```
//Q 1.2
//f: int -> int -> int
//computes k^n

//g: ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list
//takes a list and filters out any elements where p does not hold. The functions f is then applied to all the elements.

//h: T -> string
//takes input of type T, and converts it to a string.

//Q 1.1

let rec f n = function
    | 0 -> 1
    | k when k>0 -> n * (f n (k-1))
    | _ -> failwith "illegal argument";;

let rec g p f = function
    | [] -> []
    | x::xs when p x -> f x :: g p f xs
    | _::xs -> g p f xs;;

type T =
    | A of int
    | B of string
    | C of T*T;;

let rec h = function
    | A n -> string n
    | B s -> s
    | C(t1,t2) -> h t1 + h t2;;


f 10 3
let p1 n = n > 5
g p1 (f 2) [3..10]

let a1 = A 1
let b1 = B " one"
let c1 = C (a1, b1)

h c1

//Q 1.3
//1 tail recursive

let rec fA n a = function
    | 0 -> a
    | k when k>0 -> (fA n (n*a) (k-1))
    | _ -> failwith "illegal argument";;

```

```
fA 10 1 3

//2 continuation-based
let rec fC n c = function
    | 0 -> c 1
    | k when k>0 -> (fC n (fun v -> c(v*n)) (k-1))
    | _ -> failwith "illegal argument";;

fC 10 id 3

//Q 1.4
let sq = Seq.initInfinite (fun i -> 3*i);;

//sq type: seq<int> infinite sequence
//outputs the multiplication table of 3: 0, 3, 6, 9... etc.

let k j = seq {for i in sq do
                   yield (i,i-j) };;
//k type: seq<int*int>
//outputs an infinite sequence of tuples (i, i-j), where j is a constant from input, and i is 0,3,6,9...

k 9

//Q 1.5
let xs = Seq.toList (Seq.take 4 sq);;
//xs: 4 first elements of 3-table -> [0, 3, 6 and 9]
let ys = Seq.toList (Seq.take 4 (k 2));;
//ys: 4th element of (i,i-2) -> [(0,-2), (3,1), (6,4) and (9,7)]


//Q 2.1
//let ordered l = List.forall (fun x -> x = 0) l;;
let rec ordered = function
    | x::(y::ys) -> (x <= y) && ordered (y::ys)
    | _ -> true;;

ordered [1..10];;
ordered [1;3;4;1;2;9]

//Q 2.2
let smallerThanAll x xs = List.forall (fun y -> x < y) xs;;
smallerThanAll 0 [1..10]

//Q 2.3
let rec insertBefore p x = function
    | [] -> []
    | y::ys when (p y) -> x::(y::ys)
    | y::ys -> y::(insertBefore p x ys);;

let gt3 n = n > 3;;

```

```
insertBefore gt3 6 [1..10]

//Q 2.4
type Sex = | M // male
           | F // female

let sexToString = function
    | M -> "Male"
    | F -> "Female"
    | _ -> failwith "There are only 2 genders"

sexToString M
sexToString F

//Q 2.5
let rec replicate s = function
    | 0 -> ""
    | n when n > 0-> s + (replicate s (n-1))
    | _ -> failwith "n must be positive";;

replicate "abc" 1

//Q 3.1
type Name = string;;
type YearOfBirth = int;;
type FamilyTree = P of Name * Sex * YearOfBirth * Children
and Children = FamilyTree list;;

let marychildren = [(P("Peter", M, 2005,[]));
                    (P("Bob", M, 2008,[]));
                    (P("Eve", F, 2010,[]))]

let joechildren = [(P("Stanley", M, 1975,[]));
                    (P("Mary", F, 1980, marychildren));
                    (P("Jane", F, 1985,[]))]

let maychildren = [(P("Fred", M, 1970,[]));
                    (P("Joan", F, 1975,[]))]

let larrychildren = [(P("May", F, 1945,maychildren));
                    (P("Joe", M, 1950, joechildren));
                    (P("Paul", M, 1955,[]))]

let famtree = P("Larry", M, 1920, larrychildren)



let badmayc = [(P("Fred", M, 1980,[]));
                    (P("Joan", F, 1960,[]))]

let badchildren = [(P("May", F, 1922,[]));
                    (P("Joe", M, 1921, badmayc));
                    (P("Paul", M, 1921,[]))]

let badtree =  P("Larry", M, 1920, badchildren)

let rec orderOK last = function
    | P(_,_,y,P(_,_,y',c::cs)::cs') -> (last <= y) && (orderOK y c)
    | _ -> true;;

let rec OTC = function
    | [] -> true
    | P(n,s,y,[])::rest -> (OTC rest)
    | P(n,s,y,c::cs)::rest -> (List.forall (fun (P(_,_,y',c')) -> y < y') (c::cs)) && (OTC cs) && (OTC rest);;

let isWF = function
    | P(n,s,y,[]) -> true
    | P(n,s,y,c) -> OTC c;;

isWF badtree
isWF famtree

//Q 3.2
let makePerson (n,s,y) = P(n,s,y,[])
makePerson ("William",M,1955)

//Q 3.3

let check (nn,ns,ny,ncs) = function
    | [] -> true;
    | P(n',s,y,_)::cs  -> (ny <= y);;

let rec insertChildOf n (nn,ns,ny,ncs) tree =
    match tree with
    | P(n',s,y,cs) when (n=n') && (isWF tree) && (y < ny) -> insertChildOfInList n c cs
    | _ -> None
and insertChildOfInList n c = function
    | cs when (check c cs) -> (c::cs)
    | cs -> cs
    | _ -> None;;

let ytostring y = y.ToString;;

let rec toString n = function
    | P(n,s,y,c) -> n + (sexToString s) + (ytostring y) + rightString
    | _ -> ""
and rightString n = function
    | c::cs -> toString n c;;

```