BEYOND APPLICATIVE
WARMUP: TWO OF THE SAME CHAR

- Write function: take char, parse two of the char

```haskell
import Control.Applicative (Parser, <$>)

twinCharP :: Char -> Parser (Char, Char)
twinCharP c = pair <$> charP c <*> charP c
  where pair x y = (x, y)
    -- shorter: (,) <$> charP c <*> charP c
```
MAKE THIS WORK FOR ANY CHAR?

- We want a parser of this type:

```
sameTwoP :: Parser (Char, Char)
```

- A first try: brute force the cases:

```
sameTwoP =  twinCharP 'a' <|> twinCharP 'b' <|> twinCharP 'c'
-- ...
```

- This is going to be quite painful...
WANTED: NEW WAY TO COMBINE PARSERS

• Here’s what we have so far:

    itemP :: Parser Char  -- parse one of any Char

    oneMore :: Char -> Parser (Char, Char)
    oneMore c = addC <$> charP c
    where addC = \x -> (c, x)

    sameTwoP = combiner itemP oneMore

• To combine, want something of the type:

    combiner :: Parser Char
             -> (Char -> Parser (Char, Char))
             -> Parser (Char, Char)
A COMMON PATTERN FOR SEQUENCING
UNRELIABLE COMPUTATIONS

• Two unreliable computations:

foo :: a -> Maybe b
bar :: b -> Maybe c

• How to string them together?

foobar :: a -> Maybe c
foobar x = case foo x of
  Nothing -> Nothing
  Just y  -> case bar y of
    Nothing -> Nothing
    Just z  -> Just z
WHAT ABOUT ONE MORE?

foo :: a -> Maybe b
bar :: b -> Maybe c
baz :: c -> Maybe d

foobazbar :: a -> Maybe d
foobazbar x = case foo x of
  Nothing -> Nothing
  Just y  -> case bar y of
    Nothing -> Nothing
    Just z  -> case baz z of
      Nothing -> Nothing
      Just w  -> Just w

• Starting to get a bit unwieldy...
PROGRAMS WITH LOGGING

• Two computations with logging:

```haskell
foo :: a -> (b, String)
bar :: b -> (c, String)

foobar :: a -> (c, String)
foobar x = let (y, log1) = foo x in
            let (z, log2) = bar y in
            (z, log1 ++ log2)
```
WHAT ABOUT ONE MORE?

foo :: a -> (b, String)
bar :: b -> (c, String)
baz :: c -> (d, String)

foobarbaz :: a -> (c, String)
foobarbaz x = let (y, log1) = foo x in 
let (z, log2) = bar y in 
let (w, log3) = bar y in 
(w, log1 ++ log2 ++ log3)

• Starting to get a bit unwieldy...
MAINTAINING A COUNTER

- Two computations modify a counter

---

Input: init counter. Output: updated counter + output string

foo :: Int -> (Int, String)
bar :: Int -> (Int, String)

foobar :: Int -> String
foobar c = let (c', out') = foo c in
let (c'', out'') = bar c' in
out'
WHAT ABOUT ONE MORE?

-- Input: init counter. Output: updated counter + output string
foo :: Int -> (Int, String)
bar :: Int -> (Int, String)
baz :: Int -> (Int, String)

foobarbaz :: Int -> String
foobarbaz c = let (c', out') = foo c in
    let (c'', out'') = bar c'' in
    let (c''', out'''') = bar c''' in
    out'''
WHAT IS THE PATTERN?
TWO OPERATIONS

• Wrapping a normal value into a “monadic” value
  ▪ Package an “output” value with some extra data

• Transforming a monadic value
  1. first monadic value
  2. function from regular value to monadic value
     ▪ Plug pieces together to get another monadic value
THE MONAD TYPECLASS

- These two operations are called *return* and *bind*

```
class Applicative m => Monad m where
  return :: a -> m a        -- Required op. 1: return
  (>>=) :: m a -> (a -> m b) -> m b  -- Required op. 2: bind
  (>>) :: m a -> m b -> m b           -- Special case of bind
```
EXAMPLE: MAYBE

- Maybe \( a \) is either an \( a \), or nothing

```haskell
instance Monad Maybe where
  -- Given normal value, wrap it with Just
  return :: a -> Maybe a
  return val = Just val

  -- Compose two Maybe computations
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  maybe >>= f = case maybe of
    Nothing -> Nothing -- First computation failed
    Just val -> f val -- First computation OK, run second
```
**EXAMPLE: WITHLOG**

- WithLog is an a with a String log
  - Also known as the Writer monad

```haskell
data WithLog a = MkWithLog (a, String)

instance Monad WithLog where
  -- return :: a -> WithLog a
  return val = MkWithLog (val, "")
  -- ^-- Empty log

  -- (>>=) :: WithLog a -> (a -> WithLog b) -> WithLog b
  logA >>= f = let MkWithLog (output, log) = logA in
               let MkWithLog (output', log') = f output in
               MkWithLog (output', log ++ log')
               ^--join--^
```

```haskell
--
```
EXAMPLE: STATE

- State \( s \ a \) is computation that returns an \( a \)

```haskell
data State s a = MkState (s -> (a, s))

instance Monad (State s) where
    -- return :: a -> State s a
    return val = MkState (\state -> (val, state))
    -- ^-- unchanged --^  

    -- (>>=) :: State s a -> (a -> State s b) -> State s b
    (MkState stTrans1) >>= f = MkState $ \st -> 
        let (out', st') = stTrans1 st
            in f out' st' -- ^-Part 1^-  

        let (MkState stTrans2) = (f out') in stTrans2 st'  -- ^-Part 2^-  
```
NICER SYNTAX: DO-NOTATION
foo :: a -> Maybe b
bar :: b -> Maybe c
baz :: c -> Maybe d

foobarbaz :: a -> Maybe d
foobarbaz x = case foo x of
  Nothing -> Nothing
  Just y  -> case bar y of
    Nothing -> Nothing
    Just z  -> case baz z of
      Nothing -> Nothing
      Just w  -> Just w
foo :: a -> Maybe b
bar :: b -> Maybe c
baz :: c -> Maybe d

foobarbaz :: a -> Maybe d
foobarbaz x = foo x >>= (\y ->
    bar y >>= (\z ->
        baz z >>= (\w ->
            return w))))

• Common pattern: monad value, >>=, lambda
APPLYING A BIND

• Turn `monVal >>= (\var -> ...)` into

```haskell
do var <- monVal
...
```

• Variant of bind: the following are equivalent:
  - `monVal >> ...`
  - `monVal >>= (\_ -> ...)`

```haskell
do monVal
...
```
TRANSLATING OUR EXAMPLE

• Before, without do-notation

```haskell
foobarbaz x = foo x >>= (\y ->
    bar y >>= (\z ->
        baz z >>= (\w ->
            return w)))
```

• With do-notation (watch indentation)

```haskell
foobarbaz x = do y <- foo x
               z <- bar y
               w <- baz z
               return w
```
COMPACT DO-NOTATION

- Do-notation uses indentation and linebreaks

```haskell
foobarbaz x = do y <- foo x
              z <- bar y
              w <- baz z
              return w
```

- Can also use braces and semicolons

```haskell
foobarbaz x = do { y <- foo x ;
                   z <- bar y ;
                   w <- baz z ;
                   return w }
```
GENERAL ADVICE

- Do-notation is very clean, but it hides a lot
- Try to start with `>>=` and `return`
  - Unfold definition of these operations
  - Easier to see what’s going on (just functions)
  - Easier to see that types are correct
- WR3: practice do-notation
MORE MONADS
**Either**

- **Idea:** `OrErr a` is either an `a`, or an error `String`.

```haskell
data Either a b = Left a | Right b

type OrErr a = Either String a  -- give type a new name

actualInt :: OrErr Int
actualInt = Right 5000  -- Actual number 5000

ererrorInt :: OrErr Int
errorInt = Left "Couldn't think of a number"  -- Error string
```
MONAD INSTANCE?

• As always, follow the types...

```haskell
instance Monad OrErr where
    -- return :: a -> OrErr a
    return x = Right x

    -- (>>=) :: OrErr a -> (a -> OrErr b) -> (OrErr b)
    (Left err)  >>= f = Left err
    (Right val) >>= f = f val
```
LISTS

• To give monad instance, need two functions:
  - **Concat**: take lists of lists, flatten into single list
    - Takes \([1, 2], [3]\) to \([1, 2, 3]\)
  - **Map**: apply function to each element of input list
    - Map “times 2”: takes \([1, 2, 3]\) to \([2, 4, 6]\)
MONAD INSTANCE?

• As always, follow the types...

```haskell
instance Monad [] where
  -- return :: a -> [a]
  return x = [x]  -- list with just one element

  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  listA >>= f = concat $ map f listA  -- map, then concat
```
REVISITING OUR PARSER
PARSER IS A MONAD

- Looks suspiciously like the State monad!
- Actually: State + Maybe monad
- Type of state is always String: stuff to parse
- Type `a` is the “return type”: result of parse

```
data Parser a = MkParser (String -> Maybe (a, String))
data State s a = MkState (s -> (a, s))
```
RETURN FOR PARSERS

- Return: yield output value without any parsing
- Follow the types...

```haskell
-- return :: a -> Parser a
return val = MkParser $ \str -> Just (val, str)
```
Bind: sequence parser, where second parser can depend on first output

Follow the types...

```haskell
-- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
par >>= f = MkParser $ \str ->

  let firstRes = runParser par str
  in case firstRes of
    Nothing    -> Nothing
    Just (val, str') -> let par' = f val
                         in runParser par' str'

    -- Run parser 1
    -- See what we got...
    -- Choose parser 2
    -- Run parser 2 on rest
```