```
Solutions to COSC3401 Report 2
Exercise 1

;; (1)
;; Require: list is any list
;; Ensure: Result = { i,j : N_1 | i <= j <= length(list)
;;                   :: (list[i], list[j])}

(defun pair-combinations-from (list)
 (reduce 'append
  (maplist #'(lambda (l) (distl (car l) l)) list)
))

;; (2)
;; Require: list is any list
;; Ensure: Result = { i,j,k : N_1 | i <= j <= k <= length(list)
;;                   :: (list[i], list[j], list[k])}

(defun triple-combinations-from (list)
 (mapcar #'(lambda (x) (cons (car x) (cadr x)))
        (reduce 'append
         (maplist #'(lambda (l) (distl (car l)
                                       (pair-combinations-from l))) list))
))

------------------------------------------------------------------------------
--
Exercise 2

;; Set *interval-size_plus_1 to the length the "range" plus 1.

(setq *interval-size_plus_1* 5001)

;; Require: count is integer >= 1
;; Ensure: count > 1 --> Result = ((1) ... (count))
;;         count = 0 --> Result = nil

;; To avoid excessive depth of recursion the big interval is split in half
;; until the interval is small enough to use range.

(defun big-range (first last)
  (cond ((< (- last first) interval_size_plus_1) (range first last))
        (t (append (big-range first (+ first (floor (- last first) 2)))
                   (big-range (+ first (1+ (floor (- last first) 2))) last)))
))

------------------------------------------------------------------------------
--
Exercise 3 (partial solution)

The variables in the let expressions become the parameters of the lambda
functions.  The expressions assigned to the variables become
arguments for the lambda functions.

------------------------------------------------------------------------------
--
```

Exercise 4

---------- Variation 1

```
;; Need a recursive support function.

(defmacro mycase-v1 (clist plist)
 (cons 'mycond (combine-lists clist plist))
)

(defun combine-lists (clist plist)
 (cond ((null clist) nil)
       (t (cons (list (car clist) (car plist))
                (combine-lists (cdr clist) (cdr plist))))
))
```

---------- Variation 2

Mapcar is used because we want to step down both lists simultaneously.
The problem is a variation of the inner product program and matrix transpose.

```
;; Cons is required to insert the keyword mycond into the output

(defmacro mycase-v2 (clist plist)
  (cons 'mycond (mapcar 'list clist plist))
)
```

or

```
(defmacro mycase-v2 (clist plist)
  (append '(mycond) (mapcar 'list cl pl)))

;; cannot use list for append, it nests result parts too deep.
```

Exercise 5

Exercise 5.1

```
middle(List, Item) :- append(A,B,List)
                    , oneShorter(A,B)
                    , B = [Item|_].

oneShorter([],[_]).
oneShorter([_|Ra],[_|Rb]) :- oneShorter(Ra,Rb).
```

Exercise 5.2

```
middles([], []).
middles([F|R], Middles) :- F=[_|_]
                         , (   middle(F, M) , Middles = [M|Mr]
                             , middles(R, Mr)
                           ; middles(R, Middles)).
```

```
middles([F|R], Middles) :- F \= [_|_] , middles(R,Middles).

-------------------------------------------------------------------------------
--
Exercise 6

------------
Exercise 6.1

arith_prog([]).
arith_prog([_]).

arith_prog([F, S | R]) :- Diff is S - F , arith_prog([S|R],Diff).

arith_prog([_], _).
arith_prog([F, S | R], Diff) :- Diff is S - F , arith_prog([S|R],Diff).

------------
Exercise 6.2

minMaxMean([], nil).

minMaxMean([F|R], Am) :- minMaxMean(R, F, F, Am).

minMaxMean([],Min,Max,Am) :- Am is (Min+Max)/2.

minMaxMean([F|R], Min, Max, Am) :-
    ( F < Min , minMaxMean(R,F,Max,Am)
    ; Max < F , minMaxMean(R,Min,F,Am)
    ; Min =< F , F =< Max , minMaxMean(R,Min,Max,Am)
    ).

-------------------------------------------------------------------------------
--
Exercise 7

------------
Exercise 7.1


The base case.
Result is an empty list if the list is empty.  We do not care what are the
old and new values, so they are anonymous variables.

    substitute(_ , _ , [], []).

Case 1: Do a substitution --
If the head of the list is the old value and the head of the result is the
new value and substitute is correct for rest of the list and rest of the
result.

    substitute(OldValue, NewValue, [ OldValue | Lt], [ NewValue | Rt ]) :-
        substitute(OldValue, NewValue, Lt, Rt).

Case 2: No substitution means the head of the result is the same as the head
```

of the list --
If the head of the list is not the old value and the head of the list is not a list and substitute is correct for rest of the list and rest of the result.

```
    substitute(OldValue, NewValue, [Lh | Lt], [Lh | Rt]) :-
        Lh \= OldValue , Lh \= [_|_]
      , substitute(OldValue, NewValue, Lt, Rt).
```

Case 3: No substitution and the head of the list is itself a list that requires recursive descent --
If the head of the list is a list, then the head of the result is the substition
in the head of the list and the tail of the result is the substitution in the tail of the list.

```
    substitute(OldValue, NewValue, [Lh | Lt], [Rh|Rt]) :-
        Lh \= OldValue , Lh = [_|_]
      , substitute(OldValue, NewValue, Lh, Rh)
      , substitute(OldValue, NewValue, Lt, Rt).
```

Exercise 7.2

```
swap(_,_,[],[]).

swap(I1,_,[I1],[I1]).


swap(I1, I2, [I1,I2 | Lt], [I2, I1 | Rt]) :- swap(I1, I2, Lt, Rt).

swap(I1, I2, [I1, L2 | Lt], [I1, R2 | Rt]) :-
    L2 \= I2 , swap(I1, I2, [L2|Lt], [R2|Rt]).

swap(I1, I2, [Lh|Lt], [Rh|Rt]) :-
          Lh \= I1 , Lh = [_|_]
        , swap(I1, I2, Lh, Rh)
        , swap(I1, I2, Lt, Rt)

swap(I1, I2, [Lh|Lt], [Lh|Rt]) :-
          Lh \= I1, Lh \= [_|_]
        , swap(I1, I2, Lt, Rt).
```