If $f : X \to Y$, $A \subseteq Y$, and $f$ is onto, then $A \subseteq F(F^{-1}(A))$. 
If $f : A \rightarrow B$ and $g : B \rightarrow C$ are both one-to-one, then $g \circ f : A \rightarrow C$ is one-to-one.
Consider the following excerpt from *Have I Ever Told You How Lucky You Are?* by Dr. Seuss.

Out west, near Hawtch-Hawtch, there’s a Hawtch-Hawtcher Bee-Watcher. His job is to watch... is to keep both his eyes on the lazy town bee. A bee that is watched will work harder, you see. Well... he watched and he watched. But, in spite of his watch, that bee didn’t work any harder. Not mawtch. So then somebody said, “Our old bee-watching man just isn’t bee-watching as hard as he can. He ought to be watched by another Hawtch-Hawtcher.”

The thing that we need is a Bee-Watcher-Watcher.

WELL... The Bee-Watcher-Watcher watched the Bee-Watcher. He didn’t watch well. So another Hawtch-Hawtcher had to come in as a Watch-Watcher-Watcher. And today all the Hawtchers who live in Hawtch-Hawtch are watching on Watch-Watcher-Watchering-Watch, Watch-Watching the Watcher who’s watching that bee.

You’re not a Hawtch-Watcher. You’re lucky, you see.
The set of citizens of Hawtch-Hawtch, together with their bee, can be defined as the set containing the bee and all those who are employed to watch someone else in the set. We can model this using the following ML datatype:

```ml
datatype hawtchHawtcher = Bee | WatcherOf of hawtchHawtcher;
```

For example, the person watching the town bee is `WatcherOf(Bee)`, and the person watching the watcher of the bee-watcher is `WatcherOf(WatcherOf(WatcherOf(Bee)))`.

a. Write a function `numWatchers` that takes a `hawtchHawtcher` and returns the number of watchers in the chain. For example, `numWatchers(WatcherOf(WatcherOf(WatcherOf(WatcherOf(Bee)))))` would return 4.
b. Write a function beeProductivity that takes a hawtchHawtcher and two reals indicating the pollenation rate of the bee if it were unwatched and the factor by which the bee’s pollentation improves for each watcher in the chain watching it. For example, beeProductivity(WatcherOf(WatcherOf(Bee)), 0.3, 1.25) returns 0.46875 which is 1.25 * 1.25 * 0.3.

c. Write a function doubleWatchers that takes a hawtchHawtcher and returns a hawtchHawtcher like the given one except with twice as many watchers. For example, doubleWatchers(WatcherOf(Bee)) would return WatcherOf(WatcherOf(WatcherOf(Bee))).
Prove that $I(n)$ is a loop invariant for bbb. (14 points.)

$$I(n) = \text{after } n \text{ iterations, } x = 50 + i$$

fun bbb(m) =
  let
    val x = ref 50;
    val y = ref 50;
    val i = ref 0;
  in
    (while !i < m do
      (x := !x + 1;
       y := !y - 1;
       i := !i + 1);
      !x + !y)
  end;
Write a function `findExtreme` that takes a function (with type `int × int → bool`) and a list of integers and uses the function to select the extreme element (least, greatest, etc) of the list. Specifically, the function that `findExtreme` takes as a parameter defines a way to order `int`, that is, it compares two ints (say `a` and `b`) and returns true if `a` comes before `b` and false otherwise (mathematically, this function is a total order). Thus `findExtreme` is a generalization of `findGreatest`. For example, `findExtreme(fn (a, b) => a > b, [6, 4, 18, 9, 2])` would return 18. (This problem is not naturally solved using `map` or `filter`.)