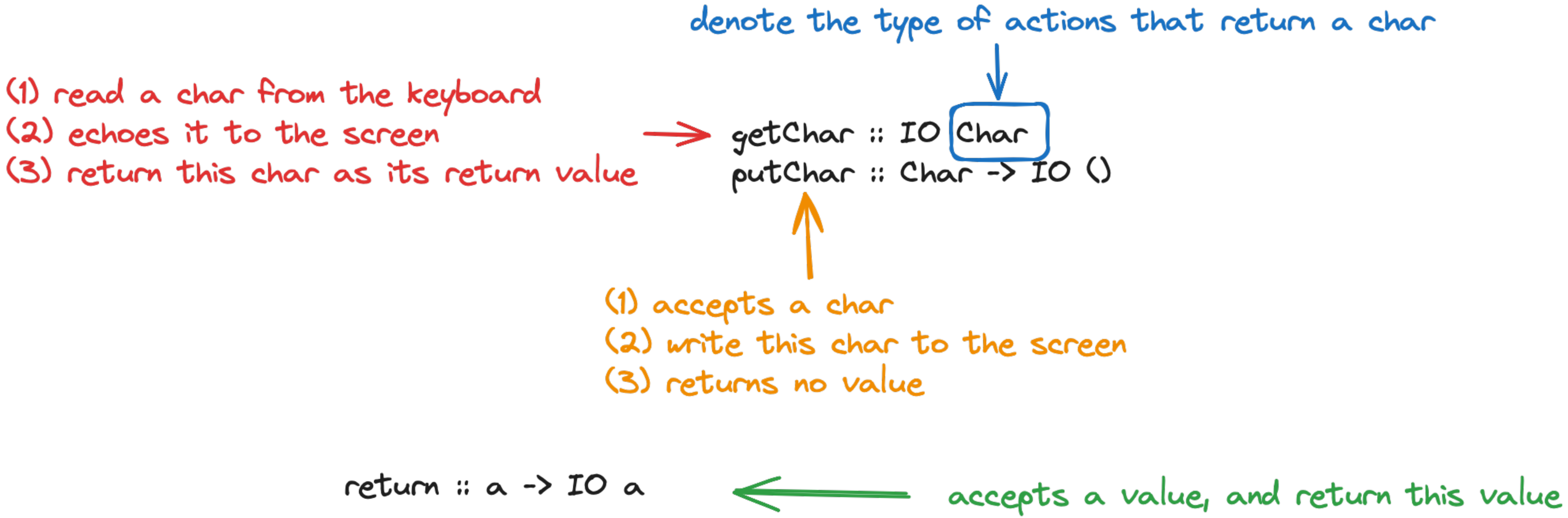
COMP3258

Functional Programming

Tutorial Session 8: IO and Monads

Review: primitives



Review: derived primitives

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
               return []
             else
               do xs <- getLine
                  return (x:xs)
```

```
getLine :: IO String
putStr :: String -> IO ()
putStrLn :: String -> IO ()
```

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

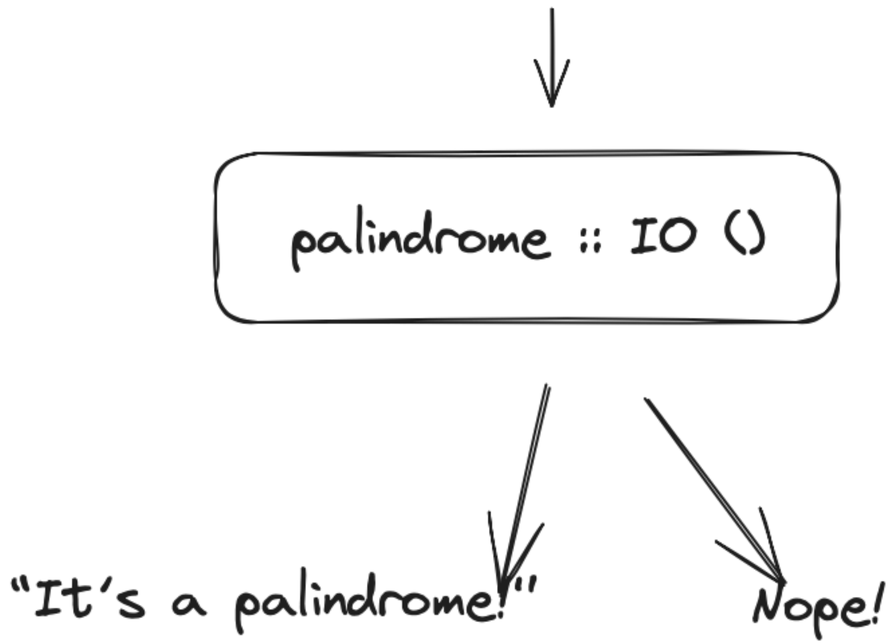
```
putstr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Q: palindrome

"Eva, can I stab bats in a cave?"

"Was it a car or a cat I saw?"

"Dammit, I'm mad!"

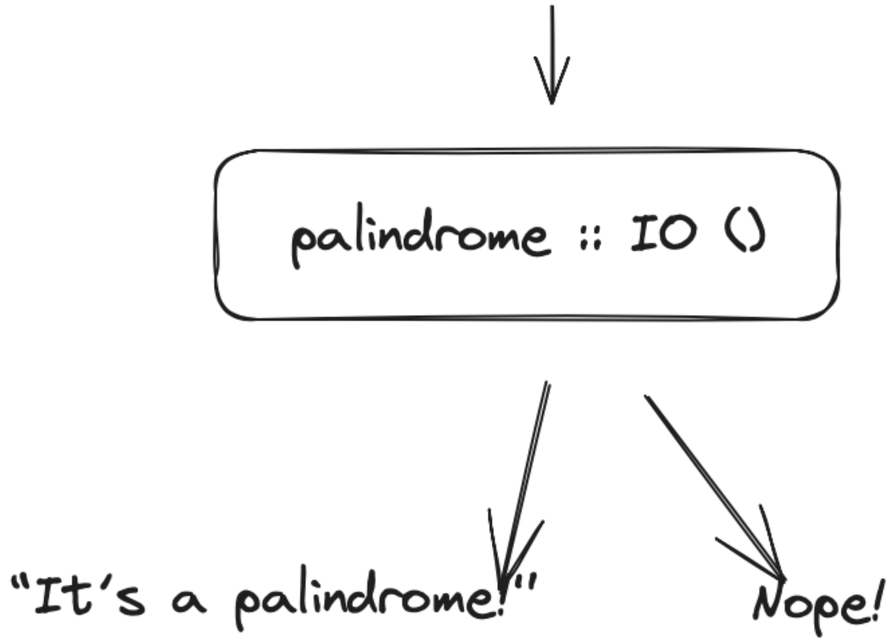


Q: palindrome

"Eva, can I stab bats in a cave?"

"Was it a car or a cat I saw?"

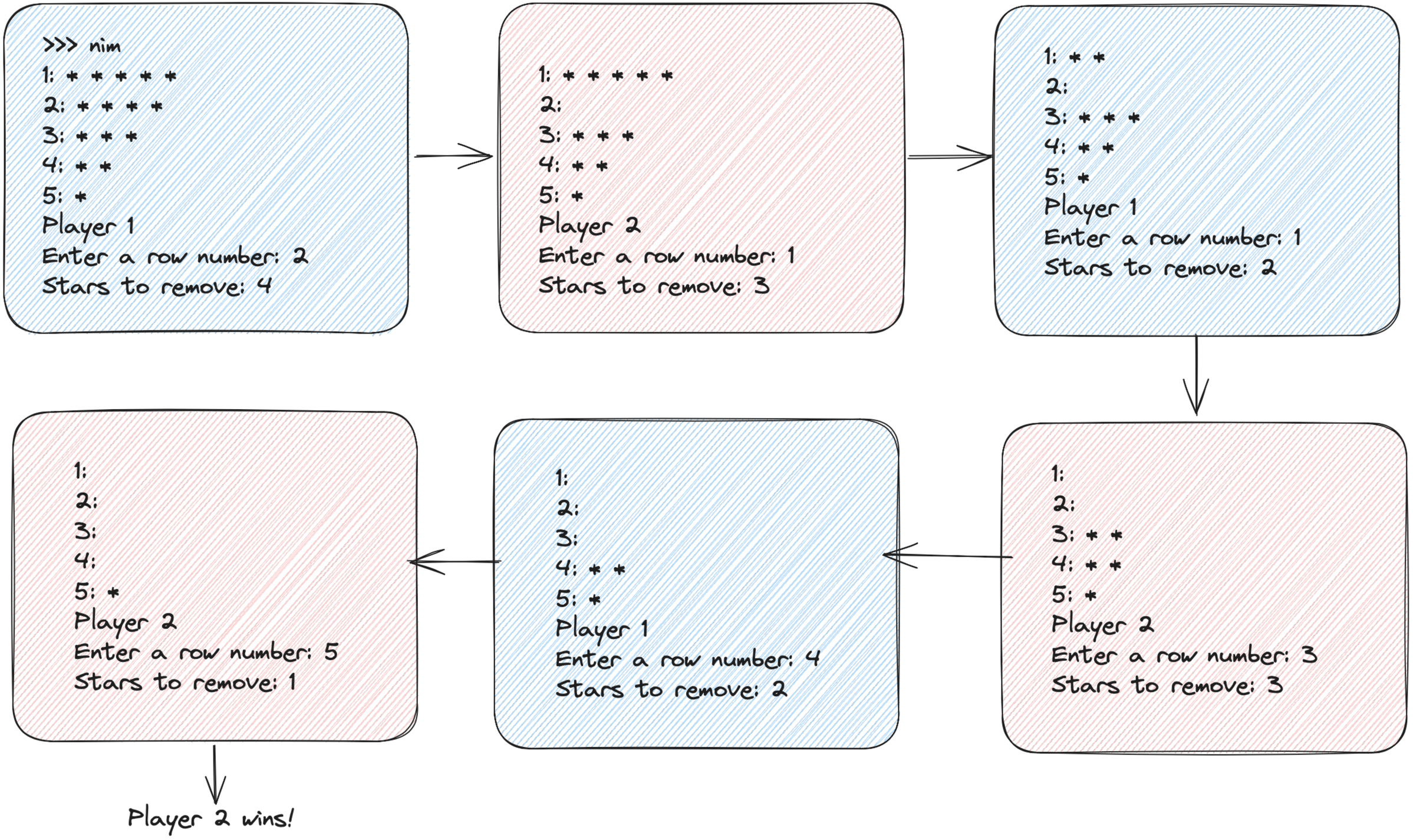
"Dammit, I'm mad!"

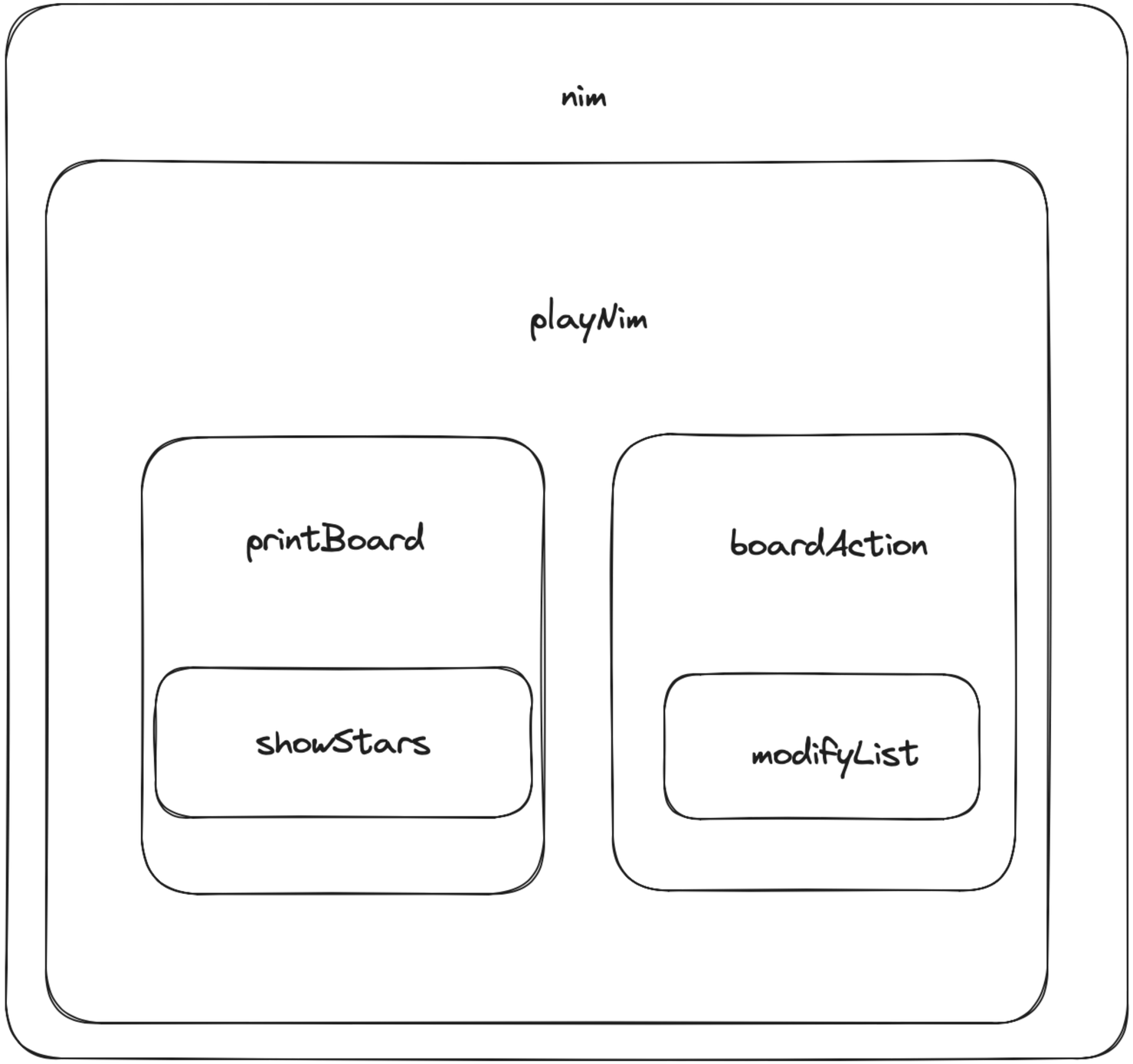



```haskell
palindrome :: IO ()
palindrome = do
  putStrLn "Enter a sentence:"
  sentence ← getLine

  let cleanedSentence = map toLower $ filter isAlpha sentence

  if isPalindrome cleanedSentence
    then putStrLn "It's a palindrome!"
    else putStrLn "Nope!"

isPalindrome :: String → Bool
isPalindrome s = s == reverse s
```

Q: Nim Game

- Two players take turns to remove one or more stars from the end of a single row.
- The winner is the player who removes the last star or stars from the board.


>>> nim
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
Player 1
Enter a row number: 2
Stars to remove: 4
1: * * * * *
2:
3: * * *
4: * *
5: *
Player 2
Enter a row number: 1
Stars to remove: 3
1: * *
2:
3: * * *
4: * *
5: *
Player 1
Enter a row number: 1
Stars to remove: 2
1:
2:
3: * *
4: * *
5: *
Player 2
Enter a row number: 3
Stars to remove: 3
1:
2:
3:
4: * *
5: *
Player 1
Enter a row number: 4
Stars to remove: 2
1:
2:
3:
4:
5: *
Player 2
Enter a row number: 5
Stars to remove: 1
Player 2 wins!




```
nim :: Int -> IO ()
playNim :: [Int] -> Player -> IO ()

printBoard :: [Int] -> IO ()
showStars :: Int -> String

boardAction :: [Int] -> IO [Int]
modifyList :: (a -> a) -> Int -> [a] -> [a]
```

Q: showStars

```
λ: showStars 0
""
λ: showStars 1
"*"
λ: showStars 3
"* * *"
λ: showStars 4
"* * * *"
```

```haskell
showStars :: Int → String
showStars = intersperse ' ' . (`replicate` '*')
```

Q: printBoard

```
λ: printBoard [1]

1: *

λ: printBoard [1,1]

1: *
2: *

λ: printBoard [1,1,2]

1: *
2: *
3: * *

λ: printBoard [1..5]

1: *
2: * *
3: * * *
4: * * * *
5: * * * * *
```

```haskell
printBoard :: [Int] -> IO ()
printBoard board = putStrLn "" >> printBoard' board >> putStrLn ""
 where
  printBoard' :: [Int] -> IO ()
  printBoard' b = forM_ (zip b [1 :: Int ..])
    $ \(ns, idx) -> putStrLn $ printf "%d: %s" idx (showStars ns)
```

Q: modifyList

```
λ: modifyList (+1) 2 [1,2,3,4]
[1,2,4,4]
λ: modifyList (subtract 1) 2 [1,2,3,4]
[1,2,2,4]
```

```haskell
modifyList :: (a → a) → Int → [a] → [a]
modifyList f 0 (h : t) = f h : t
modifyList f n (h : t) = h : modifyList f (n - 1) t
modifyList _ _ []      = []
```

Q: playNim

```haskell
nim :: Int → IO ()
nim x = do
  let board = [x, x - 1 .. 1]
  printBoard board
  playNim board P1
```

```haskell
playNim :: [Int] → Player → IO ()
playNim board p = if all (== 0) board
  then putStrLn $ printf "%s wins!" (show $ switchP p)
  else do
    putStrLn "" >> print p
    board' ← boardAction board
    printBoard board'
    playNim board' (switchP p)
 where
  boardAction :: [Int] → IO [Int]
  boardAction b = do
    putStr "Enter a row number: "
    row ← readLn
    putStr "Star to remove: "
    n ← readLn
    return $ modifyList (max 0 . subtract n) (row - 1) b
```

Monads

The basic intuition is that it combines two computations into one larger computation

We've already got this intuition from Parser monad, which combines two parser into a larger parser using do-notation (desugars to bind operation).

However, this intuition cannot apply to all monad cases.
Personally, I would like to develop each intuition for each different monads.

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

Monads

```
newtype Parser a = P (String -> [(a, String)])
instance Monad Parser where
  return v = P (\inp -> [(v, inp)])
  p >>= f = P (\inp -> case parse p inp of
                         []->[]
                         [(v, out)] -> parse (f v) out)
```

what's the type of p and f?

Monads Desugar Rules

```
do pattern <- exp
   morelines

exp >>= (\pattern -> do morelines)
```

```
do exp
   morelines

exp >>= (\_ -> do morelines)
```

```
do return exp

return exp
```

Desugar this expression:

```
p :: Parser (Char, Char)
p = do x ← item
       y ← item
       return (x, y)
```

Monad Laws

Left Identity:

```
return a >>= k = k a
```

Right Identity:

```
m >>= return = m
```

Associativity:

```
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Maybe Monad

```
instance Monad Maybe where
    return :: a → Maybe a
    return x  = Just x

    (>>=) :: Maybe a → (a → Maybe b) → Maybe b
    (>>=) m g = case m of
                  Nothing → Nothing
                  Just x  → g x
```

Propagating failures

Maybe Monad

Suppose we are validating a user registration, where they give us their email, their password, and their age. We'll provide simple functions for validating each of these input strings and converting them into newtype values:

```haskell
newtype Email = Email String
newtype Password = Password String
newtype Age = Age Int

validateEmail :: String -> Maybe Email
validateEmail input = if '@' `elem` input
  then Just (Email input)
  else Nothing

validatePassword :: String -> Maybe Password
validatePassword input = if length input > 12
  then Just (Password input)
  else Nothing

validateAge :: String -> Maybe Age
validateAge input = case (readMaybe input :: Maybe Int) of
  Nothing -> Nothing
  Just a -> Just (Age a)
```

Maybe Monad

```haskell
data User = User Email Password Age

processInputs :: (String, String, String) → Maybe User
processInputs (i1, i2, i3) = do
  email ← validateEmail i1
  password ← validatePassword i2
  age ← validateAge i3
  return $ User email password age
```

You'll get Nothing if any validation fails

One step further: Either Monad

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

instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r
```

Propagating failures with infos

Either Monad

```haskell
data ValidationError = BadEmail String
                     | BadPassword String
                     | BadAge String
                     deriving (Show)

validateEmail :: String -> Either ValidationError Email
validateEmail input = if '@' `elem` input
  then Right (Email input)
  else Left (BadEmail input)

validatePassword :: String -> Either ValidationError Password
validatePassword input = if length input > 12
  then Right (Password input)
  else Left (BadPassword input)

validateAge :: String -> Either ValidationError Age
validateAge input = case (readMaybe input :: Maybe Int) of
  Nothing -> Left (BadAge input)
  Just a -> Right (Age a)

processInputs :: (String, String, String) -> Either ValidationError User
processInputs (i1, i2, i3) = do
  email <- validateEmail i1
  password <- validatePassword i2
  age <- validateAge i3
  return $ User email password age
```

Either Monad

```
createUser :: IO (Either ValidationError User)
createUser = do
  i1 ← getLine
  i2 ← getLine
  i3 ← getLine
  let result = processInputs (i1, i2, i3)
  case result of
    Left e → print ("Validation Error: " ++ show e) >> return (Left e)
    Right u → return (Right u)

λ: createUser
someone at gmail dot com
password
42
"Validation Error: BadEmail \"someone at gmail dot com\""
```