Reduction Strategies; Lazy Evaluation

February 13th

Last Time

- We:
  - Described the runtime behaviors of a language via operational semantics
  - Identify three styles of writing operational semantics
  - Distinguish between deterministic and nondeterministic evaluation

Agenda

- We will:
  - Give an operational semantics for the lambda calculus
  - Compare different evaluation orders for the lambda calculus
  - Explore and exploit laziness in Haskell

Semantics For λ

- Recall: Lambda calculus is a language in which *everything* is a function
- Syntax:

```
t ::= x          | λx. t          | t t
```

- How to evaluate:

```
(λx. λy. y x) (λx. x) (λx. x)
```

- A reducible expression or redex is an expression which has a function call that can be made

Semantics For λ

- If everything is a function, all you can do is call functions:

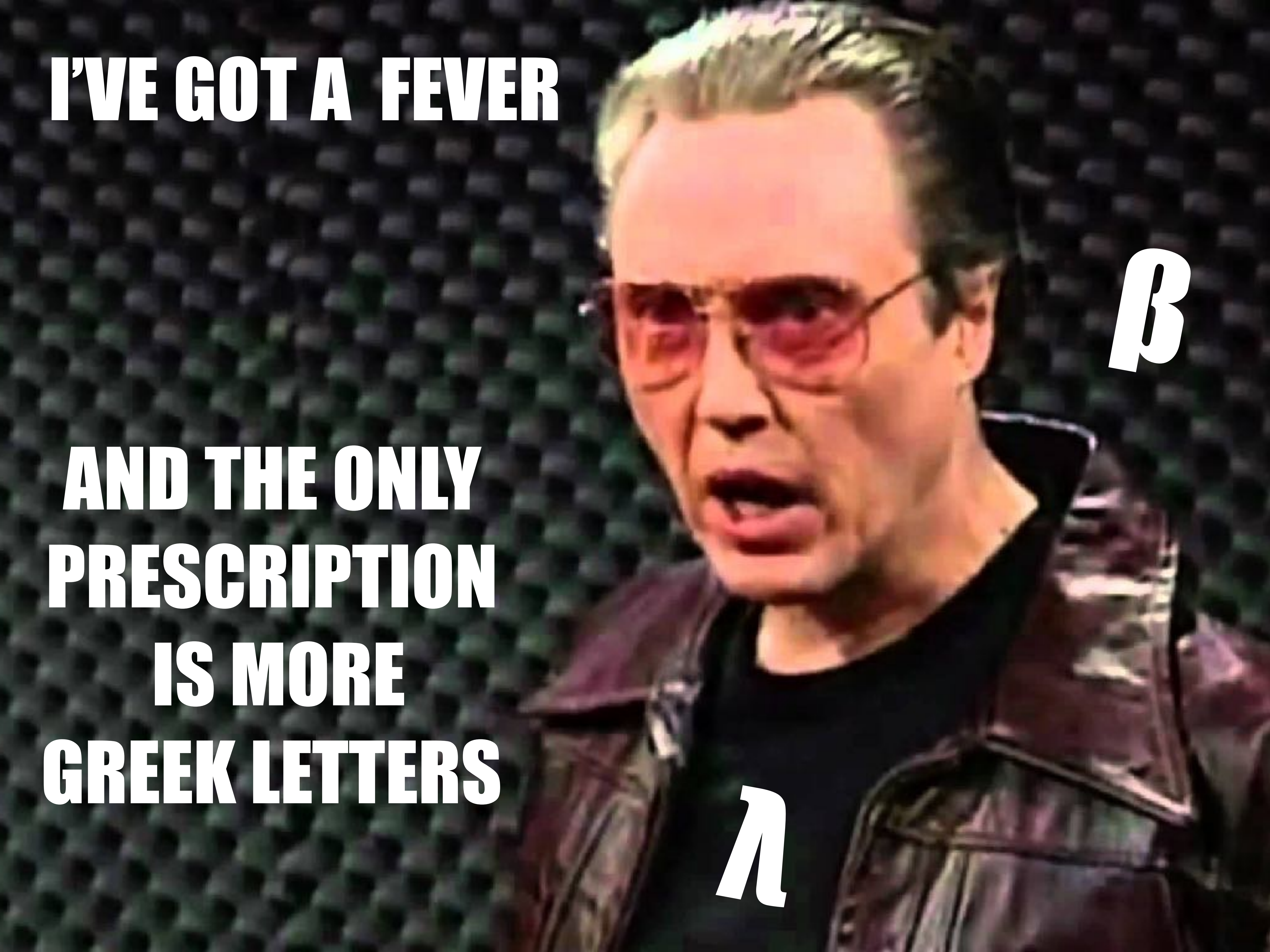
$$\frac{e_1 \longrightarrow e_1'}{e_1\ \ e_2 \longrightarrow e_1'\ \ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ \ e_2 \longrightarrow e_1\ \ e_2'}$$

$$\frac{}{(\lambda x.e_1)\ \ e_2 \longrightarrow e_1\,[x \mapsto e_2]}$$

redex

This step is called a **beta reduction**

(λx. λy. x y) (λx. x) (λx. x) ⟶


I'VE GOT A FEVER
AND THE ONLY
PRESCRIPTION
IS MORE
GREEK LETTERS
β
λ

More λ Semantics

- Alpha reduction:

$$\frac{\texttt{y} \text{ does not appear in } \texttt{e}}{\lambda\texttt{x.e} \longrightarrow \lambda\texttt{y.e}[\texttt{x} \mapsto \texttt{y}]}$$

  - In other words, variable names don't matter
- Eta reduction:

$$\frac{\texttt{x} \text{ does not appear in } \texttt{e}}{(\lambda\texttt{x.e}\ \ \texttt{x}) \longrightarrow \texttt{e}}$$

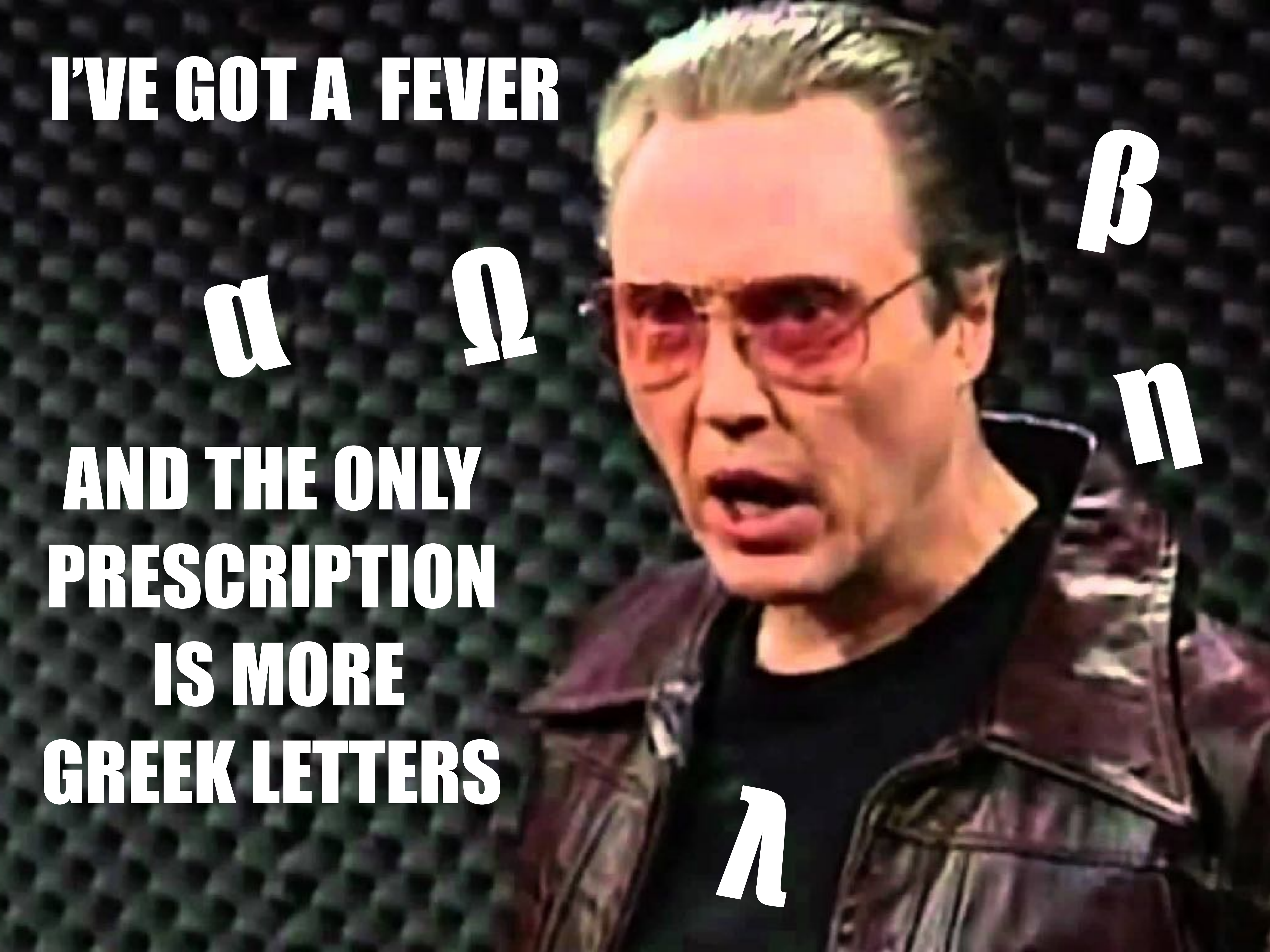
  - Can lift this to a notion of eta-expansion, lets you delay diverging computations

Ω

- Is the lambda calculus strongly normalizing under beta reduction?
  - Does every expression evaluate to a unique normal form?
- Nope:

(λx. x x) (λx. x x)

- This is a diverging computation, i.w.one that does not terminate
- We'll call this Ω


I'VE GOT A FEVER
β
α Ω
η
AND THE ONLY PRESCRIPTION IS MORE GREEK LETTERS
λ

Evaluation Strategy

- These rules are nondeterministic:

```
(λx.x) ((λx.x) ((λz.(λx.x) z))
```

- This semantics is also called full beta reduction
- Any sensible implementation needs to fix an order, i.e. choose an evaluation strategy

Call By Value

- Recall that the results of evaluation are a language's values:

  - The lambda calculus' values are the closed λ expressions:

$$\lambda x . t$$

- In the Call-By-Value semantics, only the top-most function that is applied to a value is reduced:

$$\frac{e_1 \longrightarrow e_1'}{e_1 \; e_2 \longrightarrow e_1' \; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v \; e_2 \longrightarrow v \; e_2'}$$

$$\frac{}{(\lambda x . e) \; v \longrightarrow e_1 [x \mapsto v]}$$

CBV Example

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

$$\frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}$$

$$\frac{}{(\lambda x.e)\ v \longrightarrow e_1[x \mapsto v]}$$

(λx.x) ((λx.x) ((λz.(λx.x) z))

Call By Name

- The alternative: beta-reductions are performed immediately once the left-hand side is a value:

$$\frac{}{(\lambda x.e_1)\ \ e_2 \longrightarrow e_1\,[x \mapsto e_2]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ \ e_2 \longrightarrow e_1'\ \ e_2}$$

`(λx.x) ((λx.x) ((λz.(λx.x) z))`

- Can we distinguish between the two evaluation strategies?

Y'alls Turn

- Evaluate this expression using both strategies:

$$(\lambda x.\lambda y.y\ x)\ (5 + 2)\ (\lambda x.x + 1)$$

- Can you come up with a term that loops forever, and keeps getting bigger?

Other Strategies

- Note that these rules don't reduce inside lambda expressions
- Two do:
  - Normal-order: the leftmost, outermost redex is always reduced first:

```
(λx.x) ((λx.x) ((λz.(λx.x) z))
```

  - In applicative order, arguments are reduced first:

```
(λx.x) ((λx.x) ((λz.(λx.x) z))
```

Call-By-Need

- There are pros and cons for each of these strategies:
  - Call-by-value: can perform extraneous computations:

    `(λx λy.x) ((λx.x) ((λz.(λx.x) z)) (λx.x)`

  - Call-by-name: duplicates work (and terms can grow big)

    `(λx.x x) ((λx.x) ((λz.(λx.x) z))`

Call-By-Need

- For efficient call-by-name, we need to avoid duplicate work:

```
(λx.x x) ((λx.x) ((λz.(λx.x) z))
```

- Idea: track of occurrences coming from the same argument
  - When forced to reduce one, replace the others
  - This requires sharing in the run-time representation
- This call-by-need strategy is used in Haskell

Need=Pattern Matching

- When is reduction needed in Haskell?
- Pattern Matching!

```
[1..3] ++ ([4..6] ++ [7..10])
```

- Consequence of this is we can define infinite structures:

```
stream = 1 : stream
head stream = 1
```

Demo Time

Strict Arguments

- Consider: `foldl (+) 0 [1,2,3]`

- `fold`$_L$ is always going to evaluate (+)
- Solution is declare the accumulator to be strict: forces call-by-value semantics

Recap

- We will:
  - Give an operational semantics for the lambda calculus
  - Compare different evaluation orders for the lambda calculus
  - Explore and exploit laziness in Haskell