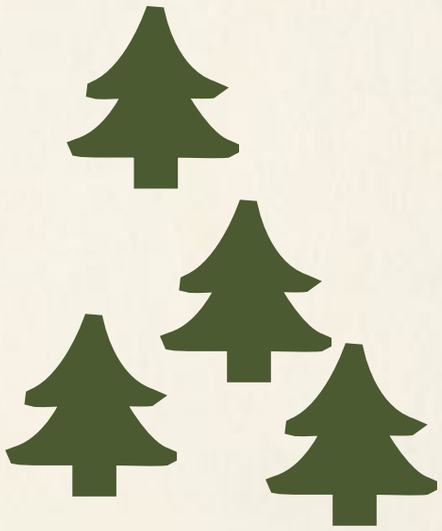
Imperative Langauges

February 15th

Last Time

- We:
  - Gave an operational semantics for the lambda calculus
  - Compared different evaluation orders for the lambda calculus
  - Explored laziness in Haskell

Agenda



- We will:
  - Look at how big-step semantics guide the implementation of an interpreter
  - Look at how to model state in operational semantics
  - Show how to use monads to implement effects in pure languages

Imp Semantics

★ Recall the syntax for IMP, our core imperative language:

```
a  ::= 𝕍 | 𝕀 | a * a | a + a
b  ::= true | false | not b | b && b
       | a == a | a < a
S  ::= 𝕍 := a | skip | S1; S2 |
       | if b then S1 else S2 | while b do S
```

★ Semantics for (basic) arithmetic expressions is straightforward:

$$\frac{}{n \Downarrow n}\ \text{E-Const}$$

$$\frac{n \Downarrow v_n \quad m \Downarrow v_m}{n \star m \Downarrow n \star m}\ \text{E-Times} \qquad \frac{n \Downarrow v_n \quad m \Downarrow v_m}{n + m \Downarrow n + m}\ \text{E-Plus}$$

Haskell Implementation

- Has a ‘natural’ implementation in Haskell:

```
data AExp = Const Int | Plus AExp AExp | Times AExp AExp
```

```
eval (Const i) = i
eval (Plus a1 a2) = (eval a1) + (eval a2)
eval (Times a1 a2) = (eval a1) * (eval a2)
```

- How could we add boolean expressions?

```
b ::= true | false | not b | b && b | a == a | a < a
```

Handling State

★ Core issue is that Variable require state
  - This is what makes IMP an interesting language

- Intuitively, how do we evaluate the following expression?

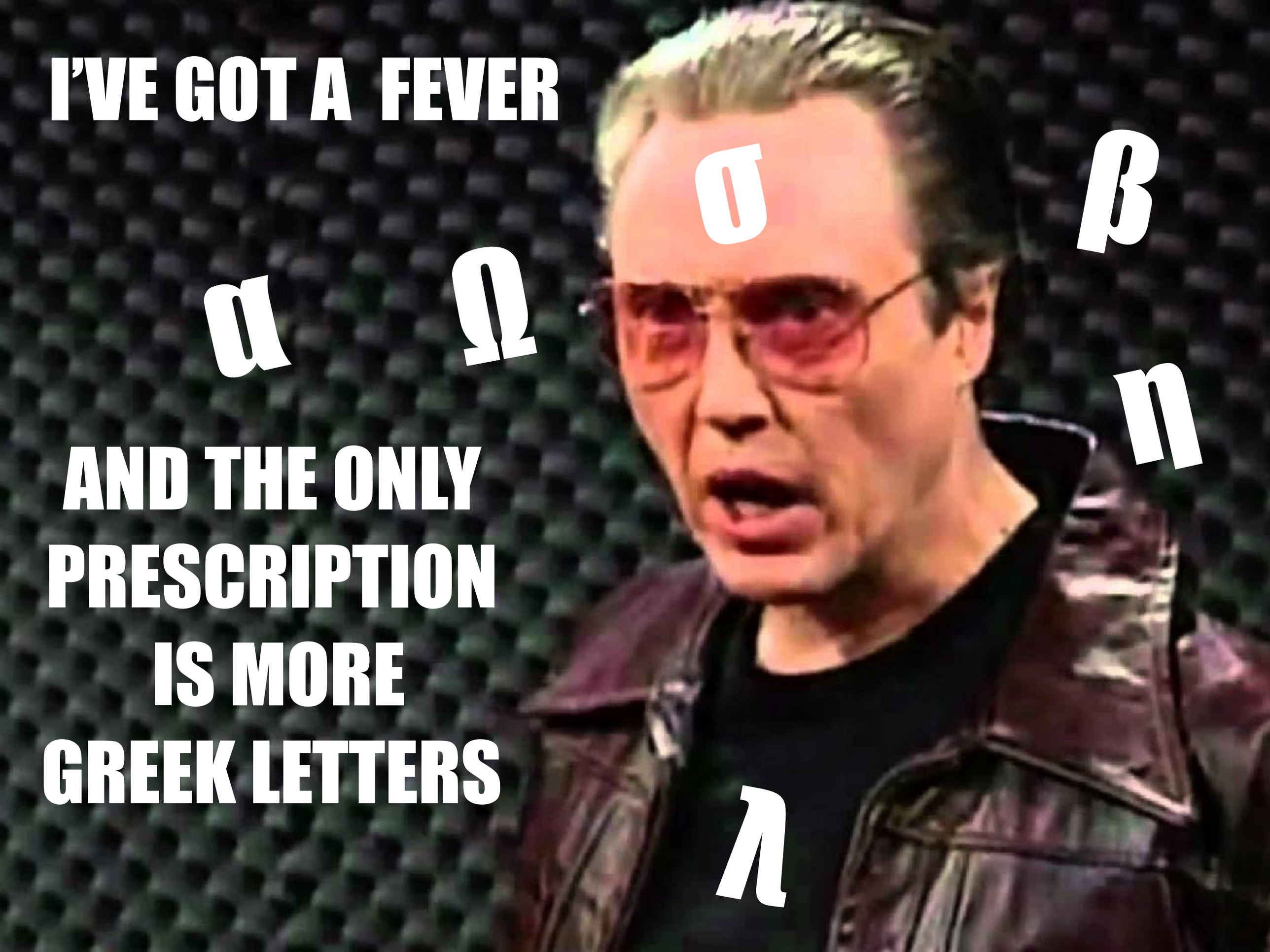
$$\frac{}{x + 3 \Downarrow}$$

- How do we represent global state in an evaluation rule?

$$\frac{}{x \Downarrow ?}\ \text{E-Var}$$

I'VE GOT A FEVER

AND THE ONLY
PRESCRIPTION
IS MORE
GREEK LETTERS



Adding State

★ Idea: evaluation becomes a three-way relation:

$$\frac{\sigma(x) = n}{\sigma, x \Downarrow n}\ \text{E-Var}$$

starting state ($\sigma :: \mathbb{V} \to \mathbb{I}$)

redex

valuation

★ Adding state to the relation lets us use it in the premise

★ We have to update other rules as well:

$$\frac{\sigma, x \Downarrow v_x \qquad \sigma, y \Downarrow v_y}{\sigma, x+y \Downarrow v_x + v_y}\ \text{E-Var}$$

Haskell Implementation

- What about statements?

```
S  ::= 𝕍 := a | skip | S1; S2 | if b then S1 else S2 |
while b do S
```

- In C or Java, we might just have a global variable for the mapping and we would update it appropriately.
- Haskell is a pure language— there are no mutable variables!
- What's the solution?

Haskell Implementation

- In C or Java, we might just have a global variable for the mapping and we would update it appropriately.
- Haskell is a pure language— there are no mutable variables!
- What's the solution?

- The key bit was threading the state parameter through in a way that was consistent with the semantics
- Passing around all these variables was a little cumbersome.

Monads

- This idiom is so common that Haskell has a standard method of approaching it:

Monads

- What is a monad:
  - “A monad is just a monoid in the category of endofunctors”

    —Phil Wadler
  - A monad is a means for wrapping values in a context and passing that context around.

Monads

★ Monad typeclass (basic) definition:

Type of Containers or Contexts

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

put a value into the container

sequence two contexts together

★ The Option Monad:

```
instance Monad Option where
  return a = Just a
  Nothing >>= f = Nothing
  Just a >>= f = f x
```

Monads

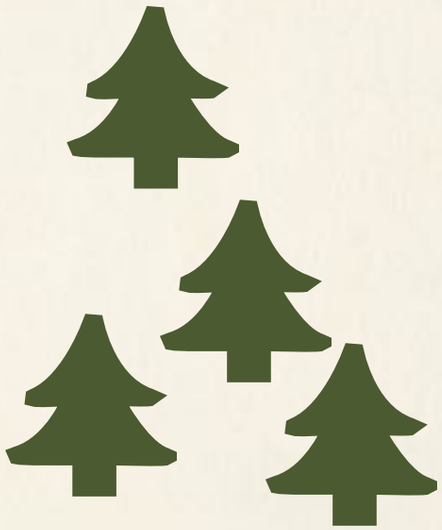
★ The State Monad:

```
data State s a =  State (s -> (a, s))
instance Monad (State s) where
      return x = State $ \s -> (x,s)
      (State h) >>= f = State $ \s -> let (a, newState) = h s
                                          (State g) = f a
                                      in g newState
```

★ The Monad laws:

- left identity: `return x >>= f ≅ f x`
- right identity: `m >>= return ≅ m`
- associativity: `(m>>=f) >>= g ≅ m >>= (\x ->f x >>= g)`

Recap



- We will:
  - Look at how big-step semantics guide the implementation of an interpreter
  - Look at how to model state in operational semantics
  - Show how to use monads to implement effects in pure languages