# A very short introduction to functional programming

## Scala and OCaml...

Will Qi

December, 2013

# Outline - Scala

- Functional programming through $\lambda$-calculus
- Scala syntax in 10 minutes
- Scala object system and pattern matching
- Scala type system and tools for building abstractions
- Reactive programming

# Outline - OCaml

- $\lambda$-calculus in OCaml
- Lists, variants and pattern matching
- Powerful module system
- Concurrent programming with Async

# Functions

- In C++ a typical function application look like:

```
int foo(int a, char b, bool c) {
  // body
}

foo (10, 'x', false);
```

Functions are not first class citizens, it's hard to compose functions, although it's much better with `std::function` and `bind` that came with C++11.

# Functions

- In Scala functions can be defined:

  ```
  def foo(a: Int, b: Char, c: Boolean): Int = {
    if (c) a
    else b.toInt
  }
  ```

  or in a curried form:

  ```
  def foo(a: Int)(b: Char)(c: Boolean): Int = {
    if (c) a
    else b.toInt
  }
  ```

  Observe two things. Scala functions do not use `return` keyword as in C++. Scala functions can be defined as a curried function. These are two distinctive features of functional programming.

# Functions

- Currying is the process of transforming a function with multiple arguments into chained partial applications, e.g.

```
val f1: Char => Boolean => Int = foo(10)
val value = f1('x')(true) // 10
```

- In OCaml this is much more concise:

```
let f a b c = if c then a else int_of_char b
> f : int -> char -> bool -> int
let f1 = f 10
let value = f1 'x' true
```

# Expressions

In functional programming, all expressions result in values, even if the expression is purely side-effecting, the return type is `unit` which is equivalent to C++'s `void` type.

```
def foo(a: Int) {
  // purely for side effects
  // does not return anything
}
> foo: Int => Unit
```

The function signature of `foo` is `foo: Int => Unit`.

# I

mmutability is an epitome of function programming.

```
val l = List() // empty list => []
val l1 = 10 :: 11 :: 12 :: l // [10, 11, 12]
val l2 = l1.filter(x => x % 2 == 0) // [10, 12]
```

Where are `l1, l` now? They are still persisted in memory and are probably shared by `l2`.

# Recursion

Recursion is used very extensively in functional programming. Here is a recursive data structure Tree and a recursive function sum.

```
abstract class Tree[B]
case class Leaf[B](v: B) extends Tree[B]
case class Node[B](l: Tree[B], r: Tree[B])
  extends Tree[B]

type IntTree = Tree[Int]

def sum(t: Tree[Int]): Int = t match {
    case Leaf(v) => v
    case Node(l, r) => sum(l) + sum(r)
}
```

Notice we used pattern match on the tree, but it's not tail recursive, it will overflow the call stack...

# Recursion

Can Scala compiler do tail recursive optimization to this function? Answser is ”no” due to limitation of JVM. Sometimes if you add `@tailrec` annotation to a function, and the last step of your function calls itself, the Scala compiler can do TCO for you. e.g.

```
import scala.annotation.tailrec
def factorial(n: Int): Int = {
  @tailrec def factorialAux(acc: Int, n: Int): Int = {
    if (n <= 1) acc
    else factorialAux(n * acc, n - 1)
  }
  factorialAux(0, n)
}
```

# Classes and Objects

- Class definition and companion object. Each class can have a companion object (like a singleton object) where we can define methods/variables shared by the whole class.

```
class Person(age: Int) {
  def speaks = "age is " + age
}

object Person {
  def apply(age: Int): Person {
    new Person(age)
  }
}

val p = Person(99)
p.speaks // prints "age is 99"
```

# Classes and Objects

Traits are similar to Java interfaces. A class or object can extend multiple traits and the interfaces inherited are linearly stacked. They can be abstract or concrete.

```
trait Quacking {
  def quack = println("Quack quack quack")
}
trait Swimming {
  def action = println("Swim swim swim")
}
trait Yellow {
  def color = println("Yellow yellow yellow")
}
class Duck { }
val duck = new Duck with Quacking with Swimming with Yell
a.quack // "Quack quack quack"
a.color // "Yellow yellow yellow"
```