```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Avoiding Unnecessary Computations

Avoiding expensive computations

Thunks let you skip expensive computations if they are not needed

Great if you take the true-branch:

```
(define (f th)
   (if (…) 0 (…   (th) …)))
```

But worse if you end up using the thunk more than once:

```
(define (f th)
   (… (if (…) 0 (… (th) …))
       (if (…) 0 (… (th) …))
        …
       (if (…) 0 (… (th) …))))
```

In general, might not know how many times a result is needed

Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

– Not compute it *until needed*

– *Remember the answer* so future uses complete immediately

Called *lazy evaluation*

Languages where most constructs, including function arguments, work this way are *lazy languages*

– Haskell

Racket predefines support for *promises*, but we can make our own

– Thunks and mutable pairs are enough