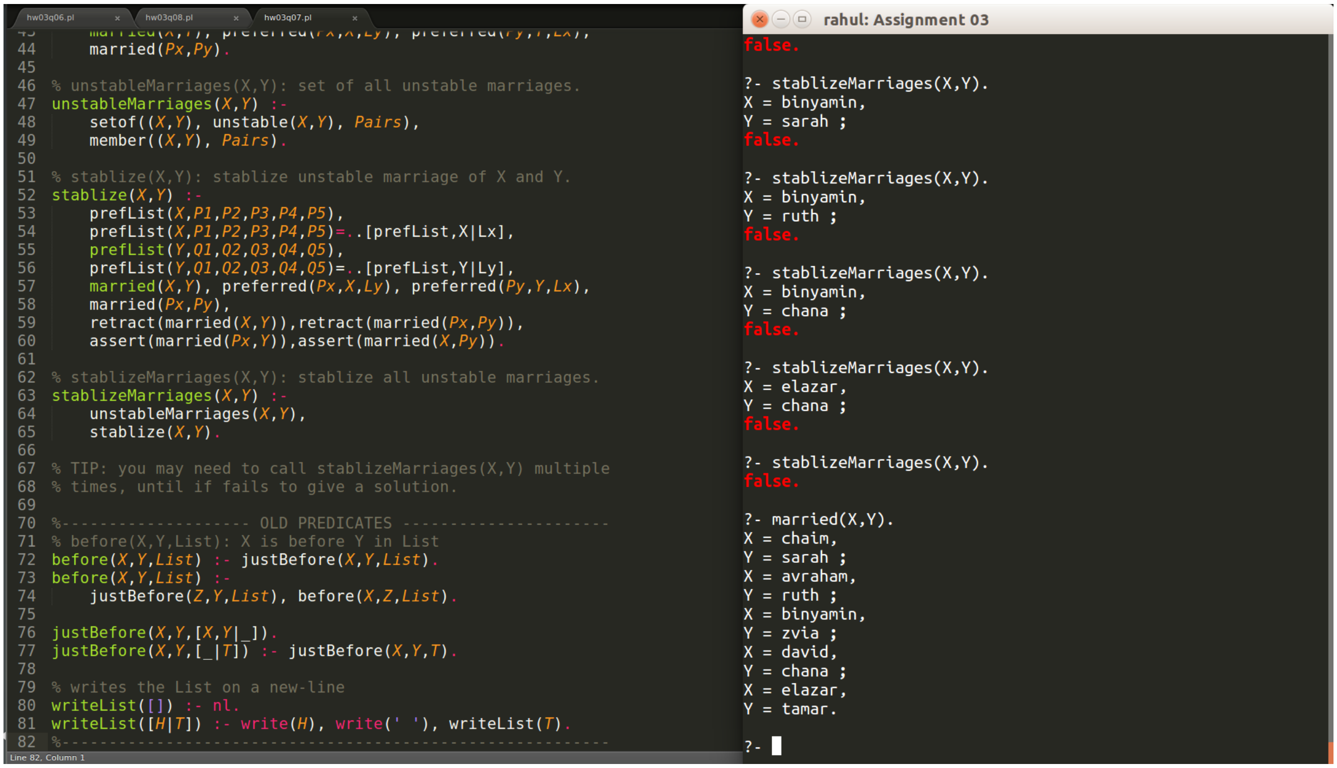
```
male(avraham). male(binyamin). male(chaim). male(david). male(elazar).

female(zvia). female(chana). female(ruth). female(sarah). female(tamar).

%female(X) :- \+ male(X).

% preferences of each person.
prefList(avraham,chana,tamar,zvia,ruth,sarah).
prefList(binyamin,zvia,chana,ruth,sarah,tamar).
prefList(chaim,chana,ruth,tamar,sarah,zvia).
prefList(david,zvia,ruth,chana,sarah,tamar).
prefList(elazar,tamar,ruth,chana,zvia,sarah).

prefList(zvia,elazar,avraham,david,binyamin,chaim).
prefList(chana,david,elazar,binyamin,avraham,chaim).
prefList(ruth,avraham,david,binyamin,chaim,elazar).
prefList(sarah,chaim,binyamin,david,avraham,elazar).
prefList(tamar,david,binyamin,chaim,elazar,avraham).

% dynamically marriages are updated.
:- dynamic married/2.
married(avraham,zvia).  married(binyamin,chana).
married(chaim,ruth).    married(david,sarah).
married(elazar,tamar).

% preferred(X,Y,List): in list List, X is preferred
% over Y, where prefList if of Z.
preferred(X,Y,List) :-
    prefList(Z,P1,P2,P3,P4,P5),
    prefList(Z,P1,P2,P3,P4,P5)=..[prefList,Z|List],
    before(X,Y,List).

% unstable(X,Y): marriage of X and Y is unstable.
unstable(X,Y) :-
    prefList(X,P1,P2,P3,P4,P5), prefList(Y,Q1,Q2,Q3,Q4,Q5),
    prefList(X,P1,P2,P3,P4,P5)=..[prefList,X|Lx],
    prefList(Y,Q1,Q2,Q3,Q4,Q5)=..[prefList,Y|Ly],
    married(X,Y), preferred(Px,X,Ly), preferred(Py,Y,Lx),
    married(Px,Py).
```

Line 6, Column 1

rahul: Assignment 03

```
rahul:Assignment 03$ swipl -q hw03q07.pl
?- unstableMarriages(X,Y).
X = avraham,
Y = zvia ;
X = chaim,
Y = ruth ;
X = david,
Y = sarah.

?- stablizeMarriages(X,Y).
X = avraham,
Y = zvia ;
X = chaim,
Y = ruth ;
X = david,
Y = sarah ;
false.

?- stablizeMarriages(X,Y).
X = avraham,
Y = tamar ;
false.

?- stablizeMarriages(X,Y).
X = chaim,
Y = tamar ;
false.

?- stablizeMarriages(X,Y).
X = binyamin,
Y = sarah ;
false.

?- stablizeMarriages(X,Y).
X = binyamin,
Y = ruth ;
false.
```

Contd...



hw03q06.pl hw03q08.pl hw03q07.pl

```
    married(Px,Py).

% unstableMarriages(X,Y): set of all unstable marriages.
unstableMarriages(X,Y) :-
    setof((X,Y), unstable(X,Y), Pairs),
    member((X,Y), Pairs).

% stablize(X,Y): stablize unstable marriage of X and Y.
stablize(X,Y) :-
    prefList(X,P1,P2,P3,P4,P5),
    prefList(X,P1,P2,P3,P4,P5)=..[prefList,X|Lx],
    prefList(Y,Q1,Q2,Q3,Q4,Q5),
    prefList(Y,Q1,Q2,Q3,Q4,Q5)=..[prefList,Y|Ly],
    married(X,Y), preferred(Px,X,Ly), preferred(Py,Y,Lx),
    married(Px,Py),
    retract(married(X,Y)),retract(married(Px,Py)),
    assert(married(Px,Y)),assert(married(X,Py)).

% stablizeMarriages(X,Y): stablize all unstable marriages.
stablizeMarriages(X,Y) :-
    unstableMarriages(X,Y),
    stablize(X,Y).

% TIP: you may need to call stablizeMarriages(X,Y) multiple
% times, until if fails to give a solution.

%------------------- OLD PREDICATES --------------------
% before(X,Y,List): X is before Y in List
before(X,Y,List) :- justBefore(X,Y,List).
before(X,Y,List) :-
    justBefore(Z,Y,List), before(X,Z,List).

justBefore(X,Y,[X,Y|_]).
justBefore(X,Y,[_|T]) :- justBefore(X,Y,T).

% writes the List on a new-line
writeList([]) :- nl.
writeList([H|T]) :- write(H), write(' '), writeList(T).
%-------------------------------------------------------
```

Line 82, Column 1

rahul: Assignment 03

```
false.

?- stablizeMarriages(X,Y).
X = binyamin,
Y = sarah ;
false.

?- stablizeMarriages(X,Y).
X = binyamin,
Y = ruth ;
false.

?- stablizeMarriages(X,Y).
X = binyamin,
Y = chana ;
false.

?- stablizeMarriages(X,Y).
X = elazar,
Y = chana ;
false.

?- stablizeMarriages(X,Y).
false.

?- married(X,Y).
X = chaim,
Y = sarah ;
X = avraham,
Y = ruth ;
X = binyamin,
Y = zvia ;
X = david,
Y = chana ;
X = elazar,
Y = tamar.

?-
```

8. Program the block worlds problem described in the book (program the intelligent behavior using choose action on page 269). Assume that there are 3 locations p, q and r, and five blocks a, b, c, d, e. Generate a plan to go from initial configuration shown below to final configuration shown below.

Initial configuration:

```
b   e
a c d
_____
p q r
```

Final configuration:

```
e   a
d b c
_____
p q r
```

Ans.

Though, I've already produced a solution (as shown below), it seems not to be optimised. As I can find only 9 transformations (instead of 113) manually, I think the following algorithm may yield considerably less transformations:

Algorithm: IC, FC and CC: Initial, Final and Current Configuration.

1. find **freeBlocks(CC=IC)**, {here, [b,c,e]}
2. Make sure to place blocks closest to the table first, i.e. try placing [d,b,c] first.

   How? And which ones first?

   a. Find **baseBlocks(FC)**, which are not placed at proper base. {here, [d,b,c]}

   b. match base blocks which are free. [d,**b**,c]

   c. A = **findBase(b,FC)**, finds the base/stack of a block from FC;
      B = **findbase(b,CC)**. *If A\=B then d1; else d2.* {here, A=q, B=p}

   d1. Empty the stack 'q' to non 'p' stack; **emptyStack(q,r)**.

   d2. Move 'b' to any non 'p' stack (say 'q'), and now **emptyStack(p,r)**

   e. Place free block 'b' to 'q'.

Repeat 1 to 2e. Until all baseBlocks are placed.
Repeat this process for every next bottom-most layer.

Here, we can enumerate the moves following the above algorithm -

1. Bottom most level:

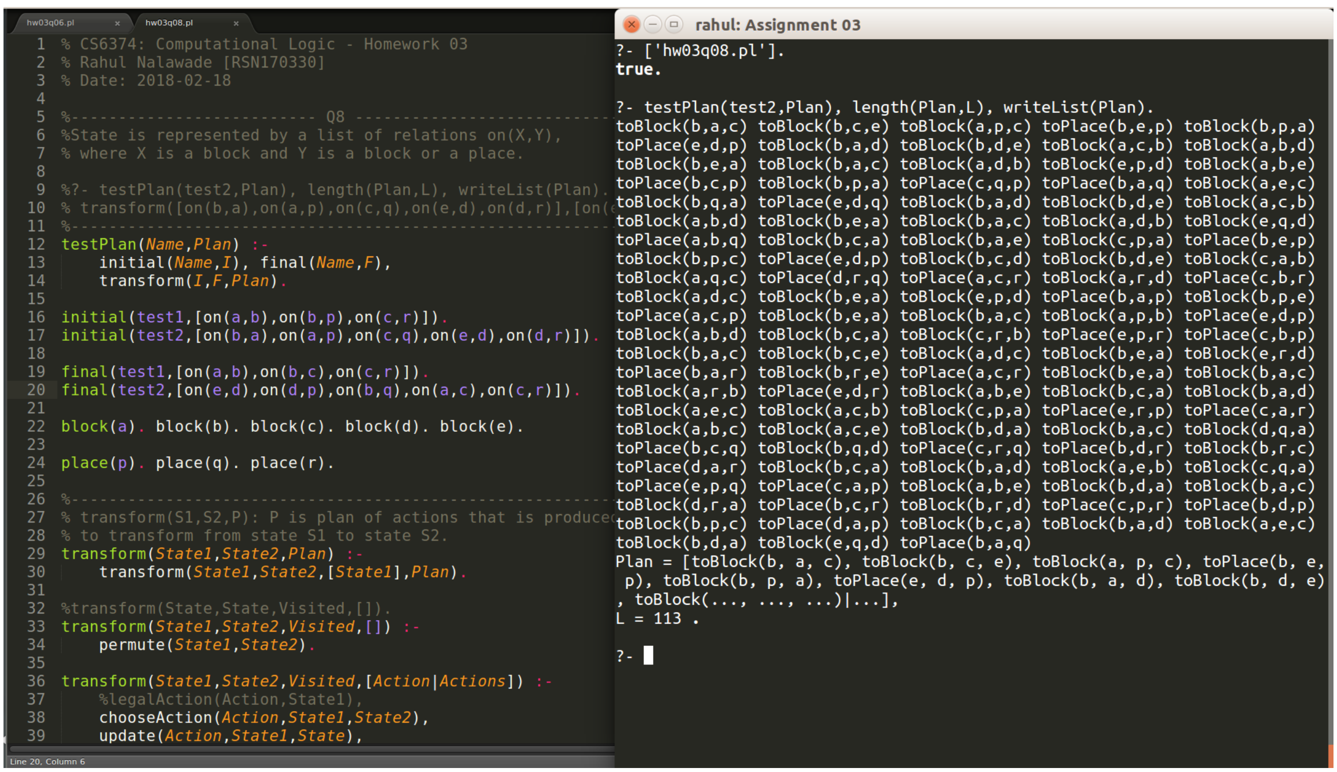
Placing b: toBlock(c,q,e), toPlace(b,p,q)	c e a b d **p q r**
Placing c: toBlock(c,e,a), toBlock(e,d,b), toBlock(d,r,e), toPlace(c,a,r)	d e a b c **p q r**
Placing d: toBlock(a,p,c), toPlace(d,e,p)	e a d b c **p q r**

2. Next level:

Placing e: toBlock(e,b,d)	e a d b c **p q r**

We might need to make it more specific, but placing level by level is the main idea here.

Anyways, here's the output. Might have some loops.



9. Consider the missionary-cannibal problem. Three missionaries and three cannibals come to a river that they want to cross and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten.
Think of this problem as a planning problem and program it in Prolog. You should print a sequence of moves (one per line) that lists the people crossing the river at each step. You can represent the 3 cannibals as c1, c2 and c3, and the three missionaries as m1, m2 and m3. For example, the move:
c1 m1
means that cannibal c1 and missionary m1 went from the bank where the boat is to the other bank. Keep in mind that the boat is always needed to go from one side to another.

Ans.
Comma ',' in the output line as shown below is the river, partitioning left(before crossing) and right(after crossing) sides.