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 values 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 applicable action 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:

```ocaml```
let b = if 3 > 4 and ((6 / 0) > 5) then 5 else 25
in b
```

If computing the next value was much more expensive than adding 1 each time, we'd only compute each explored value 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.

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

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

```ocaml```
type 'a stream = (* nth and take as before *)

let squares = (* nth and take as before *)

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

or

```ocaml```
let squares = (* nth and take as before *)

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

However, 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. Lazy.force evaluates something multiple times, so long as you stay purely functional—this in context, functions embedded in data structures:

```ocaml```
struct A = { f: int -> int }

let f a = a * a

let a = Lazy.force f 2
```

or

```ocaml```
let a = Lazy.force f 2
```

The second version incorporates a lazy pattern match. It will happen only if 

The lazy keyword forces the function; 'force' calls it (and remembers the result for future reference).

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

```ocaml```
type 'a stream = (Cons of 'a * 'a stream) + (true, lazy v)

let squares = (Cons (1, lazy squares)) + (4, Lazy.force squares)
```

or

```ocaml```
let squares = (Cons (1, lazy squares)) + (4, Lazy.force squares)
```

However, 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. Lazy.force evaluates something multiple times, so long as you stay purely functional—this in context, functions embedded in data structures:

```ocaml```
struct A = { f: int -> int };

let f a = a * a

let a = Lazy.force f 2
```

or

```ocaml```
let a = Lazy.force f 2
```

The second version incorporates a lazy pattern match. It will happen only if 

The lazy keyword forces the function; 'force' calls it (and remembers the result for future reference).

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

```ocaml```
type 'a stream = (Cons of 'a * 'a stream) + (true, lazy v)

let squares = (Cons (1, lazy squares)) + (4, Lazy.force squares)
```

or

```ocaml```
let squares = (Cons (1, lazy squares)) + (4, Lazy.force squares)
```

However, 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. Lazy.force evaluates something multiple times, so long as you stay purely functional—this in context, functions embedded in data structures:

```ocaml```
struct A = { f: int -> int };

let f a = a * a

let a = Lazy.force f 2
```

or

```ocaml```
let a = Lazy.force f 2
```