1 Functional Paradigms

1.1 Functional Programming

Mutation is where an attribute of a variable can change while the identity of the variable is maintained. For example, could define a polynomial class, then set a certain coefficient to a particular value.

Functional programming is a programming strategy that avoids mutation/reliance on state information. Immutable values are used. These can be transformed, but the idea is to minimise side effects. We don't want to pass an argument to a function that will then modify that argument. Input goes in, return value comes out with inputs unchanged.

The restricted definition of a functional programming framework is one in which there are no mutable variables, assignments, or imperative conttrol structures In a wider sense, functional programming can be carried out in any language that allows the construction of elegant programs that focus on functions

In scala, functions are first class objects. They can be treated and passed around just like any other variable (as in python)

1.2 call-by-name, call-by-value

Functions arguments in scala can be handled two ways. The argument can be called by name, or called by value.

```
//call by value
def CBVfunc(a:Int):Int = {...}
```

Function CBVfunc takes an int, returns an Int. The integer argument a is evaluated when the function is called.

```
//call by name
def CBNfunc(a: =>Int): Int = {...}
```

Function CBNfunc takes an integer argment (a), which is evaluated when it needs to be (or not at all!).

Not all functions terminate, infinte loops are a thing

Both call by value and call by name will reduce to the same outcome provided

- the reduced expression consists of pure functions (no state information/-side effects?)
- both evaluations terminate (no infinte loops)

if call by value terminates, then call by name will also converse is not guarenteed: call by name termination does not imply call by value termination in call by name, unused arguments are not evaluated

- could have a function that takes two arguments. The second argument is not used (always)

- could pass a non-terminating input to the CBN function, which is not used. no big deal
- call by value will try to evaluate it and get stuck

Below is an example of a function that will terminate when arguments are called by name but not when called by value.

```
// an infinite loop, this run indefinitely when evaluated
def loop = loop

// call by value function, arguments are evaluated once when the
    function is called
def mooseCBV(a: Int, b: Int):Int = a

// this evaluates loop, starting the infinte loop...
mooseCBV(1,loop)

// same as above, but using call-by-name (=>)
def mooseCBN(a: =>Int, b: => Int) = a

// returns 1, the loop is not evaluated
mooseCBN(1,loop)
```

Scala uses call by value by default, unless the function arguments are defined with `=>`

1.3 Conditionals

Boolean operations don't always need to evaluate the right hand operand (short circuit evaluation)

```
true && e -> e
false && e -> false
true || e -> true
false || e -> e
```

Things can be defined by name or by value. so `def x =loop` is a function, it is not evaluated untill it needs to be (call by name).

`val x =loop` evaluates to loop (call by value) immediately. this will kill your scala session/repl.

1.4 Recursion

Recursive functions must always have their return type explicitly defined (to make the compiler's life easier).

1.4.1 Tail Recursion

If a function calls itself as its last action, the stack frame can be reused Essentially it acts the same as a loop

If a functions last action is to call a function, (maybe the same, maybe different function), then the stack frame can be used - this is a tail-call (tail recursion is recursive tail-calling).

Tail recursive factorial example(works)

```
def factorial(N:Int) = {
    @scala.annotation.tailrec
    def currentProd(n:Int, prod:Int) :Int = {
        if (n==0) prod
        else currentProd(n-1,n*prod)
    }
    currentProd(N,1)
}
```

A lot of loops can be replaced by tail recursion. Usually this involves defining an inner function for the actual recursion, which accepts an accumulator argument in addition to other parameters. Recursive invocations of the inner function pass the current value of the accumulator, or return something when termination condition is met. For tail recursion (or recursion in general), it seems helpful to define the termination conditions at the very begining, then figure out the remaining logic.

1.5 Blocks and Scope

A block is defined by curly braces `{}`

Definitiones inside a block are invisible outside the block. Definitions from outside the block are visible inside, provided they have not been shadowed.

2 misc