Review

- Functional Semantics: static environment, dynamic environment, let statements, recursion
- Today:
  - Lazy vs Strict Languages
  - Thunks

Lazy Evaluation

- Recall the lambda calculus:
  - Variables
  - $\lambda x . e$ Function application
  - $\text{fn} \, x \, \Rightarrow \, e$ Anonymous functions
- For function application, call-by-value
  - i.e., when you see $e_1 \, e_2$, then evaluate $e_2$ to a value $v_2$.
  - When $e_1 = \lambda x . e_3$, then $e_1 \, e_2$ evaluates to $e_3[v_2/x]$
- Instead, call-by-name
  - When $e_1 = \lambda x . e_3$, then $e_1 \, e_2$ evaluates to $e_3[x/e_1]$
  - Don't evaluate until you have to!

Haskell and Lazy Evaluation

- Haskell has lazy evaluation
- Consider the code:

```sml
fun orL l = foldr (||) False
fun anyL p = orL . map p
fun five_million = let
  fun f x =
    if x <= 5000000
    then x :: f (x + 1)
    else []
  in
  f 0
  end

> anyL (fn x => p = ll) five_million
```

Infinite Lists

- These are infinite lists!
- Again, through the power of lazy evaluation

A Small Puzzle

Consider the following SML code:

```sml
fun or l = foldr (fn (x, y) => x orelse y) l
fun any p l = or (map p l)
val five_million = let
  fun f x =
    if x <= 5000000
    then x :: f (x + 1)
    else []
  in
  f 0
end

> anyL (fn x => p = ll) five_million
```

Infinite Lists

```sml
val ones = 1 : ones
val primes = sieve (2 ..)
where
  sieve (p : xs) = p : sieve [x | x <- xs, rem x p /= 0]
```

- These are infinite lists!
Implementing Lazy Evaluation

How do we implement lazy evaluation?

Consider this:

```haskell
datatype 'a stream =
    Empty
  | Cons of 'a * (unit -> 'a stream)

exception empty_stream

fun lhead Empty = raise empty_stream
| lhead (Cons (x, xs)) = x

fun ltail Empty = raise empty_stream
| ltail (Cons (x, xs)) = xs ()
```

Five Million Stream

- We can now write five_million as this:

```haskell
val five_million =
let
    fun f x =
        if x <= 5000000
        then Cons (x, (fn () => f (x + 1)))
        else Empty
    in
        f 0
    end
```

Memoization

- How do we keep repeated calculations from being expensive?

```haskell
triple :: Int -> Int
triple = x + x + x

trouble :: Int
trouble = triple expensive_computation
```

- Do we really need to calculate expensive_computation 3 times?
- Not if it's a pure function!
- Just save the result the first time.
- Since it's pure, the result won't change, so there's no issue here.