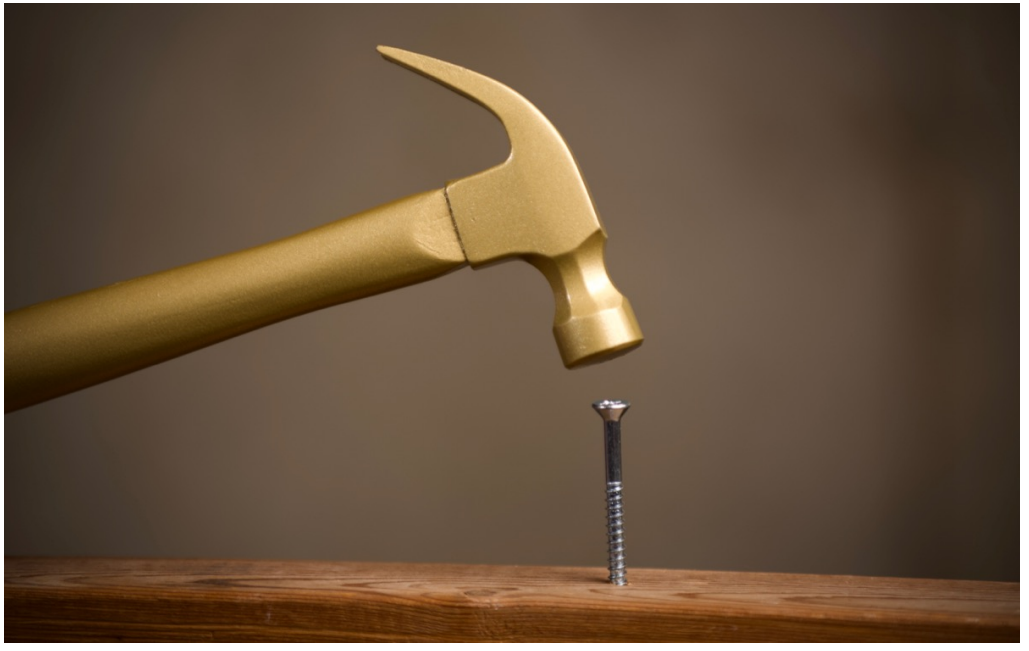
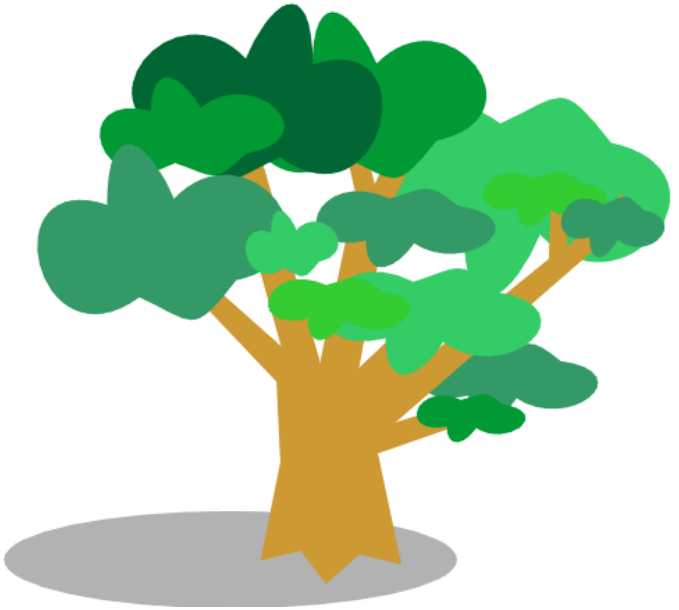
# When all the nails are trees, the hammer you need probably looks like a chainsaw

Matthieu Baechler • matthieu@baechler-craftsmanship.fr

@matthieu@framapiaf.org • @m_baechler





~~When all the nails are trees, the hammer you need probably looks like a chainsaw~~

Matthieu Baechler • matthieu@baechler-craftsmanship.fr

@matthieu@framapiaf.org • @m_baechler

# Handling trees with Chainsaw

Matthieu Baechler • matthieu@baechler-craftsmanship.fr

@matthieu@framapiaf.org • @m_baechler

# Who Am I ?

# Who Am I ?

- Software Crafter

## Who Am I ?

- Software Crafter
- Writing mainly Scala

# Who Am I ?

- Software Crafter
- Writing mainly Scala
- Recently promoted to Full Stack Developer™

# Who Am I ?

- Software Crafter
- Writing mainly Scala
- Recently promoted to Full Stack Developer™
  - writing Elm

## Who Am I ?

- Software Crafter
- Writing mainly Scala
- Recently promoted to Full Stack Developer™
  - writing Elm
- Freelancer

# What is the Agenda ?

## What is the Agenda ?

- How I became a Tree Expert

## What is the Agenda ?

- How I became a Tree Expert
- RoseTree

## What is the Agenda ?

- How I became a Tree Expert
- RoseTree
  - Definition

## What is the Agenda ?

- How I became a Tree Expert
- RoseTree
  - Definition
  - Code

# How I became a Tree Expert

# a Tree

```
case class Tree(data: String, children: List[Tree])
```

# a traversal

```
def parse(s: String): Int = 2
```

# a traversal

```
def parse(s: String): Int = 2
```

```
def recurseTree(tree: Tree): List[Int] =
    tree.children match
        case Nil => parse(tree.data) :: Nil
        case children => parse(tree.data) :: children.flatMap(recurseTree)
```

# a traversal

```
def parse(s: String): Int = 2
```

```
def recurseTree(tree: Tree): List[Int] =
    tree.children match
        case Nil => parse(tree.data) :: Nil
        case children => parse(tree.data) :: children.flatMap(recurseTree)
```

```
def recurseList(list: List[String]): List[Int] =
    list match
        case Nil => List.empty
        case head :: tail => parse(head) :: recurseList(tail)
```

# a traversal

```
def parse(s: String): Int = 2
```

```
def recurseTree(tree: Tree): List[Int] =
    tree.children match
        case Nil => parse(tree.data) :: Nil
        case children => parse(tree.data) :: children.flatMap(recurseTree)
```

```
def recurseList(list: List[String]): List[Int] =
    list match
        case Nil => List.empty
        case head :: tail => parse(head) :: recurseList(tail)
```

```
val list = List("Foo", "Bar")

recurseList(list)

list.map(parse)
```

❤️

`map`

`flatmap`

`filter`

`collect`

…

Scala 💔 Trees

Elm  Trees 

# Elm ❤️ Trees 🥳

RoseTree

# A drama in 6 acts

## 🎟 A drama in 6 acts

1. PoC with RoseTree in Elm

##  A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io

## 🎫 A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io
3. About RoseTree in Scala

## 🎫 A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io
3. About RoseTree in Scala
4. Move on at work

##  A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io
3. About RoseTree in Scala
4. Move on at work
5. PoC won't go in production anytime soon

##  A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io
3. About RoseTree in Scala
4. Move on at work
5. PoC won't go in production anytime soon
6. My talk is selected

## 🎫 A drama in 6 acts

1. PoC with RoseTree in Elm
2. Let's propose a talk for Scala.io
3. About RoseTree in Scala
4. Move on at work
5. PoC won't go in production anytime soon
6. My talk is selected

# Please be kind

# Please be kind

# Please be kind

# RoseTree

# RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
```

# RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
```

## without RoseTree

```
case class Thing(field1: String, field2: Int, children: Seq[Thing])
```

# RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
```

## without RoseTree

```
case class Thing(field1: String, field2: Int, children: Seq[Thing])
```

## with RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
case class ThingLabel(field1: String, field2: Int)
type Thing = Tree[ThingLabel]
```

# RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
```

## without RoseTree

```
case class Thing(field1: String, field2: Int, children: Seq[Thing])
```

## with RoseTree

```
case class Tree[T](label: T, children: Seq[Tree[T]])
case class ThingLabel(field1: String, field2: Int)
type Thing = Tree[ThingLabel]
```

## brings `map` `filter` `collect` … ❤️

# What about Chainsaw ?

## What about Chainsaw ?

- Alpha-level library

## What about Chainsaw ?

- Alpha-level library
- Ported from `gampleman / elm-rosetree`

## What about Chainsaw ?

- Alpha-level library
- Ported from `gampleman / elm-rosetree`
- ~~Not yet on maven central~~

## What about Chainsaw ?

- Alpha-level library
- Ported from `gampleman / elm-rosetree`
- ~~Not yet on maven central~~
- No stable API

## What about Chainsaw ?

- Alpha-level library
- Ported from `gampleman / elm-rosetree`
- ~~Not yet on maven central~~
- No stable API
- 🤗

## What about Chainsaw ?

- Alpha-level library
- Ported from `gampleman / elm-rosetree`
- ~~Not yet on maven central~~
- No stable API
- 🤗

Talk is cheap, show me the code

## Map on List

```
enum TypedTree:
  case 🌳
  case 🌴
  case 🌲
```

```
val parse: String => TypedTree = {
  case "Oak"       => 🌳
  case "PalmTree"  => 🌴
  case "Evergreen" => 🌲
}
```

```
List("Oak", "PalmTree", "Evergreen").map(parse).show()
// res5: String = "[🌳, 🌴, 🌲]"
```

## Map on Tree

```
val parse: String => HomeOrTree = {
  case "Oak"           => 🌳
  case "PalmTree"      => 🌴
  case "Evergreen"     => 🌲
  case "Neighbourhood" => `🏘`
  case "House"         => 🏡
}
val tree =
  Tree("Neighbourhood",
      Tree("House",
            Tree("Oak"),
            Tree("PalmTree")),
      Tree("House",
            Tree("Evergreen"),
            Tree("Evergreen"))
)
```

```
tree.map(parse).show("types")
// res7: String = """types
// 🏘
//  \--🏡
//      \--🌳
//      \--🌴
//  \--🏡
//      \--🌲
//      \--🌲
// """
```

# Filter List

```
val things = List(`🏘`, 🏡, 🌳, 🌴, 🏡, 🌲, 🌲)

val filteredTrees = things.filter {
  case 🌳 | 🌴 | 🌲 => false
  case `🏘` | 🏡   => true
}
```

```
filteredTrees.show()
// res9: String = "[🏘, 🏡, 🏡]"
```

# Filter Tree (1)

```
val tree =
    Tree(`🏘`,
        Tree(🏡,
              Tree(🌳),
              Tree(🌴)),
        Tree(🏡,
              Tree(🌲),
              Tree(🌲)))

val filteredTrees = tree.filter {
  case 🌳 | 🌴 | 🌲 => false
  case `🏘` | 🏡    => true
}.get
```

```
filteredTrees.show("without trees")
// res11: String = """without trees
// 🏘
//  \--🏡
//  \--🏡
// """
```

# Filter Tree (2)

```
val tree =
    Tree(`🏘`,
        Tree(🏡,
              Tree(🌳),
              Tree(🌴)),
        Tree(🏡,
              Tree(🌲),
              Tree(🌲)))

val filteredHomes = tree.filter {
  case 🏡                  => false
  case `🏘` | 🌳 | 🌴 | 🌲 => true
}.get
```

```
filteredHomes.show("without homes")
// res13: String = """without homes
// 🏘
// """
```

## Collect List

```
val parse: PartialFunction[String, HomeOrTree] = {
  case "Oak"       => 🌳
  case "PalmTree"  => 🌴
  case "Evergreen" => 🌲
}
```

```
List("Oak", "PalmTree", "Evergreen", "Olive Tree").collect(parse).show()
// res15: String = "[🌳, 🌴, 🌲]"
```

## Collect Tree

```
val parse: PartialFunction[String, HomeOrTree] = {
  case "Oak"           => 🌳
  case "PalmTree"      => 🌴
  case "Evergreen"     => 🌲
  case "Neighbourhood" => `🏘`
  case "House"         => 🏡
}
val tree =
    Tree("Neighbourhood",
      Tree("Neighbourhood", Tree("Oak"), Tree("PalmTree")),
      Tree("House", Tree("Oak"), Tree("PalmTree")),
      Tree("House", Tree("Evergreen"), Tree("Evergreen"), Tree("Olive Tree")),
      Tree("Building", Tree("Evergreen"), Tree("Evergreen"))
    )
```

```
tree.collect(parse).get.show()
// res17: String = """🏘
//  \--🏘
//      \--🌳
//      \--🌴
//  \--🏡
//      \--🌳
//      \--🌴
//  \--🏡
//      \--🌲
//      \--🌲
// """
```

## FoldLeft List

```
val energy: HomeOrTree => Int = {
  case 🌳 => 100
  case 🌴 => 5
  case 🌲 => 20
  case _  => 0
}
```

```
val result = List(🌳, 🌴, 🌲).foldLeft(0)((acc, tree) => energy(tree) + acc)
// result: Int = 125

s"$result eq. KWh"
// res19: String = "125 eq. KWh"
```

# FoldLeft Tree (1)

```
val tree = Tree(
  `🏠`,
  Tree(🏡, Tree(🌳), Tree(🌴)),
  Tree(🏡, Tree(🌲), Tree(🌲))
)

val energy: HomeOrTree => Int = {
  case 🌳 => 100
  case 🌴 => 5
  case 🌲 => 20
  case _  => 0
}
```

```
val result = tree.foldLeft[Int](0)((tree, acc) => energy(tree) + acc)
// result: Int = 145

s"$result eq. KWh"
// res21: String = "145 eq. KWh"
```

# FoldLeft Tree (2)

```
val tree = Tree(
  `🏚`,
  Tree(🏡, Tree(🌳), Tree(🌴)),
  Tree(🏡, Tree(🌲), Tree(🌲))
)
```

```
tree.zipWithPath.foldLeft(List.empty)((tree, acc) => acc :+ tree)
// res23: List[Any] = List(
//   (Path(elements = ArraySeq(0)), 🏚),
//   (Path(elements = ArraySeq(0, 0)), 🏡),
//   (Path(elements = ArraySeq(0, 0, 0)), 🌳),
//   (Path(elements = ArraySeq(0, 0, 1)), 🌴),
//   (Path(elements = ArraySeq(0, 1)), 🏡),
//   (Path(elements = ArraySeq(0, 1, 0)), 🌲),
//   (Path(elements = ArraySeq(0, 1, 1)), 🌲)
// )
```

# FoldRight Tree

```
val tree = Tree(
  `🏚️`,
  Tree(🏡, Tree(🌳), Tree(🌴)),
  Tree(🏡, Tree(🌲), Tree(🌲))
)
```

```
tree.zipWithPath.foldRight(List.empty)((tree, acc) => acc :+ tree)
// res25: List[Any] = List(
//   (Path(elements = ArraySeq(0, 1, 1)), 🌲),
//   (Path(elements = ArraySeq(0, 1, 0)), 🌲),
//   (Path(elements = ArraySeq(0, 1)), 🏡),
//   (Path(elements = ArraySeq(0, 0, 1)), 🌴),
//   (Path(elements = ArraySeq(0, 0, 0)), 🌳),
//   (Path(elements = ArraySeq(0, 0)), 🏡),
//   (Path(elements = ArraySeq(0)), 🏚️)
// )
```

# Traverse List

```
val parse: String => Either[String, HomeOrTree] = {
  case "Oak"       => Right(🌳)
  case "PalmTree"  => Right(🌴)
  case "Evergreen" => Right(🌲)
  case s           => Left(s"$s is not a tree")
}
```

```
val left = List("Oak", "PalmTree", "Evergreen", "Olive Tree")
        .traverse(parse)
        .fold(identity, _.show())
// left: String = "Olive Tree is not a tree"

val right = List("Oak", "PalmTree", "Evergreen")
        .traverse(parse)
        .fold(identity, _.show())
// right: String = "[🌳, 🌴, 🌲]"
```

# Traverse Tree

```
val parse: String => Either[String, HomeOrTree] = {
  case "Oak"           => Right(🌳)
  case "PalmTree"      => Right(🌴)
  case "Evergreen"     => Right(🌲)
  case "Neighbourhood" => Right(`🏘`)
  case "House"         => Right(🏡)
  case s               => Left(s"parse error for $s")
}
val valid = Tree(
  "Neighbourhood",
  Tree("House", Tree("Oak"), Tree("PalmTree")),
  Tree("House", Tree("Evergreen"), Tree("Evergreen"))
)
val invalid = Tree(
  "Neighbourhood",
  Tree("House", Tree("Oak"), Tree("PalmTree")),
  Tree("House", Tree("Evergreen"), Tree("Evergreen"), Tree("Olive Tree"))
)
```

```
val left = invalid.traverse(parse).fold(identity, _.show())
// left: String = "parse error for Olive Tree"
val right = valid.traverse(parse).fold(identity, _.show())
// right: String = """🏘
//  \--🏡
//      \--🌳
//      \--🌴
//  \--🏡
//      \--🌲
//      \--🌲
// """
```

# Conclusion

## Conclusion

- Trees are like most datastructure your know

## Conclusion

- Trees are like most datastructure your know
- But you didn't see any flatmap, right?

# Conclusion

- Trees are like most datastructure your know
- But you didn't see any flatmap, right?
- Many other operators
  - zip, ❤️ mapAccum ❤️, indexedMap, unfold ...

# Conclusion

- Trees are like most datastructure your know
- But you didn't see any flatmap, right?
- Many other operators
  - zip, ❤️ mapAccum ❤️, indexedMap, unfold ...
- Come contribute to Chainsaw

# Thanks!

Code and slides at mbaechler/chainsaw-slides on GitHub

Project at mbaechler/chainsaw on GitHub

## Questions?