From the Prelude:

```haskell
data Bool = False | True
not :: Bool -> Bool
not False = True
not True = False
```

```haskell
data Shape = Circle Radius | Rect Width Height deriving (Eq, Show)

Some values of type Shape: Circle 1.0
```
**Maybe**

From the Prelude:

```haskell
data Maybe a = Nothing | Just a  
   deriving (Eq, Show)
```

Some values of type Maybe: Nothing :: Maybe a

---

**Lists**

From the Prelude:

```haskell
data [a] = [] | (:) a [a]  
   deriving Eq
```

The result of deriving Eq:

```haskell
instance Eq a => Eq [a] where  
  []    == []    = True  
  (x:xs) == (y:ys) = x == y && xs == ys  
  _      == _      = False
```

---

**Lists**

From the Prelude:

```haskell
data [a] = [] | (:) a [a]  
   deriving Eq
```

The result of deriving Eq:

```haskell
instance Eq a => Eq [a] where  
  []    == []    = True  
  (x:xs) == (y:ys) = x == y && xs == ys  
  _      == _      = False
```

---

**Tree**

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

---

**Tree**

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

---

**Tree**

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

---

**Tree**

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

---

Defined explicitly:

```haskell
instance Show a => Show [a] where  
    show xs = "[" ++ concat cs ++ "]"
```
data Tree a = Empty | Node a (Tree a) (Tree a)
deriving (Eq, Show)

Some trees:
  Empty
  Node 1 Empty Empty
  Node 1 (Node 2 Empty Empty) Empty

Tree

find :: a -> Tree a -> Bool
find :: Ord a => a -> Tree a -> Bool

find _ Empty = False
find x (Node a l r)
  | x < a = find x l
  | a < x = find x r
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
  | x < a  = find x l
  | a < x  = find x r
  | otherwise = True

-- assumption: < is a linear ordering
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
  | x < a  = find x l
  | a < x  = find x r
  | otherwise = True

insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)

insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a = Node a (insert x l) r
  | a < x = Node a l (insert x r)
  | otherwise = Node a l r
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
    | x < a = Node a (insert x l) r
    | a < x = Node a l (insert x r)
    | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))

insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
    | x < a = Node a (insert x l) r
    | a < x = Node a l (insert x r)
    | otherwise = Node a l r

Example
insert 6 (Node 5 Empty (Node 7 Empty Empty))
 = Node 5 Empty (insert 6 (Node 7 Empty Empty))
 = Node 5 Empty (Node 7 (insert 6 Empty) Empty)
QuickCheck for Tree

```haskell
import Control.Monad
import Test.QuickCheck

-- for QuickCheck: test data generator for Trees
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree
    where
      tree 0 = return Empty
      tree n | n > 0 =
        oneof [return Empty,
               liftM3 Node arbitrary (tree (n `div` 2))
                      (tree (n `div` 2))]

prop_find_insert x y t =
  find x (insert y t) == (x == y || find x t)
```

```haskell
prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
  find x (insert y t) == (x == y || find x t)
```
prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
    find x (insert y t) == (x == y || find x t)

(Int not optimal for QuickCheck)

prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
    find x (insert y t) == (x == y || find x t)

(Int not optimal for QuickCheck)

prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
    find x (insert y t) == (x == y || find x t)

(Int not optimal for QuickCheck)
Problem: how to get from one word to another, with a minimal number of "edits".

Example: from "fish" to "chips"
data Edit = Change Char
  | Copy
  | Delete
  | Insert Char
  deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys =
data Edit = Change Char
  | Copy
  | Delete
  | Insert Char
deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
data Edit = Change Char
  | Copy
  | Delete
  | Insert Char
deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
  | x == y    = Copy : transform xs ys
  | otherwise = best [Change y
data Edit = Change Char
    | Copy
    | Delete
    | Insert Char
deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys = map Insert ys
transform xs [] = replicate (length xs) Delete
transform (x:xs) (y:ys)
    | x == y     = Copy : transform xs ys
    | otherwise  = best [Change y : transform xs ys,
                        Delete : transform (x:xs) (y:ys)]

best :: [Edit] -> [Edit]
best [x] = x
best :: [[Edit]] -> [Edit]
best [x] = x
best (x:xs)
  | cost x <= cost b = x
  | otherwise = b
  where b = best xs

Example: What is the edit distance from "trittin" to "tarantino"?

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
best :: [[Edit]] -> [Edit]
best [x] = x
best (x:xs)
    | cost x <= cost b = x
    | otherwise = b
    where b = best xs

cost :: [Edit] -> Int

Example: What is the edit distance from "trittin" to "tarantino"?
transform "trittin" "tarantino" = ?

Complexity of transform: time $O(\cdot)$
best :: [[Edit]] -> [Edit]
best [x] = x
best (x:x8)

8.2 The general case

data T a₁ ... aₚ =
    C₁ t₁₁ ... t₁k₁ \mid
    \vdots
    Cₙ tₙ₁ ... tₙkₙ

defines the constructors

    C₁ :: t₁₁ -> ... t₁k₁ -> T a₁ ... aₚ
    \vdots
    Cₙ :: tₙ₁ -> ... tₙkₙ -> T a₁ ... aₚ

Example: What is the edit distance from "trittin" to "tarantino"?
transform "trittin" "tarantino" = ?

Complexity of transform: time \(O(3^{m+n})\)
Constructors are functions too!

Constructors can be used just like other functions.

Example
map Just [1, 2, 3] = [Just 1, Just 2, Just 3]

But constructors can also occur in patterns!

Patterns revisited

Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A pattern can be
- a variable (incl. _)
- a literal like 1, 'a', "xyz", ...
- a tuple \( (p_1, \ldots, p_n) \) where each \( p_i \) is a pattern

Patterns revisited

Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A pattern can be
- a variable (incl. _)
- a literal like 1, 'a', "xyz", ...
- a tuple \( (p_1, \ldots, p_n) \) where each \( p_i \) is a pattern
- a constructor pattern \( C \ p_1 \ldots \ p_n \) where
  \( C \) is a data constructor (incl. True, False, [] and (\))
  and each \( p_i \) is a pattern
Patterns are expressions that consist only of constructors and variables (which must not occur twice):

A pattern can be

- a variable (incl. _)
- a literal like 1, 'a', "xyz", ...
- a tuple \( (p_1, \ldots, p_n) \) where each \( p_i \) is a pattern
- a constructor pattern \( C \ p_1 \ldots \ p_n \) where
  - \( C \) is a data constructor (incl. True, False, [] and ())
  - and each \( p_i \) is a pattern