Script **generated by TTT**

Title: Nipkow: Info2 (03.12.2013)

Date: Tue Dec 03 15:33:08 CET 2013

Duration: 73:25 min

Pages: 72

Bool

From the Prelude:

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

Bool

From the Prelude:

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


```
not :: Bool -> Bool
not False  =  True
not True   =  False
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

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

Some values of type Shape: `Circle 1.0`

Maybe

From the Prelude:

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

Some values of type Maybe: `Nothing ::   Maybe a`

Lists

From the Prelude:

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

The result of deriving Eq:

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

Lists

From the Prelude:

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

The result of deriving Eq:

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

Defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
```

Tree

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

Tree

```
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

```
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
Node 1 Empty (Node 2 Empty Empty)
```

Tree

```
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
Node 1 Empty (Node 2 Empty Empty)
Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
```

```
find ::           a -> Tree a -> Bool
```

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

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty  =  False
find x (Node a l r)
```

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

```
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 :: Ord a => a -> Tree a -> Tree a
insert x Empty  =  Node x Empty Empty
```

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

```haskell
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
```

```haskell
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)
```

```haskell
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
```

```haskell
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

```haskell
insert 6 (Node 5 Empty (Node 7 Empty Empty))
```

```haskell
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

```haskell
insert 6 (Node 5 Empty (Node 7 Empty Empty))
= Node 5 Empty (insert 6 (Node 7 Empty Empty))
```

```haskell
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

```haskell
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)
```

```haskell
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

```haskell
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)
= Node 5 Empty (Node 7 (Node 6 Empty Empty) Empty)
```

QuickCheck for Tree

```
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) == ???
```

```
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)
```

```
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)

```
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)

Edit distance (see Thompson)

Problem: how to get from one word to another,

Edit distance (see Thompson)

Problem: how to get from one word to another,
with a *minimal* number of "edits".

Edit distance (see Thompson)

Problem: how to get from one word to another,
with a *minimal* number of "edits".

Example: from `"fish"` to `"chips"`

```
data Edit =
```

```haskell
data Edit = Change Char
```

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)
```

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

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

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

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

transform [] ys  =
```

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

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

transform [] ys  =  map Insert ys
```

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

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

transform [] ys  =  map Insert ys
transform xs []  =
```

```haskell
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
```

```haskell
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)
```

```haskell
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       =
```

```haskell
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
```

```haskell
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
```

```haskell
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
```

```haskell
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
```

```haskell
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 xs (y:ys),
                          Insert y : transform
```

```haskell
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 xs (y:ys),
                          Insert y : transform (x:xs) ys]
```

```haskell
best :: [[Edit]] -> [Edit]
best [x]    = x
```

```
best :: [[Edit]] -> [Edit]
best [x]    = x
best (x:xs)
```

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

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

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

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

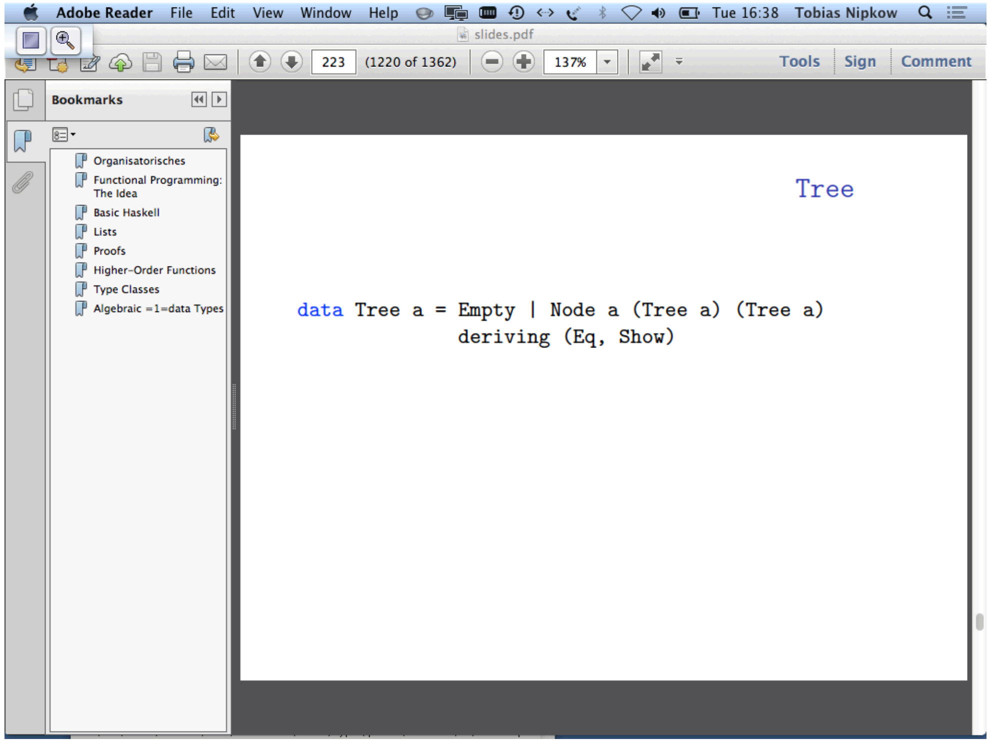
```
best :: [[Edit]] -> [Edit]
best [x]    = x
best (x:xs)
  | cost x <= cost b     = x
  | otherwise            = b
  where b = best xs

cost :: [Edit] -> Int
```

```
best :: [[Edit]] -> [Edit]
best [x]    = x
```

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

`transform "trittin" "tarantino" = ?`

Complexity of `transform`: time $O($



```
best :: [[Edit]] -> [Edit]
best [x]    = x
best (x:xs)
```

8.2 The general case

data T a_1 ... a_p =
  C_1 t_{11} ... t_{1k_1} |
  $\vdots$
  C_n t_{n1} ... t_{nk_n}

defines the *constructors*

C_1 :: t_{11} -> ... t_{1k_1} -> T a_1 ... a_p
$\vdots$
C_n :: t_{n1} -> ... t_{nk_n} -> T a_1 ... a_p

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

`transform "trittin" "tarantino" = ?`

Complexity of `transform`: time $O(3^{m+n})$

8.2 The general case

data T a_1 ... a_p =
  C_1 t_{11} ... t_{1k_1} |
  $\vdots$
  C_n t_{n1} ... t_{nk_n}

defines the *constructors*

C_1 :: t_{11} -> ... t_{1k_1} -> T a_1 ... a_p
$\vdots$
C_n :: t_{n1} -> ... t_{nk_n} -> T a_1 ... a_p

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

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`, ..., `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`, ..., `p_n`)` where each p_i is a pattern
- a constructor pattern C p_1 ... p_n where
  C is a `data` constructor (incl. `True`, `False`, `[]` and `(:)`)
  and 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