Class 6: Type Classes

February 21

parametricity

```
f :: a -> a -> a
```

```
f :: a -> a -> a
f a1 a2 = a1 && a2
```

```haskell
f :: a -> a -> a
f a1 a2 = case (typeOf a1) of
  Int -> a1 + a2
  Bool -> a1 && a2
  _ -> a1
```

```
f :: a -> a -> a
```

parametric polymorphism

```haskell
f :: a -> a -> a
f a1 a2 = a1

f :: a -> a -> a
f a1 a2 = a2
```

parametric polymorphism

```
f :: a -> a
```

parametric polymorphism

```
f :: a -> b
```

parametric polymorphism

```
f :: (a -> a) -> a
```

parametric polymorphism

```
f :: (a -> a) -> a -> a
```

parametric polymorphism

type classes

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
show :: Show a => a -> String
```

look at these type signatures

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char
```

```haskell
instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char

instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2  Int -> Int -> Bool
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2  Char -> Char -> Bool
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

the Eq type class

```haskell
data Foo = A Int | B Char


instance Eq Foo where
  (==) :: Foo -> Foo -> Bool
  (A i1) == (A i2) = i1 == i2
  (B c1) == (B c2) = c1 == c2
  _ == _ = False

  (/=) :: Foo -> Foo -> Bool
  foo1 /= foo2 = not (foo1 == foo2)
```

`Foo -> Foo -> Bool`

the Eq type class

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

the Eq type class

(tree exercise + example)

type class constraint

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys) = x == y || elem x ys
```

type class polymorphic functions

vs. Java interfaces?

```
class Show a where
  show :: a -> String
```

the Show type class

```haskell
data Foo = A Int | B Char


instance Show Foo where
  show :: Foo -> String
  show (A i) = “A” ++ show i
  show (B c) = “B” ++ show c
```

the Show type class

```
data Foo = A Int | B Char
  deriving (Eq, Show)
```

deriving

(duration exercise)

other standard type classes

(homework overview)