L5: More Functional Programming

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

January 21st, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Functional Language

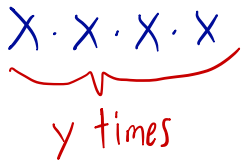
- Now that you played around with Scala

- Let’s formally define what is a functional language

Repetion (Instead of a Loop)

- Recursion is the answer to looping
  - Calling the function itself again → next iteration
- You can write
  def func_name(v1: Type, …, vN: Type): RetType = {
  …
  }
  - The return type is optional unless your function is recursive

Example



- Let's say you want to write pow(x: Int, y: Int): Int
  - For x!=0 and y>=0, and evaluates to x^y
- You can write
  def pow(x: Int, y: Int): Int = if(y ==0) 1 else pow(x, y-1)*x

- While Scala has a loop, recursion is a more powerful construct than a loop
  - So, use recursion for now
- Functions are values → defining a function just binds the expression to a name
  - No actual execution happen during the binding process

Compound Data

- So far, we have talked about a single data item
  - Number
  - Boolean
  - Conditionals
  - Variables
  - Functions
- Let’s look at a way to build up data with multiple parts

Tuple

- A fixed number of items, each can have different type
- Example:
  - (“hello”, 1) will results in a value of type (String, Int)

Tuple – A Pair

- For a pair, the rule is simple
  - Value: if e1 → v1 and e2 → v2, then (e1,e2) → (v1,v2)
  - Type: if e1 : t1 and e2 → t2, then (e1, e2) : (t1, t2)

- You can bind a pair to a name
  - Val t = (“hello”, 1)
- Accessing each component by
  - t._1 for the first item
  - t._2 for the second item

Val t = (1, 2, 3, "C", 4)

(Int, Int, Int, String, Int)

t._3 => 3

- Generally, you can use ._k to get the k^{th} item
- You can also unpack the pair using val
  - val (a, b) = t

Examples

def swap(t: (Int, Int)): (Int, Int) = (t._2, t._1)

return type

- def swap(p: (String, Int)) = (p._2, p._1)
- def swapInts(p: (Int, Int)) = (p._2, p._1)
- def sum(p: (int, int)) = (p._1 + p._2)
- def order(p: (int, Int)) = if (p._1 < p._2) p else swapInts(p)

Already Sorted

- Then, you can have k-tuple by using this same concept
- You can also have nested tuples (tuples within a tuple)

List

- Lists in Scala and most functional language are front-access lists
- List() makes an empty list
  - Type List[Nothing]
  - We can force a type by saying List[Type]
    - Example: List(): List[Int]
- You can also make a list with elements in it
  - List(1,2,3,1,2)
- You can stick element to the front of the list
  - You will get a brand new list
- Specifically
  - If e1→v, e2→l = [v1, v2, … vn], where e1: T and e2: List[T] then e1::e2 has the type List[T] and represent [v, v1, v2, … vn]

Append to the front

Type

v::(v1, v2, v3)

(v, v1, v2, v3)

Accessing a List

- Let's discuss some standard functions for a list
- Check if a list is empty
  - If L is a list, then L.isEmpty is true if L is empty
- Access the head
  - If L is non-empty, L.head evaluates to the head element of L
  - Else, you get an exception
- Access the tail (Subtract the first item)
  - If L = [v1, v2, … ,vn], then L.tail is the list [v2, …, vn]
  - Notice how you got the remaining elements

List L

if L is empty → return (List of nothing)

else (L.tail)

↓

get last item

if return nothing (L.head and L.tail are equal)

↳ L.head

Examples

- How can I summarize items in my list xs?
  - def sumList(xs: List[Int]): Int = if (xs.isEmpty) 0 else xs.head + sumList(xs.tail)
- How can I create a descending list of range n?
  - def descRange(n: Int): List[Int] = if(n==0) Nil else n::descRange(n-1)
- How can I concatenate two lists together?
  - Def concat(xs: List[Int], ys: List[Int]): List[Int] = if(xs.isEmpty) ys else (xs.head)::concat(xs.tail, ys)

sum of empty list = 0

head

List of the rest except the head

List of n down to 1

Add to the front (n, n-1, n-2, ..., 1)

End of recursion

↳ Append head of xs to ys

Pattern Matching

- Can I do switch-case in functional programming?
- Yes! Use
  - selector match { alternative }

- Example
  - Def sumList(xs: List[Int]): Int = xs match {
    case Nil => 0 List is empty
    case h::t → h + sumList(t)
    }
    xs.head :: xs.tail
- This pattern-match your list with each cases

Pattern Matching

- Benefits:
  - Gets a warning if you are missing any cases
  - Gets a warning if you have duplicate cases
  - Most concise, and hopefully more readable
    - Compared to tons of functions …
- Example: You can also use pattern matching to break down tuples in a list
  - Let's say you have a list of (Int, String)
    xs match {
    case (number, name)::t => … (non empty list case)
    …. // Other cases here
    }
  - This will break down to the numbers and names for you

Styles

- For the following code We know the last value already (n)

```
def countUpFrom1(n: Int): List[Int] = {
    def count(from: Int, to: Int): List[Int] =
        if (from == to) Nil else from :: count(from+1, to)
    count(1, n)
}
```

- In this case, you definitely know to = n so you can write

```
def countUpFrom1(n: Int): List[Int] = {
    def count(from: Int): List[Int] =
        if (from == n) Nil else from :: count(from+1)
    count(1) → Count from 1 to n
}
```

Before We Leave Today

In-class Exercise 5

- Write the following functions:
  - sumPairList(xs: List[(Int, Int)]): Int adds up all the numbers (both in the first and second coordinates).
  - firsts(xs: List[(Int, Int)]): List[Int] returns a list that extracts the first coordiate.
  - seconds(xs: List[(Int, Int)]): List[Int] returns a list that extracts the second coordiate.
  - pairSumList(xs: List[(Int, Int)]): (Int, Int) returns a pair where the first number is the sum of the first coordiates, and the second number is the sum of the second coordiates

- Submit them to in-class exercise 5