

Types, Type Classes, Polymorphism, and Pattern Matching

Stephen A. Edwards

Columbia University

Fall 2026



Algebraic Data Types: The Built-in Bool Type

```
data Bool = False | True  -- type constructor = data constructor | data...  
    deriving Show          -- Use the default printing rules
```

```
infixr 3 &&                -- a && b && c = a && (b && c)  
(&&) :: Bool -> Bool -> Bool -- Top-level type declaration  
(&&) False _ = False       -- Data constructor as a pattern  
(&&) True  x = x
```

```
ghci> True && True && True
```

```
True
```

```
ghci> :t Bool
```

```
<interactive>:1:1: error: [GHC-31891]
```

```
    * Illegal term-level use of the type constructor or class 'Bool'
```

```
ghci> :info Bool
```

```
type Bool :: *                -- Its kind: a simple type; no polymorphism
```

```
data Bool = False | True     -- Its definition
```

```
instance [safe] Show Bool    -- Its typeclasses
```

if then else

```
ifthenelse :: Bool -> a -> a -> a  
ifthenelse p t e = case p of  
  True  -> t  
  False -> e
```

```
ghci> let x = 183 in ifthenelse (x > 180) "Tall" "Short"  
"Tall"  
ghci> let x = 160 in ifthenelse (x > 180) "Tall" "Short"  
"Short"  
ghci> let x = 183 in if x > 180 then "Tall" else "Short"  
"Tall"  
ghci> let x = 160 in if x > 180 then "Tall" else "Short"  
"Short"
```

if must always have a **then** and an **else** clause

The types of **then** and **else** must match

Guards

```
height h = case h > 180 of  
  True  -> "Tall"  
  False -> "Short"
```

```
height' h = if h > 180 then "Tall" else "Short"
```

```
height'' h | h > 180 = "Tall"      -- Guard  
           | otherwise = "Short"
```

A | following a pattern is a *guard*: a Boolean expression that must be true for the pattern to completely match

Guards are tested in order

otherwise is a synonym for **True**

Filter: Keep List Elements That Satisfy a Predicate

odd and *filter* are Standard Prelude functions

```
odd n = n `rem` 2 == 1
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
```

```
filter p (x:xs) | p x      = x : filter p xs  
                | otherwise = filter p xs
```

```
ghci> filter odd [1..10]
```

```
[1,3,5,7,9]
```


Built-in Types (Part of the Standard Prelude)

Bool	Booleans: True or False
Char	A single Unicode character, about 25 bits
Word	Word-sized unsigned integers. E.g., 64 bits on my x86_64 Linux desktop
Int	Word-sized integers; the usual integer type. E.g., 64 bits on my x86_64 Linux desktop
Integer	Unbounded integers. Less efficient, so only use if you need <i>really</i> big integers
Float	Single-precision floating point
Double	Double-precision floating point

Programming Challenge: A Three-Function Calculator

File calc.hs:

```
data Op = Add | Sub | Mul
      deriving Show                -- Default printing rules

data Expr = BinOp Expr Op Expr -- E.g., 5 + 3
          | Neg  Expr          -- E.g., -7
          | Lit  Int           -- E.g., 42
      deriving Show
```

```
ghci> :load calc
[1 of 2] Compiling Main                ( calc.hs, interpreted )
Ok, one module loaded.
ghci> Lit 5
Lit 5
ghci> :t BinOp (Lit 5) Add (Lit 7)  -- 5 + 7
BinOp (Lit 5) Add (Lit 7) :: Expr
```

Expression Evaluation with Pattern Matching

```
data Op    = Add | Sub | Mul
      deriving Show

data Expr = BinOp Expr Op Expr
          | Neg   Expr
          | Lit   Int
      deriving Show
```

```
eval :: Expr -> Int
eval (Lit n) = n
eval (Neg e) = negate $ eval e
eval (BinOp e1 op e2) =
    case op of Add -> e1' + e2'
              Sub -> e1' - e2'
              Mul -> e1' * e2'
    where e1' = eval e1
          e2' = eval e2
```

```
ghci> eval $ Lit 5
5
ghci> eval $ BinOp (Lit 5) Add (Lit 7)      --      5 + 7
12
ghci> eval $ BinOp (Neg (Lit 5)) Add (Lit 7) --      -5 + 7
2
```

Pretty Printing: Split into precedence levels

```
term :: Expr -> String
term (BinOp e1 Add e2) = term e1 ++ " + " ++ fact e2
term (BinOp e1 Sub e2) = term e1 ++ " - " ++ fact e2
term e                  = fact e

fact :: Expr -> String
fact (BinOp e1 Mul e2) = fact e1 ++ " * " ++ atom e2
fact e                 = atom e

atom :: Expr -> String
atom (Lit n) = show n
atom (Neg e) = '-' : atom e
atom e      = "(" ++ term e ++ ")"
```

```
ghci> term $ BinOp (BinOp (BinOp (Lit 1) Sub (Lit 2)) Sub (BinOp (Lit 3) Sub (Lit 4)))
"1 - 2 - (3 - 4)"
```

The Num Typeclass

The usual arithmetic operators and even numeric literals are polymorphic

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t (-)
(-) :: Num a => a -> a -> a
ghci> :t (*)
(*) :: Num a => a -> a -> a
ghci> :t 42
42 :: Num a => a
```

“+ operates on any type that implements the **Num** typeclass”

“42 is of any type that implements the **Num** typeclass”

A programming trick: let's make the **Expr** type implement the **Num** typeclass so we can, e.g., add expressions

Using the Num Typeclass to Construct Expressions

:info Num gives

```
class Num a where  
  (+) :: a -> a -> a  
  (-) :: a -> a -> a  
  (*) :: a -> a -> a  
  negate :: a -> a  
  abs :: a -> a  
  signum :: a -> a  
  fromInteger :: Integer -> a
```

Using the Num Typeclass to Construct Expressions

:info Num gives

```
class Num a where
```

```
  (+) :: a -> a -> a
```

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

```
  (*) :: a -> a -> a
```

```
  negate :: a -> a
```

```
  abs :: a -> a
```

```
  signum :: a -> a
```

```
  fromInteger :: Integer -> a
```

```
{-# MINIMAL (+), (*), abs, signum,  
  fromInteger, (negate | (-)) #-}
```

Using the Num Typeclass to Construct Expressions

:info Num gives

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum,
    fromInteger, (negate | (-)) #-}

instance Num Double
instance Num Float
instance Num Int
instance Num Integer
instance Num Word
```

Using the Num Typeclass to Construct Expressions

:info Num gives

```
class Num a where
```

```
  (+) :: a -> a -> a
```

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

```
  (*) :: a -> a -> a
```

```
  negate :: a -> a
```

```
  abs :: a -> a
```

```
  signum :: a -> a
```

```
  fromInteger :: Integer -> a
```

```
  {-# MINIMAL (+), (*), abs, signum,  
    fromInteger, (negate | (-)) #-}
```

```
instance Num Double
```

```
instance Num Float
```

```
instance Num Int
```

```
instance Num Integer
```

```
instance Num Word
```

```
instance Num Expr where
```

```
  e1 + e2 = BinOp e1 Add e2
```

```
  e1 - e2 = BinOp e1 Sub e2
```

```
  e1 * e2 = BinOp e1 Mul e2
```

```
  negate e = Neg e
```

```
  abs _ = undefined
```

```
  signum _ = undefined
```

```
  fromInteger n =
```

```
    Lit (fromInteger n)
```

Using the Num Typeclass to Construct Expressions

:info Num gives

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum,
    fromInteger, (negate | (-)) #-}
```

```
instance Num Double
instance Num Float
instance Num Int
instance Num Integer
instance Num Word
```

```
instance Num Expr where
  e1 + e2 = BinOp e1 Add e2
  e1 - e2 = BinOp e1 Sub e2
  e1 * e2 = BinOp e1 Mul e2
  negate e = Neg e
  abs _ = undefined
  signum _ = undefined
  fromInteger n =
    Lit (fromInteger n)
```

```
ghci> 1 + 2
3
ghci> 1 + 2 :: Expr
BinOp (Lit 1) Add (Lit 2)
ghci> term $ (1-2)-3-(4-5)
"1 - 2 - 3 - (4 - 5)"
ghci> term $ 2+3-4*5* (6 + 7)
"2 + 3 - 4 * 5 * (6 + 7)"
```

Common Typeclasses: Eq, Ord, Enum

```
class Eq a where
```

```
  (==) :: a -> a -> Bool
```

```
  (/=) :: a -> a -> Bool
```

```
class Eq a => Ord a where
```

```
  compare :: a -> a -> Ordering
```

```
  (<) :: a -> a -> Bool
```

```
  (<=) :: a -> a -> Bool
```

```
  (>) :: a -> a -> Bool
```

```
  (>=) :: a -> a -> Bool
```

```
  max :: a -> a -> a
```

```
  min :: a -> a -> a
```

```
ghci> (compare 3 5, compare 3 3,
```

```
ghci|      compare 5 3)
```

```
(LT,EQ,GT)
```

```
class Enum a where
```

```
  succ :: a -> a
```

```
  pred :: a -> a
```

```
  toEnum :: Int -> a
```

```
  fromEnum :: a -> Int
```

```
  enumFrom :: a -> [a]
```

```
  enumFromThen :: a -> a -> [a]
```

```
  enumFromTo :: a -> a -> [a]
```

```
  enumFromThenTo ::  
                        a -> a -> a -> [a]
```

```
ghci> fromEnum 'A'
```

```
65
```

```
ghci> enumFromThenTo 'a' 'c' 'z'  
"acegikmoqsuwy"
```

Common Typeclasses: Bounded, Num, Real, Integral

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

```
class (Num a, Ord a)
  => Real a where
  toRational :: a -> Rational
```

```
class (Real a, Enum a)
  => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

```
ghci> quotRem 13 5
(2,3)
ghci> quotRem 13 (-5)
(-2,3)
ghci> divMod 13 (-5)
(-3,-2)
```

Default Implementations of Common Typeclasses

```
data OneFour = One | Two | Three | Four  
deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

```
ghci> One == One    -- Eq OneFour  
True  
ghci> One == Four  
False  
ghci> Two < Three   -- Ord OneFour  
True  
ghci> One > Two  
False  
ghci> succ One      -- Enum OneFour  
Two  
ghci> succ Three  
Four  
ghci> fromEnum Three  
2  
ghci> toEnum 1 :: OneFour  
Two
```

```
ghci> read "Two" :: OneFour  
Two  
ghci> show Three  
"Three"  
ghci> Four  
Four  
-- Show OneFour  
ghci> minBound :: OneFour  
One  
ghci> maxBound :: OneFour  
Four
```

Records: Naming Product Type Fields

```
data Person = Person { firstName :: String
                        , lastName :: String
                        , age :: Int
                        , height :: Float
                        , phoneNumber :: String
                        , flavor :: String
                        } deriving Show
```

```
hbc = Person { lastName = "Curry", firstName = "Haskell",
              age = 42, height = 6.0, phoneNumber = "555-1212",
              flavor = "Curry" }
```

```
ghci> :t lastName
lastName :: Person -> String
ghci> lastName hbc
"Curry"
```

Updating and Pattern-Matching Records

```
ghci> hbc
```

```
Person {firstName = "Haskell", lastName = "Curry", age = 42,  
        height = 6.0, phoneNumber = "555-1212", flavor = "Curry"}
```

```
ghci> hbc { age = 43, flavor = "Vanilla" }
```

```
Person {firstName = "Haskell", lastName = "Curry", age = 43,  
        height = 6.0, phoneNumber = "555-1212", flavor = "Vanilla"}
```

```
ghci> sae = Person "Stephen" "Edwards" 49 6.0 "555-1234" "Durian"
```

```
fullName :: Person -> String
```

```
fullName (Person { firstName = f, lastName = l }) = f ++ " " ++ l
```

```
ghci> map fullName [hbc, sae]
```

```
["Haskell Curry", "Stephen Edwards"]
```

Record Named Field Puns In Patterns

`:set -XNamedFieldPuns` in GHCi or put a pragma at the beginning of the file

```
{-# LANGUAGE NamedFieldPuns #-}
```

```
favorite :: Person -> String
favorite (Person { firstName, flavor } ) =
    firstName ++ " loves " ++ flavor
```

```
ghci> favorite hbc
"Haskell loves Curry"
```

Omitting a field when constructing a record is a compile-time error unless you `:set -Wno-missing-fields`, which allows uninitialized fields. Evaluating an uninitialized field throws an exception.

Record Wildcards

:set -XRecordWildCards in GHCi or add a pragma:

```
{-# LANGUAGE RecordWildCards #-}
```

```
favorite :: Person -> String
favorite Person {..} = firstName ++ " loves " ++ flavor
-- like Person { firstName = firstName, lastName = lastName, .. }
sae = let lastName = "Edwards"
      firstName = "Stephen"
      age = 50
      height = 6.0
      phoneNumber = "555-2121" in
      Person {flavor = "Pizza", ..} -- Picks up lastName, etc.
```

```
ghci> favorite hbc
"Haskell loves Curry"
ghci> firstName sae
"Stephen"
```

Polymorphic Types: Maybe

A safe replacement for null pointers

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

The *Maybe* type constructor is a function with a type parameter (*a*) that returns a type (*Maybe a*).

```
ghci> :k Maybe
```

```
Maybe :: * -> *      — A type function: takes a type as an argument
```

```
ghci> Just "your luck"
```

```
Just "your luck"
```

```
ghci> :t Just "your luck"
```

```
Just "your luck" :: Maybe String
```

```
ghci> :t Nothing
```

```
Nothing :: Maybe a      — Polymorphic by itself
```

```
ghci> :t Just (10 :: Int)
```

```
Just (10 :: Int) :: Maybe Int — Constrained the literal type
```

Association Lists: A good use of Maybe

```
phoneBook = [("Jenny", "867-5309")
              ,("Morris", "777-9311")
              ,("Alessia", "273-8255")
              ,("Alicia", "489-4608")
              ]
```

```
lookup' :: Eq a => a -> [(a,b)] -> Maybe b    -- lookup in Prelude
```

```
lookup' _ [] = Nothing
```

```
lookup' key ((k,v):xs) | k == key = Just v    -- Requires Eq a
                        | otherwise = lookup' key xs
```

```
ghci> lookup "Jenny" phoneBook
```

Just "867-5309"

```
ghci> lookup' "Alicia" phoneBook
```

Just "489-4608"

```
ghci> lookup "Nobody" phoneBook
```

Nothing

Either: Success or Noisy Failure

```
data Either a b = Left a | Right b  -- Left failure / Right success
    deriving (Eq, Ord, Read, Show)
```

```
lookup'' :: String -> [(String,a)] -> Either String a
lookup'' key [] = Left $ key ++ " not found"
lookup'' key ((k,v):xs) | k == key = Right v
                        | otherwise = lookup'' key xs
```

```
ghci> :k Either
Either :: * -> * -> *  -- Takes two type arguments
ghci> lookup'' "Alicia" phoneBook
Right "489-4608"
ghci> lookup'' "Nobody" phoneBook
Left "Nobody not found"
ghci> :t lookup'' "Nobody" phoneBook
lookup'' "Nobody" phoneBook :: Either String String
```

Polymorphic Types: Lists

```
data List a = Nil           -- The Empty list
              | Cons a (List a) -- A list cell: the payload value + the tail
deriving Show
```

```
foldr' :: (a -> b -> b) -> b -> List a -> b
foldr' _ z Nil           = z
foldr' f z (Cons x xs) = f x (foldr' f z xs)
```

```
ghci> :k List
List :: * -> *
ghci> l1 = Cons 1 (Cons 2 (Cons 3 Nil))
ghci> l1
Cons 1 (Cons 2 (Cons 3 Nil))
ghci> foldr' (+) 0 l1
6
```

Polymorphic Types: Lists

```
infixr 5 :  
data [a]    = []  
          | a : [a]  
    deriving Show
```

-- Note: Syntactically incorrect Haskell
-- The Empty list
-- A list cell: the payload value + the tail

```
foldr  :: (a -> b -> b) -> b -> List a -> b  
foldr  _ z  []          = z  
foldr  f z (x : xs)     = f x (foldr  f z xs)
```

```
ghci> :k []  
[] :: * -> *  
ghci> l1 = 1 : 2 : 3 : []  
ghci> l1  
[1,2,3]  
ghci> foldr (+) 0 l1  
6
```

Introducing type aliases with `type`

```
type AssocList k v = [(k, v)]    -- AssocList is just an alias
lookup''' :: Eq k => k -> AssocList k v -> Maybe v
lookup''' _ []                    = Nothing
lookup''' key ((k,v):xs) | k == key = Just v
                               | otherwise = lookup''' key xs
```

```
ghci> :t lookup'''
lookup''' :: Eq k => k -> AssocList k v -> Maybe v
ghci> lookup''' "Jenny" phoneBook
Just "867-5309"
ghci> :t lookup''' "Jenny"
lookup''' "Jenny" :: AssocList String v -> Maybe v
```

The Functor Type Class: Should be “Mappable”†

```
infixl 4 <$  
class Functor f where  
  fmap    :: (a -> b) -> f a -> f b    — Must have fmap id = id  
  
  (<$)    :: a -> f b -> f a            — Replace f b with f a  
  m <$ a = fmap (\_ -> a)              — Default implementation of <$
```

If $f :: a \rightarrow b$,

$bs = \text{fmap } f \text{ as}$

applies f to every a in as to give bs

$bs = as \text{ <$ } x$ replaces every a in as with x .

Here, f is a type constructor that takes an argument, like Maybe or List

† “Functor” is from Category Theory

Instance of Functor for Maybe

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- Must have fmap id = id

data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f Nothing  = ?
  fmap f (Just x) =
```

What type goes here? How do we construct such an object?

Instance of Functor for Maybe

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b  — Must have fmap id = id
```

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

```
instance Functor Maybe where
```

```
  fmap _ Nothing  = Nothing
```

```
  fmap f (Just x) = ?
```

What type goes here? How do we construct such an object?

Instance of Functor for Maybe

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b  — Must have fmap id = id
```

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

```
instance Functor Maybe where
```

```
  fmap _ Nothing = Nothing
```

```
  fmap f (Just x) = Just ?
```

What type goes here? How do we construct such an object?

Instance of Functor for Maybe

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b  — Must have fmap id = id
```

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

```
instance Functor Maybe where
```

```
  fmap _ Nothing  = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

“Apply f to the a in the box, if any, and leave it there”

Instance of Functor for Lists

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b    -- Must have fmap id = id
```

```
data List a = Nil | Cons a (List a)  
  deriving Show
```

```
instance Functor List where
```

```
  fmap f Nil          = ?
```

```
  fmap f (Cons x xs) =
```

What type goes here? How do we construct such an object?

Instance of Functor for Lists

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b    -- Must have fmap id = id  
  
data List a = Nil | Cons a (List a)  
    deriving Show  
  
instance Functor List where  
    fmap _ Nil          = Nil  
    fmap f (Cons x xs) = ?
```

What type goes here? How do we construct such an object?

Instance of Functor for Lists

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b    -- Must have fmap id = id
```

```
data List a = Nil | Cons a (List a)  
  deriving Show
```

```
instance Functor List where
```

```
  fmap _ Nil          = Nil
```

```
  fmap f (Cons x xs) = Cons ?
```

What type goes here? How do we construct such an object?

Instance of Functor for Lists

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b    -- Must have fmap id = id
```

```
data List a = Nil | Cons a (List a)  
  deriving Show
```

```
instance Functor List where
```

```
  fmap _ Nil          = Nil  
  fmap f (Cons x xs) = Cons (f x) ?
```

What type goes here? How do we construct such an object?

Instance of Functor for Lists

```
class Functor f where  
  fmap :: (a -> b) -> f a -> f b    -- Must have fmap id = id  
  
data List a = Nil | Cons a (List a)  
  deriving Show  
  
instance Functor List where  
  fmap _ Nil          = Nil  
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

```
ghci> fmap (+10) $ Cons 1 (Cons 2 (Cons 3 Nil))  
Cons 11 (Cons 12 (Cons 13 Nil))
```

Exactly the familiar `map` function

Instance of Functor for Either

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- Must have fmap id = id

data Either a b = Left a | Right b

instance Either a where
  fmap _ (Left x)  = Left x
  fmap f (Right x) = Right (f x)
```

Functor takes a type constructor with one argument (kind is $* \rightarrow *$); **Either** takes two arguments ($* \rightarrow * \rightarrow *$). Solution is to fix the **Left** type and be polymorphic in the **Right** type.