**WHY RECURSION?**

Recursion avoids mutable state associated with loops.

LECTURE 1

Scala code produces Java bytecode

A scala worksheet is a REPL (Read, Eval, Print, Loop)

In a worksheet, each line of code is an expression that evaluates to something

Val x = 1 – value which cannot be changed via a side-effect

Val y = 1 – variable that can be changed

Structure of a function

def <functionName> (param list): <returnType> = <expression>

-in Scala we can use newline liberally. We can use indentation just like in Python (true for Scala 3)

-during this lecture, each if expression must be accompanied by an else. An if expression must evaluate to something

-a good coding practice is to write if expressions as follows:

If(<condition) <then\_expression>

else <else\_expression>

ABOUT SIDE EFFECTS

The “imperative” style for computing the factorial (C or Java) is:

Int fact =1

For(int I = 2;i<=n;i++)

Fact \*=i

We use side-effects: variable “fact is continuously updated during the execution of the program

The functional style for implementing factorial:

-we use recursion and recursive calls

Fibonacii function

-fibAux is an inner function and its visibility scope is restricted

We can define code blocks using {}

Even code blocks are expressions that evaluate to something

A code block will always evaluate to its last expression

GET FAMILIAR WITH TAIL RECURSIVE FUNCTIONS

LECTURE 3

Higher order functions-may take other functions as param, may return other functions

Why higher order functions are useful?

Suppose we want to compute of the sum of all vals of [start,stop].-sumOf()

Suppose now we want to write the sum of squares. squaresOf() [start^2…..stop^2]

The code is similar so we can generalize it.

[start,stop] = a1,a2…an f(x) = x

sumOf() : a1+a2+…an f(x) = x\*x

squaresOf() : a1^2 + a^2 + …… +an^2

How can we generalize? : f(a1) + f(a2)+….f(an)

Types seen so far: Int, Boolean, Double, Char

Function types: represented used “=>”. Int => Int -takes an int, returns an int

(Int,Boolean) = >Int – takes an int and boolean, returns an int

Anonymus function: just a function (inline function-short one) which we use only ones and we do not want to assign a name to it (example : def id(x:Int): Int = x and def square(x:Int): Int = x\*x)

We can simply write sumWithf((x:Int) => x,0,10)

SECOND PART: Higher order functions return other functions

Withf will take a function f:Int=> Int as input and return another function (Int,Int) => which will perform the sum f(a1) + f(a2)+….f(an)

We have a special syntax when function returns other functions

Curried(or curry) functions vs uncurry(uncurried) functions

Haskel Curry – logician

Curryf – takes its parameters IN TURN –

Curryf(1) - function with 2 params

Curryf(1)(2) - function with 1 param

Curryf(1)(2)(3) – the val 6

A function is uncurry is it takes all its parameter “at once”

We can use higher order function TO SEPARATE FUNCTIONALITY or our code

Suppose we would like to find an interval [x,x+10] such that the sum of the range has a certain property (it is an even number)

We start with x=0, we compute 0 + 1+…+10, we repeat for x =1…

A better way and more general is to **separate** the computation process, from the **search** process

**LECTURE 3**

HW1:Sets are functions

Functions as param returns functions

LISTS – collection of elements together with an ordering

First elem of list-head -> tail-pointer of references to the other elements(rest of the list)

Insert-at the beginning of the list in scala

Complexity – constant

Retrieving – linear complexity of n(size of list)

Lists are immutable – operations do not change the existing lists but create new lists from existing ones

Construnctors -NIL-empty list

Cons(e,l)-new list whose head is e and tail is l

LIST observers -isempty(l), head(l), tail(l)