CS 558 17 Sept 2024

Note Title 2024-09-17

Built-in lists are in two variants, empty [] and non-empty x:xs
:: [a] :: [a]

```
data Shape = Circle Double          -- the Double value describes the reading
           | Square Double
           | Rectangle Double Double
           deriving Show
```

a new type

"data constructor"

Type Shape is not polymorphic / not recursive

```
Circle 5.0 :: Shape
Rectangle 1.5 2.5 :: Shape

Circle :: Double -> Shape
Rectangle :: Double -> Double -> Shape
```

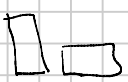
```
shapes :: [Shape]
shapes = [Rectangle 1 2, Circle 5]

circles :: [Shape]
circles = map Circle [1.0, 2.0, 3.0]

area :: Shape -> Double
area (Square a) = a * a
area (Rectangle a b) = a * b
area (Circle r) = 3 * r * r

perimeter :: Shape -> Double
perimeter s =
    case s of
        Square a    -> 4 * a
        Circle r    -> 6 * r
        Rectangle a b -> 2 * (a + b)
```

Shape printing:

show (Circle 5)
→ "Circle 5"

we can have this instead of the deriving Show clause

```
instance Show Shape where
    show (Circle r) = "circle with radius" ++ show r
    show (Rectangle a b) = ...
    show (Square a) = ...
```



```
instance Eq Shape where
    Circle r == Circle r' = r == r'
    Rectangle a b == Rectangle a' b' = (a == a' && b == b') || (a == b' && b == a')
    Square a == Square a' = a == a'
    Rectangle a b == Square c = a == b && a == c
    Square c == Rectangle a b = a == b && a == c
    _ == _ = False
```

Polymorphic, but not recursive:

```
data Maybe a = Nothing
               | Just a
```

(a: type parameter)

```
Nothing :: Maybe a
Just :: a -> Maybe a
Just 3 :: Maybe Int
Just ["a", "b"] :: Maybe [String]
```

```
f :: a -> Maybe b
⋮
case f x of
    Nothing -> ....
    Just y  -> ....
```

(y: b)

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

Recursive, but not polymorphic:

```
data Nat = Zero
         | Succ Nat

one :: Nat
one = Succ Zero
two :: Nat
two = Succ one

Succ (Succ (Succ Zero)) :: Nat

add :: Nat -> Nat -> Nat
add Zero y = y
add (Succ x) y = add x (Succ y)

instance Eq Nat where
  Succ x == Succ y = x == y
  Zero == Zero = True
  Succ x == Zero = False
  Zero == Succ = False
```