Introduction

- FP becomes more popular (finally!)
- Time to have a look
- There are many useful patterns
  - Monoids
  - Better Error Handling
Lego vs Duplo

pictures from shop.lego.com
Lego vs Duplo

pictures from shop.lego.com
Lego vs Duplo

- **Duplo** favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse
Lego vs Duplo

- **Duplo** favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse

- **Lego** focuses on small composable building blocks
  - blocks can conveniently be reused for other purposes
  - limited use of specialized building blocks
Lego vs Duplo

- Duplo favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse
- Lego focuses on small composable building blocks
  - blocks can conveniently be reused for other purposes
  - limited use of specialized building blocks

OO tends to be like Duplo, FP tends to be like Lego
Monoids

- intuition: “combine stuff”
- you can create values from thin air via `Monoid.empty`
- combine two values via `Monoid.combine` / `|+|`
- additionally: laws (don’t write buggy implementations)
Monoid Laws

// 1) left identity
empty |+| x === x

// 2) right identity
x |+| empty === x

// 3) associative
x |+| (y |+| z) === (x |+| y) |+| z
Monoid Typeclass

```scala
trait Monoid[A] {
  def empty: A
  def combine(lhs: A, rhs: A): A
  // infix operator: |+|
}

implicit val plus: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine(lhs: Int, rhs: Int): Int = lhs + rhs
}

object Monoid {
  def apply[A: Monoid]: Monoid[A] = implicitly
}
```
Using Our Monoid

> Monoid[Int].empty
0
> Monoid[Int].combine(1,2)
3
> 1 |+| 2
3
> 42 |+| Monoid[Int].empty
42
> List(1,2,3).foldLeft(Monoid[Int].empty)(_ |+| _)
6
More Monoids

- **Monoid** instance not unique: addition / multiplication / min / max
- most collections are **Monoids**: List / Vector / Set
- let’s see some more examples
Monoid Zoo

List[A] is a Monoid
Monoid Zoo

\textbf{List[A]} is a \textbf{Monoid} \( A \Rightarrow B \) if \( B \) is a \textbf{Monoid}.
Monoid Zoo

List[A] is a Monoid if B is a Monoid
A => B if A and B are Monoids
(A,B)
Monoid Zoo

List[A] is a Monoid if B is a Monoid
A => B
(A,B)
Future[A] if A is a Monoid and B are Monoids

val m1 = Map("as" -> 21, "bs" -> 4)
val m2 = Map("as" -> 21, "cs" -> 2)
m1 |+| m2
// Map("as" -> 42, "bs" -> 4, "cs" -> 2)
Monoid Zoo

List\([A]\)
\(A \Rightarrow B\)
\((A, B)\)

Future\([A]\)

Map\([A, B]\)

is a Monoid
if B is a Monoid
if A and B are Monoids
if A is a Monoid
if B is a Monoid
Monoid Zoo

List[A]  
A => B  
(A,B)  
Future[A]  
Map[A,B] 
is a Monoid  
if B is a Monoid  
if A and B are Monoids  
if A is a Monoid  
if B is a Monoid

val m1 = Map("as" -> 21, "bs" -> 4)  
val m2 = Map("as" -> 21, "cs" -> 2)  
m1 |+| m2  
// Map("as" -> 42, "bs" -> 4, "cs" -> 2)
Monoids Compose (Lego Principle)

Config => A
Monoids Compose (Lego Principle)

\[
\text{Config} \Rightarrow A
\]

\[
\text{Config} \Rightarrow \text{Future}[A]
\]
Monoids Compose (Lego Principle)

- Config => A

- Config => Future[A]

- Config => Future[Map[String,A]]
Monoids Compose (Lego Principle)

Config => A

Config => Future[A]

Config => Future[Map[String,A]]

Config => Future[Map[String,(A,B)]]
Monoids Compose (Lego Principle)

1. Config => A
2. Config => Future[A]
3. Config => Future[Map[String,A]]
4. Config => Future[Map[String,(A,B)]]
5. Config => Future[Map[String,(A,Option[B])] ]
Monoids Compose (Lego Principle)

- Config => A
- Config => Future[A]
- Config => Future[Map[String,A]]
- Config => Future[Map[String,(A,B)]]
- Config => Future[Map[String,(A,Option[B])]}
- Config => Future[Map[String,(Set[A],Option[B)]]}
Lot’s of theory . . .
• analysis of a potentially huge text
• calculate metrics over text
  • word count
  • char count
  • min/max word length
  • avg word length
  • . . . (be flexible)
• **goal**: single traversal ↔ easy composition
RDDs and Folds

abstract class RDD[T] {
  /**
   * Aggregate the elements of each partition,
   * and then the results for all the partitions,
   * using a given associative function and a
   * neutral "zero value".
   */
  def fold(zeroValue: T)(op: (T, T) => T): T
}

Markus Hauck (@markus1189) Functional Programming Awesomeness 15 / 41
Monoidal RDDs

```scala
implicit class MonoidRDD[T](val rdd: RDD[T]) {

  // avoid conflicts with fold/reduce etc
  def combine(implicit M: Monoid[T]): T =
    rdd.fold(M.empty)(M.combine(_, _))

}
```
The Program

```scala
val sc: SparkContext = ???
val file: String = ???

val data = sc.textFile(file). // read the file
  .flatMap(_.split("\W+")) // split into words
  .map(expand) // action!

def expand(w: String) = (1, w.length, Map(w -> 1))

val (words, chars, wordMap) = data.combine
```

Markus Hauck (@markus1189)
Running this program

Scala Meetup: Rhein-Main Scala Enthusiasts
Running this program

Scala Meetup: Rhein-Main Scala Enthusiasts

Seq("Scala","Meetup","Rhein","Main","Scala", ...)

Markus Hauck (@markus1189)
Running this program

Scala Meetup: Rhein-Main Scala Enthusiasts

```
Seq("Scala","Meetup","Rhein","Main","Scala", ...)
```

```
Seq(  // expand(w: String) = (1,w.length,Map(w->1))
   (1, 5, Map("Scala" -> 1)),
   (1, 6, Map("Meetup" -> 1)),
   (1, 5, Map("Rhein" -> 1)),
   (1, 4, Map("Main" -> 1)),
   // ...
)
```
Running this program

Scala Meetup: Rhein–Main Scala Enthusiasts

Seq("Scala","Meetup","Rhein","Main","Scala", ...)

Seq(  // expand(w: String) = (1,w.length,Map(w->1))
    (1, 5, Map("Scala" -> 1)),
    (1, 6, Map("Meetup" -> 1)),
    (1, 5, Map("Rhein" -> 1)),
    (1, 4, Map("Main" -> 1)),
    // ...
)

(6,36,Map("Scala" -> 2,"Meetup" -> 1,...))
Easy Extension

```scala
val data: RDD[String] = ??? // as before

def expand(w: String) = (
  1,
  Max(w.length), // max word length
  Min(w.length), // min word length
  Map(w.length -> Set(w)) // words by count
)

val (count, max, min, byCount) = data.combine
```
The new program

Scala Meetup: Rhein-Main Scala Enthusiasts
The new program

Scala Meetup: Rhein–Main Scala Enthusiasts

Seq("Scala","Meetup","Rhein","Main","Scala", ...)

Markus Hauck (@markus1189)
The new program

Scala Meetup: Rhein-Main Scala Enthusiasts

Seq("Scala","Meetup","Rhein","Main","Scala", ...)

Seq(
(1, Max(5), Min(5), Map(5 -> Set("Scala"))),
(1, Max(6), Min(6), Map(6 -> Set("Meetup"))),
(1, Max(5), Min(5), Map(5 -> Set("Rhein"))),
(1, Max(4), Min(4), Map(4 -> Set("Main"))),
(1, Max(5), Min(5), Map(5 -> Set("Scala"))),
// ...
)
The new program

Scala Meetup: Rhein–Main Scala Enthusiasts

Seq("Scala","Meetup","Rhein","Main","Scala", ...)

Seq(
    (1, Max(5), Min(5), Map(5 -> Set("Scala"))),
    (1, Max(6), Min(6), Map(6 -> Set("Meetup"))),
    (1, Max(5), Min(5), Map(5 -> Set("Rhein"))),
    (1, Max(4), Min(4), Map(4 -> Set("Main"))),
    (1, Max(5), Min(5), Map(5 -> Set("Scala"))),
    // ...
)

(6,Max(11),Min(4),Map(5->Set("Scala","Rhein"),...))

Markus Hauck (@markus1189)
More Monoid Tricks

- “filter” values via `mempty` value
- `map + reduce` == two phase computation via monoids
- finger trees, choose Monoid, get random access list / queue /
  ...
- Monoids Theme and Variations (Functional Pearl)
Part Two: Errors
The Traditional Way

- Java style: `try/catch/finally`
- `checked` exceptions
  - compiler help
  - reduce return value checking
- `unchecked` exceptions
  - only visible via docs, if documented at all
  - the “way to go” in Java?
- errors? return `null` / custom classes
The Functional Way

- Scala: only *unchecked* exceptions
- but: *throw* and *catch* discouraged in FP anyway
- FP: type system + first class values = referential transparency!
Referential Transparency

- referential transparency (sounds scary?)
- enables you to replace $40 + 2$ with $42$
- local reasoning & equational reasoning ♡
- parts can be reused independently like lego bricks
Out Of The Box

- **Either / Try**
  - Either unbiased (before 2.12)
  - Try not a lawful monad...

- get the most by using a FP library
Try and Catch

```scala
def convert(is: String*): List[Int] =
  is.map(_.toInt).toList

convert("1","2","3","Hello World!","5")
```

```java
java.lang.NumberFormatException: For input string: "Hello World"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
...
```
Try and Catch: Not Compositional

```
try {
    convert("1","2","3","Hello World!","5")
} catch { 
    case e: NumberFormatException => println("Oops")
}
```

- type says nothing about errors
- only one option: fail fast
- what about getting all errors or non-fatal warnings
- problem is that `try/catch` does not compose (Duplo)
Example Time

- password validation
- constraints:
  - length ≥ 8
  - contains at least one number
  - contains no spaces
  - contains at least one upper char
Validating Passwords

```scala
sealed trait LoginError
case object PwTooShort extends LoginError
case object PwContainsSpace extends LoginError

def checkLength(s: String): Option[LoginError]
def checkSpace(s: String): Option[LoginError]
def convert[E](o: Option[E]): Either[E, Unit]

def login(s: String): Either[LoginError, Token] =
  for {
    _ <- convert(checkLength(input))
    _ <- convert(checkSpace(input))
    // ...
    token = createToken()
  } yield token
```
Using our Function

> cabbage
  Sorry the password must be more than 8 chars
> boiled cabbage
  Sorry, the password must contain at least one digit
> 1 boiled cabbage
  Sorry, the password cannot have spaces
> 50damnedboiledcabbages
  Sorry, the password must contain at least one upper char
Using our Function
Validated

- scenario: multiple unrelated error conditions, result is fail/success
- `Either`, `Xor`, etc. are made for fail-fast, or short circuiting
- solution: `Validated / Validation`
Passwords with Validated

```scala
sealed trait LoginError
  case object PwTooShort extends LoginError
  case object PwContainsSpace extends LoginError
  case object PwContainsNoDigit extends LoginError

  def checkLength(s: String): Option[LoginError]
  def checkSpace(s: String): Option[LoginError]
```

Markus Hauck (@markus1189)
Passwords with Validated

```scala
def convert[E](o: Option[E]): ValidatedNel[E, Unit]

def login(s: String):
  ValidatedNel[LoginError, Token] = {
    (convert(checkLength(input)) |+|
    convert(checkSpace(input)) |+|
    ...
  ).as(createToken())
}
```
Using Validated

```haskell
login("cabbage")
Invalid(NonEmptyList(PwTooShort,
  PwContainsNoDigit,
  PwContainsNoSpecialChar))
```

> cabbage
Sorry the password is too short (min. 8 chars), contains no digit and no special character
Using Validated
Ior (cats) and \&/ (scalaz)

- what about “non-fatal” exceptions
- instead of short circuiting, continue with warning
- there might still be fatal situations with short circuiting
- solution: Ior and \&/
First Class Errors

- replace `try / catch` with first class values
- `Either / Xor` and friends = fail-fast
- `Validated / Validation` = multiple independent errors
- `Ior` = fail/succeed/succeed with warnings
• FP has some nice patterns for you
• Monoids: combine stuff
• Functional Error Handling
Enough Duplo, time for Lego!

THANKS!