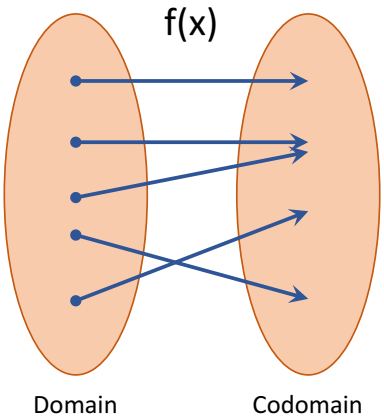
Functional Programming in Scala

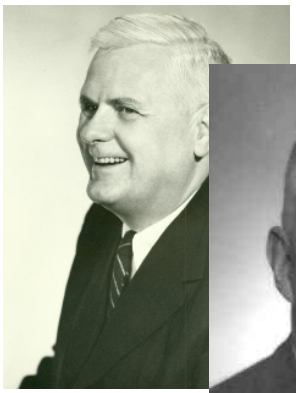
What is Functional Programming?

- Functional Programming views computation as function evaluation
  - in the mathematical sense
  - avoiding state and mutable data
- FP started with Lisp in the 1950s
  - but the ideas are much older than that…
- Today, many languages support FP
  - Haskell
  - Clojure
  - Scala
  - Java
  - C#
  - C++
  - Python
  - …

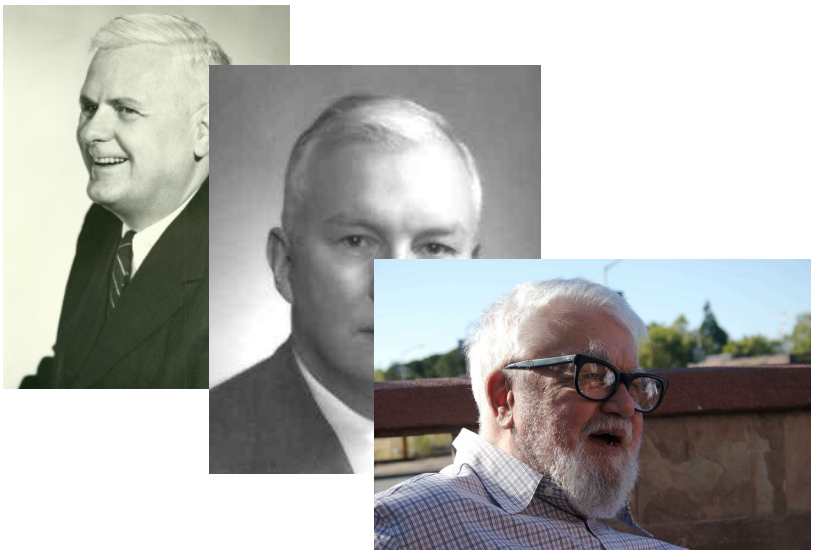


What is Functional Programming?

What is Functional Programming?



What is Functional Programming?

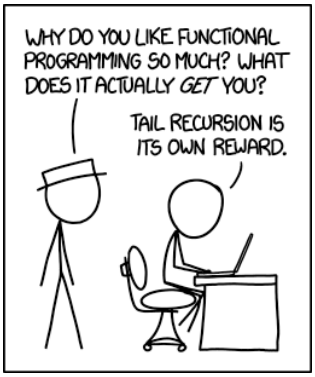


What is Functional Programming?

What is Functional Programming?

What Characterises Functional Programming?

- First-class and higher-order functions
  - lambdas
  - closures
- Immutable state
- Use of recursion
- Declarative style
- Lazy evaluation
- Type inference



What are FP's Strong Points?

- Higher-order functions and declarative style
  - concise expression of algorithms
  - efficient implementation
- Immutable state
  - less concern with concurrency problems
  - easy parallelization
- Especially well suited to Big Data processing
  - declarative style allows flexibility of implementation
  - thanks to ease of parallelization

Functional Programming in Scala

- We have already seen "functions" in Scala

```
scala> def twice ( i: Int ) = i * 2
twice: (i: Int)Int

scala> twice(3)
res59: Int = 6
```

- These are not strictly speaking first class functions
  - E.g. cannot be assigned

```
scala> val doubleIt = twice
<console>:12: error: missing argument list for method twice
…
```

Functional Programming in Scala

- First class functions are objects

```
scala> val doubleIt = (n: Int) => n * 2
doubleIt: Int => Int = <function1>

scala> doubleIt(3)
res60: Int = 6
```

- doubleIt is an instance of a type that extends Function1[Int, Int]
  - trait
  - How does the function get called?
- Many other Function*n* types – up to Function22[…]

Function Literals

- AKA Lambda Expressions

```
scala> (n: Int) => n * n
res61: Int => Int = <function1>

scala> ( (n:Int) => n * n )(10)
res62: Int = 100

scala> val squareIt = (n: Int) => n * n
squareIt: Int => Int = <function1>

scala> (a: Int, b: Int) => a + b
res65: (Int, Int) => Int = <function2>

scala> ((a: Int, b: Int) => a + b)(4,3)
res66: Int = 7
```

Higher Order Functions

- Functions that take other functions as arguments

```
scala> val doIt = ( i: Int, f: Int => Int ) => f(i)
doIt: (Int, Int => Int) => Int = <function2>

scala> doIt(10, doubleIt)
res63: Int = 20

scala> doIt(10, squareIt)
res64: Int = 100
```


Function Composition

- Chaining function invocations, building a pipeline of processing
  - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4
```

Function Composition

- Chaining function invocations, building a pipeline of processing
  - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>

scala> val h = f compose g
h: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4

scala> h(2)
res202: Int = 5
```


Function Composition

- Chaining function invocations, building a pipeline of processing
  - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>

scala> val j = g compose f
j: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4

scala> j(2)
res203: Int = 6
```

Function Composition

- Chaining function invocations, building a pipeline of processing
  - Fundamental principle of functional programming

```
scala> val f = (x: Int) => x + 1
f: Int => Int = <function1>

scala> val g = (x: Int) => x * 2
g: Int => Int = <function1>

scala> val k = f andThen g
k: Int => Int = <function1>
```

```
scala> f(2)
res199: Int = 3

scala> g(2)
res200: Int = 4

scala> k(2)
res201: Int = 6
```

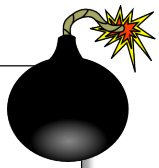

Higher Order Functions

- Functions that return functions

```
scala> val multBy = (n: Int) => ( (x: Int) => x * n )
multBy: Int => (Int => Int) = <function1>

scala> val double = multBy(2)
double: Int => Int = <function1>

scala> val triple = multBy(3)
triple: Int => Int = <function1>

scala> double(4)
res67: Int = 8

scala> triple(4)
res68: Int = 12
```

Closures

- Function definition may refer to external values
- Function "captures" or "closes over" these external values
- Beware closing over mutable state!



```
scala> var extVal = 100
extVal: Int = 100

scala> def foo(n: Int): Int = n * extVal
foo: (n: Int)Int

scala> foo(2)
res0: Int = 200

scala> extVal = 200
extVal: Int = 200

scala> foo(2)
res1: Int = 400
```


Methods and Functions

- Method can be converted to a function object
  - "lifting"
- Use explicit type

```
def myMethod(a: Int) = a + 10
```

```
scala> val myMethod3: Int => Int = myMethod
myMethod3: Int => Int = <function1>

scala> myMethod3(10)
res10: Int = 20
```

- Use special syntax

```
scala> val myMethod2 = myMethod _
myMethod2: Int => Int = <function1>

scala> myMethod2(10)
res8: Int = 20
```