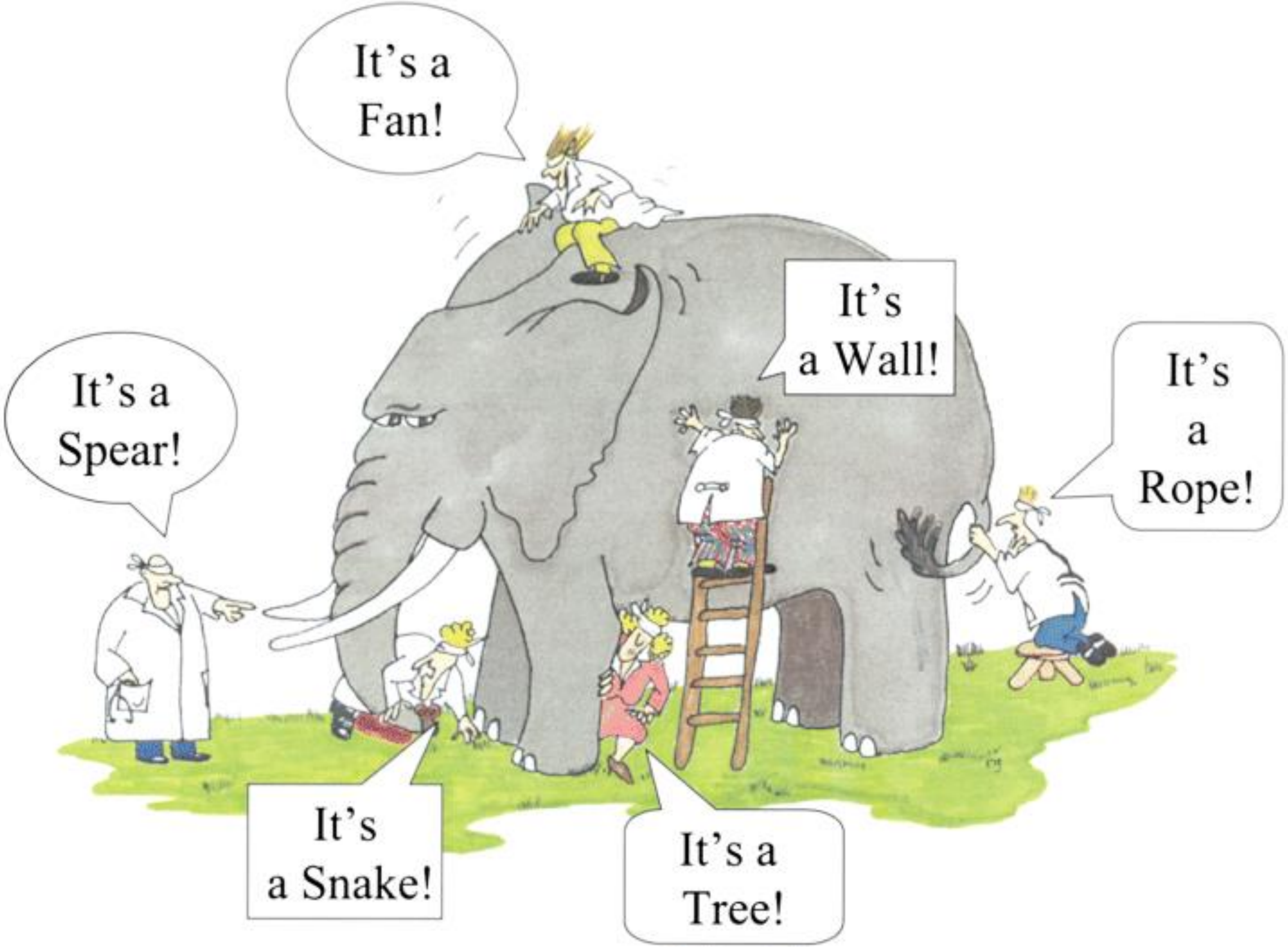
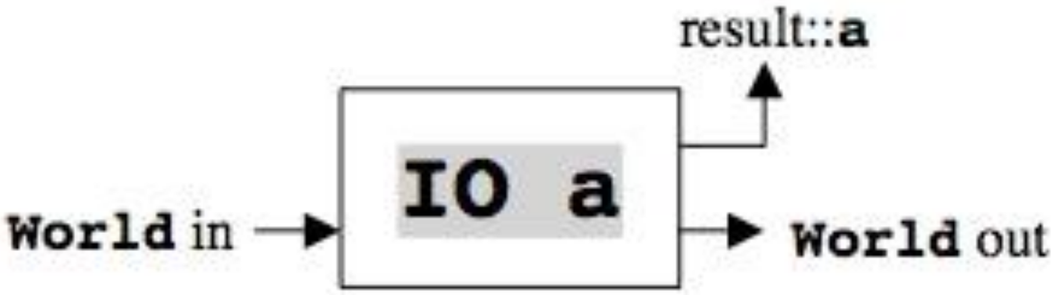
So, what's a monad?



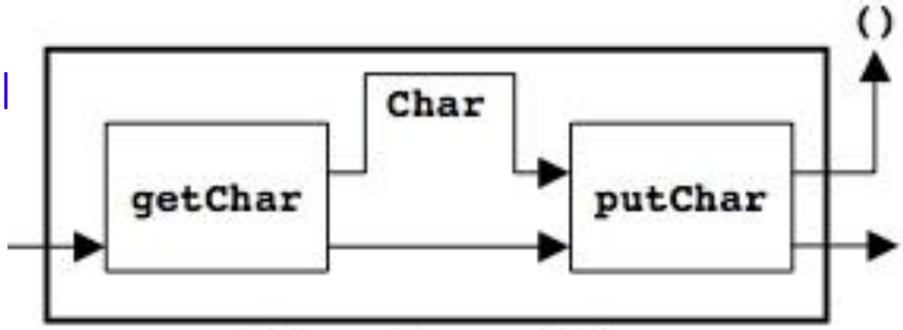
Dealing with state

- To have state *and* pure functions, the old state of the world must be passed in as a parameter, and the new state of the world returned as a result
- A monad is a way of automatically maintaining state
- `IO a` can be thought of as a function whose type is `World -> (a, World)`



The “bind” operator, >>=

- We will want to take the “state of the world” resulting from one function, and pass it into the next function
- Suppose we want to read a character and then print it
- Types:
  - `getChar :: IO Char`
  - `putChar :: Char -> IO ()`
- The result of `getChar` isn’t something that can be given to `putChar`
  - The `IO Char` “contains” a `Char` that has to be extracted to be given to `putChar`
  - `(>>=) :: IO a -> (a -> IO b) -> IO b`
- Hence,
-
```
Prelude> getChar >>= putCl
a
aPrelude>
```



getChar >>= putChar

The “then” operator, >>

- The second argument to `>>=` is a function (such as `putChar`)
  - This is what we need for passing along a result
  - It is convenient to have another function that doesn’t demand a function as its second argument

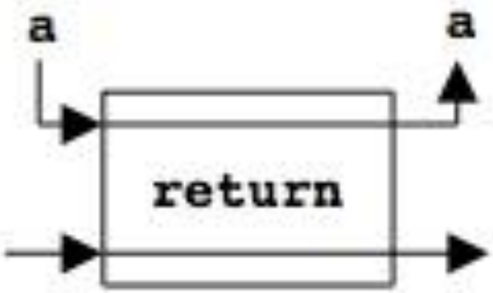
- The “then” operator simply throws away its contents

  -
    ```
    (>>) :: IO a -> IO b -> IO b
    ```

  -
    ```
    Prelude> putChar 'a' >> putChar 'b' >>
    putChar '\n'
    ab
    Prelude>
    ```

The `return` function

- Finally, it is helpful to be able to create a monad container for arbitrary values
- `return :: a -> IO a`
- The action `(return` *v*`)` is an action that does no I/O, and immediately returns *v* without having any side effects
-
```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \ c1 ->
                             getChar >>= \ c2 ->
                             return (c1,c2)
```



do notation

- From the last slide:

  -
    ```
    getTwoChars :: IO (Char,Char)
    getTwoChars = getChar >>= \ c1 ->
                           getChar >>= \ c2 ->
                           return (c1,c2)
    ```
- That's pretty hard to read
- The `do` provides "syntactic sugar"

  -
    ```
    get2Chars :: IO (Char,Char)
    get2Chars = do
            c1 <- getChar
            c2 <- getChar
            return (c1,c2)
    ```
  - The `do` also allows the `let` form (but without `in`)

Formal definition of a monad

- A monad consists of three things:
  - A type constructor M
  - A bind operation, `(>>=) :: (Monad m) => m a -> (a -> m b) -> m b`
  - A return operation, `return :: (Monad m) => a -> m a`
- And the operations must obey some simple rules:
  - `return x >>= f    =    f x`
    - `return` just sends its result to the next function
  - `m >>= return      =     m`
    - Returning the result of an action is equivalent to just doing the action
  - `do {x <- m1; y <- m2; m3}  =`
    `do {y <- do {x <- m1; m2} m3}`
    - `>>=` is associative

sequence

- sequence takes a list of I/O actions and produces a list of results
- `sequence :: [IO a] -> IO [a]`

  -
    ```
    main = do
        rs <- sequence [getLine, getLine, getLine]
        print rs
    ```
  - is equivalent to

  -
    ```
    main = do
        a <- getLine
        b <- getLine
        c <- getLine
        print [a,b,c]
    ```