Class 6: Lazy Evaluation

February 27

what is laziness?

```
f :: Int -> Int -> Int
f = …
```

to evaluate

```
f 5 (3^1000)
```

need to first evaluate

```
3^1000
```

```
f :: Int -> Int -> Int
f x y = x + y
```

to evaluate

```
f 5 (3^1000)
```

need to first evaluate

```
3^1000
```

```haskell
f :: Int -> Int -> Int
f x _ = x + 2
```

to evaluate

```haskell
f 5 (3^1000)
```

need to first evaluate

```haskell
3^1000
```

strict evaluation

evaluation of function arguments is delayed as long as possible.

lazy evaluation

```
f :: Int -> Int -> Int
f x _ = x + 2
```

to evaluate

```
f 5 (3^1000)
```

don't need to evaluate

```
3^1000
```

lazy evaluation

```
f ():
  print “hello”;
  return 1;

g ():
  print “bye”;
  return 2;
```

```
add (f(), g());
```

```
add (x, y):
  return x + y;
```

side effects

```
f ():
  print “hello”;
  return 1;

g ():
  print “bye”;
  return 2;
```

```
add (f(), g());
```

```
add (x, y):
  return y + x;
```

side effects

purity enables laziness.
laziness enforces purity.

side effects

pattern matching drives evaluation

```haskell
f1 :: Maybe a -> [Maybe a]
f1 m = [m, m]

f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

pattern matching drives evaluation

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m, m]

  f1 (safeHead [3^1000, 5])
= [safeHead [3^1000, 5],
   safeHead [3^1000, 5]]
```

pattern matching drives evaluation

```
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

```
  f2 (safeHead [3^1000, 5])
= f2 (Just (3^1000))
= [3^1000]
```

pattern matching drives evaluation

complexity

short circuiting

```
(||) :: Bool -> Bool -> Bool
True  || _ = True
False || b = b

(||!) :: Bool -> Bool -> Bool
True  ||! True = True
True  ||! False = True
False ||! True = True
False ||! False = False
```

(||) operator

```
(||)  :: Bool -> Bool -> Bool
(||!) :: Bool -> Bool -> Bool

True ||  (expensive computation)
True ||! (expensive computation)
```

(||) operator

```
(||)  :: Bool -> Bool -> Bool
(||!) :: Bool -> Bool -> Bool

True ||  (error “bad”)
True ||! (error “bad”)
```

(&&) operator

```
if :: Bool -> a -> a -> a
if True  x _ =  x
if False _ y = y
```

user-defined control structures

```
foo = if condition
      then
        let x = expensive in
        do something with x
      else
        do something else
```

where clauses

```
foo = let x = expensive in
      if condition
      then
        do something with x
      else
        do something else
```

where clauses

```
foo = if condition
      then
        do something with x
      else
        do something else
  where
    x = expensive
```

where clauses

infinite data structures

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  take 2 (repeat 7)
=
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  take 2 (repeat 7)
= take 2 (7 : repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x : xs)   = x : take (n - 1) xs
```

```
  take 2 (7 : repeat 7)
= 7 : take (2 - 1) (repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  7 : take (2 - 1) (repeat 7)
= 7 : take 1 (repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  7 : take 1 (repeat 7)
= 7 : take 1 (7 : repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  7 : take 1 (7 : repeat 7)
= 7 : 7 : take (1 - 1) (repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []          = []
take n (x : xs)    = x : take (n - 1) xs
```

```
  7 : 7 : take (1 - 1) (repeat 7)
= 7 : 7 : take 0 (repeat 7)
```

infinite list: repeat

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x : xs)   = x : take (n - 1) xs
```

```
  7 : 7 : take (1 - 1) (repeat 7)
= 7 : 7 : []
```

infinite list: repeat

```
nats :: [Int]
nats = 0 : map (+ 1) nats
```

infinite list: natural numbers

```
take 2 nats
= take 2 (0 : map (+ 1) nats)
= 0 : take 1 (map (+ 1) nats)
= 0 : take 1 (map (+ 1) (0 : map (+ 1) nats))
= 0 : take 1 (1 : map (+ 1) (map (+ 1) nats))
= 0 : 1 : take 0 (map (+ 1) (map (+ 1) nats))
= 0 : 1 : []
```

infinite list: natural numbers

```
upTo10 :: [Int]
upTo10 = [0..9]
```

infinite list: natural numbers

```haskell
upTo10 :: [Int]
upTo10 = [0..9]

nats :: [Int]
nats = [0..]
```

infinite list: natural numbers

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

infinite list: fibonacci numbers

```haskell
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

fib :: Int -> Int
fib n = fibs !! n
```

infinite list: fibonacci numbers

```
data Stream a = Cons a (Stream a)
```

```haskell
streamToList :: Stream a -> [a]
streamToList (Cons x xs) = x : streamToList xs

streamTake :: Int -> Stream a -> [a]
streamTake n = take n . streamToList
```

(exercise: streamIterate)

function reuse

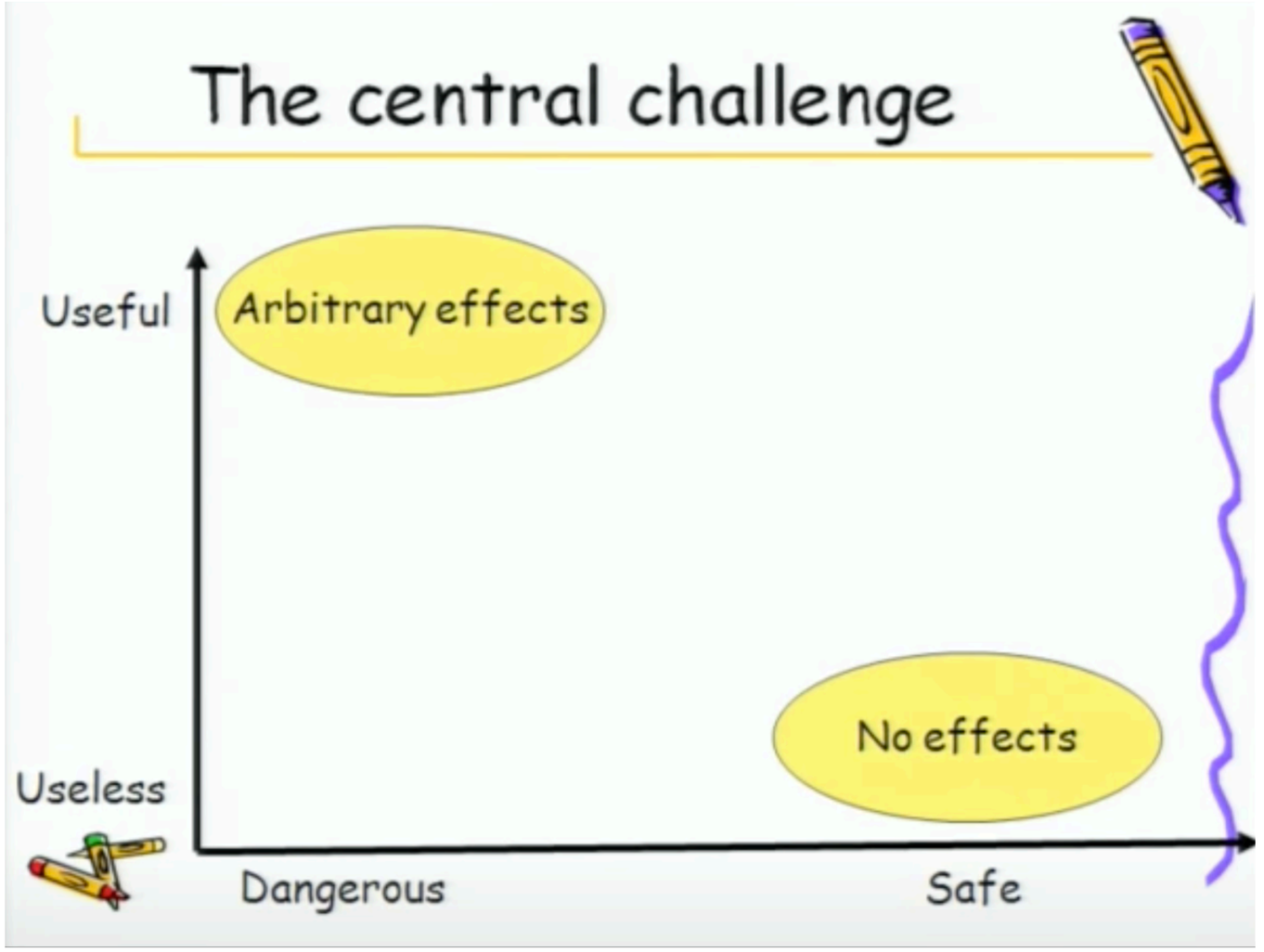
```
any :: (a -> Bool) -> [a] -> Bool
any f [] = False
any f (x : xs) = f x || any f xs
```

any

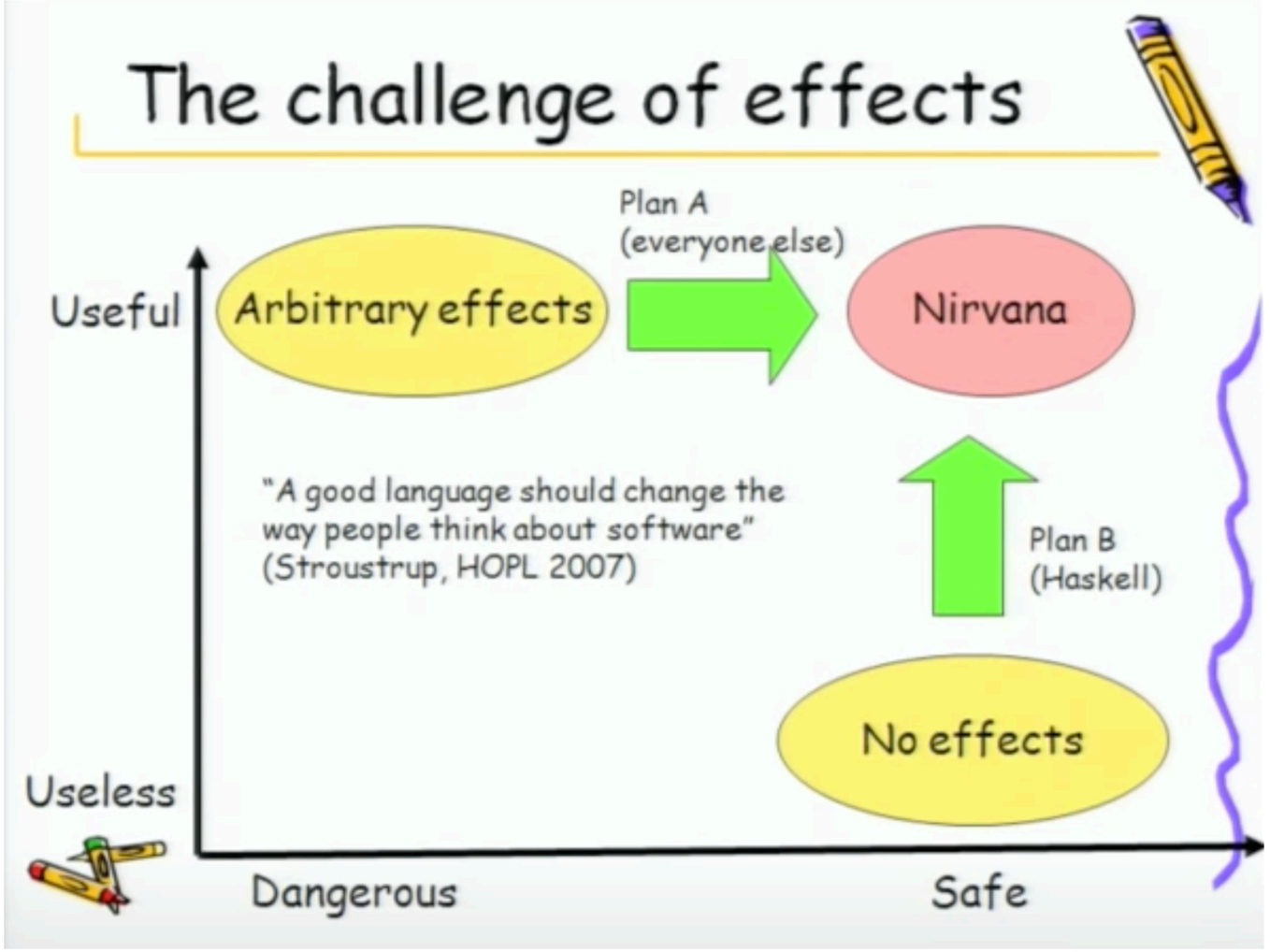
```haskell
or :: [Bool] -> Bool
or = foldr (||) False

any :: (a -> Bool) -> [a] -> Bool
any f = or . map f
```

any

big picture



“a history of Haskell: being lazy with class”



"a history of Haskell: being lazy with class"