# Lazy Evaluation

## Idea

data is **not** evaluated unless it becomes **necessary**

- **SML** uses **eager** (non lazy) evaluation, like C or Java
- Some languages, most notably **Haskell**, use only **lazy evaluation**

## Example

```
fun if_then_else (p,x,y) = if p then x else y;
if_then_else(1>2, hd [], hd [1]);
```

won't work in **SML** but would in **Haskell**

Application: **infinite** data structures (e.g., infinite sequences)

## Example (in Haskell)

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

What is the output if assuming lazy evaluation or eager evaluation?

## Example

```
fun f(x:int):bool = (print "evaluate body\n"; true);
fun arg():int = (print "evaluate argument\n"; 3);

f(arg());
```

```
signature SEQ = sig

type 'a seq
val cons: 'a * (unit -> 'a seq) -> 'a seq

val hd : 'a seq -> 'a
val tl : 'a seq -> 'a seq
val take : 'a seq * int -> 'a list
val drop : 'a seq * int -> 'a seq
val append : 'a list * 'a seq -> 'a seq

val map : ('a -> 'b) -> 'a seq -> 'b seq
val filter : ('a -> bool) -> 'a seq -> 'a seq
val find : int -> ('a -> bool) -> 'a seq -> 'a option

val tabulate : (int -> 'a) -> 'a seq
val iter : ('a -> 'a) -> 'a -> 'a seq
val iterList : ('a list -> 'a) -> 'a list -> 'a seq
val repeat : 'a list -> 'a seq

val merge : 'a seq * 'a seq -> 'a seq
val mergeList1 : 'a seq list -> 'a seq
val mergeList2 : 'a list seq -> 'a seq
val mergeSeq : 'a seq seq -> 'a seq

val Naturals : int seq
val upTo : int -> int seq
val Primes : int seq
end
```

```
local
    open Seq
    fun addyes () = cons ("yes", addno)
    and addno () = cons ("no", addyes)
in
val yesno = addyes()

- yesno;
val it = Cons (SOME "yes",fn) : string Seq.seq

- Seq.hd yesno;
val it = "yes" : string

- Seq.take(yesno, 10);
val it = ["yes","no","yes","no","yes","no","yes","no","yes","no"]
  : string list

- Seq.hd ( Seq.tl yesno);
val it = "no" : string
```

```
datatype 'a seq = ::: of 'a option * (unit -> 'a seq)
infix 5 :::
fun cons (x, f) = SOME x ::: f


datatype 'a seq = Cons of 'a option * (unit -> 'a seq)
fun cons (x, f) = Cons(SOME x, f)

fun  hd (Cons(SOME x, _)) = x
   | hd (Cons(NONE, f)) = hd(f())
```

```
(* will not pass term tests; not "lazy" *)
fun tl (Cons(NONE, f)) = tl(f())
    |  tl (Cons(SOME _, f)) = f()

(* won't type check *)
fun tl (Cons(NONE, f)) =  Cons(NONE, tl(f()))
    |  tl (Cons(SOME x, f)) = Cons(NONE, f)

(* now correct type with an anonymous function; evaluation of tl(f()) is delayed *)
fun tl (Cons(NONE, f)) =  Cons(NONE, fn () => tl(f()))
    |  tl (Cons(SOME x, f)) = Cons(NONE, f)

(* use function composition operator o for a more elegant version*)
fun tl (Cons(NONE, f)) =  Cons(NONE, tl o f)
    |  tl (Cons(SOME x, f)) = Cons(NONE, f)
```

```
fun map g (Cons(SOME x, f)) = Cons(SOME (g x), fn () => map g (f()) )
    | map g (Cons(NONE, f)) = Cons(NONE, fn () => map g (f()))
```

```
fun map g (Cons(SOME x, f)) = Cons(SOME (g x),  map g o f)
    | map g (Cons(NONE, f)) = Cons(NONE,  map g o f)
```