==========
Evaluation order (more in Chapters 6 and 9)

Applicative order
what you're used to in imperative languages
evaluate all arguments before passing to function
usually faster
Normal order
don't evaluate arg until you need it
sometimes faster -- more overhead to remember ***how*** to compute things, but avoid computing if you never need them
terminates if anything will (Church-Rosser theorem)
in naive form, can require re-evaluating something multiple times

Lazy evaluation gives the best of both worlds, so long as you stay purely functional: it evaluates only when it has to, but remembers the value so it doesn't compute it more than once.
The implementation is said to use ***memoization*** (as in, it creates a memo)
Can give unexpected answers in the presence of side effects.

A ***strict*** language requires all arguments to be well-defined, so applicative order can be used.

A ***non-strict*** language does not require all arguments to be well-defined; it requires normal-order or lazy evaluation.

Lisp and ML are strict by default. Haskell is non-strict by default (you can ask for eager evaluation when you want it).

So in OCaml the following will result in a `Division_by_zero` exception:

```
let choose c t e =
  if c then t else e in
  choose (3 < 4) 5 (6 / 0);;
```

But in Haskell the following evaluates just fine:

```
choose c t e =
  if c then t else e         -- declaration of choose
choose (3 < 4) 5 (6 / 0)        => 5
```

Note that in both languages `if (3 < 4) then 5 else (6 / 0)` evaluates to 5. That's because `if..then..else` is a ***special form***, built into the language, whose arguments are lazily evaluated.
In a similar vein, the following evaluate without error in both OCaml and Haskell—neither throws an exception in either language:

```
(3 < 4) or ((6 / 0) > 5)        => true
(3 > 4) and ((6 / 0) > 5)       => false
```

We talk more about lazy evaluation in Chapter 6.

In a language like OCaml you can build lazy evaluation for contexts in which it isn't otherwise provided using higher-order functions—in this case, functions embedded in data structures:

```
type 'a stream =
  Cons of 'a * (unit -> 'a stream);;
let hd : 'a stream -> 'a = function Cons(h, _) -> h;;
let tl : 'a stream -> 'a stream =
  function Cons(_, t) -> t ();;
```

We can use this machinery to create an "infinite list" of, say, the squares of the natural numbers:

```
let squares =
   let rec next n = Cons (n*n, fun () -> next (n+1)) in
   next 1;;

let rec nth n s =
   if n = 1 then hd s
   else nth (n-1) (tl s);;

let rec take n s =
   if n = 0 then []
   else (hd s) :: take (n-1) (tl s);;

hd squares;;              => 1
hd (tl squares);;         => 4
hd (tl (tl squares));;    => 9
nth 5 squares;;           => 25
take 5 squares;;          => [1; 4; 9; 16; 25]
```

Unfortunately, if we evaluate "`nth 123 squares`" multiple times, we'll compute the whole list multiple times. Wouldn't it be nice to ***remember*** the computed part of the list, and keep it around?
There's a standard library called `Lazy` that does this for you.

```
   let a = lazy (expensive_function x);;
   let b = if unlikely_condition then Lazy.force a else 0;;
or
   let b = match unlikely_condition, a with
   | true, lazy v -> v
   | _ -> 0;;
```

The second version incorporates a ***lazy pattern match***. It will happen only if `unlikely_condition` evaluates to true.

Lazy works by creating functions that compute the values you want, but only when called. The '`lazy`' keyword creates the function; '`force`' calls it (and remembers the result for future reference).

Here's a re-write of the stream type:

```
type 'a stream =
  Cons of 'a * 'a stream Lazy.t;;
let hd : 'a stream -> 'a = function Cons(h, _) -> h;;
let tl : 'a stream -> 'a stream =
  function Cons(_, t) -> Lazy.force t;;

let squares =
  let rec next n =
    Cons (n*n, lazy (next (n+1))) in next 1;;

(* nth and take as before *)
```

Because of memoization, the list will actually be fleshed out in memory, and kept, as needed. If computing the next value was much more expensive than adding 1 each time, we'd only compute each explored value once.