# Abstraction

Pepe García

2020-04-20

# Pure functions

In functional programming we say that functions have some special features.

# They're values

Functions can be treated as any other value in your program. They can be assigned to `val`s, returned from methods, taken as parameters...

# Purity

They're pure, meaning that they perform no side effects

# Idempotence

given the same input, always return the same output

## Side effects

- Throwing exceptions
- IO
- mutating variables
- Random

# Referential transparency

They're referentially transparent. We can safely substitute any function call with its return value and it won't have any effect in the overall program.

# Totality

They're total. Functions should operate on all values from it's input type. If they fail with an input, they're not total.

# Totality

They're total. Functions should operate on all values from it's input type. If they fail with an input, they're not total.

When a function is not total, we say it's partial.

# Examples

```
def sum(a: Int, b: Int): Int = a + b
```

Is this pure?

# Examples

```
def sum(a: Int, b: Int): Int = a + b
```

Is this pure?

**yes**

# Examples

```
def sum(a: Int, b: Int): Int = {
  println(s"summing $a + $b")
  a + b
}
```

Is this pure?

# Examples

```
def sum(a: Int, b: Int): Int = {
  println(s"summing $a + $b")
  a + b
}
```

Is this pure?

**Nope, it's side effectful!**

# Examples

```
def operation(): Unit = {
  launchMissiles()
  ()
}
```

Is this pure?

# Examples

```
def operation(): Unit = {
  launchMissiles()
  ()
}
```

Is this pure?

**Nope, it's side effectful!**

# Examples

```
def findPhone(name: String): Option[String] = None
```

Is this pure?

# Examples

```
def findPhone(name: String): Option[String] = None
```

Is this pure?

**Yes!**

# Examples

```
def findUser(id: Int): String = {
  db.findUser(id) match {
    case Some(x) => x
  case None => throw new Exception("No user found")
  }
}
```

Is this pure?

# Examples

```
def findUser(id: Int): String = {
  db.findUser(id) match {
    case Some(x) => x
  case None => throw new Exception("No user found")
  }
}
```

Is this pure?

**Nope, it's side effectful!**

# Examples

```
def toString(id: Int): String = id match {
  case 1 => "one"
}
```

Is this pure?

# Examples

```
def toString(id: Int): String = id match {
  case 1 => "one"
}
```

Is this pure?

**Nope, it's partial!**

# Recap

Yesterday we implemented a binary tree

```
sealed trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Node[A](
  l: Tree[A],
  a: A, r: Tree[A]
) extends Tree[A]
```

# Recap

```
val myTree = Node(
  Node(Empty(), 2, Empty()),
  1,
  Node(
    Node(Empty(), 4, Empty()),
    3,
    Node(
      Node(Empty(), 6, Empty()),
      5,
      Node(Empty(), 7, Empty())
    )
  )
)
```

# Recap

Abstraction is the ultimate goal of functional programming. If you see the implementations we ended up creating for yesterday's exercises, you'll see that there's a pattern:

# Recap

```
def height[A](tree: Tree[A]): Int =
  tree match {
    case Empty() => 0
    case Node(l, _, r) =>
      1 + (height(l).max(height(r)))
  }

height(myTree)
// res0: Int = 4
```

# Recap

```
def sum(tree: Tree[Int]): Int = tree match {
  case Empty() => 0
  case Node(l, x, r) => x + sum(l) + sum(r)
}

sum(myTree)
// res1: Int = 28
```

# Recap

```
def count[A](tree: Tree[A]): Int =
  tree match {
    case Empty() => 0
    case Node(l, _, r) =>
      1 + count(l) + count(r)
  }

count(myTree)
// res2: Int = 7
```

# Recap

```
def toStringNodes(tree: Tree[Int]): Tree[String] =
  tree match {
    case Empty() => Empty()
    case Node(l, x, r) =>
      Node(
        toStringNodes(l),
        x.toString,
        toStringNodes(r))
  }

toStringNodes(myTree)
// res3: Tree[String] = Node(
//   Node(Empty(), "2", Empty()),
//   "1",
//   Node(
//     Node(Empty(), "4", Empty()),
//     "3",
//     Node(Node(Empty(), "6", Empty()), "5", Node(Empty()
```

# Recap

```
def squared(tree: Tree[Int]): Tree[Int] =
  tree match {
    case Empty() => Empty()
    case Node(l, x, r) =>
    Node(
      squared(l),
      x * x,
      squared(r)
    )
  }

squared(myTree)
// res4: Tree[Int] = Node(
//   Node(Empty(), 4, Empty()),
//   1,
//   Node(
//     Node(Empty(), 16, Empty()),
//     9,
```

# Exercise 3

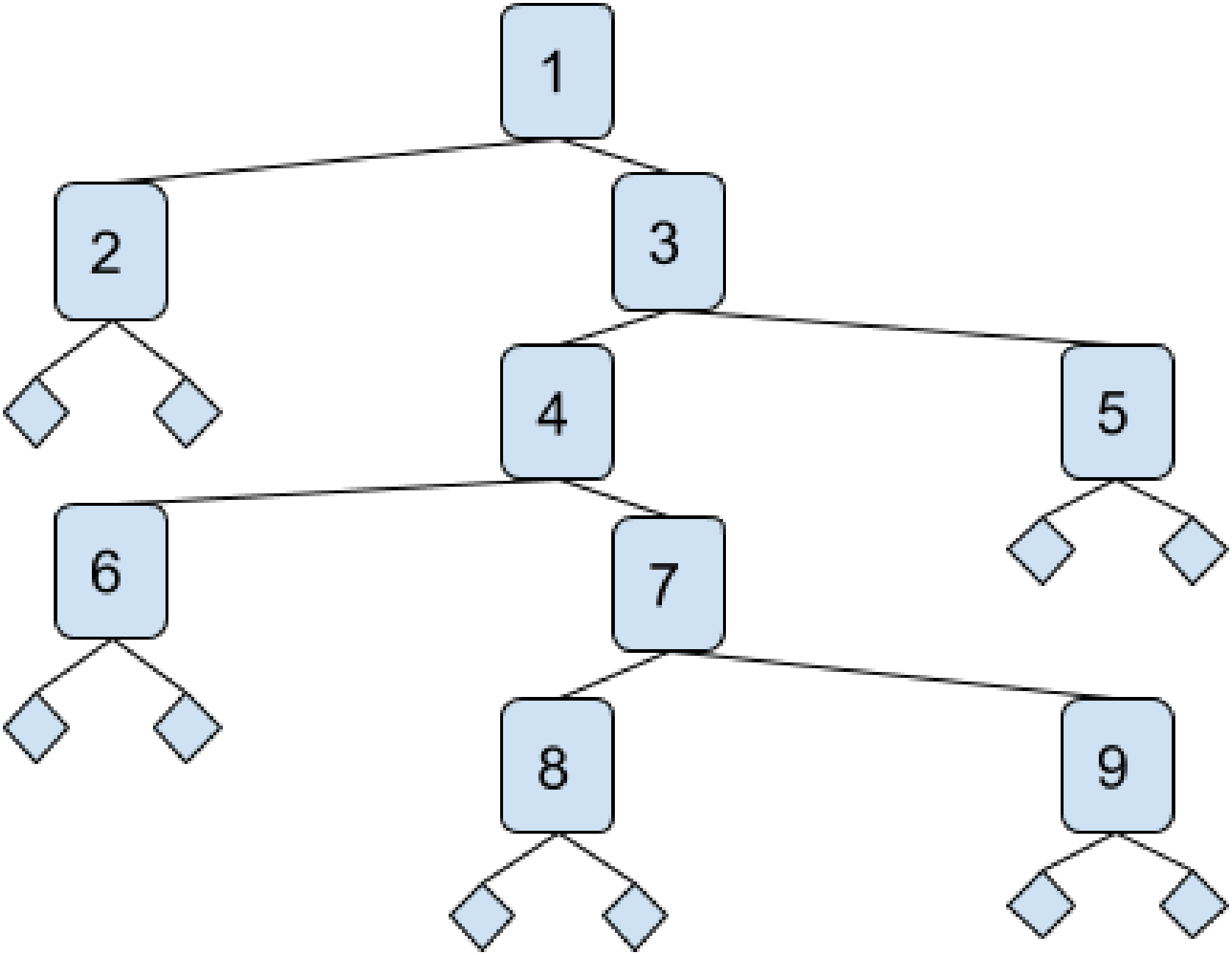
Can you create a higher order function with the common parts of previous functions?

# Identifying common functions

The pattern we've identified here is called `fold`, or more specifically, `catamorphism`.

Folds consume structures and create values out of them.

# Identifying common functions



# Identifying common functions

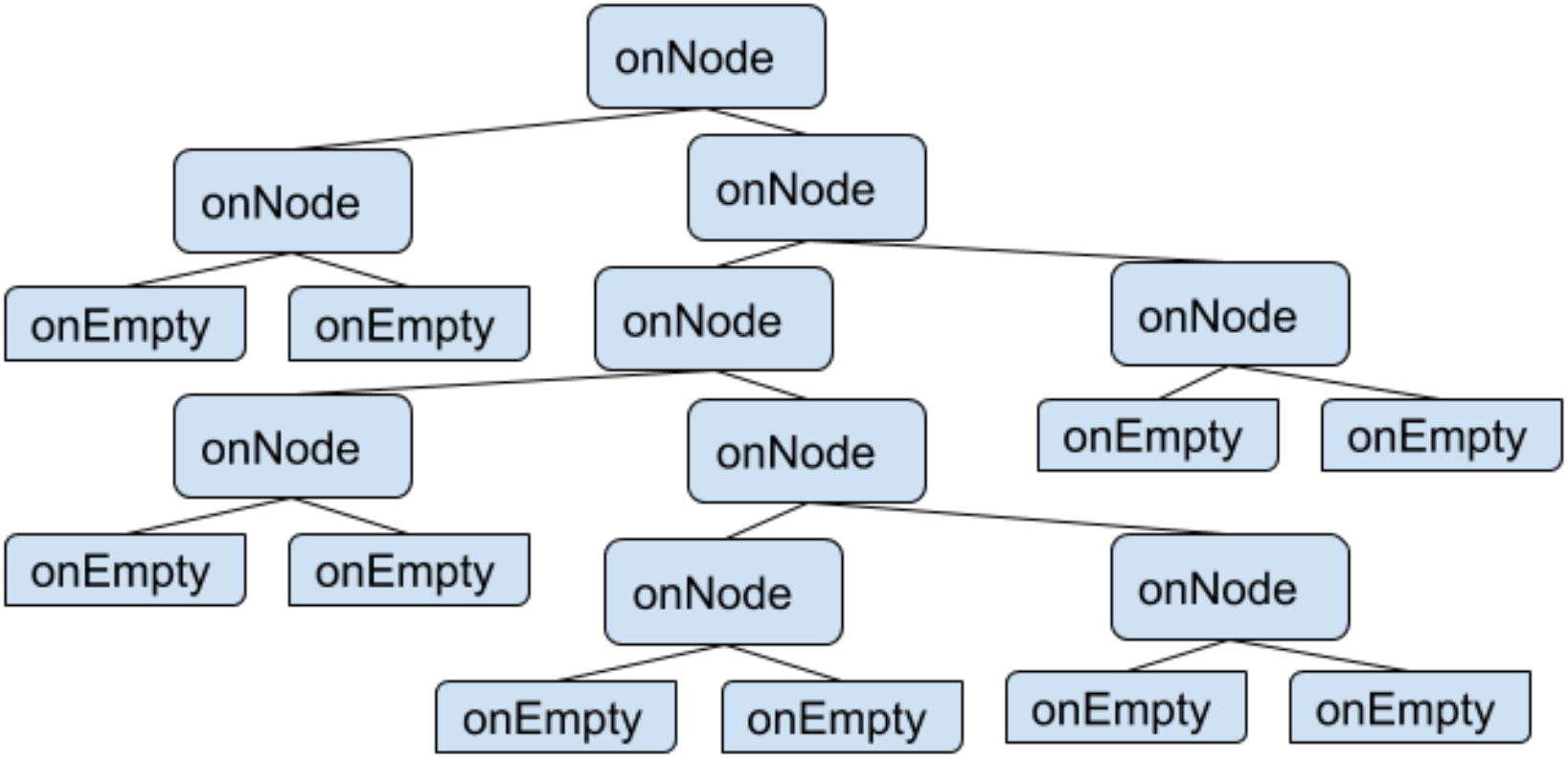



Figure 2: fold

# fold

```
def fold[A, B](tree: Tree[A])(
  onEmpty: B,
  onNode: (B, A, B) => B
): B = tree match {
  case Empty() => onEmpty
  case Node(left, a, right) => onNode(
    fold(left)(onEmpty, onNode),
    a,
    fold(right)(onEmpty, onNode)
  )
}
```

# abstraction

Now that we have created the `fold` function, let's reimplement the other functions based on it!

```
def heightWithFold[A](tree: Tree[A]): Int =
  fold(tree)(0, { (l: Int, _: A, r: Int) =>
    1 + (l.max(r))
  })

heightWithFold(myTree)
// res5: Int = 4
```

# abstraction

Now that we have created the `fold` function, let's reimplement the other functions based on it!

```
def sumWithFold(tree: Tree[Int]): Int =
  fold(tree)(0, { (l: Int, a: Int, r: Int) =>
    l + a + r
  })

sumWithFold(myTree)
// res6: Int = 28
```

# abstraction

Now that we have created the `fold` function, let's reimplement the other functions based on it!

```
def countWithFold[A](tree: Tree[A]): Int =
  fold(tree)(0, { (l: Int, _: A, r: Int) =>
    1 + l + r
  })

countWithFold(myTree)
// res7: Int = 7
```

# abstraction

Now that we have created the `fold` function, let's reimplement the other functions based on it!

```
def toStringNodesWithFold(tree: Tree[Int]): Tree[String] =
  fold(tree)(Empty[String](), { (l: Tree[String], a: Int, r
    Node[String](l, a.toString, r)
  })

toStringNodesWithFold(myTree)
// res8: Tree[String] = Node(
//   Node(Empty(), "2", Empty()),
//   "1",
//   Node(
//     Node(Empty(), "4", Empty()),
//     "3",
//     Node(Node(Empty(), "6", Empty()), "5", Node(Empty()
//   )
// )
```

# abstraction

Now that we have created the `fold` function, let's reimplement the other functions based on it!

```
def squaredWithFold(tree: Tree[Int]): Tree[Int] =
  fold(tree)(Empty[Int](), { (l: Tree[Int], a: Int, r: Tree
    Node[Int](l, a * a, r)
  })

squaredWithFold(myTree)
// res9: Tree[Int] = Node(
//   Node(Empty(), 4, Empty()),
//   1,
//   Node(
//     Node(Empty(), 16, Empty()),
//     9,
//     Node(Node(Empty(), 36, Empty()), 25, Node(Empty(),
//   )
// )
```

# abstraction

If we look closer at our last abstractions using fold, we can notice another pattern in the last two functions. They're transforming the value of the nodes, but leaving the structure intact.

That's a `map`! and we can implement it based on `fold`!

# abstraction

```
def map[A, B](tree: Tree[A])(fn: A => B): Tree[B] =
  fold(tree)(Empty[B](), { (l: Tree[B], a: A, r: Tree[B]) =
    Node[B](l, fn(a), r)
  })
```

# abstraction

And finally, implement those based on `map`!

```
def toStringNodesWithMap(tree: Tree[Int]): Tree[String] =
  map(tree)(_.toString)

toStringNodesWithMap(myTree)
// res10: Tree[String] = Node(
//   Node(Empty(), "2", Empty()),
//   "1",
//   Node(
//     Node(Empty(), "4", Empty()),
//     "3",
//     Node(Node(Empty(), "6", Empty()), "5", Node(Empty()
//   )
// )
```

# abstraction

And finally, implement those based on `map`!

```
def squaredWithMap(tree: Tree[Int]): Tree[Int] =
  map(tree)(x => x*x)

squaredWithMap(myTree)
// res11: Tree[Int] = Node(
//   Node(Empty(), 4, Empty()),
//   1,
//   Node(
//     Node(Empty(), 16, Empty()),
//     9,
//     Node(Node(Empty(), 36, Empty()), 25, Node(Empty(),
//   )
// )
```

# Intermezzo: Higher Kinded Types

We are used to have generics in other languages now, in Java for example, we don't need to implement a new List for every datatype we want to put inside, we have a *Generic* `List<A>` that can be used for all cases.

# Intermezzo: Higher Kinded Types

In Scala, we have already seen how to implement generics, we use **square brackets** to suround the generic parameters.

```
class Container[A](value: A)

def testGeneric[A](value: Int): A = ???
```

# Intermezzo: Higher Kinded Types

In the same way we have abstracted over types, we can abstract our functions over type constructors, or Higher Kinded Types:

```
def fn[M[_], A, B](
  ma: M[A]
)(
  fn: A => M[B]
): M[B] = ???
```

*Trivia*: Can you give a name to this particular function?

# Identifying functional patterns

That's all we've done in these last examples, find repetitions and try to abstract them. That's what FP is all about!

There are some well know abstractions we should be aware of

# fold

# fold

`fold` consumes a structure (`Tree` in our case) and produces a value out of it.

```
def fold[F[_], A, B](
  f: F[A]
)(
  onEmpty: B,
  onNode: (B, A, B) => B
): B = ???
```

# map

# map

`map` transforms each element given a function `A => B`

```
def map[F[_], A, B](f: F[A])(fn: A => B): F[B] = ???
```

# flatMap

# flatMap

`flatMap` is similar to `map`, but the lambda we pass to it returns a `F[B]` instead of a `B`

```
def flatMap[F[_], A, B](f: F[A])(fn: A => F[B]): F[B] = ???
```

# filter

# filter

`filter` returns a new structure `F[_]` with elements that doesn't adjust to a predicate `fn: A => Boolean` filtered out.

```
def filter[F[_], A](f: F[A])(fn: A => Boolean): F[A] = ???
```

# find

# find

`find` returns the first element in a structure `F[_]` that matches a predicate.

```
def find[F[_], A](f: F[A])(fn: A => Boolean): Option[A] = 
```

# functional datatypes

# functional datatypes

There are some datatypes in the scala standard library that help a lot with common tasks.

# Option

Option is used when something may be not present. Use it whenever you'd use null `null`.

## With `nulls`

```
def httpConnection: String = {
  if (hostDefined) {
    getHost
  } else {
    null
  }
}
```

# Option

Option is used when something may be not present. Use it whenever you'd use null `null`.

## With `Option`

```
def httpConnectionWithOption: Option[String] = {
  if (hostDefined) {
    Some(getHost)
  } else {
    None
  }
}
```

# Option

You can construct options with its two constrcutors `Some` & `None`, and you can also use the `Option` constructor, that will convert nulls in `None` if they occur

# Try

Try captures exceptions and represents them as `Failures` instead of throwing them!

This is very important because exceptions will now follow the path of normal values instead of bubbling on their own.

# Try

## Without Try:

```
def user: String = try {
  findUser(3)
} catch {
  case e: Exception =>
    throw e
}
```

# Try

With `Try`:

```
import scala.util.Try

def userWithTry: Try[String] = Try(findUser(3))
```

# Either

Either represents computations that can return two different values. One of the many use cases for `Either` is validations:

## Without Either

```
case class ValidationError() extends Exception()

def validatePhone(
  phone: String
): String = if (phone.length == 9) {
    phone
  } else {
    throw ValidationError()
  }
```

# Either

## With Either:

```
def validatePhoneWithEither(
  phone: String
): Either[ValidationError, String] =
  if (phone.length == 9) {
    Right(phone)
  } else {
    Left(ValidationError())
  }
```

# Future

Future represents computations detached from time. You know that those computations will happen, but do not when. It's useful when you have an expensive operation and don't want to block the current thread.

# Future

## Without Future

```
val callDB: Runnable = new Runnable {
  def run(): Unit = {
    db.findUser(3)
  }
}

new Thread(callDB).start
```

# Future

## with Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val callDBFuture: Future[String] = Future(findUser(3))
```

# Exercise 4