How do you make a monad outside of Haskell?

- Two core functions
  - bind
  - return
- Implementations of effectful operations
  - failure for an error monad
  - get and put for a state monad
  - callcc for a continuation monad
  - whatever else you want
- The rest is mainly syntax, higher-order functions and polymorphism

Monads and Lisp fit well

- Monadic syntax can be first-class (macros)
  - Better than Haskell – where do-notation is not first-class
- Lisp has higher-order functions and closures
  - Required for bind, monadic map and many other monad utility functions
- Dynamic types provide polymorphism
  - Many monadic functions are also polymorphic (e.g. bind, return)
  - Tradeoff: mixing effects can be trickier

Example: Environment Monad

- Effect: computations that implicitly thread some read-only information

```
(define (return v) (lambda (env) v)
(define (bind f mv)
  (lambda (env) (f (mv env) env)))
```

- In this bind function, the same environment is used twice
  - Compare this to bind for a state monad

Environment Monad Operations

```
; access the local environment
(define capture-env
   (lambda (env) env)

; change the environment for a
; sub-computation
(define (local-env f mv)
   (lambda (env)
      (mv (f env))))
```

Exercise: List Monad

- Represents ambiguous computations with varying numbers of results
- `(return v)` has exactly one result
- `m-zero` means no results
- `(m-plus a b c ...)` joins together the possible results of a and b
- as always, `bind` handles the sequencing

- How would you write these functions?

Exercise: List Monad (continued)

- Sample Scheme code is available at: monad-tutorial/exercises/SchemeMonads/List.ss
- Exercise goals:
  - Building a monad implementation in a familiar language
  - Understanding direct monadic programming with bind and return

Exercise: Monadic Syntax

- Direct programming with bind and return isn't terribly convenient
- Haskell has do-notation to deal with this
- Lisp macros can be used to make more convenient syntax
- Example: `letM` and `letM*` macros
- Exercise instructions at: monad-tutorial/exercises/SchemeMonads/Monad.ss
- Goal: Understanding how to build useful monadic syntax in Lisp

Putting this all together: A monadic evaluator fragment

```
(define (analyze-function name body)
  (let ((body-code (analyze body)))
  (letM ((env capture-env))
  (return
    (lambda (val)
      ((with-binding name val body)
        env))))))
```

- monad-tutorial/scheme/Env.ss and EnvInterp.ss has the full code

Exercise: Monadic String Parsing

- Uses List monad from first exercise
- Parse a string into a word or a number (decimal or hexadecimal)
- Scheme starter code in: monad-tutorial/exercises/SchemeMonads/ParseString.ss
- Goal: Write an interesting monadic program using monadic syntax