Week 3: Functional Programming

What is Functional?

- At its core, functional programming can be described as having as few statements and side effects as necessary.
- "Functions" are taken to be mathematical functions, in which they take and input and return a value, and do nothing else to the environment
- Functional programming is often "short and sweet".

Functional in Scala

- Scala's functional programming is most similar to Haskell's. However, in Scala, functional and OO are two sides of the same coin.
- Functional programming allows what would taken multiple statements in Java to be done in what takes one expression.

A Functional Example

```java
// Java
int[] ints = {1, 2, 3, 4, 5};
for(int i : ints) {
    if(i % 2 == 0)
        System.out.println(i * 3)
}
```

```scala
// Scala
println(
    List(1, 2, 3, 4, 5)
        .filter(_ % 2 == 0)
        .map(_ * 3)
        .mkString("\n")
)
```

Side Effects

- A "side effect" is anything a function does to change its environment outside its scope, such as writing a file.
- Side effects are conventionally avoided as much as possible in functional contexts.
- In Scala, nilary functions are declared with empty parens iff they have side-effects. `Unit` functions exist solely for side-effects.

Immutability

- Immutability is also common in functional programming. This means values cannot change after being assigned.
- Wide use of immutability makes closures easier to construct, and makes understanding program state less difficult, because greater immutability means less state to keep track of.

Lists

- Lists are extremely common in functional programming, as are operations to transform them.
- There are many list types in Scala, but `List` is one of the most commonly used.
- Unlike `scala.Array`, which is just a wrapping for hava's native arrays, `List` is a linked list; that is, it is a series of nodes that are an element and the rest of the list after it.

Lists

- `Lists` are constructed by prepending elements to `Nil`, the unique empty `List`, using `::` (pronounced cons).
- `List(1, 2, 3)` can be expressed as `1 :: 2 :: 3 :: Nil`.
- Beware that linked lists are best for list-wide operations, because element access is slow.

Function Literals (Lambdas)

- Lambda expressions, named for the λ-calculus, are functions with no name meant to be used once.
- Lambdas are a feature of Scala's functions-as-objects, which allows functions to be passed as values.
- Lambdas can be used to encapsulate behavior and avoid code repetition.

Function Literals (Lambdas)

```
// canonical syntax
(i: Int, j: Int) => { x + y }
// let the compiler guess the signature from context
(i, j) => x + y
// nilary function
() => println("hai")
// another nilary function
val a = { val i = something(); i + 1 }
// underscore notation
List(1, 2, 3).map(_ + 1)
```

Higher Order Functions

- A higher order function is any function that takes a function object as an argument.

- Higher order functions can be used to abstract away behavior.

```
// (Int, Int) => Int is the type of a function that
// takes two Ints and returns another Int
def combineAndSquare(x: Int, y: Int)(f: (Int, Int) => Int): Int {
    val res = f(x, y)
    res * res
}

combineAndSquare(2, 5)(_ - _ + 1) // 4
```

Currying

```
def foo(i: Int)(s: String) = s * i
val a = foo _ // type: (Int) => (String) => String
val b = foo(3) _ // type: (String) => String

println(b("foo")) // foofoofoo
println(a(2)("foo")) // foofoo
```

List Operations

- A list comprehension is transforming a list into something else.
- Scala provides a very large library of list comprehension functions.
- Four are more important, because the rest are based on them: `map`, `flatMap`, `foreach`, and `filter`

List Operations

```
val list = 1 to 5 // list of ints from 1 to 5

// map returns a list where each
// element is the result of a function
list.map(5 - _)
// flatMap is like map, but it flattens
// the resulting list of lists
list.flatMap(i => (1 to 3).map(_ * i))
// foreach does an operation on each
// element and ignores the result
list.foreach(println(_))
// filter returns a list of elements
// that match a given predicate
list.filter(_ < 3)
```

`for`-comprehensions

- `for`-comprehensions are a type of syntactic sugar for the four main list operations, and an analogue to the java `for(:)` loop.
- `for`-comprehensions can be used for looping as well as for creating new lists with `yield`.

`for`-comprehensions

```
// prints numbers 1 to 10
for(i <- 1 to 10)
    println(10)
// prints all combinations of the numbers 1 to 10
// except the ones where both numbers are the same
for(i <- 1 to 10; j <- 1 to 10; if i != j)
    println(s"$i $j")
// assigns a list of powers of 5 to `powers`
val powers =
    for(i <- 1 to 10; j = 5)
        yield Math.pow(j, i)
// equivalent to list.map(f)
for(el <- list)
    yield f(el)
```

Pattern Matching

- Pattern matching is similar to Java's `switch`, but more general and powerful.
- Matching can be used to concisely analyze the structure of an input, and to create type tests.
- Matching also returns a value, and does not have fall-throughs.
- Unlike in Java, a failed match will throw an exception. The compiler will warn when a match can fail.

Using matching

```
def mash(o: Option[_]): Int = o match {
    case Some("one") => 1
    case Some(i: Int) => i
    case Some(l: List[_]) => l.length
    case Some(s: String) => s.length
    case Some(_) => 0
    case None => -1
}

case class Foo(x: Any, y: Any)
def mash(f: Foo): String = f match {
    case Foo("toString", x) => x.toString
    case Foo(_, "toString") => "toString"
    case Foo(x: String, y: String) => x + y
    case Foo(_, _) => "nada"
}
```

Anatomy of a `case`

```
case x @ Foo(a @ Bar(1, y), b) if b.bar == y.baz => // ...
```

- A case is made up of two parts: a *pattern* and an optional *guard*.
- A pattern can be anything from a literal, to a type-ascribed variable, to a combination of variable bindings (the `@`s) and extractors (`Foo(...)`).

Extractor Objects

- An extractor is any object with an `unapply` method that takes a single argument and returns an Option with results.
- Extractors can be used in patterns, and usually reverse an equivalent `apply` method.
- Extractors are quite complex, and some details won't be covered.

match and List

```
val list = List(1,2,3,4,5)
val thing = list match {
   // extract the whole list
   case List(1, a, 3, b, 5) => List(a, b)
   // match on the list head
   case a :: 2 :: rest => a :: rest
   // same as above
   case List(a, 2, rest @ _*) => a :: rest
   // match an empty list
   case Nil => List()
   // why is this match not exhaustive?
}
```

Example list ops

- `reduce` - combines all elements of the list with a binary operator
- `zip` - combines two functions into tuples of elements at the same index
- `find` - returns the first element matching a predicate
- `forall` - checks that a predicate matches all elements
- `partition` - splits the list into the elements that do and don't match a predicate
- `take` - returns the first n elements of a list
- `drop` - returns the list without the first n elements
- Complete API: http://www.scala-lang.org/api/current/#scala.collection.immutable.List

Tail recursion

- Scala has loops as part of its baggage from Java. Pure functional programming languages don't have looping structures however.

- In Haskell, loops are implemented by making the return statement of a function a call to itself. The compiler can unroll the recursive call into a cheaper loop.

- Scala's compiler has a similar feature, which transforms such functions into `while` loops.

Tail recursion

```
def factorial(n: Int): Int =
    if(n <= 1) 1
    else factorial(n - 1) * n

def fibs(n: Int): List[Int] =
    if(n <= 2) List(1, 0)
    else {
        val last = fibs(n - 1)
        last.take(2).reduce(_ + _) :: last
    }
```

Laziness

- There are two important forms of lazy evaluation in scala: `lazy val` and call-by-name.
- A `lazy val` is a val whose value is not computed until the first time it is accessed, after which it is cached.
- Call-by-name is a mechanism that allows parameters to be evaluated lazily by turning them into lambdas.

Laziness

```
lazy val expensiveResult = {
    val x = myExpensiveComputation()
    x * 10
}

def until(cond: => Boolean)(f: => Unit): Unit = while(!cond) f

var foo = 10
until (foo == 0) {
    println(foo)
    foo -= 1
}
```