# Minicurso de Haskell

Tipos de Dados Algébricos
Typeclasses

# Definindo Novos Tipos

## Algébricos porque:

Têm mais de um construtor separados por /

Cada construtor pode ter n parâmetros

Podem ser recursivos

```
data Bool = True | False
```

```
data Person = Person String Int
```

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

```
data Tree a
     = Leaf a
     | Node a (Tree a) (Tree a)
```

# Instâncias

## Polimorfismo!

Definem o comportamento de certas funções para os nossos tipos

```haskell
data Person = Person String Int

instance Show Person where
    show (Person name age)  -- pattern matching
        = name ++ " tem " ++ show age ++ " anos"

instance Eq Person where
    Person name1 age1 == Person name2 age2
        = name1 == name2 && age1 == age2
```

# Deriving

## Polimorfismo!

Implementa automaticamente as instâncias para nós

```
data Person = Person String Int
    deriving (Eq, Show)

data Tree a
    = Leaf a
    | Node a (Tree a) (Tree a)
    deriving (Eq, Show)
```

# Type Variables em Tipos

## Polimorfismo!

Um tipo definido pode depender de seus tipos internos

Em outras linguagens isso aparece como *Tree<a>*

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

```
data Tree a
    = Leaf a
    | Node a (Tree a) (Tree a)
    deriving (Eq, Show)
```

```
data List a
    = Elem a
    | Cons a (List a)
```

# Constraints

As vezes é necessário limitar a existência de uma instância

*Maybe a* só pode ser printável se *a* também for

```
instance Show a => Show (Maybe a) where
    show (Just x) = "Just " ++ show x
    show Nothing = "Nothing"
```

# Pequeno Exercício

## Defina as instâncias de Eq, Show e Functor para

```
data List a
    = Elem a
    | Cons a (List a)
```

```
instance Show a => Show (Maybe a) where
    show (Just x) = "Just " ++ show x
    show Nothing = "Nothing"
```

-- lembre-se das constraints

```
instance Show Person where
    show (Person name age)
        = name ++ " tem " ++ show age ++ " anos"
```

-- lembre-se da sintaxe

```haskell
instance Show a => Show (List a) where
    show (Elem x) = show x
    show (Cons x xs) = show x ++ " : " ++ show xs

instance Eq a => Eq (List a) where
    (Elem x) == (Elem y) = x == y
    (Cons x xs) == (Cons y ys) = x == y && xs == ys
    _ == _ = False

instance Functor List where
    fmap f (Elem x) = Elem (f x)
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

# Typeclasses

Definem funções que podem ser utilizadas nos tipos que as intanciam

Em outras linguagens aparece como *interface* ou *trait*

| Typeclass | Funções | Tipos |
|---|---|---|
| Eq | (==), (/=) | Int, Bool, Char, Double |
| Show | show | Int, Bool, Char, Double |
| Ord | (<), (<=), (>), (>=), min, max | Int, Bool, Char, Double |
| Num | (+), (-), (*), abs | Int, Integer, Double, Float |
| Fractional | (/) | Double, Float |
| Floating | sin, cos, logBase, pi, sqrt, (**) | Double, Float |
| Bounded | minBound, maxBound | Char, Float, Int, Bool |
| Functor | fmap, (<$>) | [a], Maybe a, (b,a) |

# Constraint em Funções

```haskell
elem' :: Eq a => a -> List a -> Bool
elem' x (Elem y)     = x == y
elem' x (Cons y ys) = x == y || elem' x ys

multAll :: Num a => a -> List a -> List a
multAll x list = fmap (*x) list
```

```haskell
show :: Show a => a -> String
```

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

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

```haskell
(+) :: Num a => a -> a -> a
```

```haskell
print :: Show a => a -> IO ()
```

# Ergonomia

## Type Synonyms

Cria um sinônimo de tipo, apenas substitui o nome. Não pode ser instanciado em typeclasses.

```
type String = [Char]
type Idade = Int
type Nome = String
type Ponto = (Float, Float)
```

## Record Syntax

Cria funções automaticamente para facilitar acesso aos valores dentro de um tipo criado.

```
data Person = Person {
    name :: String,
    age :: Int
} deriving (Eq, Show)
```