```
/*
Implement some simple counting functions wrapped in
code to perform type checking and to assure variables
take values only from the appropriate range.
This argument testing decreases code speed (more
instructions get executed per predicate call)
but increases code safety.

Michael E. Sparks, 18 December 2020

SAMPLE USAGE:

1) How many arrangements of 5 distinct blocks on
   5 distinct places are possible?

?- factorial(5,Y).
Y = 120.

2) There are 20 dogs living in a given neighborhood,
   which includes one professional dog walker who
   keeps 5 leashes of differing colors in his inventory.
   He walks 5 dogs in any given dog walking session,
   no more, no less. How many arrangements of
   dogs X leashes are possible for each such session?

?- permutation(20,5,Y).
Y = 1860480.

3) A Hogwarts admissions officer is tasked with
   selecting a group of 50 students from 300
   applicants to matriculate in the coming year,
   ideally such that students will interact with
   one another in an optimal manner. How many
   distinct groupings of students must she
   contemplate? (Note that all Hogwarts employees
   have magical powers, and thus work really fast.)

?- combination(300,50,Y).
Y = 31127323256409201907978577788548204261215346953434685588l6.
*/

/*
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1)))))

;; syntax covering all unary & binary
;; functions considered here
(define-syntax safe-arg?
  (syntax-rules ()
    ((_ n) (and (integer? n) (>= n 0)))
    ((_ n r) (and (integer? n) (integer? r)
                  (>= n 0) (>= r 0) (>= n r)))))

(define (safe-factorial x)
  (if (safe-arg? x)
      (factorial x)
      #f))
```

```
scheme@(guile-user)> (safe-factorial 5)
$2 = 120
scheme@(guile-user)> (safe-factorial -5)
$3 = #f
scheme@(guile-user)> (safe-factorial "five")
$4 = #f
*/
factorial0(0,Y) :-
    Y is 1, !.

factorial0(X,Y) :-
    integer(X),
    X > 0,
    X1 is X - 1,
    factorial0(X1,Y1), % not tail recursive! :(
    Y is X * Y1.

% tail recursion's achieved using an accumulator
factorial(X,Y) :-
    integer(X),
    X >= 0,
    factorial_aux(X,1,Y).

factorial_aux(0,Y,Y) :- !.

factorial_aux(X,Accumulator,Y) :-
    Accumulator1 is X * Accumulator,
    X1 is X - 1,
    factorial_aux(X1,Accumulator1,Y).

/*
(define (permutation n r)
  (/ (factorial n)
     (factorial (- n r))))

(define (safe-permutation n r)
  (if (safe-arg? n r)
      (permutation n r)
      #f))

scheme@(guile-user)> (safe-permutation 20 5)
$5 = 1860480
scheme@(guile-user)> (safe-permutation 3.14 "COVID")
$6 = #f
scheme@(guile-user)> (safe-permutation 5 20)
$7 = #f
*/
permutation(N,R,Y) :-
    integer(N),
    integer(R),
    N >= 0,
    R >= 0,
    R =< N,
    factorial(N,Numer),
    X1 is N - R,
    factorial(X1,Denom),
    Y is Numer / Denom.
```

```
/*
(define (combination n r)
  (/ (permutation n r)
     (factorial r)))

;; here's another mechanism to implement new syntax
;; in Guile Scheme, similar to Common Lisp's defmacro
(define-macro (safe-combination n r)
  `(apply
    (lambda (x y)
      (if (safe-arg? x y)
          (combination x y)
          #f))
    (list ,n ,r)))

scheme@(guile-user)> (safe-combination 300 50)
$8 = 311273232564092019079785777885482042612153469534346855881 6
scheme@(guile-user)> (safe-combination "300" 50)
$9 = #f
*/
combination(N,R,Y) :-
    integer(N),
    integer(R),
    N >= 0,
    R >= 0,
    R =< N,
    permutation(N,R,Numer),
    factorial(R,Denom),
    Y is Numer / Denom.
```